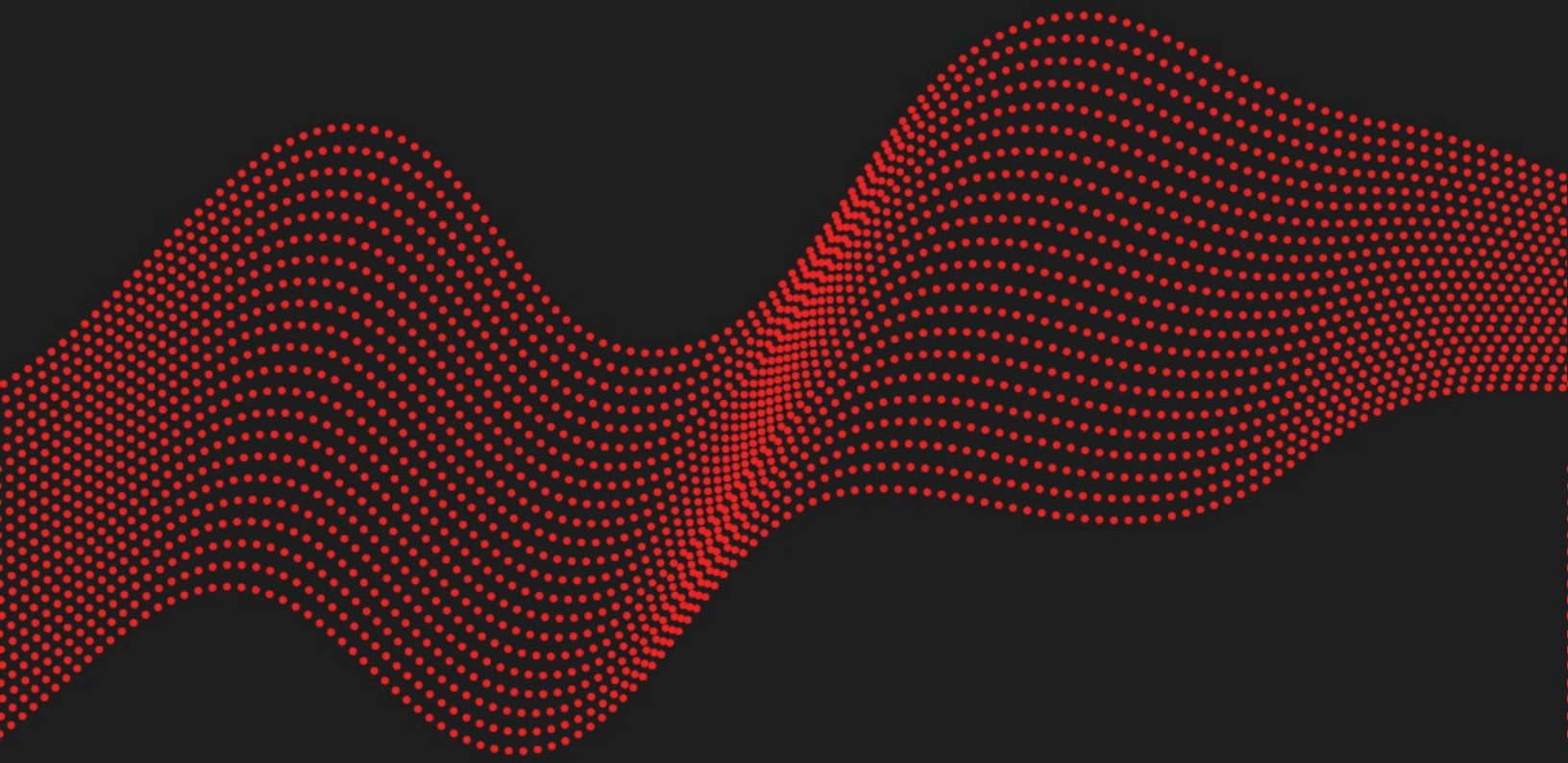
SYNACKTIV

ANSSI – CSPN Technical Evaluation Report

KeePassXC

2025/10/28



Identification of the document

Document specifications

Objet	RTE - Technical report
Nombre de pages	167
Diffusion	Public
Reference	CSPN-2024-KEEPASSXC

Document history

Version	Date	State
0.9	2024/12/18	Before validation
1.0	2024/12/10	First version – Validated
1.1	2025/07/17	ANSSI's remarks corrections
1.2	2025/09/11	ANSSI's remarks corrections
1.3	2025/10/28	PUBLIC classification

Contents

Identification of the evaluation technical report

Name of the evaluated product	KeePassXC
RTE Reference	CSPN-2024-KEEPASSXC
Authors	Synacktiv

Identification of the evaluated product and the security target

References and versions of the target of evaluation

Company Editor	KeePassXC Team (https://keepassxc.org/team/)
Product Name	KeePassXC
Name of the target of evaluation	-
Evaluated version	2.7.9
Potential fixes applied	-
Security target	1.7 (2025-10-28)
Product Category	Secure storage
Miscallenous	-

Identification procedure of the evaluated product

The version of KeePassXC is available through the "About" and "Help" menu:

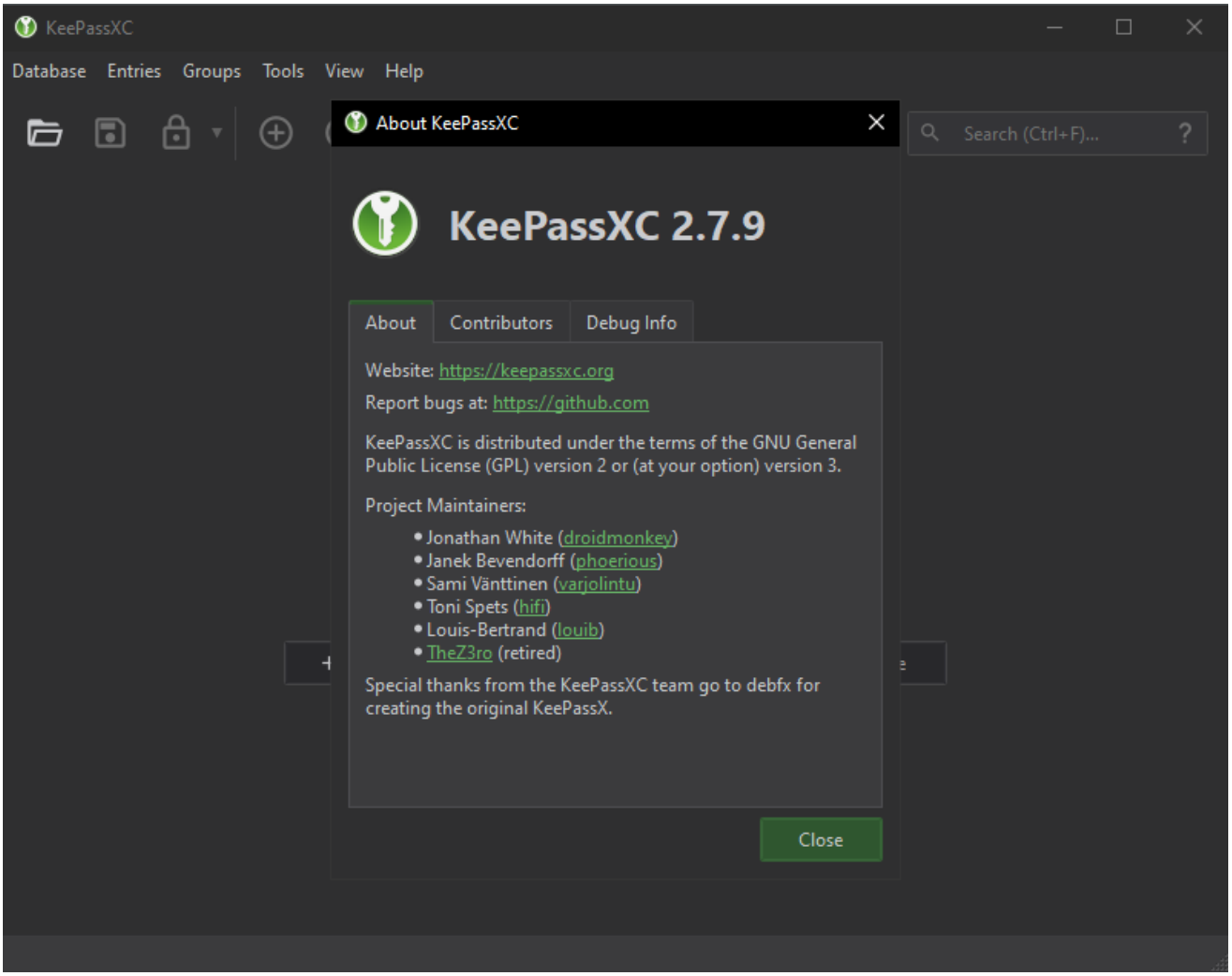


Illustration 1: KeePassXC version.

Details of the evaluation work

Compliance and robustness analysis

Environment and security problem

Requirements specification and security problem

Consistent with paragraph 2 of the security target document [Synacktiv-KeePassXC-cible_cspn]

Usage and environment / Product description

Consistent with paragraph 2.2 and 2.3 of the security target document [Synacktiv-KeePassXC-cible_cspn]

Expert opinion and identified potential vulnerabilities

During the evaluation, experts did not find any vulnerability nor inconsistency with the security target.

Application of the product

Installation

The evaluation aims to assess the security of the KeePassXC program as installed and configured by most users on the Windows operating system. The KeePassXC software was downloaded from the official website: https://keepassxc.org.:

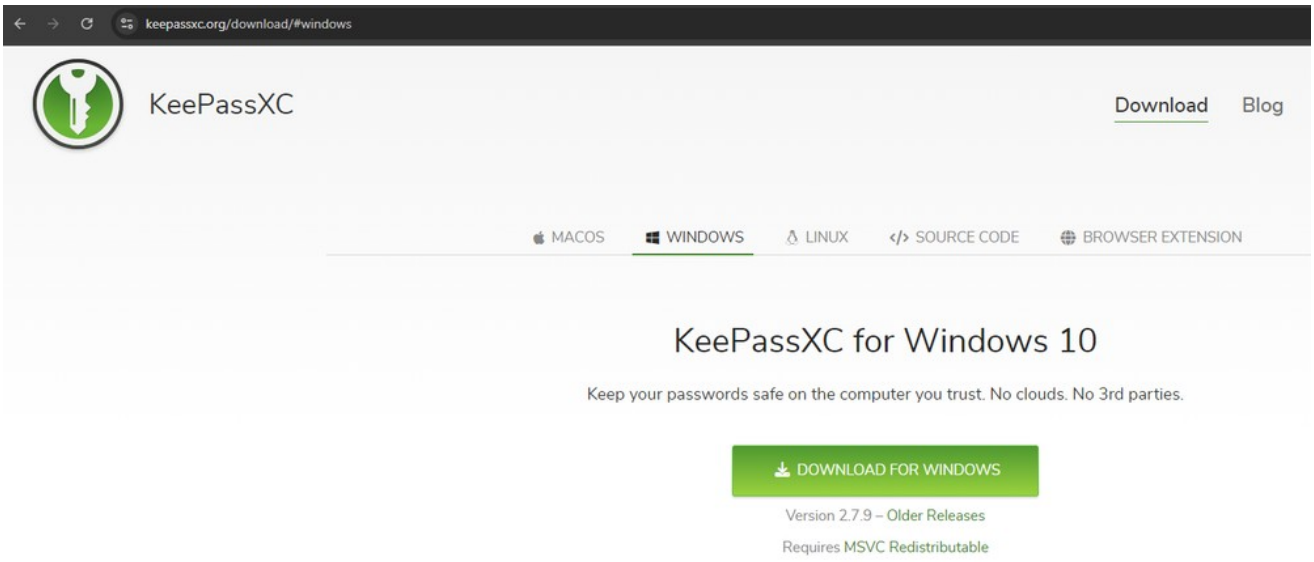


Illustration 2: KeePassXC official website.

The program comes as an MSI package, which is the traditional Windows package format. The SHA-256 digest of the MSI file is available on the same page as well as the PGP signature and a link to procedure explaining how to verify PGP signature and the digest correspondence. Both procedures are clear and properly walk through the operation.

The PGP key signing key is hosted in the **keys.opengpg.org** repository:

```
$ gpg --keyserver keys.openpgp.org --recv-keys BF5A669F2272CF4324C1FDA8CFB4C2166397D0D2
gpg: key CFB4C2166397D0D2: public key "KeePassXC Release <release@keepassxc.org>"
imported
gpg: Total number processed: 1
gpg:               imported: 1
```

The procedure also invites users verifying the publisher: DroidMonkey Apps, LLC.

Once the verification was performed, the MSI package can be unpacked and its content being extracted to **C:\Program Files\KeePassXC** by default. The overall installation process is fast and is similar to any classic Windows program installations. For this evaluation, no specific configuration nor modification of the environment was performed.

During the evaluation, the KeePassXC program was used in its AMD64 version compiled for Windows 10.

Ease of use

KeePassXC behaves as most Password Manager solution, it allows managing different password databases, opening them and adding, editing or deleting entries inside them. The most sensitive feature that could be misused by users is choosing a weak database's passphrase, but KeePassXC kindly warns the user when weak passphrases are chosen, and it encourages the user to choose a more robust one:

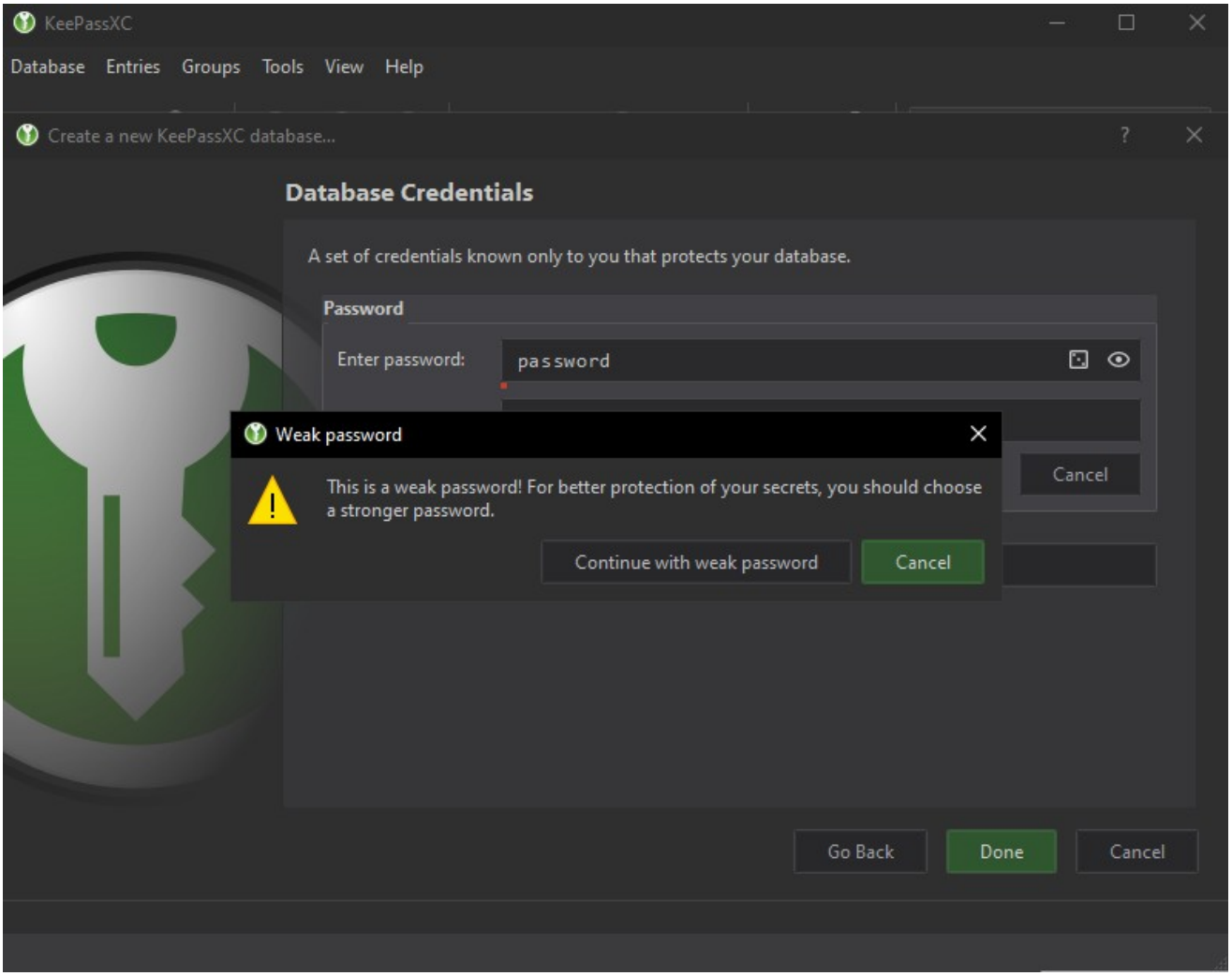


Illustration 3: KeePassXC warning.

Furthermore, default options are sufficiently secured by default and do not put the user or its data at risk. The options are well described and their use and impact are clear.

Expert opinion and identified potential vulnerabilities

The product is considered easy to use by users, with no possibility of inadvertently overriding safety measures..

Design and development

Documentation and supplies

KeePassXC 's documentation is available at https://keepassxc.org/docs/.

Moreover, the target of evaluation [Synacktiv-KeePassXC-cible_cspn] and cryptographic specifications [Synacktiv-KeePassXC-specs_crypto] were provided for the analysis.

KeePassXC is an open-source tool, so the full source code was available for the analysis.

Moreover, the official GitHub of the product provides a wiki[1] containing precious information developers.

The official "Getting Started" guide[2] is thorough and easy to access. It contains every installation step as well as a description of each feature and how to use it. Security advice regarding good practice are noted in the documentation, for example regarding the database key.

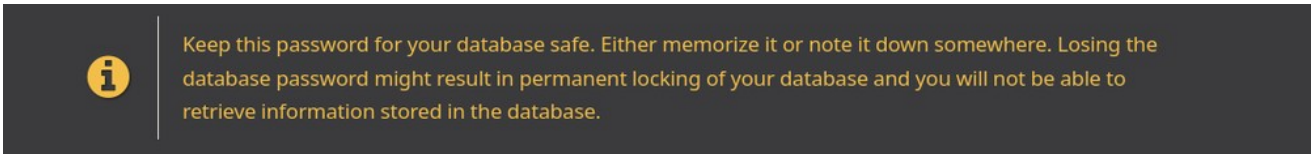


Illustration 4: Warning regarding database password.

Other messages warn user about practice that would decrease the security level, for example regarding the storage of a TOTP code in the same database as the password.

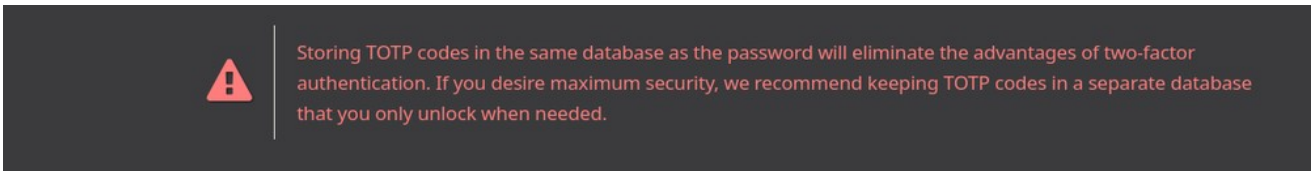


Illustration 5: Warning message regarding TOTP code storage.

The document targets general users of the tool, even users without security knowledge. With this documentation, users are guided toward good practices.

Dependencies analysis

KeePassXC being open source, the project provides the dependencies to download for the build process. On the GitHub repository, these dependencies are described in the file `vcpkg.json` (https://github.com/keepassxreboot/keepassxc/blob/develop/vcpkg.json). As part of this evaluation, each component has been reviewed to check whether they were affected by known vulnerabilities.

Component	Version >=	Vulnerabilities
`argon2`	20190702	No vulnerability identified affecting this version. However, the project seems unmaintained.
`botan`	3.1.1	CVE-2024-34702: Denial of Service Due to Excessive Name Constraints CVE-2024-39312: Authorization Error due to Name

1 https://github.com/keepassxreboot/keepassxc/wiki

2 https://keepassxc.org/docs/KeePassXC_GettingStarted

		Constraint Decoding Bug CVE-2024-34703: DoS due to oversized elliptic curve parameters The three bugs occurred in X.509 and ASN.1 parsing which is not used by KeePassXC.
`minizip`	1.3	CVE-2023-45853: MiniZip in zlib through 1.3 has an integer overflow and resultant heap-based buffer overflow in zipOpenNewFileInZip4_64. KeePassXC uses the affected function in the KeeShare feature when exporting compressed databases. However, the impact is low as the vulnerability is reachable by the current database's user.
`libqrencode`	4.1.1	The version is the latest available.
`libusb`	1.0.26.11791	No vulnerability identified affecting this version.
`libxi`	1.8	No vulnerability identified affecting this version.
`libxslt`	1.2.4	No vulnerability identified affecting this version.
`qt5`	5.15.11	No vulnerability identified affecting this version.
`qt5-imageformats`	5.1.5.11	
`qt5-macextras`	5.1.5.11	
`qt5-svg`	5.1.5.11	
`qt5-tools`	5.1.5.11	
`qt5-translations`	5.1.5.11	
`qt5-wayland`	5.1.5.11	
`qt5-x11extras`	5.1.5.11	
`readline`	0#5	No vulnerability identified affecting this version.
`zlib`	1.3	CVE-2023-45853: MiniZip in zlib through 1.3 has an integer overflow and resultant heap-based buffer overflow in zipOpenNewFileInZip4_64. KeePassXC uses the affected function in the KeeShare feature when exporting compressed databases. However, the impact is low as the vulnerability is reachable by the current database's user.

Cryptographic specifications analysis

The cryptographic specifications were written by Synacktiv alongside the security target and based on a quick source code analysis. During the evaluation, some differences have been noted between the

provided specifications and what was observed during the evaluation work. This chapter is thus mostly irrelevant, the cryptographic analysis was eventually performed by a more thorough source code analysis and is documented in a dedicated chapter: Cryptographic mechanisms page 117.

The only notable difference between the source and the specifications is regard with symmetric ciphers. KeePassXC supported several other algorithms.

Symmetric Ciphers are used for the database encryption and decryption. The default algorithm is AES-256-CBC, but TwoFish 256 or Chacha20 are selectable.

The ANSSI Cryptography Guide [ANSSI-PG-083] recommends the use of AES (page 15) and Chacha20 (page 16).

It is also important to note that the provided cryptographic specifications were written by Synacktiv alongside the security target.

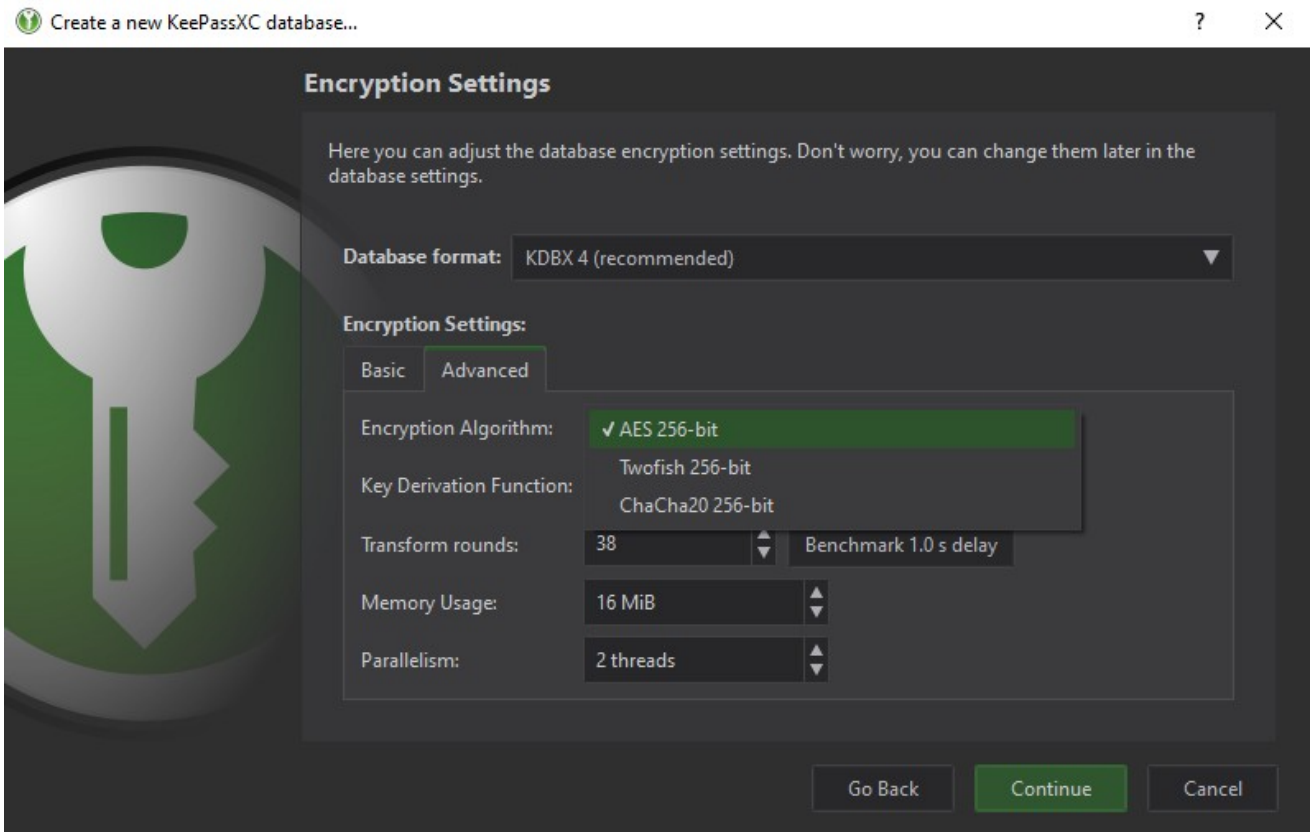


Illustration 6: KeePassXC database encryption selection.

Chacha20 is used to encrypt specific protected fields such as the password or any additional fields with the “protected” attribute.

The code supports more symmetric cipher suites, for compatibility with older versions.

```
// src/crypto/SymmetricCipher.cpp:173
SymmetricCipher::Mode SymmetricCipher::stringToMode(const QString& cipher)
```

```
{
    auto cs = Qt::CaseInsensitive;
    if (cipher.compare("aes-128-cbc", cs) == 0 || cipher.compare("aes128-cbc", cs) ==
0) {
        return Aes128_CBC;
    } else if (cipher.compare("aes-256-cbc", cs) == 0 || cipher.compare("aes256-cbc",
cs) == 0) {
        return Aes256_CBC;
    } else if (cipher.compare("aes-128-ctr", cs) == 0 || cipher.compare("aes128-ctr",
cs) == 0) {
        return Aes128_CTR;
    } else if (cipher.compare("aes-256-ctr", cs) == 0 || cipher.compare("aes256-ctr",
cs) == 0) {
        return Aes256_CTR;
    } else if (cipher.compare("aes-256-gcm", cs) == 0 || cipher.compare("aes256-gcm",
cs) == 0) {
        return Aes256_GCM;
    } else if (cipher.startsWith("twofish", cs)) {
        return Twofish_CBC;
    } else if (cipher.startsWith("salsa", cs)) {
        return Salsa20;
    } else if (cipher.startsWith("chacha", cs)) {
        return ChaCha20;
    } else {
        return InvalidMode;
    }
}
```

The other sections of the cryptographic specifications are compliant with what has been observed during the analysis.

Interview with developers

Communication with the developers did not happen during the analysis.

Expert opinion and identified potential vulnerabilities

The KeePassXC documentation is thorough and allows understanding of its inner mechanisms. Many security aspects are addressed, and the product design follows the state of the art regarding cryptography [Synacktiv-KeePassXC-cible_cspn]. In addition, security scans of the TOE are performed throughout its development using static analysis and fuzzing [FUZZING].

Regarding dependencies, only one was found with an effective vulnerability. However, the only location where the vulnerable function is called in the KeeShare export feature which is controlled by the database's user.

No impactful issue has been identified while analysing the TOE documentation and design.

Compliance and robustness of mechanisms and features

Summary of analysed / non analysed features

The following table summarizes the security features indicated in the security target:

Feature	Tested	Compliance with security target	Compliance with state of the art
SF1: Anti screenshot/recording	Yes	Compliant	Compliant
SF2: Clipboard and auto-type protection	Yes	Compliant	Compliant
SF3: Database protection	Yes	Compliant	Compliant Default algorithms are compliants with [ANSSI-PG-083]
SF4: Memory protection	Yes	Compliant	Compliant The TOE successfully protects its memory against access from unprivileged users.
SF5: Password prompt in browser extension	Yes	Compliant	Compliant Default algorithms are compliant with [ANSSI-PG-083]
SF6: KeeShare segregation	Yes	Compliant	Compliant
SF7: ssh-agent interaction	Yes	Compliant	Compliant

Details of the security features compliance analysis work

SF1: Anti-screenshot/recording

```
SF1: The KeePassXC window disappears when being recorded or screenshot.
```

KeePassXC provides a protection against screen capture. It basically disappears from the screen that is recorded. This section describes how this feature works by reviewing the source code and dynamically testing to capture the screen.

Screenshot prevention is handled by the `MainWindow` class, in the method `OSUtils::setPreventScreenCapture`.

```
// src/gui/MainWindow.cpp:1747
void MainWindow::focusWindowChanged(QWindow* window)
{
    if (window != windowHandle()) {
        m_lastFocusOutTime = Clock::currentMilliSecondsSinceEpoch();
    }

    if (!osUtils->setPreventScreenCapture(window, !m_allowScreenCapture) && !
m_allowScreenCapture) {
        displayGlobalMessage(QObject::tr("Warning: Failed to block screenshot capture
on a top-level window."),
                             MessageWidget::Error);
    }
}
```

The **MainWindow** object inherits from the **QMainWindow** class of the Qt framework (https://doc.qt.io/qt-6/qmainwindow.html).

For each supported operating system, KeePassXC provides an implementation of the **setPreventScreenCapture**. The windows implementation is:

```
// src/gui/osutils/winutils/WinUtils.cpp:59
bool WinUtils::setPreventScreenCapture(QWindow* window, bool prevent) const
{
    bool ret = true;
    if (window) {
        HWND handle = reinterpret_cast<HWND>(window->winId());
        ret = SetWindowDisplayAffinity(handle, prevent ? WDA_EXCLUDEFROMCAPTURE :
WDA_NONE);
    }
    return ret;
}
```

Referring to the Windows official documentation, the function **SetWindowDisplayAffinity** specifies where the content of the window can be displayed.

```
// https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-
setwindowdisplayaffinity
- WDA_NONE (0x00000000): Imposes no restrictions on where the window can be displayed.
- WDA_EXCLUDEFROMCAPTURE (0x00000011) : The window is displayed only on a monitor.
Everywhere else, the window does not appear at all. One use for this affinity is for
windows that show video recording controls, so that the controls are not included in
the capture.
Introduced in Windows 10 Version 2004. See remarks about compatibility regarding
previous versions of Windows.
```

The variable **m_allowScreenCapture** is set with a call to **setAllowScreenCapture**.

```
// src/gui/MainWindow.cpp:1734

void MainWindow::setAllowScreenCapture(bool state)
{
    m_allowScreenCapture = state;
    for (auto window : qApp->topLevelWindows()) {
        if (window->isVisible()) {
            osUtils->setPreventScreenCapture(window, !m_allowScreenCapture);
        }
    }
    m_ui->actionAllowScreenCapture->blockSignals(true);
    m_ui->actionAllowScreenCapture->setChecked(m_allowScreenCapture);
    m_ui->actionAllowScreenCapture->blockSignals(false);
}
```

By default, the **m_allowScreenCapture** variable is set to **false**, except if the **allowScreenCaptureOption** is set.

```
// src/main.cpp:187

// Disable screen capture if not explicitly allowed
// This ensures any top-level windows (Main Window, Modal Dialogs, etc.) are excluded
from screenshots
mainWindow.setAllowScreenCapture(parser.isSet(allowScreenCaptureOption));
```

Screen captures can be enabled by the command-line argument **allow-screencapture** or by toogling the corresponding check box in the KeePassXC settings.

```
// src/main.cpp:80
QCommandLineOption allowScreenCaptureOption("allow-screencapture",
                                            QObject::tr("allow screenshots and app
recording (Windows/macOS)"));
// src/main.cpp:604
    connect(m_ui->actionAllowScreenCapture, &QAction::toggled, this,
&MainWindow::setAllowScreenCapture);
```

To verify the behavior, a screenshot was taken with the option **--allow-screencapture** and without this option. See the TEST-SCREENSHOT in SF1: Anti-screenshot/recording page 137.

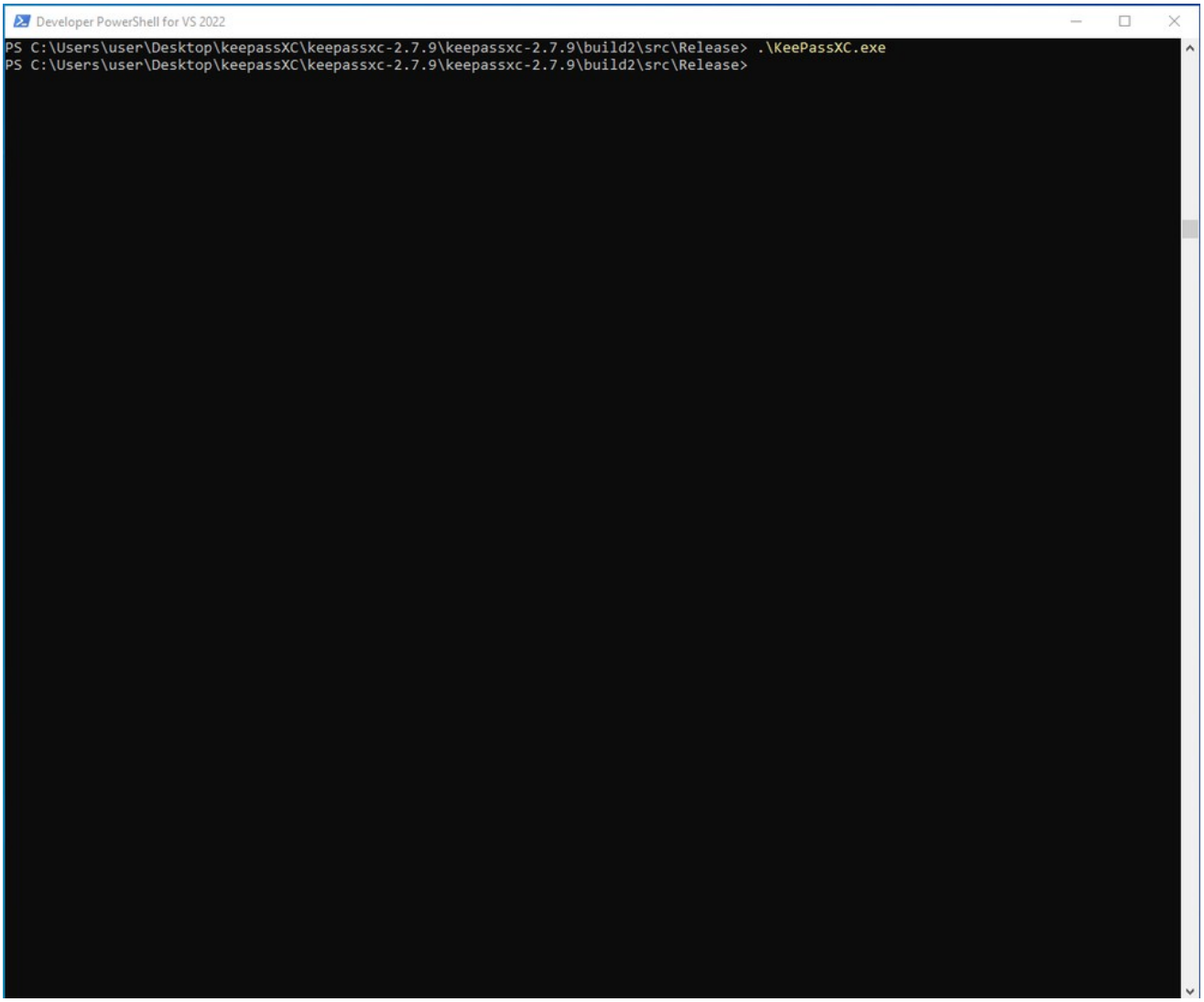


Illustration 7: Screencapture with the default options

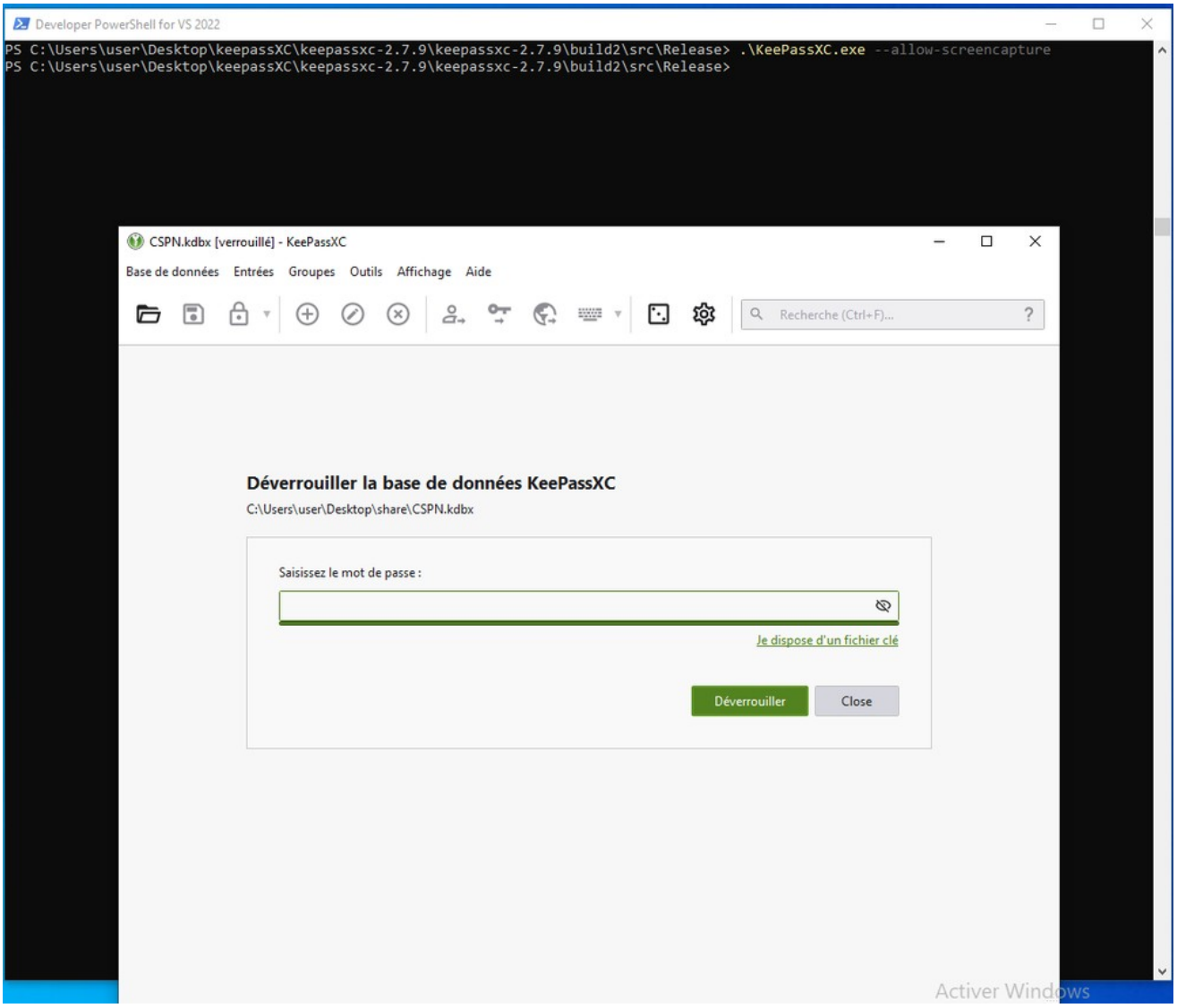


Illustration 8: Screencapture with the --allow-screencapture option

The same operation was performed for screen recording, see TEST-SCREENRECORD in SF1: Anti-screenshot/recording page 137.

```
KeePassXC implements two protections against clipboard snooping:

    • The autotype feature that directly types the password's characters into the
target window.
    • The clearing of the clipboard after a certain timeframe.
```

The auto-type feature consists in typing the credentials stored in KeePassXC to another windows that needs some fields to be filled.

The windows in which to type the password is either:

- The last active windows before KeePassXC, when enabling auto-type from a KeePassXC entry
- The current active windows, when enabling auto-type with the global shortcut. In that case, a pop-up asks the user to select an entry to auto-type.

The feature does not use the clipboard as every character is going to be typed character by character, by using the ::SendInput method.

```
// src/autotype/windows/AutoTypeWindows.cpp:84

void AutoTypePlatformWin::sendChar(const QChar& ch)
{
    INPUT in[2];
    in[0].type = INPUT_KEYBOARD;
    in[0].ki.wVk = 0;
    in[0].ki.wScan = ch.unicode();
    in[0].ki.dwFlags = KEYEVENTF_UNICODE;
    in[0].ki.time = 0;
    in[0].ki.dwExtraInfo = ::GetMessageExtraInfo();

    in[1] = in[0];
    in[1].ki.dwFlags |= KEYEVENTF_KEYUP;

    ::SendInput(2, &in[0], sizeof(INPUT));
}
```

The format of the username and password to be auto-typed is specified in the "Note" field of the KeePassXC entry, using place-holders for special characters and fields. For example, if the note field contains: `Auto-Type: {USERNAME}{TAB}{PASSWORD}` the auto-type feature will type the username, then a tab, then the password.

The parser of the AutoType feature uses the following `QRegularExpression` regular expression to parse the content of the format:

```
("([+%^#]*)(?:({{((?>[^{}]+?|(?2))+?)(?:\\s+(\\d+))?})|(.))");
```

The parsing of the KeePassXC format was not analyzed further. It is considered that the decrypted content of the database is trusted by the user.

To dynamically verify that the content auto-typed is not stored in the clipboard, a test has been issued as described in TEST-AUTOTYPE in SF2: Clipboard and auto-type protection page 138.

SF2: Clearing of the clipboard after a delay

Every field of a KeePassXC database can be copied into the clipboard. In the general settings, users can choose a delay which after the clipboard is erased. By default, this delay is set to 10 seconds. This part describes how this feature is implemented in the code and how it has been dynamically tested.

The clipboard object is defined in the files `src/gui/Clipboard.h` and `src/gui/Clipboard.cpp`.

A text is saved in the clipboard using the method `Clipboard::setText`:

```
// src/gui/Clipboard.cpp:47
void Clipboard::setText(const QString& text, bool clear)
{
    auto* clipboard = QApplication::clipboard();
//[...]
    auto* mime = new QMimeData;
    mime->setText(text);
// [...]
    mime->setData("ExcludeClipboardContentFromMonitorProcessing",
QByteArrayLiteral("1"));
// [...]
    if (clear) {
        m_lastCopied = text;
        if (config()->get(Config::Security_ClearClipboard).toBool()) {
            int timeout = config()-
>get(Config::Security_ClearClipboardTimeout).toInt();
            if (timeout > 0) {
                m_secondsToClear = timeout;
                sendCountdownStatus();
                m_timer->start(1000);
            } else {
                clearCopiedText();
            }
        }
    }
}
```

According to the MSDN documentation, the clipboard variable `ExcludeClipboardContentFromMonitorProcessing` has the following effect: place any data on the clipboard in this format to prevent all clipboard formats being included in the clipboard history or synchronized to the user's other devices.

Moreover, in this function, after the content is inserted into the clipboard, the value of `m_secondsToClear` is set to the timeout value in the configuration's variable `Config::Security_ClearClipboardTimeout`. This variable is decremented in the method `Clipboard::countdownTick()` until it is lower than 0. In that case, the clipboard is cleared. Otherwise, the UI is refreshed displaying the remaining time before the clipboard is cleared.

```
// src/gui/Clipboard.cpp:116
void Clipboard::countdownTick()
{
    if (--m_secondsToClear <= 0) {
        clearCopiedText();
    } else {
        sendCountdownStatus();
    }
}
```

The `countdownTick` method is called periodically by the `m_timer` object. The link between the timer and the method is configured when the Clipboard object is created.

```
// src/gui/Clipboard.cpp:34
Clipboard::Clipboard(QObject* parent)
    : QObject(parent)
    , m_timer(new QTimer(this))
{
// [...]
    connect(m_timer, SIGNAL(timeout()), SLOT(countdownTick()));
    connect(qApp, SIGNAL(aboutToQuit()), SLOT(clearCopiedText()));
}
```

The period of the `m_timer` is 1 second, due to the line `m_timer->start(1000);` in the `setText` method.

The cleaning of the password effectively happens in the method `Clipboard::clearCopiedText()`. In this function, if the content of the clipboard is equal to the content that was written by the process in the clipboard, then the method QClipboard::clear is called on the clipboard.

```
// src/gui/Clipboard.cpp:90
void Clipboard::clearCopiedText()
```

```
{
    m_timer->stop();
    emit updateCountdown(-1, "");

    auto* clipboard = QApplication::clipboard();
    if (!clipboard) {
        qWarning("Unable to access the clipboard.");
        return;
    }

    if (m_lastCopied == clipboard->text(QClipboard::Clipboard)
        || m_lastCopied == clipboard->text(QClipboard::Selection)) {
        clipboard->clear(QClipboard::Clipboard);
        clipboard->clear(QClipboard::Selection);
// [...]
    }

    m_lastCopied.clear();
}
```

In order to dynamically verify that the clipboard is effectively cleared after 10 seconds, the following script prints the content of the clipboard every second, as described in TEST-CLIPBOARD in SF2: Clipboard and auto-type protection page 138.

```
PS C:\Users\user> for ($i=0 ; $i -lt 100; $i+=1) { sleep 1 ; Get-Date; Get-Clipboard ;}

mardi 29 octobre 2024 14:30:37
mardi 29 octobre 2024 14:30:38
$V%$5!eZqf@LvSL#<h|;
mardi 29 octobre 2024 14:30:39
$V%$5!eZqf@LvSL#<h|;
mardi 29 octobre 2024 14:30:40
$V%$5!eZqf@LvSL#<h|;
mardi 29 octobre 2024 14:30:41
$V%$5!eZqf@LvSL#<h|;
mardi 29 octobre 2024 14:30:42
$V%$5!eZqf@LvSL#<h|;
mardi 29 octobre 2024 14:30:43
$V%$5!eZqf@LvSL#<h|;
mardi 29 octobre 2024 14:30:44
$V%$5!eZqf@LvSL#<h|;
mardi 29 octobre 2024 14:30:45
$V%$5!eZqf@LvSL#<h|;
mardi 29 octobre 2024 14:30:46
$V%$5!eZqf@LvSL#<h|;
```

```
mardi 29 octobre 2024 14:30:47
$V%$5!eZqf@LvSL#<h|;
mardi 29 octobre 2024 14:30:48
mardi 29 octobre 2024 14:30:49
mardi 29 octobre 2024 14:30:50
mardi 29 octobre 2024 14:30:51
```

```
SF3: Database protection

Master passwords are computed using cryptography's best practices (Argon2) into a
master key that is used to encrypt the database. The algorithm can be parameterized in
order to take a certain amount of time to decrypt the database. The encryption and
decryption process also include an integrity check in order to identify whether the
database file was tampered with or corrupted.

KeePassXC advises regarding the robustness of user-supplied master password and/or key
file and if requested, generates strong random passwords/key files.
```

The database is stored in the KDBX4 format. For the database protection, several aspects are important:

- The confidentiality of the database
- The authenticity and integrity of the database

In this part, the format of the KeePassXC database as well as the key derivation mechanism will be described. After that, the integrity and authenticity checks will be explained, as well as the decryption process. Next, the writing of the database is case change occured will be described.

Finally, every cryptographic algorithm used will be summarized.

The tests in this part are performed by source code auditing and the dynamic tests are based on a database created with the default parameters.

KDBX4 Format

By default, the latest version of KeePassXC uses the KDBX4 format. KeePassXC implements a KBDX4 parser. This section will describe the KDBX4 format and how KeePassXC parses it, using code review and dynamic analysis using Frinet and Frida.

The parser starts by reading the header fields of the database. The header starts with the fields described in the following table:

Name	Description	Size
Magic1	Magic of the file, **`03 d9 a2 9a`**	4
Version	Latest is Keepass2	4
Magic 2	Magic of the format: **`fb 4b b5`**	4
File version	4 (for KDBX4)	4
Crypto parameters	A list of OuterHeaderFields	variable

SHA256	SHA256 of the header	32
HMAC-SHA256	HMAC-SHA256 of the header	32

The crypto parameters are formatted as a `OuterHeaderFields`, described in the following table:

Name	Size (byte)
ID	1
Size	4
Data	size

The different identifiers of the `OuterHeaderFields` are listed below. Not all fields are mandatory and only the useful one are present in the header.

```
// src/format/KeePass2.h:72
    enum class HeaderFieldID
    {
        EndOfHeader = 0,
        Comment = 1,
        CipherID = 2,
        CompressionFlags = 3,
        MasterSeed = 4,
        TransformSeed = 5,
        TransformRounds = 6,
        EncryptionIV = 7,
        ProtectedStreamKey = 8,
        StreamStartBytes = 9,
        InnerRandomStreamID = 10,
        KdfParameters = 11,
        PublicCustomData = 12
    };
```

As an example, here are the different values of the header of a KDBX4 database:

```
- primary_identifier : 03 d9 a2 9a
- version : Version.keepass2
- secondary_identifier : fb 4b b5
- file_version : 0x40000
- cipher_id : b'1\xc1\xf2\xe6\xbfqCP\xbeX\x05!j\xfcZ\xff' (CIPHER_AES256)
- compression_flags : DynamicHeaderTypeCompressionList.gzip
- master_seed : 23 6a 28 ec 09 2e fb 6c 64 73 a4 80 07 07 92 d1 75 bb 63 ca f9 4f 6e 3a
c3 0a d7 ce ab e7 de 27
```

```
- encryption_iv : 65 29 31 3a ea 71 f6 3e 00 52 f8 23 d4 85 e6 91
- kdf_parameters :
    $UUID : ef 63 6d df 8c 29 44 4b 91 f7 a9 a4 03 e3 0a 0c (KDF_ARGON2D)
    I : 0x29
    M : 0x4000000
    P : 0x2
    S : 89 57 65 39 86 a6 0a c0 bf 91 c1 4b 23 29 24 97 18 a9 44 74 9b e5 4f 16 a6 1d
8e 91 fa b8 0b cf
    V : 0x13
- end : 0d 0a 0d 0a
- Header's SHA256: 87 1a 5b 98 76 df 4c 3c 9f a0 97 82 85 f0 02 8f ff d4 7c ef dc 6b 96
26 cb 8b 55 97 bb 21 9f 41
- Header's HMAC: cd 78 06 cd 47 de fc ea de 6e 84 f9 7b c2 cb 39 e5 8a 4c 03 9b 1d 60
7e 29 d8 43 28 c3 e3 ef 73
```

When a database is opened in KeePassXC, the method **`KdbxReader::readDatabase`** is called. This function parses the header by calling the **`readMagicNumbers`**, **`readHeaderField`** methods and finally **`readDatabaseImpl`**.

```
// src/format/KdbxReader.cpp:63

bool KdbxReader::readDatabase(QIODevice* device, QSharedPointer<const CompositeKey>
key, Database* db)
{
    device->seek(0);

    m_db = db;
    m_masterSeed.clear();
    m_encryptionIV.clear();
    m_streamStartBytes.clear();
    m_protectedStreamKey.clear();

    StoreDataStream headerStream(device);
    headerStream.open(QIODevice::ReadOnly);

    // read KDBX magic numbers
    quint32 sig1, sig2, version;
    if (!readMagicNumbers(&headerStream, sig1, sig2, version)) {
        return false;
    }
    m_kdbxSignature = qMakePair(sig1, sig2);
    m_db->setFormatVersion(version);

    // read header fields
    while (readHeaderField(headerStream, m_db) && !hasError()) {
    }
```

```
    headerStream.close();

    if (hasError()) {
        return false;
    }

    // read payload
    return readDatabaseImpl(device, headerStream.storedData(), std::move(key), db);
}
```

The first fields are fixed sized and are read in the function **readMagicNumbers**. For the other header fields, the parsing happens inside the **readHeaderField** function. The size is always verified before it is copied to the allocated buffer, inside the **read** function, and the final size is verified to be equal to the effective number of bytes read.

```
// src/format/KdbxReader.cpp:157
bool Kdbx4Reader::readHeaderField(StoreDataStream& device, Database* db)
{
    QByteArray fieldIDArray = device.read(1);
    if (fieldIDArray.size() != 1) {
        raiseError(tr("Invalid header id size"));
        return false;
    }
    char fieldID = fieldIDArray.at(0);

    bool ok;
    auto fieldLen = Endian::readSizedInt<quint32>(&device, KeePass2::BYTEORDER, &ok);
    if (!ok) {
        raiseError(tr("Invalid header field length: field %1").arg(fieldID));
        return false;
    }

    QByteArray fieldData;
    if (fieldLen != 0) {
        fieldData = device.read(fieldLen);
        if (static_cast<quint32>(fieldData.size()) != fieldLen) {
            raiseError(tr("Invalid header data length: field %1, %2 expected, %3
found")
                               .arg(static_cast<int>(fieldID))
                               .arg(fieldLen)
                               .arg(fieldData.size()));
            return false;
        }
    }
```

After the header comes the encrypted database. Before data can be extracted from it, it must be decrypted.

Password and keyfile generation

The KeepassXC Database can be protected using:

- A user defined password,
- A randomly generated key file,
- A password and a key file.

The KeepassXC UI provides a random generator in case a user wants a randomly generated password.

The key file can have 2 different formats:

- A file containing 32 raw random bytes,
- A .keyx file organized in an XML structure, containing 32 random bytes

Here is an example of an XML Keyfile:

```
<?xml version="1.0" encoding="UTF-8"?>
<KeyFile>
    <Meta>
        <Version>2.0</Version>
    </Meta>
    <Key>
        <Data Hash="90514B52">
            47849CCD F0D95334 5095F917 48EAF2B2
            C532301F 2423CC5D 7437A2FD 3626775D
        </Data>
    </Key>
</KeyFile>
```

The KeyFile generation is triggered by the UI Widget:

```
//src/gui/databasekey/KeyFileEditWidget.cpp:105
void KeyFileEditWidget::createKeyFile()
{
```

```
    Q_ASSERT(m_compEditWidget);
    if (!m_compEditWidget) {
        return;
    }
    QString filters = QString("%1 (*.keyx *.key);;%2 (*)").arg(tr("Key files"), tr("All
files"));
    QString fileName = fileDialog()->getSaveFileName(this, tr("Create Key File…"),
QString(), filters);

    if (!fileName.isEmpty()) {
        QString errorMsg;
        bool created = FileKey::create(fileName, &errorMsg);
        if (!created) {
            MessageBox::critical(getMainWindow(),
                                 tr("Error creating key file"),
                                 tr("Unable to create key file: %1").arg(errorMsg),
                                 QMessageBox::Button::Ok);
        } else {
            m_compUi->keyFileLineEdit->setText(fileName);
        }
    }
}
```

The widget function callback will retrieve the filename where the keyfile is to be stored, and call the FileKey::create method.

This function will create the output file and upon success call createRandom or createXMLv2 if the file extension is .keyx.

```
// src/keys/FileKey.cpp:255
/**
 * Create a new key file from random bytes.
 *
 * @param fileName output file name
 * @param errorMsg error message if generation failed
 * @param number of random bytes to generate
 * @return true on successful creation
 */
bool FileKey::create(const QString& fileName, QString* errorMsg)
{
    QFile file(fileName);
    if (!file.open(QFile::WriteOnly)) {
        if (errorMsg) {
            *errorMsg = file.errorString();
        }
        return false;
```

```
    }
    if (fileName.endsWith(".keyx")) {
        createXMLv2(&file);
    } else {
        createRandom(&file);
    }
    file.close();
    file.setPermissions(QFile::ReadUser);
// ...
```

These functions are declared in the Filekey.h file, with a default size parameter.

```
static void createRandom(QIODevice* device, int size = 128);
static void createXMLv2(QIODevice* device, int size = 32);
```

The createRandom function will generate size random bytes and write them to the file

```
/**
 * Generate a new key file with random bytes.
 *
 * @param device output device
 * @param number of random bytes to generate
 */
void FileKey::createRandom(QIODevice* device, int size)
{
    device->write(randomGen()->randomArray(size));
}
```

The randomGen() uses the random generator of the library Botan.

The createXMLv2 function will output the data in an XML form. The data is also randomly generated using the Botan library.

After allocating and manipulating the key, a call to **`Botan::secure_scrub_memory`** is issued in order to write zero at the location where the key array was stored.

```
/**
 * Generate a new key file in the KeePass2 XML format v2.
```

```
 *
 * @param device output device
 * @param number of random bytes to generate
 */
void FileKey::createXMLv2(QIODevice* device, int size)
{
    QXmlStreamWriter w(device);
    w.setAutoFormatting(true);
    w.setAutoFormattingIndent(4);
    w.writeStartDocument();

    w.writeStartElement("KeyFile");

    w.writeStartElement("Meta");
    w.writeTextElement("Version", "2.0");
    w.writeEndElement();

    w.writeStartElement("Key");
    w.writeStartElement("Data");

    QByteArray key = randomGen()->randomArray(size);
    CryptoHash hash(CryptoHash::Sha256);
    hash.addData(key);
    QByteArray result = hash.result().left(4);
    key = key.toHex().toUpper();

    w.writeAttribute("Hash", result.toHex().toUpper());
    w.writeCharacters("\n            ");
    for (int i = 0; i < key.size(); ++i) {
        // Pretty-print hex value (not strictly necessary, but nicer to read and
KeePass2 does it)
        if (i != 0 && i % 32 == 0) {
            w.writeCharacters("\n            ");
        } else if (i != 0 && i % 8 == 0) {
            w.writeCharacters(" ");
        }
        w.writeCharacters(QChar(key[i]));
    }
    Botan::secure_scrub_memory(key.data(), static_cast<std::size_t>(key.capacity()));
    w.writeCharacters("\n        ");

    w.writeEndElement();
    w.writeEndElement();

    w.writeEndDocument();
}
```

To conclude, the key file is randomly generated using the Botan library. The file protection is the responsibility of the database user, so the file stores the key in plaintext inside the keyfile.

Key Derivation

In order to compute the key used to decrypt the database, the user can specify either a key file and/or a password.

The password is a string of any length, although a weak password raises a warning, the user can still create a database with a one character password.

The key file is an XML file, with a "Data Hash" value.

```
<?xml version="1.0" encoding="UTF-8"?>
<KeyFile>
    <Meta>
        <Version>2.0</Version>
    </Meta>
    <Key>
        <Data Hash="90514B52">
            47849CCD F0D95334 5095F917 48EAF2B2
            C532301F 2423CC5D 7437A2FD 3626775D
        </Data>
    </Key>
</KeyFile>
```

The method **`FileKey::loadXml`** is responsible for the extraction and validation of the key. The first XML element must be a **`Keyfile`** element, otherwise the parsing stops. The **`Meta`** and **`Key`** element are then extracted. In the **`Key`** element, the **`Data`** subkey corresponds to the first 4 bytes of the SHA256 of the key, and the other characters are the hexadecimal representation of the actual database key.

Here, the hash has the role of a checksum. Finally, the data from the file is copied into its final destination, and the size is correctly checked. After the copy has occurred, the memory is cleared using the function **`Botan::secure_scrub_memory`**.

```
// src/keys/FileKey.cpp:297
bool FileKey::loadXml(QIODevice* device, QString* errorMsg)
{
    QXmlStreamReader xmlReader(device);
// [...]
    if (xmlReader.readNextStartElement() && xmlReader.name() != "KeyFile") {
        return false;
    }
```

```
    struct
    {
        QString version;
        QByteArray hash;
        QByteArray data;
    } keyFileData;

    while (!xmlReader.error() && xmlReader.readNextStartElement()) {
        if (xmlReader.name() == "Meta") {
            while (!xmlReader.error() && xmlReader.readNextStartElement()) {
                if (xmlReader.name() == "Version") {
                    keyFileData.version = xmlReader.readElementText();
     // [...]
            }
        } else if (xmlReader.name() == "Key") {
            while (!xmlReader.error() && xmlReader.readNextStartElement()) {
                if (xmlReader.name() == "Data") {
                    keyFileData.hash =
QByteArray::fromHex(xmlReader.attributes().value("Hash").toLatin1());
                    keyFileData.data =
xmlReader.readElementText().simplified().replace(" ", "").toLatin1();
                    // [...]
                    if (keyFileData.version == "2.0" && Tools::isHex(keyFileData.data))
{
                        keyFileData.data = QByteArray::fromHex(keyFileData.data);

                        CryptoHash hash(CryptoHash::Sha256);
                        hash.addData(keyFileData.data);
                        QByteArray result = hash.result().left(4);
                        if (keyFileData.hash != result) {
                            if (errorMsg) {
                                *errorMsg = QObject::tr("Checksum mismatch! Key file
may be corrupt.");
                            }
                            return false;
                        }
                    } else {
                        if (errorMsg) {
                            *errorMsg = QObject::tr("Unexpected key file data! Key file
may be corrupt.");
                        }
                        return false;
                    }
                }
            }
        }
    }
```

```
    bool ok = false;
    if (!xmlReader.error() && !keyFileData.data.isEmpty()) {
        std::memcpy(m_key.data(), keyFileData.data.data(), std::min(SHA256_SIZE,
keyFileData.data.size()));
        ok = true;
    }

    Botan::secure_scrub_memory(keyFileData.data.data(),
static_cast<std::size_t>(keyFileData.data.capacity()));

    return ok;
}
```

In order to decrypt the database, a user can either specify a password, provide a key file or both of these options. The function **buildDatabaseKey** computes the key based on the provided password and/or key file. The code starts by extracting the data in the password field of the GUI. A **CompositeKey** object is created to store the **databaseKey**. It will contain an aggregate of the password and/or the key file key.

```
// src/gui/DatabaseOpenWidget.cpp:384
QSharedPointer<CompositeKey> DatabaseOpenWidget::buildDatabaseKey()
{
    auto databaseKey = QSharedPointer<CompositeKey>::create();

// [...]
    if (!m_ui->editPassword->text().isEmpty() || m_retryUnlockWithEmptyPassword) {
        databaseKey->addKey(QSharedPointer<PasswordKey>::create(m_ui->editPassword-
>text()));
    }
// [...]
    auto key = QSharedPointer<FileKey>::create();
    QString keyFilename = m_ui->keyFileLineEdit->text();
    if (!keyFilename.isEmpty()) {
        QString errorMsg;
        if (!key->load(keyFilename, &errorMsg)) {
            m_ui->messageWidget->showMessage(tr("Failed to open key file:
%1").arg(errorMsg), MessageWidget::Error);
            return {};
        }
// [...]
        databaseKey->addKey(key);
        lastKeyFiles.insert(m_filename, keyFilename);
    }

    if (config()->get(Config::RememberLastKeyFiles).toBool()) {
```

```
        config()->set(Config::LastKeyFiles, lastKeyFiles);
    }

    return databaseKey;
}
```

Regarding the keyfile key, it is loaded and added to the `databaseKey` after the password part if it was present.

Regarding the password, internally, the `QSharedPointer<PasswordKey>::create()` calls the constructor of `PasswordKey`, which computes the SHA256 digest of the data given as argument.

```
// src/keys/PasswordKey.cpp:35
PasswordKey::PasswordKey(const QString& password)
    : Key(UUID)
    , m_key(SHA256_SIZE)
{
    setPassword(password);
}

//src/keys/PasswordKey.cpp:62
void PasswordKey::setPassword(const QString& password)
{
    setRawKey(CryptoHash::hash(password.toUtf8(), CryptoHash::Sha256));
}
```

To summarize, the database key is a `CompositeKey` object containing :

- `SHA256(password)` if only a password is provided,
- `keyfile` if only a keyfile is provided,
- `SHA256(password) . keyfile` if both are provided.

When the `CompositeKey` is extracted, another SHA256 will be computed. The code of the function `CompositeKey::rawKey` will compute the SHA256 of all data in the internal `m_key` attribute. The `m_key` attribute is not securely erased after usage.

```
// src/keys/CompositeKey.cpp:74
QByteArray CompositeKey::rawKey(const QByteArray* transformSeed, bool* ok, QString*
error) const
```

```
{
    CryptoHash cryptoHash(CryptoHash::Sha256);

    for (auto const& key : m_keys) {
        cryptoHash.addData(key->rawKey());
    }

    if (ok) {
        *ok = true;
    }

// [...]
    return cryptoHash.result();
}
```

In the end, the raw `CompositeKey` is either:

- `SHA256(SHA256(password))` if only a password is provided,
- `SHA256(keyfile)` if only a keyfile is provided,
- `SHA256(SHA256(password) . keyfile)` if both are provided.

In the example database whose header was presented in the KDBX4 section, the password is `TestCSPN123!` and the resulting `compositeKey` is `AB97A3A157604B5EE2D345E1405326424E91D81350758AA408FC0CAD1B3F6A2D`.

The `CompositeKey` and the parsed header parameters are used in the `Kdbx4Reader::readDatabaseImpl` function:

```
// src/format/Kdbx4Reader.cpp:34
bool Kdbx4Reader::readDatabaseImpl(QIODevice* device,
                                   const QByteArray& headerData,
                                   QSharedPointer<const CompositeKey> key,
                                   Database* db)
{
```

The code verifies that the parameters are present in the header:

```
// src/format/Kdbx4Reader.cpp:47
// check if all required headers were present
    if (m_masterSeed.isEmpty() || m_encryptionIV.isEmpty() || db->cipher().isNull()) {
        raiseError(tr("missing database headers"));
        return false;
```

```
    }
```

The database key is then derived using the KDF parameter from the header, in the method **`Database::setKey`**.

```
// src/format/Kdbx4Reader.cpp:53
    bool ok = AsyncTask::runAndWaitForFuture([&] { return db->setKey(key, false,
false); });
    if (!ok) {
        raiseError(tr("Unable to calculate database key: %1").arg(db->keyError()));
        return false;
    }
```

The **`setKey`** method will transform the key parameter using the proper key derivation function. Based on the dynamic trace, the **`transformKey`** parameter is not explicitely provided but it is equal to **`True`** in the trace.

```
// src/core/Database.cpp:822
bool Database::setKey(const QSharedPointer<const CompositeKey>& key,
                      bool updateChangedTime,
                      bool updateTransformSalt,
                      bool transformKey)
{
[...]
    QByteArray transformedDatabaseKey;

    if (!transformKey) {
        transformedDatabaseKey = QByteArray(oldTransformedDatabaseKey.rawKey());
    } else if (!key->transform(*m_data.kdf, transformedDatabaseKey, &m_keyError)) {
        return false;
    }

//src/keys/CompositeKey.cpp:121
/*
 * @param kdf key derivation function
 * @param result transformed key hash
 * @return true on success
 */
bool CompositeKey::transform(const Kdf& kdf, QByteArray& result, QString* error) const
{
// [...]
    QByteArray seed = kdf.seed();
    Q_ASSERT(!seed.isEmpty());
    bool ok = false;
```

```
    return kdf.transform(rawKey(&seed, &ok, error), result) && ok;
}
```

For KDBX4, this function is `Argon2d`. So, in the end, the `Argon2Kdf::transform` method is called.

```
// src/crypto/Argon2Kdf.cpp:164
bool Argon2Kdf::transform(const QByteArray& raw, QByteArray& result) const
{
    result.clear();
    result.resize(32);
    // Time Cost, Mem Cost, Threads/Lanes, Password, length, Salt, length, out, length

    int rc = argon2_hash(rounds(),
                         memory(),
                         parallelism(),
                         raw.data(),
                         raw.size(),
                         seed().data(),
                         seed().size(),
                         result.data(),
                         result.size(),
                         nullptr,
                         0,
                         type() == Type::Argon2d ? Argon2_d : Argon2_id,
                         version());
    if (rc != ARGON2_OK) {
        qWarning("Argon2 error: %s", argon2_error_message(rc));
        return false;
    }

    return true;
}

// src/crypto/Argon2Kdf.cpp:59
Argon2Kdf::Type Argon2Kdf::type() const
{
    return uuid() == KeePass2::KDF_ARGON2D ? Type::Argon2d : Type::Argon2id;
}
```

The `argon2_hash` function is defined in the reference C implementation, included with the `#include <argon2.h>` statement.

The parameters resolve as follows for our database:

- rounds: 41 (0x29)

- memory: 0x10000
- parallelism: 2
- raw data: AB97A3A157604B5EE2D345E1405326424E91D81350758AA408FC0CAD1B3F6A2D (`compositeKey`)
- raw.size: 32
- seed data: 8957653986A60AC0BF91C14B2329249718A944749BE54F16A61D8E91FAB80BCF (`kdf_parameters.S` in header)
- result data: CDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCD
- result size: 32
- type: 0 (KDF_ARGON2D)
- version: 0x13 (19)

To summarize, the `derivedKey` is the result of the `Argon2d` algorithm based on the input:

- `SHA256(SHA256(password))` if only a password is provided,
- `SHA256(keyfile)` if only a keyfile is provided,
- `SHA256(SHA256(password) . keyfile)` if both are provided.

Integrity verification

KeePassXC checks for the integrity of the database before the decryption process starts.

After deriving the key with the KDF algorithm, it checks that the SHA256 checksum written at the end of the header corresponds to the actual SHA256 sum of the `headerData`.

```
// src/format/Kdbx4Reader.cpp:64
    QByteArray headerSha256 = device->read(32);
    QByteArray headerHmac = device->read(32);
    if (headerSha256.size() != 32 || headerHmac.size() != 32) {
        raiseError(tr("Invalid header checksum size"));
        return false;
    }
    if (headerSha256 != CryptoHash::hash(headerData, CryptoHash::Sha256)) {
        raiseError(tr("Header SHA256 mismatch"));
```

```
        return false;
    }
```

This feature protects the header against header data corruption, without knowing any secrets. It does not prevent an attacker from modifying the header. To protect against malicious data tampering, a HMAC of the header data is computed using the `derivedKey`.

The KeePassXC database header is protected with a HMAC-SHA256. Moreover, each block of 1 MB has its own HMAC-SHA256.

For the header, the HMAC value corresponds to:

```
hmac_header = MAC_SHA256(hmacHeaderKey)(headerData)
```

For every block *i*, the HMAC value corresponds to:

```
hmac_block_i = MAC_SHA256(blockHmacKey_i)(block_index_i . block_size_i .
encrypted_block_i)
```

The headerHmacKey is equal to:

```
headerHmacKey = SHA512(0xFFFFFFFFFFFFFFFF . SHA512(masterSeed . derivedKey . "\x01"))
```

And for each block i, the blockHmacKey_i is equal to :

```
blockHmacKey_i = SHA512(block_index_i . SHA512(masterSeed . derivedKey . "\x01"))
```

For a better readability, the value `SHA512(masterSeed . derivedKey . "\x01")` will be renamed `interHmacKey` in this report.

A code review of the parts relating to the previous computation is described in this section.

The HMAC of the header relies on three different function calls.

```
// src/format/Kdbx4Reader.cpp:103
    QByteArray hmacKey = KeePass2::hmacKey(m_masterSeed, db->transformedDatabaseKey());
    if (headerHmac != CryptoHash::hmac(headerData,
HmacBlockStream::getHmacKey(UINT64_MAX, hmacKey), CryptoHash::Sha256)) {
        raiseError(tr("Invalid credentials were provided, please try again.\n"
                      "If this reoccurs, then your database file may be corrupt.") + "
" + tr("(HMAC mismatch)"));
        return false;
    }
```

The three steps of this computation are described below:

- The call to **`KeePass2::hmacKey(m_masterSeed, db->transformedDatabaseKey())`**
- The call to **`HmacBlockStream::getHmacKey(UINT64_MAX, interHmacKey)`**
- The call to **`CryptoHash::hmac(headerData, HmacBlockStream::getHmacKey(UINT64_MAX, hmacKey), CryptoHash::Sha256)`**

1. Call to **`KeePass2::hmacKey(m_masterSeed, db->transformedDatabaseKey())`**

An **`intermediate HMACKey`** is obtained by first computing the **`SHA512(masterSeed . derivedKey . "\x01")`**

```
// src/format/KeePass2.cpp:55
QByteArray KeePass2::hmacKey(const QByteArray& masterSeed, const QByteArray&
transformedMasterKey)
{
    CryptoHash hmacKeyHash(CryptoHash::Sha512);
    hmacKeyHash.addData(masterSeed);
    hmacKeyHash.addData(transformedMasterKey);
    hmacKeyHash.addData(QByteArray(1, '\x01'));
    return hmacKeyHash.result();
}
```

2. Call to **`HmacBlockStream::getHmacKey(UINT64_MAX, interHmacKey)`**

Then, the intermediate HMAC key is processed by the function: **`HmacBlockStream::getHmacKey(UINT64_MAX, interHmacKey)`**. This function computes the **`SHA512(block_idx . interHmacKey)`**. For the header, the **`block_idx`** is **`0xFFFFFFFFFFFFFFFF`**.

```
// src/streams/HmacBlockStream.cpp:248
QByteArray HmacBlockStream::getHmacKey(quint64 blockIndex, const QByteArray& key)
{
    Q_ASSERT(key.size() == 64);
    QByteArray indexBytes = Endian::sizedIntToBytes<quint64>(blockIndex, ByteOrder);
    CryptoHash hasher(CryptoHash::Sha512);
    hasher.addData(indexBytes);
    hasher.addData(key);
    return hasher.result();
}
```

3. Call to **`CryptoHash::hmac(headerData, HmacBlockStream::getHmacKey(UINT64_MAX, interHmacKey), CryptoHash::Sha256)`**

Finally, the HMAC will be computed over the data with the previously generated HMAC key, that is unique for the header.

The HMAC function calls the **`CryptoHash`** implementation:

```
// src/crypto/CryptoHash.cpp:119
QByteArray CryptoHash::hmac(const QByteArray& data, const QByteArray& key, Algorithm
algo)
{
    CryptoHash cryptoHash(algo, true);
    cryptoHash.setKey(key);
    cryptoHash.addData(data);
    return cryptoHash.result();
}

//src/crypto/CryptoHash.cpp:99

QByteArray CryptoHash::result() const
{
    Q_D(const CryptoHash);

    Botan::secure_vector<uint8_t> result;
    if (d->hmacFunction) {
        result = d->hmacFunction->final();
    } else if (d->hashFunction) {
        result = d->hashFunction->final();
    }
    return QByteArray(reinterpret_cast<const char*>(result.data()), result.size());
}

//src/crypto/CryptoHash.cpp:29
    QScopedPointer<Botan::HashFunction> hashFunction;
    QScopedPointer<Botan::MessageAuthenticationCode> hmacFunction;
```

The final MAC function called is from the Botan library, the documentation for this library is available here: https://botan.randombit.net/handbook/api_ref/message_auth_codes.html#_CPPv425MessageAuthenticationCode. It computes a HMAC_SHA256.

After the header has been checked with the SHA256 and HMAC validation, the database decryption starts.

Database decryption

In order to decrypt the database, the decryption key or **`finalKey`** is equal to **`SHA256(masterSeed + derivedKey)`**.

```
// src/format/Kdbx4Reader.cpp:59

    CryptoHash hash(CryptoHash::Sha256);
    hash.addData(m_masterSeed);
    hash.addData(db->transformedDatabaseKey());
    QByteArray finalKey = hash.result();
```

In the code, the database file reader is wrapped in a **`HmacBlockStream`** object, wrapped in a **`SymmetricCipherStream`** object.

```
// src/format/Kdbx4Reader.cpp:82
    HmacBlockStream hmacStream(device, hmacKey);
    if (!hmacStream.open(QIODevice::ReadOnly)) {
        raiseError(hmacStream.errorString());
        return false;
    }

    auto mode = SymmetricCipher::cipherUuidToMode(db->cipher());
    if (mode == SymmetricCipher::InvalidMode) {
        raiseError(tr("Unknown cipher"));
        return false;
    }
    SymmetricCipherStream cipherStream(&hmacStream);
    if (!cipherStream.init(mode, SymmetricCipher::Decrypt, finalKey, m_encryptionIV)) {
        raiseError(cipherStream.errorString());
        return false;
    }
    if (!cipherStream.open(QIODevice::ReadOnly)) {
        raiseError(cipherStream.errorString());
```

```
        return false;
    }
```

In the **`HmacBlockStream`** class, the default block size in 1 MB. When data is read from the device, the method **`readHashedBlock`** will be called. It reads the first 32 bytes of the block, which is the expected HMAC of the block. Then, it reads 4 bytes corresponding to the size of the block. And next, it uses the size to read the whole block and to check the HMAC for that block. The HMAC corresponds to the SHA256 digest parameterized by **`hmacKey`** previously computed, the **`blockIndex`** (0 for the first block, and incremented for each next block), the size of the block and finally the data of the buffer.

For each block, a different HMAC key will be computed and is equal to **`SHA512(blockIndex . interHmacKey)`**.

```
// src/streams/HmacBlockStream.cpp:
HmacBlockStream::HmacBlockStream(QIODevice* baseDevice, QByteArray key)
    : LayeredStream(baseDevice)
    , m_blockSize(1024 * 1024)
    , m_key(std::move(key))
{
 // [...]

// src/streams/HmacBlockStream.cpp:125
bool HmacBlockStream::readHashedBlock()
{
// [...]
    QByteArray hmac = m_baseDevice->read(32);
// [...]
    QByteArray blockSizeBytes = m_baseDevice->read(4);
// [...]
    auto blockSize = Endian::bytesToSizedInt<qint32>(blockSizeBytes, ByteOrder);
// [...]
    m_buffer = m_baseDevice->read(blockSize);
// [...]
    CryptoHash hasher(CryptoHash::Sha256, true);
    hasher.setKey(getCurrentHmacKey());
    hasher.addData(Endian::sizedIntToBytes<quint64>(m_blockIndex, ByteOrder));
    hasher.addData(blockSizeBytes);
    hasher.addData(m_buffer);

    if (hmac != hasher.result()) {
        m_error = true;
        setErrorString("Mismatch between hash and data.");
        return false;
    }
// [...]
```

```
}

// src/streams/HmacBlockStream.cpp:243
QbyteArray HmacBlockStream::getCurrentHmacKey() const
{
  return getHmacKey(m_blockIndex, m_key)
}
```

After the HMAC has been verified, the data decryption process can start. A **`SymmetricCipher`** object is instantiated and initialized with the **`finalKey`** that was computed previously.

```
// src/streams/SymmetricCipherStream.cpp:20
SymmetricCipherStream::SymmetricCipherStream(QIODevice* baseDevice)
    : LayeredStream(baseDevice)
    , m_cipher(new SymmetricCipher())
    // [...]

//src/streams/SymmetricCipherStream.cpp:37
bool SymmetricCipherStream::init(SymmetricCipher::Mode mode,
                                 SymmetricCipher::Direction direction,
                                 const QByteArray& key,
                                 const QByteArray& iv)
{
    m_isInitialized = m_cipher->init(mode, direction, key, iv);
    // [...]

//src/streams/SymmetricCipherStream.cpp:124
bool SymmetricCipherStream::readBlock()
{
    QByteArray newData;
/// [...]
    int readResult = m_baseDevice->read(newData.data(), newData.size());
// [...]
        m_buffer.append(newData.left(readResult));
// [...]
        if (!m_streamCipher && m_baseDevice->atEnd()) {
            if (!m_cipher->finish(m_buffer)) {
                m_error = true;
                setErrorString(m_cipher->errorString());
                return false;
            }
        } else if (m_buffer.size() > 0) {
            if (!m_cipher->process(m_buffer)) {
                m_error = true;
```

```
                setErrorString(m_cipher->errorString());
                return false;
            }
        }
        return m_buffer.size() > 0;
    }
}
```

The **SymmetricCipher** object wraps the encryption process performed by the Botan library (documented here https://botan.randombit.net/doxygen/classBotan_1_1Cipher__Mode.html#ac76be9b4d73678708b2c243bb821154b). It basically decrypts the data using the algorithm and parameters passed in the **init** method.

```
// src/crypto/SymmetricCipher.h:80
QSharedPointer<Botan::Cipher_Mode> m_cipher;

// src/crypto/SymmetricCipher.cpp:26
bool SymmetricCipher::init(Mode mode, Direction direction, const QByteArray& key, const
QByteArray& iv)
{
  //[...]
        auto cipher = Botan::Cipher_Mode::create_or_throw(botanMode.toStdString(),
botanDirection);
        m_cipher.reset(cipher.release());
        m_cipher->set_key(reinterpret_cast<const uint8_t*>(key.data()), key.size());

        if (!m_cipher->valid_nonce_length(iv.size())) {
            m_mode = InvalidMode;
            m_cipher.reset();
            m_error = QObject::tr("SymmetricCipher::init: Invalid IV size of %1 for
%2.").arg(iv.size()).arg(botanMode);
            return false;
        }
        m_cipher->start(reinterpret_cast<const uint8_t*>(iv.data()), iv.size());
// [...]
    return true;
}

// src/crypto/SymmetricCipher.cpp:71
bool SymmetricCipher::process(char* data, int len)
{
 // [...]
    try {
        // Block size is checked by Botan, an exception is thrown if invalid
```

```
        m_cipher->process(reinterpret_cast<uint8_t*>(data), len);
        return true;
    } catch (std::exception& e) {
        m_error = e.what();
        return false;
    }
}
```

After the decryption is done, the decompression can start. If the database is compressed, it is then decompressed using the **`QtIOCompressor`** class. The decompression format is **`gzip`**.

```
// src/format/Kdbx4Reader.cpp:104

    QIODevice* xmlDevice = nullptr;
    QScopedPointer<QtIOCompressor> ioCompressor;

    if (db->compressionAlgorithm() == Database::CompressionNone) {
        xmlDevice = &cipherStream;
    } else {
        ioCompressor.reset(new QtIOCompressor(&cipherStream));
        ioCompressor->setStreamFormat(QtIOCompressor::GzipFormat);
        if (!ioCompressor->open(QIODevice::ReadOnly)) {
            raiseError(ioCompressor->errorString());
            return false;
        }
        xmlDevice = ioCompressor.data();
    }
```

When the database is decrypted, and decompressed if needed, the inner format parsing can be achieved. The **`InnerHeader`** is very similar to the outside header, it starts with a field ID on 1 byte, followed by a length field on 4 bytes and the corresponding content.

The difference between the **`InnerHeader`** anerd the **`OuterHeader`** is the field ID meaning. For the inner header:

ID	Name	Description
0	**`KeePass2::InnerHeaderFieldID::End`**	End of inner header
1	**`KeePass2::InnerHeaderFieldID::InnerRandomStreamID`**	Stream cipher algorithm
2	**`KeePass2::InnerHeaderFieldID::InnerRandomStreamKey`**	Stream cipher key
3	**`KeePass2::InnerHeaderFieldID::Binary`**	Binary data before the database: Attachments for example

The following code snippet performs the inner header parsing. It checks that the size written in the fields match their actual size ?

```
// src/format/Kdbx4Reader.cpp:119
    while (readInnerHeaderField(xmlDevice) && !hasError()) {
    }

// src/format/Kdbx4Reader.cpp:252
bool Kdbx4Reader::readInnerHeaderField(QIODevice* device)
{
    QByteArray fieldIDArray = device->read(1);
    if (fieldIDArray.size() != 1) {
        raiseError(tr("Invalid inner header id size"));
        return false;
    }
    auto fieldID = static_cast<KeePass2::InnerHeaderFieldID>(fieldIDArray.at(0));

    bool ok;
    auto fieldLen = Endian::readSizedInt<quint32>(device, KeePass2::BYTEORDER, &ok);
    if (!ok) {
        raiseError(tr("Invalid inner header field length: field
%1").arg(static_cast<int>(fieldID)));
        return false;
    }

    QByteArray fieldData;
    if (fieldLen != 0) {
        fieldData = device->read(fieldLen);
        if (static_cast<quint32>(fieldData.size()) != fieldLen) {
            raiseError(tr("Invalid inner header data length: field %1, %2 expected, %3
found")
                           .arg(static_cast<int>(fieldID))
                           .arg(fieldLen)
                           .arg(fieldData.size()));
            return false;
        }
    }

    switch (fieldID) {
    case KeePass2::InnerHeaderFieldID::End:
        return false;

    case KeePass2::InnerHeaderFieldID::InnerRandomStreamID:
        setInnerRandomStreamID(fieldData);
        break;

    case KeePass2::InnerHeaderFieldID::InnerRandomStreamKey:
        setProtectedStreamKey(fieldData);
```

```
        break;

    case KeePass2::InnerHeaderFieldID::Binary: {
        if (fieldLen < 1) {
            raiseError(tr("Invalid inner header binary size"));
            return false;
        }
        auto data = fieldData.mid(1);
        m_binaryPool.insert(QString::number(m_binaryPool.size()), data);
        break;
    }
    }

    return true;
}
```

These parameters will be used to initialize a **`StreamCipher`**, in order to decrypt protected values inside the decrypted XML file. Chacha20 is the default algorithm for KDBX4 format. Previous formats also used to support Salsa20, but Chacha20 is the standard for new databases.

```
// src/format/Kdbx4Reader.cpp:132
        mode = SymmetricCipher::ChaCha20;
//[...]
    KeePass2RandomStream randomStream;
    if (!randomStream.init(mode, m_protectedStreamKey)) {
        raiseError(randomStream.errorString());
        return false;
    }

    Q_ASSERT(xmlDevice);

    KdbxXmlReader xmlReader(KeePass2::FILE_VERSION_4, binaryPool());
    xmlReader.readDatabase(xmlDevice, db, &randomStream);


// src/format/KeePass2RandomStream.cpp:32
bool KeePass2RandomStream::init(SymmetricCipher::Mode mode, const QByteArray& key)
{
// [...]
    case SymmetricCipher::ChaCha20: {
        QByteArray keyIv = CryptoHash::hash(key, CryptoHash::Sha512);
        return m_cipher.init(mode,
SymmetricCipher::Encrypt, keyIv.left(32), keyIv.mid(32, 12));
    }
```

```
// src/format/KdbxXmlReader.cpp:1014
QString KdbxXmlReader::readString(bool& isProtected, bool& protectInMemory)
{
    QXmlStreamAttributes attr = m_xml.attributes();
    isProtected = isTrueValue(attr.value("Protected"));
    protectInMemory = isTrueValue(attr.value("ProtectInMemory"));
    QString value = m_xml.readElementText();

    if (isProtected && !value.isEmpty()) {
        QByteArray ciphertext = QByteArray::fromBase64(value.toLatin1());
        bool ok;
        QByteArray plaintext = m_randomStream->process(ciphertext, &ok);
// [...]
        value = QString::fromUtf8(plaintext);
    }

    return value;
}
```

The stream cipher key for the hashed password correspond to the first 32 bytes of the `SHA512(StreamKey_in_innerheader)`, and the nonce is the following 12 bytes. This is conform to the RFC7539.

In order to verify dynamically that a corrupted database was rejected by KeePassXC, some tampering were made to the database as described in TEST-CORRUPTFILE in SF3: Database protection page 139. Indeed, KeePassXC should not decrypt a database that is corrupted. When tampering with the HMAC of the encrypted data blocks, the error message was related to the inner header `id` size. The inner header is part of the decrypted data, and should not be parsed if the HMAC is invalid.

Illustration 9: KeePassXC error on invalid block HMAC

The error shown comes from the following code snippet:

```
// src/format/Kdbx4Reader.cpp:159
    QByteArray fieldIDArray = device.read(1);
    if (fieldIDArray.size() != 1) {
        raiseError(tr("Invalid header id size"));
        return false;
```

```
    }
```

Using the tool frinet, Synacktiv experts were able to trace the different calls implied by `device.read(1)`:

```
py.exe C:\[...]\\frinet\tracer\trace.py attach 1384 KeePassXC.exe 0x3F36F0
```

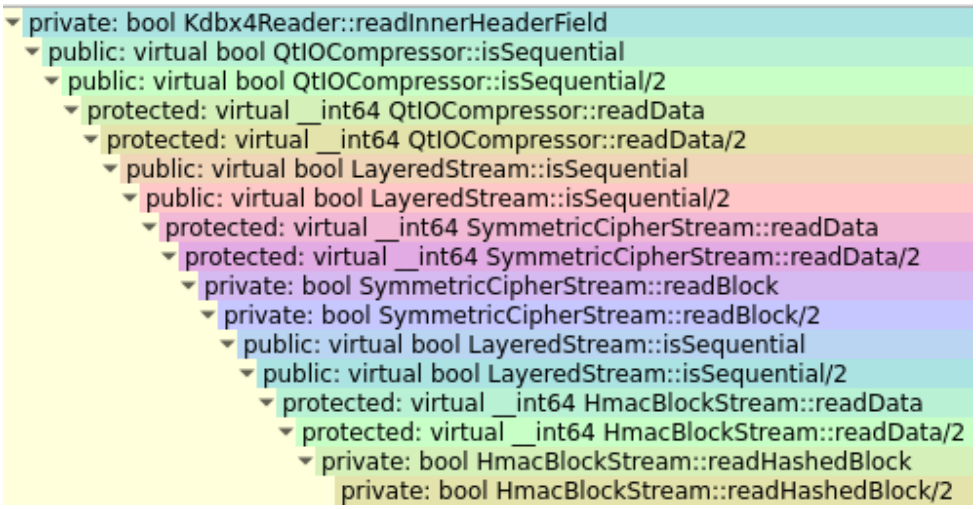


Illustration 10: Different calls implied by device.read(1)

As shown on the call tree view, the functions `QtIOCompressor::readData`, `SymmetricCipherStream::readData`, and `HmacBlockStream::readData` are called. By following the dynamic trace, each of these function returns -1, and the `fieldIDArray.size()` is performed on an object whose value is 0xFFFFFFFF and size is 0.

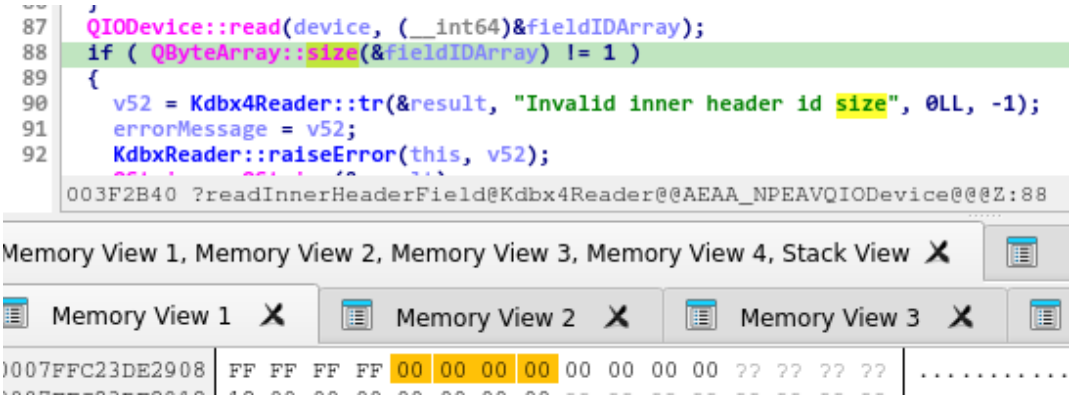


Illustration 11: Value tested for the fieldIDArray.size

Finally, the HMAC is properly verified and decryption does not happen when the HMAC is incorrect. Only the error message was misleading.

Saving the database

When the database has been modified, the seeds are regenerated so the whole encryption keys change, even if the passphrase or key file remain the same.

In a test file, the **master_seed**, **encryption_iv** and **KDF seed** have been modified after a modification in the file occurred.

```
// Initial file
- master_seed : 23 6a 28 ec 09 2e fb 6c 64 73 a4 80 07 07 92 d1 75 bb 63 ca f9 4f 6e 3a
c3 0a d7 ce ab e7 de 27
- encryption_iv : 65 29 31 3a ea 71 f6 3e 00 52 f8 23 d4 85 e6 91
- kdf_parameters :
    S : 89 57 65 39 86 a6 0a c0 bf 91 c1 4b 23 29 24 97 18 a9 44 74 9b e5 4f 16 a6 1d
8e 91 fa b8 0b cf
- Header's SHA256: 87 1a 5b 98 76 df 4c 3c 9f a0 97 82 85 f0 02 8f ff d4 7c ef dc 6b 96
26 cb 8b 55 97 bb 21 9f 41
- Header's HMAC: cd 78 06 cd 47 de fc ea de 6e 84 f9 7b c2 cb 39 e5 8a 4c 03 9b 1d 60
7e 29 d8 43 28 c3 e3 ef 73

// After a modification
- master_seed : e5 80 d7 bf f3 77 80 95 47 a2 25 f9 38 7b 3e 06 0b 14 69 b1 70 d0 8b 70
fc af 51 b5 b1 cd a4 65
- encryption_iv : 9b 70 d3 8b d4 4a 8a dd e9 d8 ec 7f f9 53 97 a7
- kdf_parameters :
    S : 0d 71 cc f9 04 13 96 3b 89 67 e3 ee 93 6e 03 aa 52 cc 41 14 08 6b 9c fa 12 96
be 6b 23 9d 1e 10
- Header's SHA256: 31 81 cb 43 2e 52 f1 cb b8 e6 a3 a7 07 86 e7 5b b3 3e af 32 a3 58 ba
0e c8 98 0e 8f 85 e7 0a a2
- Header's HMAC 73 32 a9 a0 14 9a 51 41 a5 1a fd 4c 65 51 a4 6a ce cf b0 07 3f c1 30 87
f4 02 78 c4 97 c3 12 74
```

Frida-trace can be used to find the proper calls to **writeDatabase** when the database is saved.

```
frida-trace.exe -p 7892 -a KeePassXC.exe!0x3c01 -a KeePassXC.exe!0x4a9d -a
KeePassXC.exe!0x6537 -a KeePassXC.exe!0x7e00 -a KeePassXC.exe!0xda21 -a KeePassXC.exe!
0xe16a -a KeePassXC.exe!0x12c1a -a KeePassXC.exe!0xb3cd0 -a KeePassXC.exe!0x203300 -a
KeePassXC.exe!0x203410 -a KeePassXC.exe!0x3f7070 -a KeePassXC.exe!0x3fa150 -a
KeePassXC.exe!0x43f1d0 -a KeePassXC.exe!0x43f440 -a KeePassXC.exe!0x5a2629 -a
KeePassXC.exe!0x5a264c -a KeePassXC.exe!0x5a2673 -a KeePassXC.exe!0x5a2699
```

When a database is modified and then saved, the call traces of the function shows:

```
           /* TID 0x1790 */
  9568 ms  Database::writeDatabase()
  9568 ms     | KeePass2Writer::writeDatabase()
  9568 ms     |    | Kdbx4Writer::writeDatabase()
 13386 ms     |    |    | KdbxXmlWriter::writeDatabase()
```

In the source code, the function **`Kdbx4Writer::writeDatabase`** does renew the secrets:

```
// src/format/Kdbx4Writer.cpp:34
bool Kdbx4Writer::writeDatabase(QIODevice* device, Database* db)
{
// [...]
    auto mode = SymmetricCipher::cipherUuidToMode(db->cipher());
// [...]
    int ivSize = SymmetricCipher::defaultIvSize(mode);
// [...]
    QByteArray masterSeed = randomGen()->randomArray(32);
    QByteArray encryptionIV = randomGen()->randomArray(ivSize);
    QByteArray protectedStreamKey = randomGen()->randomArray(64);
    QByteArray endOfHeader = "\r\n\r\n";
```

The function **`setKey`** is then called, the same as described in Key derivation section above, for the database read. The omitted parameter **`transformKey`** is **`True`** in the dynamic trace. The **`setKey`** will derive the database key using Argon2, and result in the derived key stored in the **`db->transformedDatabaseKey()`**.

```
// src/format/Kdbx4Writer.cpp:55
    if (!db->setKey(db->key(), false, true)) {
        raiseError(tr("Unable to calculate database key: %1").arg(db->keyError()));
        return false;
    }
```

As opposed to previous call to the **`setKey`** function, the parameter updateTransformSalt is set to true (3rd parameter). This leads to the following code being executed:

```
// src/core/Database.cpp:843
    if (updateTransformSalt) {
        m_data.kdf->randomizeSeed();
        Q_ASSERT(!m_data.kdf->seed().isEmpty());
    }
```

The randomizeSeed function will regenerate a random seed, using the Botan random generator object.

```
void Kdf::randomizeSeed()
{
    setSeed(randomGen()->randomArray(m_seed.size()));
}
```

The new encryption key, named the **finalKey** in the code, is then computed identically as in the read code.

```
// src/format/Kdbx4Writer.cpp:60
    // generate transformed database key
    CryptoHash hash(CryptoHash::Sha256);
    hash.addData(masterSeed);
    Q_ASSERT(!db->transformedDatabaseKey().isEmpty());
    hash.addData(db->transformedDatabaseKey());
    QByteArray finalKey = hash.result();
```

Next, the header is written:

```
// src/format/Kdbx4Writer.cpp:67
    // write header
    QByteArray headerData;
    {
        QBuffer header;
        header.open(QIODevice::WriteOnly);

        writeMagicNumbers(&header, KeePass2::SIGNATURE_1, KeePass2::SIGNATURE_2, db-
>formatVersion());

        CHECK_RETURN_FALSE(
            writeHeaderField<quint32>(&header, KeePass2::HeaderFieldID::CipherID, db-
>cipher().toRfc4122()));
        CHECK_RETURN_FALSE(writeHeaderField<quint32>(
            &header,
            KeePass2::HeaderFieldID::CompressionFlags,
            Endian::sizedIntToBytes(static_cast<int>(db->compressionAlgorithm()),
KeePass2::BYTEORDER)));
        CHECK_RETURN_FALSE(writeHeaderField<quint32>(&header,
KeePass2::HeaderFieldID::MasterSeed, masterSeed));
        CHECK_RETURN_FALSE(writeHeaderField<quint32>(&header,
KeePass2::HeaderFieldID::EncryptionIV, encryptionIV));

        // convert current Kdf to basic parameters
        QVariantMap kdfParams = KeePass2::kdfToParameters(db->kdf());
        QByteArray kdfParamBytes;
        if (!serializeVariantMap(kdfParams, kdfParamBytes)) {
            //: Translation comment: variant map = data structure for storing meta data
            raiseError(tr("Failed to serialize KDF parameters variant map"));
            return false;
        }
```

```
        CHECK_RETURN_FALSE(writeHeaderField<quint32>(&header,
KeePass2::HeaderFieldID::KdfParameters, kdfParamBytes));
        QVariantMap publicCustomData = db->publicCustomData();
        if (!publicCustomData.isEmpty()) {
            QByteArray serialized;
            serializeVariantMap(publicCustomData, serialized);
            CHECK_RETURN_FALSE(
                writeHeaderField<quint32>(&header,
KeePass2::HeaderFieldID::PublicCustomData, serialized));
        }

        CHECK_RETURN_FALSE(writeHeaderField<quint32>(&header,
KeePass2::HeaderFieldID::EndOfHeader, endOfHeader));
        header.close();
        headerData = header.data();
    }
CHECK_RETURN_FALSE(writeData(device, headerData));
```

The header is hashed, the database HMAC key is derived using the **masterSeed** and the **composite key**. And finally, the HMAC of the header is computed.

```
// src/format/Kdbx4Writer.cpp:108
    // hash header
    QByteArray headerHash = CryptoHash::hash(headerData, CryptoHash::Sha256);

    // write HMAC-authenticated cipher stream
    QByteArray hmacKey = KeePass2::hmacKey(masterSeed, db->transformedDatabaseKey());
    QByteArray headerHmac =
        CryptoHash::hmac(headerData, HmacBlockStream::getHmacKey(UINT64_MAX, hmacKey),
CryptoHash::Sha256);
    CHECK_RETURN_FALSE(writeData(device, headerHash));
    CHECK_RETURN_FALSE(writeData(device, headerHmac));
```

SymmetricCipherStream and **HmacBlockStream** are initialized for the encryption of the database blocks and the HMAC computation.

```
// src/format/Kdbx4Writer.cpp:118
    QScopedPointer<HmacBlockStream> hmacBlockStream;
    QScopedPointer<SymmetricCipherStream> cipherStream;

    hmacBlockStream.reset(new HmacBlockStream(device, hmacKey));
    if (!hmacBlockStream->open(QIODevice::WriteOnly)) {
        raiseError(hmacBlockStream->errorString());
        return false;
```

```
    }

    cipherStream.reset(new SymmetricCipherStream(hmacBlockStream.data()));

    if (!cipherStream->init(mode, SymmetricCipher::Encrypt, finalKey, encryptionIV)) {
        raiseError(cipherStream->errorString());
        return false;
    }
    if (!cipherStream->open(QIODevice::WriteOnly)) {
        raiseError(cipherStream->errorString());
        return false;
    }
```

If enabled on database creation, a compressor is initialized. The **outputDevice** object is now a **QIODevice**, so that anything written on it will be compressed, then encrypted and finally authenticated by an HMAC.

```
// src/format/Kdbx4Writer.cpp:138
  QIODevice* outputDevice = nullptr;
    QScopedPointer<QtIOCompressor> ioCompressor;

    if (db->compressionAlgorithm() == Database::CompressionNone) {
        outputDevice = cipherStream.data();
    } else {
        ioCompressor.reset(new QtIOCompressor(cipherStream.data()));
        ioCompressor->setStreamFormat(QtIOCompressor::GzipFormat);
        if (!ioCompressor->open(QIODevice::WriteOnly)) {
            raiseError(ioCompressor->errorString());
            return false;
        }
        outputDevice = ioCompressor.data();
    }

    Q_ASSERT(outputDevice);
```

The encrypted database data is written. First, the **innerHeader** followed by the XML data of the database. The protected fields are also encrypted using the Chacha20 algorithm.

```
// src/format/Kdbx4Writer.cpp:155
    CHECK_RETURN_FALSE(writeInnerHeaderField(
        outputDevice,
        KeePass2::InnerHeaderFieldID::InnerRandomStreamID,
```

```
Endian::sizedIntToBytes(static_cast<int>(KeePass2::ProtectedStreamAlgo::ChaCha20),
KeePass2::BYTEORDER)));
    CHECK_RETURN_FALSE(
        writeInnerHeaderField(outputDevice,
KeePass2::InnerHeaderFieldID::InnerRandomStreamKey, protectedStreamKey));

    // Write attachments to the inner header
    auto idxMap = writeAttachments(outputDevice, db);

    CHECK_RETURN_FALSE(writeInnerHeaderField(outputDevice,
KeePass2::InnerHeaderFieldID::End, QByteArray()));

    KeePass2RandomStream randomStream;
    if (!randomStream.init(SymmetricCipher::ChaCha20, protectedStreamKey)) {
        raiseError(randomStream.errorString());
        return false;
    }

    KdbxXmlWriter xmlWriter(db->formatVersion(), idxMap);
    xmlWriter.writeDatabase(outputDevice, db, &randomStream, headerHash);
```

The nonce and key used for the Chacha20 encryption are extracted from the SHA512 of the **`protectedStreamKey`** : first 32 bytes are the key and following 12 bytes are the nonce.

```
// src/format/Keepass2RandomStream.cpp:23

bool KeePass2RandomStream::init(SymmetricCipher::Mode mode, const QByteArray& key)
{
    switch (mode) {
    case SymmetricCipher::Salsa20: {
        return m_cipher.init(mode,
                             SymmetricCipher::Encrypt,
                             CryptoHash::hash(key, CryptoHash::Sha256),
                             KeePass2::INNER_STREAM_SALSA20_IV);
    }
    case SymmetricCipher::ChaCha20: {
        QByteArray keyIv = CryptoHash::hash(key, CryptoHash::Sha512);
        return m_cipher.init(mode,
SymmetricCipher::Encrypt, keyIv.left(32), keyIv.mid(32, 12));
    }
```

Compared to the database read process, there is an additional step before the **`KeePass2::InnerHeaderFieldID::End`**, which consists in writting the **`Attachments`**.

Attachments are used to store private SSH keys for example. The details of SSH key handling are described in part SF7: ssh-agent interaction page 100.

```
// src/format/Kdbx4Writer.cpp:163
    // Write attachments to the inner header
    auto idxMap = writeAttachments(outputDevice, db);
```

The attachments can be linked to an entry. In the resulting XML, attachments will be referenced by their ID:

```
<Binary>
     <Key>attachment.txt.txt</Key>
     <Value Ref="0"/>
</Binary>
```

The raw files are written in the **innerHeader** under the **Binary** type. The first **binary innerheader** will have the reference 0, the second one 1, and so on. In our example database, the parsing of the inner header displays:

```
Start reading inner header
id: inner_random_stream_id, size: 0x4, b'\x03\x00\x00\x00'
id: inner_random_stream_key, size: 0x40,
b'\x9a\x82\xe2\x92\xb5b\xd5\nv\xd4m\x8fB\xf7\xc4\x86E\x0f\xa4\xb5\x1a\xdc2\xc0\x99n\xb0
\xd7G\xabD\x1fl\x8e\xc6"\x9a\xc3\x80\xfeL>\xe5\x1dj\x08\x080\xcb=2\xa4\xaa\xb2Am\xc6c\x
82\x0e\x1f\xf6xR'
id: binary, size: 0x1b, b'\x01This is a test attachment.'
id: end, size: 0x0, b'
```

The binary entry corresponds to the raw content of the attached file.

Conclusion

To conclude, KeePassXC transforms the passphrase entered by the user and/or the keyfile provided in pseudo-random bytes using a key derivation function. Precisely,

- if only a password is provided, **derivedKey = Argon2d[41 rounds](SHA256(SHA256(passphrase)))**
- if only a keyfile is provided, **derivedKey = Argon2d[41 rounds](SHA256(key_in_keyfile))**
- if both passphrase and keyfile are provided, **derivedKey = Argon2d[41 rounds](SHA256(SHA256(passphrase) . key_in_keyfile))**

Moreover, KeePassXC uses several algorithms to protect the database against data corruption, data tampering and to provide data confidentiality:

- The header integrity is protected using the **SHA256** algorithm
- The header authenticity is protected using the **HMAC_SHA256** algorithm, with:

```
hmac_header = MAC_SHA256(hmacHeaderKey)(headerData)
```

where the headerHmacKey is equal to:

```
headerHmacKey = SHA512(0xFFFFFFFFFFFFFFFF . SHA512(masterSeed . derivedKey . "\x01"))
```

- The blocks' authenticity is protected using the **HMAC_SHA256** algorithm, with:

```
hmac_block_i = MAC_SHA256(blockHmacKey_i)(block_index_i . block_size_i .
encrypted_block_i)
```

where the blockHmacKey_i is equal to :

```
blockHmacKey_i = SHA512(block_index_i . SHA512(masterSeed . derivedKey . "\x01"))
```

- The blocks confidentiality is provided using the **AES_CBC** algorithm, with:

```
finalKey = SHA256(masterSeed . derivedKey)
```

The IV is stored in the header.

- The passwords are obfuscated inside the XML file, using the **Chacha20** algorithm with a key composed of random bytes stored in the block header of the block where the password is stored.

Finally, the real encryption key of the database changes every time a change has been done to the database and all the random seeds. The re-encryption happens on database save.

SF4: Memory protection

```
SF4: Memory Protection
```

```
KeePassXC renders its memory impossible to access for unprivileged users. This
protection is not effective against attackers with administration privileges. This
protection is persistent after the process KeePassXC has been terminated as well as
when the database is locked.
```

Containing sensitive secrets in memory, KeePassXC protects its memory against unprivileged users but also the current user. Hence, high privileges on the system are required to access its memory and resources.

Process protections

These protections are triggered during the initialization phase of KeePassXC. Two mechanisms are effectively configured for Windows builds:

- **`SetupSearchPaths()`**, which disables the current directory for searching DLL and configures the safe search to prioritize the program directory (logically **`C:\Program Files\KeePassXC`**) and then the system directories.
- Enforce specific ACL to the current process to prevent other unprivileged users and the current users to open the process with interesting permissions.

These mitigations are triggered in the bootstrap class (**`core/Bootstrap.cpp`**), in the constructor, called from the program entry point:

```
// src/main.cpp:51

int main(int argc, char** argv)
{
   QT_REQUIRE_VERSION(argc, argv, QT_VERSION_STR)
   QApplication::setAttribute(Qt::AA_EnableHighDpiScaling);
   QGuiApplication::setAttribute(Qt::AA_UseHighDpiPixmaps);
#if QT_VERSION >= QT_VERSION_CHECK(5, 14, 0) && defined(Q_OS_WIN)

QGuiApplication::setHighDpiScaleFactorRoundingPolicy(Qt::HighDpiScaleFactorRoundingPoli
cy::PassThrough);
#endif
   Application app(argc, argv);
   [...] /* argument parsing and initialization */
   Application::bootstrap(config()->get(Config::GUI_Language).toString());
```

The Bootstrap constructors calls three local functions:

```
// src/core/Bootstrap.cpp:66
```

```
void bootstrap(const QString& uiLanguage)
{
#ifdef QT_NO_DEBUG
    disableCoreDumps();
#endif
    setupSearchPaths();
    applyEarlyQNetworkAccessManagerWorkaround();
    Translator::installTranslators(uiLanguage);
}
```

The function `disableCoreDumps` performs multiple operations, but only the `createWindowsDACL` applies to a Windows build. This function creates three ACEs that restrict the permissions by creating an allow list of actions applied to Windows accounts. Setting these three ACE will by design forbid any other access not explicitly described by these ACE.

First, the permissions applied to the current user account, running the KeePassXC process are restricted to querying limited information (PID, process name and state, mostly), synchronizing (for multithreading) and terminating the process.

```
 // src/core/Bootstrap.cpp:121
 bool createWindowsDACL()
    {
    // [...]
       pTokenUser = static_cast<PTOKEN_USER>(HeapAlloc(GetProcessHeap(), 0,
cbBufferSize));
       if (pTokenUser == nullptr) {
           goto Cleanup;
       }
       if (!GetTokenInformation(hToken, TokenUser, pTokenUser, cbBufferSize,
&cbBufferSize)) {
           goto Cleanup;
       }
       if (!IsValidSid(pTokenUser->User.Sid)) {
           goto Cleanup;
       }
       [...]
       // Add allowed access control entries, everything else is denied
       if (!AddAccessAllowedAce(
               pACL,
               ACL_REVISION,
               SYNCHRONIZE | PROCESS_QUERY_LIMITED_INFORMATION | PROCESS_TERMINATE, //
same as protected process
               pTokenUser->User.Sid // pointer to the trustee's SID
               )) {
           goto Cleanup;
```

```
        }
```

Next, the process is configured to give only READ permissions to the Process Owner SID:

```
// src/core/Bootstrap.cpp:170

        // Retrieve CreatorOwnerRights SID
        pOwnerRightsSid = static_cast<PSID>(HeapAlloc(GetProcessHeap(), 0,
pOwnerRightsSidSize));
        if (pOwnerRightsSid == nullptr) {
            goto Cleanup;
        }
        if (!CreateWellKnownSid(WinCreatorOwnerRightsSid, nullptr, pOwnerRightsSid,
&pOwnerRightsSidSize)) {
            auto error = GetLastError();
            goto Cleanup;
        }
        [...]
         // Explicitly set "Process Owner" rights to Read Only. The default is Full
Control.
        if (!AddAccessAllowedAce(pACL, ACL_REVISION, READ_CONTROL, pOwnerRightsSid)) {
            goto Cleanup;
        }
```

Without this permission, the Process Owner (the current user, belonging to the **`CreatorOwnerRights`** group) would be capable of modifying the DACL and revert the applied configuration.

Then, the permissions applied to the account **`LocalSystem`** are restricted to querying information about the KeePassXC process and duplicating their handles. The documentation indicates that this permission is needed by the ssh-agent service to be used for the SSH agent integration feature. If this feature is not enabled during the build, this code is also omitted:

```
// src/core/Bootstrap.cpp:160
        [...]
        // Retrieve LocalSystem account SID
        pLocalSystemSid = static_cast<PSID>(HeapAlloc(GetProcessHeap(), 0,
pLocalSystemSidSize));
        if (pLocalSystemSid == nullptr) {
            goto Cleanup;
        }
        if (!CreateWellKnownSid(WinLocalSystemSid, nullptr, pLocalSystemSid,
&pLocalSystemSidSize)) {
            goto Cleanup;
        }
```

```
       [...]
#ifdef WITH_XC_SSHAGENT
       // OpenSSH for Windows ssh-agent service is running as LocalSystem
       if (!AddAccessAllowedAce(pACL,
                                ACL_REVISION,
                                PROCESS_QUERY_INFORMATION | PROCESS_DUP_HANDLE, // just
enough for ssh-agent
                                pLocalSystemSid // known LocalSystem sid
                                )) {
           goto Cleanup;
       }
#endif
```

Finally, the ACE are set:

```
// src/core/Bootstrap.cpp:222

       // Set discretionary access control list
       bSuccess = ERROR_SUCCESS
                  == SetSecurityInfo(GetCurrentProcess(), // object handle
                                     SE_KERNEL_OBJECT, // type of object
                                     DACL_SECURITY_INFORMATION, // change only the
objects DACL
                                     nullptr,
                                     nullptr, // do not change owner or group
                                     pACL, // DACL specified
                                     nullptr // do not change SACL
                  );
```

The resulting DACL can be summarized as:

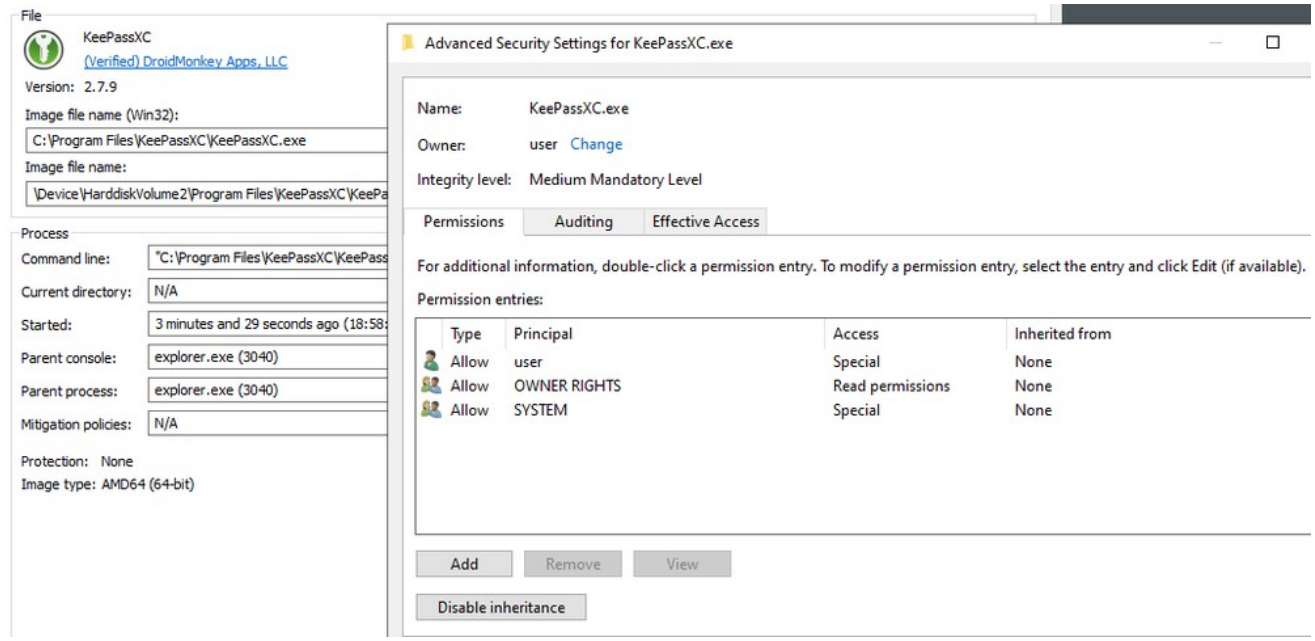


Illustration 12: DACL applied to the KeePassXC process.

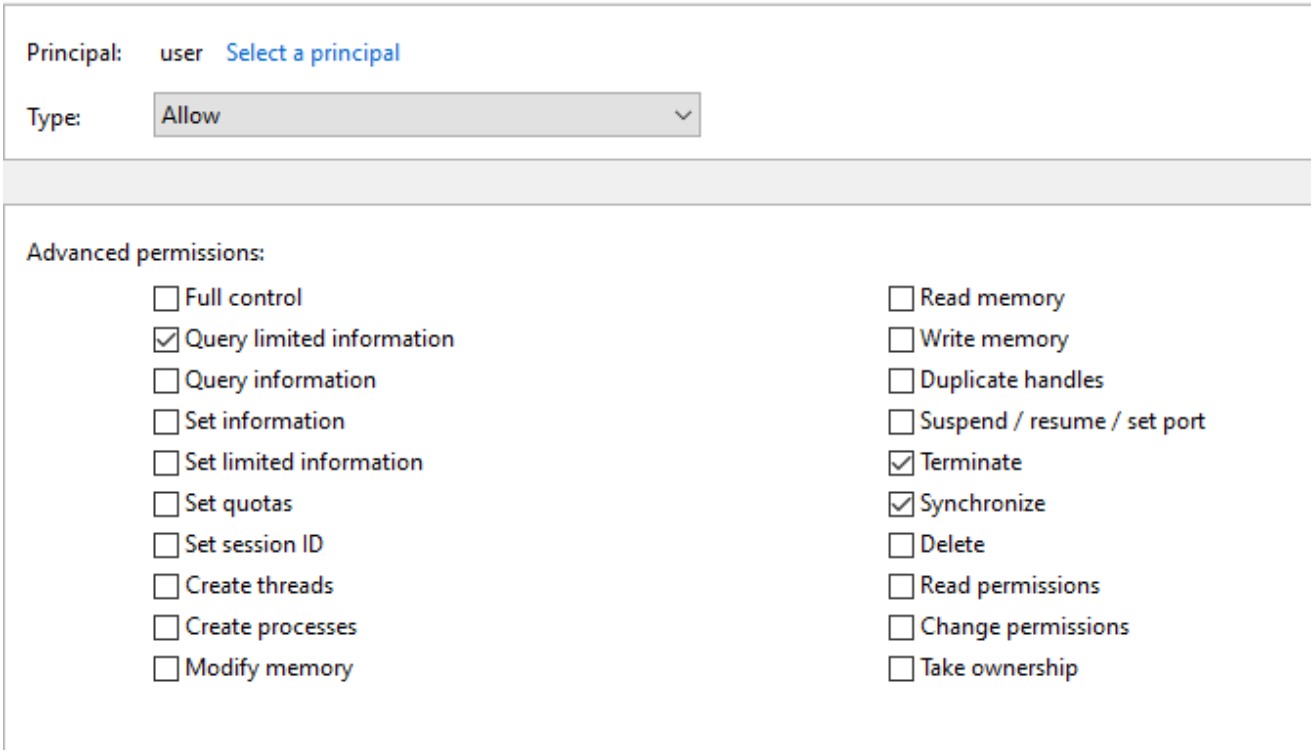


Illustration 13: DACL applied to the current user.

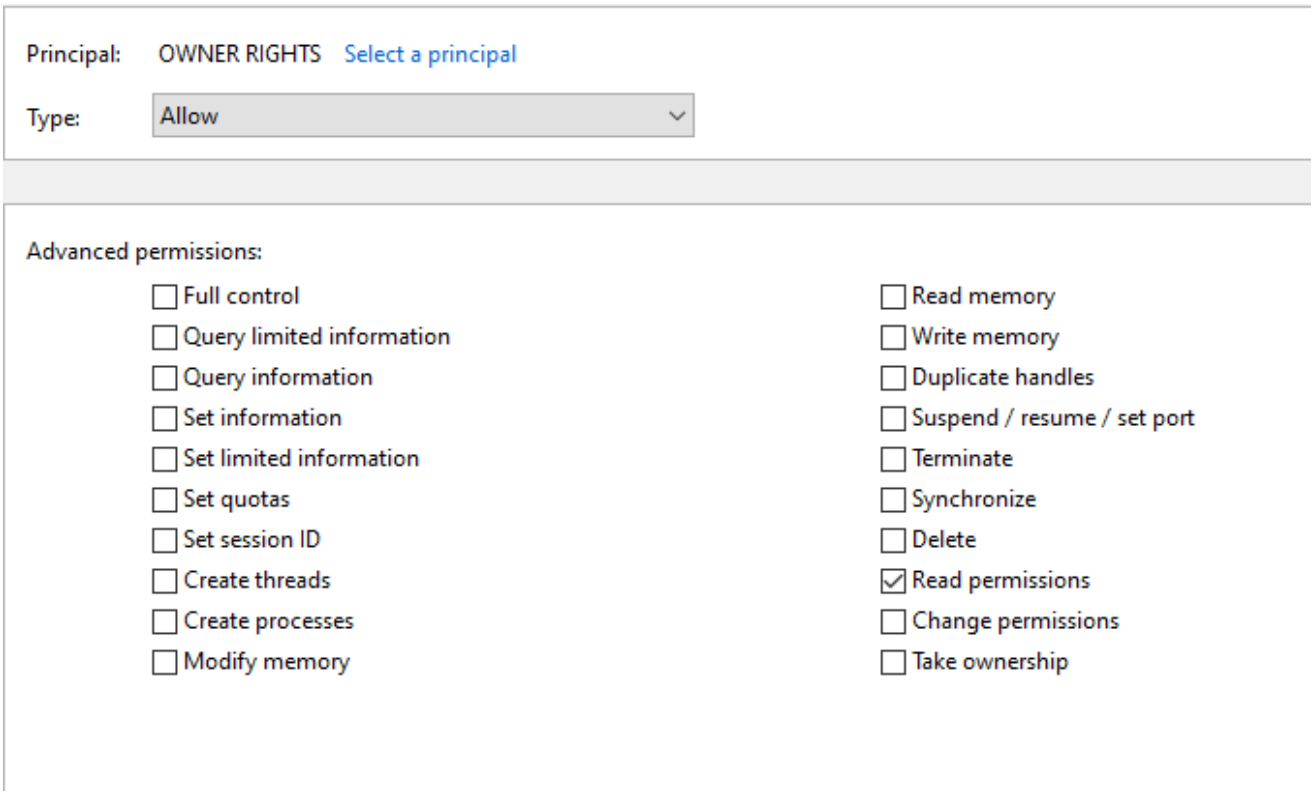


Illustration 14: DACL applied to the Process Owner.

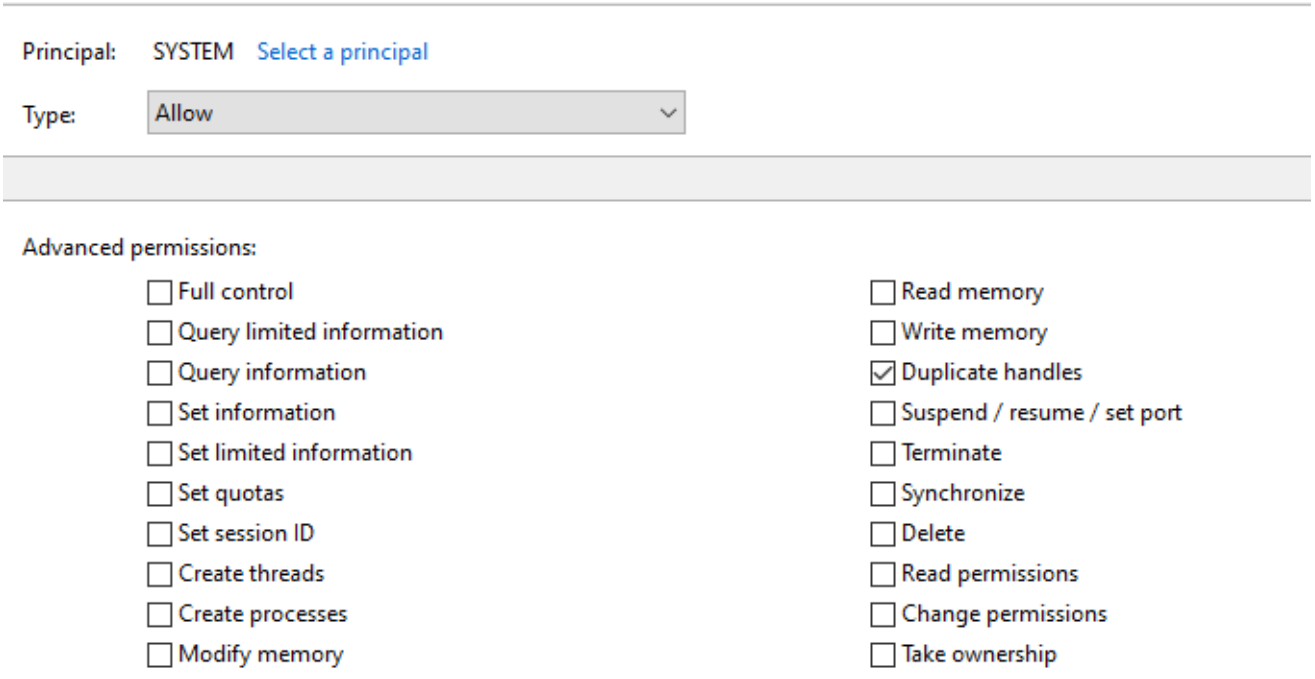


Illustration 15: DACL applied to the System process.

However, if any of these operations fails, the error propagated from this function to the `disableCoreDump` static method only results in a warning message:

```
// src/core/Bootstrap.cpp:104

#ifdef Q_OS_WIN
       success = success && createWindowsDACL();
#endif
       if (!success) {
           qWarning("Unable to disable core dumps.");
       }
   }
```

Synacktiv experts patched a custom build of KeePassXC to return false in the beginning of the `createWindowsDACL`:

```
 // src/core/Bootstrap.cpp:121
    bool createWindowsDACL()
   {
       bool bSuccess = false;
#ifdef Q_OS_WIN
       // Process token and user
       HANDLE hToken = nullptr;
       PTOKEN_USER pTokenUser = nullptr;
       DWORD cbBufferSize = 0;
       PSID pLocalSystemSid = nullptr;
       DWORD pLocalSystemSidSize = SECURITY_MAX_SID_SIZE;
       PSID pOwnerRightsSid = nullptr;
       DWORD pOwnerRightsSidSize = SECURITY_MAX_SID_SIZE;
       // Access control list
       PACL pACL = nullptr;
       DWORD cbACL = 0;

       return false;
```

It was then possible to indeed access the process memory and for example read the strings stored in memory:

```
PS C:\Users\user> .\Desktop\Tools\SysinternalsSuite\procdump.exe -ma keepassxc.exe
ProcDump v11.0 - Sysinternals process dump utility
Copyright (C) 2009-2022 Mark Russinovich and Andrew Richards
Sysinternals - www.sysinternals.com
[18:57:34] Dump 1 initiated: C:\Users\user\KeePassXC.exe_241113_185734.dmp
[18:57:34] Dump 1 writing: Estimated dump file size is 250 MB.
[18:57:37] Dump 1 complete: 250 MB written in 2.8 seconds
[18:57:37] Dump count reached.
```

```
PS C:\Users\user>
```

While with the standard KeePassXC build, the dump fails due to the applied DACL described above:

```
PS C:\Users\user> .\Desktop\Tools\SysinternalsSuite\procdump.exe -ma keepassxc.exe
ProcDump v11.0 - Sysinternals process dump utility
Copyright (C) 2009-2022 Mark Russinovich and Andrew Richards
Sysinternals - www.sysinternals.com
Error opening KeePassXC.exe (6812):
Access is denied. (0x00000005, 5)
```

Secrets in memory

Apart from the protection implemented on the KeePassXC process, Synacktiv experts studied the management of the secrets stored in the process memory. This analysis aimed to identify what secrets an attacker able to access the process memory could extract. Even though, such attacker was not considered in the security target. In particular, the locking mechanism was studied to ensure that secrets were correctly cleaned when a database is locked.

To perform this analysis, Synacktiv experts rebuilt KeePassXC in order to have the associated PDB file and performed different execution traces to study the lifecycle of secrets within the process memory. These traces were taken using Frinet (https://github.com/synacktiv/frinet) and processed through the IDA plugin Tenet (https://github.com/hexa-synacktiv/tenet). The PDB file (program database) gave information regarding the offset of each function and allowed to specifically target and trace chunks of assembly code.

These analyses, combined with a study of the source code, confirmed that the following secrets were correctly cleaned:

- The passphrase, coming from a Qt text label, manipulated in the `DatabaseOpenWidget::openDatabase` and `DatabaseOpenWidget::buildDatabaseKey` is wrapped in a `QSharedPointer<CompositeKey>` object and correctly scoped, therefore destroyed at the end of the `DatabaseOpenWidget::openDatabase`.
- Similarly, cryptographic resources used to decrypt the database are also scoped to the `Kdbx4Reader::readDatabaseImpl` and deallocated at the end of the function, only remains the database objects holding the database tree.
- The database tree containing the data and the decrypted password remains in memory unencrypted. However, when the database is locked the `db` object is recursively purged.

Cleaning of the passphrase in memory

The following paragraphs illustrate the checks performed by Synacktiv experts to ensure that the passphrase provided by the user was correctly cleaned from the process memory.

First, the unlocking passphrase was searched in the memory. To keep the analysis simple, the unlocking did not involve a key file. In the KeePassXC source code, the passphrase is retrieved from the Qt text label within the `DatabaseOpenWidget::buildDatabaseKey` function:

```
// src/gui/DatabaseOpenWidget:384

QSharedPointer<CompositeKey> DatabaseOpenWidget::buildDatabaseKey()
{
    auto databaseKey = QSharedPointer<CompositeKey>::create();

    if (!m_db.isNull() && canPerformQuickUnlock()) {
        // try to retrieve the stored password using Windows Hello
        QByteArray keyData;
        if (!getQuickUnlock()->getKey(m_db->publicUuid(), keyData)) {
            m_ui->messageWidget->showMessage(
                tr("Failed to authenticate with Quick Unlock:
%1").arg(getQuickUnlock()->errorString()),
                MessageWidget::Error);
            return {};
        }
        databaseKey->setRawKey(keyData);
        return databaseKey;
    }

    if (!m_ui->editPassword->text().isEmpty() || m_retryUnlockWithEmptyPassword) {
        databaseKey->addKey(QSharedPointer<PasswordKey>::create(m_ui->editPassword-
>text()));
    }
```

An execution trace was generated using the following command:

```
$ python .\tracer\trace.py attach 3144 KeePassXC.exe 0x2355D0
```

`0x2355D0` is the offset of the `DatabaseOpenWidget::buildDatabaseKey` function.

The trace reveals that the passphrase is written in memory by the function `QLineEdit::text`:

```
.text:000000014027FFE0 ?text@PasswordWidget@@QEAA?AVQString@@XZ proc near
.text:000000014027FFE0                                         ; CODE XREF: PasswordWidget::text(void)↑j
.text:000000014027FFE0                                         ; DATA XREF: .pdata:0000000140C203B8↓o
.text:000000014027FFE0
.text:000000014027FFE0 var_18          = dword ptr -18h
.text:000000014027FFE0 var_10          = qword ptr -10h
.text:000000014027FFE0 this            = qword ptr  8
.text:000000014027FFE0 arg_8           = qword ptr  10h
.text:000000014027FFE0
.text:000000014027FFE0                 mov     [rsp+arg_8], rdx
.text:000000014027FFE5                 mov     [rsp+this], rcx
.text:000000014027FFEA                 push    rdi
.text:000000014027FFEB                 sub     rsp, 30h
.text:000000014027FFEF                 mov     [rsp+38h+var_18], 0
.text:000000014027FFF7                 mov     rax, [rsp+38h+this]
.text:000000014027FFFC                 add     rax, 30h ; '0'
.text:0000000140280000                 mov     rcx, rax        ; this
.text:0000000140280003                 call    j_??C?$QScopedPointer@VPasswordWidget@Ui@@U?$QScopedPointerDeleter@VPas
.text:0000000140280008                 mov     rax, [rax+8]
.text:000000014028000C                 mov     [rsp+38h+var_10], rax
.text:0000000140280011                 mov     rdx, [rsp+38h+arg_8]
.text:0000000140280016                 mov     rcx, [rsp+38h+var_10]
.text:000000014028001B                 call    cs:__imp_?text@QLineEdit@@QEBA?AVQString@@XZ ; QLineEdit::text(void)
.text:0000000140280021                 mov     eax, [rsp+38h+var_18]
.text:0000000140280025                 or      eax, 1
.text:0000000140280028                 mov     [rsp+38h+var_18], eax
.text:000000014028002C                 mov     rax, [rsp+38h+arg_8]
.text:0000000140280031                 add     rsp, 30h
.text:0000000140280035                 pop     rdi
.text:0000000140280036                 retn
```

Illustration 16: Call to QLineEdit::text from PasswordWidget::text.

The function returns a pointer to **QString**, passed by reference in a stack variable. The **QString** is allocated in the heap:

```
000002661D1DCB80 | 01 00 00 00 20 00 00 00 61 00 00 00 CD CD CD CD | .... ...a.......
000002661D1DCB90 | 18 00 00 00 00 00 00 00 76 6F 68 74 68 61 65 68 | ........vohthaeh
000002661D1DCBA0 | 6F 68 73 65 74 68 32 6F 68 68 34 67 65 4E 67 61 | ohseth2ohh4geNga
000002661D1DCBB0 | 61 67 6F 61 63 38 74 68 00 CD CD CD CD CD CD CD | agoac8th........
000002661D1DCBC0 | CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD | ................
```

Illustration 17: Database passphrase located in the heap.

The pointer is then reused prior to computing the **CompositeKey** that will be used as input to the database decryption routine (already described in the chapter SF3: Database protection page 24).

To perform the computation, the passphrase is encoded in UTF8, and hashed using SHA-256:

```
// src/keys/PasswordKey.cpp:62

void PasswordKey::setPassword(const QString& password)
{
    setRawKey(CryptoHash::hash(password.toUtf8(), CryptoHash::Sha256));
}
```

```
.text:00000001400B1040 ?setPassword@PasswordKey@@QEAAXAEBVQString@@@Z proc near
.text:00000001400B1040                                         ; CODE XREF: PasswordKey::setPassword(QString const &)↑j
.text:00000001400B1040                                         ; DATA XREF: .pdata:0000000140C0C564↓o
.text:00000001400B1040
.text:00000001400B1040 var_48          = qword ptr -48h
.text:00000001400B1040 result          = QByteArray ptr -40h
.text:00000001400B1040 var_38          = qword ptr -38h
.text:00000001400B1040 var_30          = qword ptr -30h
.text:00000001400B1040 data            = qword ptr -28h
.text:00000001400B1040 var_20          = qword ptr -20h
.text:00000001400B1040 var_18          = qword ptr -18h
.text:00000001400B1040 var_10          = qword ptr -10h
.text:00000001400B1040 this            = qword ptr  8
.text:00000001400B1040 password        = qword ptr  10h
.text:00000001400B1040
.text:00000001400B1040 ; __unwind { // __CxxFrameHandler4_0
.text:00000001400B1040                 mov     [rsp+password], rdx
.text:00000001400B1045                 mov     [rsp+this], rcx
.text:00000001400B104A                 push    rdi
.text:00000001400B104B                 sub     rsp, 60h
.text:00000001400B104F                 mov     rax, [rsp+68h+this]
.text:00000001400B1054                 mov     rax, [rax]
.text:00000001400B1057                 mov     rax, [rax+10h]
.text:00000001400B105B                 mov     [rsp+68h+var_38], rax
.text:00000001400B1060                 lea     rdx, [rsp+68h+var_48]
.text:00000001400B1065                 mov     rcx, [rsp+68h+password]
.text:00000001400B106A                 call    cs:__imp_?toUtf8@QString@@QEGBA?AVQByteArray@@XZ ; QString::toUtf8(void)
```

Illustration 18: Assembly of PasswordKey::setPassword.

The UTF8 encoding copies the original **`QString`** and creates a new one also stored in the heap:

```
000002661D1DCB80 | 01 00 00 00 20 00 00 00 61 00 00 00 CD CD CD CD | .... ...a.......
000002661D1DCB90 | 18 00 00 00 00 00 00 00 76 6F 68 74 68 61 65 68 | ........vohthaeh
000002661D1DCBA0 | 6F 68 73 65 74 68 32 6F 68 68 34 67 65 4E 67 61 | ohseth2ohh4geNga
000002661D1DCBB0 | 61 67 6F 61 63 38 74 68 00 CD CD CD CD CD CD CD | agoac8th........
000002661D1DCBC0 | CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD | ................
```

Illustration 19: Copy of the passphrase in UTF8.

This **`QString`** is correctly destroyed at the end of the **`PasswordKey::setPassword`**:

```
.text:00000001400B10B8 ;   } // starts at 1400B10A0
.text:00000001400B10B9                 lea     rcx, [rsp+68h+result]
.text:00000001400B10BE                 call    cs:__imp_??1QByteArray@@QEAA@XZ ; QByteArray::~QByteArray(void)
.text:00000001400B10C4                 nop
.text:00000001400B10C5                 lea     rcx, [rsp+68h+var_48]
.text:00000001400B10CA                 call    cs:__imp_??1QByteArray@@QEAA@XZ ; QByteArray::~QByteArray(void)
.text:00000001400B10D0                 nop
.text:00000001400B10D1                 add     rsp, 60h
.text:00000001400B10D5                 pop     rdi
.text:00000001400B10D6                 retn
```

Illustration 20: QString destructor called on the UTF8 encoded passphrase (rsp+68h+var_48).

The original **`QString`** coming from the Qt text label is replaced by a call to **`QLineEdit::setText`** called from the **`DatabaseOpenWidget::clearForms`** function:

```
// src/gui/DatabaseOpenWidget.cpp:253

void DatabaseOpenWidget::clearForms()
{
    setUserInteractionLock(false);
```

```
   m_ui->editPassword->setText("");
   m_ui->editPassword->setShowPassword(false);
[...]
```

The memory is cleaned as a side effect of the **`setText`** method with an empty string.

```
000002661E223400 | DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD | ................
000002661E223410 | DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD | ................
000002661E223420 | DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD | ................
000002661E223430 | DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD | ................
000002661E223440 | DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD | ................
000002661E223450 | DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD | ................
```

Illustration 21: The memory containing the passphrase is cleaned.

The other secrets are similarly cleaned, mostly by using scoped variables that are automatically destroyed at the end of the scope (function). There is no explicit secure erasing of the editPassword field.

However, the case of the database object holding the password entries is different. This structure remains in memory until the end of the process or the lock of the database. When the database is locked, KeePassXC executes the function **`DatabaseWidget::lock`** that calls **`DatabaseWidget::replaceDatabase`**:

```
// src/gui/DatabaseWidget:1776

bool DatabaseWidget::lock()
{
[...]
   auto newDb = QSharedPointer<Database>::create(m_db->filePath());
   replaceDatabase(newDb);
[...]
```

This function ends by releasing data of the old database:

```
// src/gui/DatabaseWidget:446

void DatabaseWidget::replaceDatabase(QSharedPointer<Database> db)
{
[...]
   auto oldDb = m_db;
   m_db = std::move(db);
   [...]
   oldDb->releaseData();
```

```
}
```

The `Database::releaseData` function deletes the structure containing the Database entries:

```
// src/core/Database.cpp:487

void Database::releaseData()
{
   [...]
   // Reset and delete the root group
   auto oldGroup = setRootGroup(new Group());
   delete oldGroup;
   [...]
}
```

The `oldgroup` deletion triggers the group destructors that recursively delete the entries and subgroups:

```
// src/core/Group.cpp:55

Group::~Group()
{
   setUpdateTimeinfo(false);
   // Destroy entries and children manually so DeletedObjects can be added
   // to database.
   const QList<Entry*> entries = m_entries;
   for (Entry* entry : entries) {
       delete entry;
   }
   const QList<Group*> children = m_children;
   for (Group* group : children) {
       delete group;
   }
   if (m_db && m_parent) {
       DeletedObject delGroup;
       delGroup.deletionTime = Clock::currentDateTimeUtc();
       delGroup.uuid = m_uuid;
       m_db->addDeletedObject(delGroup);
   }
   cleanupParent();
}
```

A Frinet trace confirmed the triggering of the entries destructors:

Illustration 22: Destructors called when deleting the entries root.

Passphrase in memory

However, despite these mechanisms, Synacktiv experts noticed that the passphrase was lingering within the process memory and findable from a minidump when the passphrase was pasted:

```
PS Z:\dumps> C:\Users\user\Desktop\Tools\SysinternalsSuite\procdump.exe -ma
keepassxc.exe
ProcDump v11.0 - Sysinternals process dump utility
Copyright (C) 2009-2022 Mark Russinovich and Andrew Richards
Sysinternals - www.sysinternals.com
[16:11:00] Dump 1 initiated: Z:\dumps\KeePassXC.exe_241115_161100.dmp
[16:11:00] Dump 1 writing: Estimated dump file size is 325 MB.
[16:11:01] Dump 1 complete: 325 MB written in 1.0 seconds
[16:11:01] Dump count reached.
$ strings -e l KeePassXC.exe_241115_161100.dmp | grep -P
'voh thaehohseth2ohh4geNgaagoac8th'
voh thaehohseth2ohh4geNgaagoac8th
```

Using WinDBG, it was observed that passphrase was present in the heap part of the KeePassXC process:

```
0:000> s -u 000001661E223350 000002661E223350 L?800000000000
"vohthaehohseth2ohh4geNgaagoac8th"
00000266`1db2f100  0076 006f 0068 0074 0068 0061 0065 0068  v.o.h.t.h.a.e.h.
0:000> d 00000266`1db2f100
00000266`1db2f100  76 00 6f 00 68 00 74 00-68 00 61 00 65 00 68 00  v.o.h.t.h.a.e.h.
00000266`1db2f110  6f 00 68 00 73 00 65 00-74 00 68 00 32 00 6f 00  o.h.s.e.t.h.2.o.
00000266`1db2f120  68 00 68 00 34 00 67 00-65 00 4e 00 67 00 61 00  h.h.4.g.e.N.g.a.
00000266`1db2f130  61 00 67 00 6f 00 61 00-63 00 38 00 74 00 68 00  a.g.o.a.c.8.t.h.
00000266`1db2f140  00 00 dd dd dd dd dd dd-81 e8 cb 23 00 b0 02 8c  ...........#....
00000266`1db2f150  10 8b 2c 1d 66 02 00 00-60 1f 16 1e 66 02 00 00  ..,.f...`...f...
00000266`1db2f160  00 00 00 00 00 00 00 00-00 00 00 00 01 00 00 00  ................
00000266`1db2f170  10 00 00 00 00 00 00 00-a7 f7 0d 00 fd fd fd fd  ................
```

For example, given a database with the passphrase **vohthaehohseth2ohh4geNgaagoac8th**, entered manually, containing the passphrase of another database (**ieZoox0toh4mohlaegaethaPh1airooc**). When this second passphrase is copied and pasted to open the second database, it is still present in memory even though the database is locked or completely closed:

```
0:012> s -u 0 L?80000000000 ieZoox0toh4mohlaegaethaPh1airooc
00000205`10793c68  0069 0065 005a 006f 006f 0078 0030 0074  i.e.Z.o.o.x.0.t.
0:012> d 00000205`10793c68
00000205`10793c68  69 00 65 00 5a 00 6f 00-6f 00 78 00 30 00 74 00  i.e.Z.o.o.x.0.t.
00000205`10793c78  6f 00 68 00 34 00 6d 00-6f 00 68 00 6c 00 61 00  o.h.4.m.o.h.l.a.
00000205`10793c88  65 00 67 00 61 00 65 00-74 00 68 00 61 00 50 00  e.g.a.e.t.h.a.P.
00000205`10793c98  68 00 31 00 61 00 69 00-72 00 6f 00 6f 00 63 00  h.1.a.i.r.o.o.c.
00000205`10793ca8  00 00 fd fd fd fd ab ab-ab ab ab ab ab ab ab ab  ................
00000205`10793cb8  ab ab ab ab ab ab ee fe-00 00 00 00 00 00 00 00  ................
00000205`10793cc8  00 00 00 00 00 00 00 00-ee fe ee fe ee fe ee fe  ................
00000205`10793cd8  2e 3c 31 5e 47 66 0c 34-20 5b 0b 0e 05 02 00 00  .<1^Gf.4 [......
```

Overall, the multiple mechanisms involved in the security feature efficiently prevent arbitrary access to the process memory from unprivileged users. Regarding the secrets used to unlock the database, it has been identified that a copied and pasted passphrase lingered in memory even after closing the corresponding database. This behaviour is quite common for administrators who might store the password of a KeePassXC B in a KeePassXC A database. However, this issue needs administration privileges to be exploited by debugging the process or accessing its memory. As a reminder, a threat agent with such privileges was not considered in the scope of this evaluation.

SF5: Prompt of each password access from a browser extension

```
SF5: Prompt of each password access from a browser extension

Each access to passwords or passkey triggers pop-ups from the KeePassXC process
prompting for acceptance regarding the requested access.
```

The KeePassXC browser extension interacts with KeePassXC using a proxy. The proxy allows the extensions to communicate over the dedicated Named Pipe opened by KeePassXC. Concretely, the interaction follows the flows below:

- Upon activation of the browser support in the KeePassXC configuration, the process will create a Named Pipe and listen for connections. The Named Pipe has the following path: **`\\.\\pipe\\org.keepassxc.KeePassXC.BrowserServer_<USERNAME>`**.
- The KeePassXC process will also create a JSON file in a directory accessible to the chosen browser, for example: **`%localAppData%\KeePassXC\org.keepassxc.keepassxc_browser_brave.json`**. This file will contain the path of the proxy to be used by the browser and also described an allowlist of authorized extensions able to run the proxy. This allowlist works on the extension ID that are synchronized with their official builds. The browser extension registers a registry key pointing to this path (**`HKCU:\Software\Google\Chrome\NativeMessagingHosts\org.keepassxc.keepassxc_browser`**).
- When the extension is used for the first time, the browser will launch the proxy program (**`keepassxc-proxy.exe`**) and write to its standard output and read from its standard input. The proxy will in turn forward the data from or to the browser with the Named Pipe created by the KeePassXC process.

To limit the risk of other processes eavesdropping on the communication between the browser extension and the KeePassXC process, both parties exchange a public key and then encrypt every message. This encryption relies on the sodium library (NaCl).

To study the communication protocol, Synacktiv experts rebuilt the KeePassXC with a modified Named Pipe name and deployed a Python proxy intercepting the communication between **`keepassxc-proxy`** (and thus the browser extension) by binding the expected Named Pipe (Interception script for the browser extension communications page 147). At first, the Python proxy merely forwarded the packets from the browser extension to the KeePassXC process and displayed the received messages:

```
PS Z:\> python .\server.py
Someone is connected to \\.\\pipe\\org.keepassxc.KeePassXC.BrowserServer_user
>> Browser >> {"action":"change-public-
keys","publicKey":"8u1DVwLJ6UBVoHLD1hkHfbzE9HChZEbyAbnEq8os8Xg=","nonce":"wJp+
+zCflX4hy1hBXCqfWmD7bKQfmadE","clientID":"ptQyCNSfesotfI1gK8srCkqa6HiZlvlg"}
```

```
>> KeePassXC >> {"action":"change-public-keys","nonce":"wZp+
+zCflX4hy1hBXCqfWmD7bKQfmadE","publicKey":"ut1N8r8/rVnQH1zJID7ImQK3vtln57SA7j/
o6Wb27hM=","success":"true","version":"2.7.9"}

>> Browser >> {"action":"get-
databasehash","message":"CvRYe3CrI8ZOf1ozqQV+YpUtumroKLBzyZpUVhzT+XnOj2b0ySJwdE+67matPH
RZztd1XcRniyZdcGqAuPhmcOA0zZkpORhi8MohuZXaMx+8DOoBf42bE3f00WFKwgvioZOCzba7HCC783J5iPXYO
cbU+uvUpPpKGkE2zM68pqhDtQ==","nonce":"IRYm5xHCLqGRzFnbhYozfHV53ecMyyGz","clientID":"ptQ
yCNSfesotfI1gK8srCkqa6HiZlvlg"}

>> KeePassXC
>> {"action":"get-databasehash","message":"PUYxC+uomrTXGk/jZj918563SfwKFPFtKyuIJytarrF9
5oDPpKUb2IZvUS8/
YNzagRyi+J6GNPk+YvVZO3pkmcE+I9f9+xxtBYwtUe45kKNaOXVIqvPgp6dQWgQputHCgXdF6UbQwpbpm86KMys
TPWQP0pFMQ2mntZa91bj8ZpR6FPdqhtroqfhtX6N6sHT3dqhy4GL7dXOlJKrYxOhIJDv/
aBvZVU2zBwPSzCFqklSRYN9Nmz9ggxRDlMcUNKyLVZFS","nonce":"IhYm5xHCLqGRzFnbhYozfHV53ecMyyGz
"}

>> Browser >> {"action":"get-
databasehash","message":"kZvJO9K9t76yo+lVpltt5sbijzooVC3Y9u0nMcythO0b98b6DgC2n7xZs3dslb
+97S92/uKYDeAd0dSmGjfw9kqVeYCHhaymzkTu/
pX5OZm7qPPx92FY8dodT96LHNGMWKA16FTepEybL80nfYQbl1Zq6ci7YpNB4UMQaaVQQEmxuQ==","nonce":"4
Fy+Xaj6I/XkP4W015RJlQHyCxnVh9OF","clientID":"ptQyCNSfesotfI1gK8srCkqa6HiZlvlg"}

>> KeePassXC
>> {"action":"get-databasehash","message":"/TfnJPmpPwEP7T9NqcLa/NikdFFRvrRbSFv+No0kMqyD
fBElvUrme+ANMJZgTfQRIXpXBg10rBtYOmqUEvRfV/
n8hIWXLjJY2gZYivdm1q3HHn0AEUr17kpb0yV3RZu3RcwVaPfx8Iccj3v0k97aF91NVx6CGTa2iYPpzTuZqneiG
onnBiooippzXgko29+ion12nABk69ibnCaeePG2a6aPsjciVl4zBZZtxz7aOQ8SoOP6e8jBdtO5Ifqd/
Jjj9MKY","nonce":"4Vy+Xaj6I/XkP4W015RJlQHyCxnVh9OF"}

>> Browser >> {"action":"test-
associate","message":"ARH6eJp3hiUM9a8UvS1rAeTb4v5bJ5Op1yZ/
zBTW+uBevxSzXQsIO5jgqfPU6ySjpG/7qExfn6kMq92jpK/
8EgfoQ72ZV8WmZCBmzbxLUQmSewl2PhBUfVZsuUnFI3NvOHj07lwGKMXnm4qT9nd5cLNDqCEA","nonce":"KPa
sRSb1ajC2cKFLuepvExE4iKCkCz8g","clientID":"ptQyCNSfesotfI1gK8srCkqa6HiZlvlg"}

>> KeePassXC >> {"action":"test-
associate","message":"hNHgmJ2G5JAYfGJXN8y1+HGCk3l1HqieXpnsJ1x6pGbeaRVCT1GoL1AWomfli/
CF1Gd8Dr/qWV7B+aRU/ux1dEDHo6ULkk/6hEt5wMY8X3UTnlug5F9Z897atYACU1vVZnLCuln6Ykob/
lBTObMMKtjn7q3+
+KcplDDftU1pPC1+uKQIzyUO+f4w9JnVod8VP7BXTIFIzRjeghuHERXnUwfbkI3qeqbaNRdaporInscNc/
Tng6NqcUXi3d4UMfy03Ld2Zhf/
jE3FE0jipff34zWLwQsAVATetkk9ARaE","nonce":"KfasRSb1ajC2cKFLuepvExE4iKCkCz8g"}
```

The key exchange is visible in the beginning of the trace above. Then, for each subsequent message, the payload is encrypted using the public key of the recipient. Next, the Python proxy was modified to modify the exchanged public keys in order to be able to decrypt the messages:

```
PS Z:\> python .\server.py
Someone is connected to \\.\\pipe\\org.keepassxc.KeePassXC.BrowserServer_user
```

```
>> Browser >> {'action': 'get-databasehash',
'connectedKeys': ['2f2254913c8f0897ed6b1e85e90a489fdeaf5a29d6898247547826e3b6ab8bd0']}

>> KeePassXC >> {'hash':
'2f2254913c8f0897ed6b1e85e90a489fdeaf5a29d6898247547826e3b6ab8bd0',
'nonce': 'hGf5X/Lhy/3a6l11VBx+8Hn8A9dFQZ2m',
'success': 'true',
'version': '2.7.9'}

>> Browser >> {'action': 'test-associate',
'id': 'chrome-laptop',
'key': 'Yhpl+HJk99J2AXEyVYL1+df+LAeyNab+pSMEKznV6Qw='}

>> KeePassXC >> {'hash':
'2f2254913c8f0897ed6b1e85e90a489fdeaf5a29d6898247547826e3b6ab8bd0',
'id': 'chrome-laptop',
'nonce': '+jzqJdeXLwUrgVaifz6rJodDfScxcOW3',
'success': 'true',
'version': '2.7.9'}

>> Browser >> {'action': 'get-logins',
'id': 'chrome-laptop',
'keys': [{'id': 'chrome-laptop',
'key': 'Yhpl+HJk99J2AXEyVYL1+df+LAeyNab+pSMEKznV6Qw='}],
'submitUrl': 'https://www.cesti.lab/ap/signin',
'url': 'https://www.cesti.lab/ap/signin?[...]'}

>> KeePassXC >> {'count': 1,
'entries': [{'group': 'KeePassXC-Browser Passwords',
'login': 'test@s1n.fr',
'name': 'www.cesti.lab',
'password': "~hrM[oNvA+F/('LUw*_3",
'stringFields': [],
'uuid': 'fcca66395dae40b987fe8bbd38ef34a1'}],
'hash': '2f2254913c8f0897ed6b1e85e90a489fdeaf5a29d6898247547826e3b6ab8bd0',
'id': 'chrome-laptop',
'nonce': '+SmsRP/Fe+NzfP8+Eh5839qjx3HGSR4y',
'success': 'true',
'version': '2.7.9'}
```

The trace illustrates the connection until the query of a login for the url **`www.cesti.lab`**, the encrypted requests have not been displayed to simplify the trace.

The overall communication is protected by several mechanisms preventing the arbitrary retrieval of passwords :

- The Named Pipe path is fixed for one user session, if the path is already taken, the KeePassXC process does not bind it. Therefore, it is not feasible to set up a Man-in-the-Middle without

modifying the executables (KeePassXC.exe or keepassxc-proxy.exe) to make them use another path.

- Once the public keys are exchanged, communications are encrypted using robust cryptography based on the following algorithms that provides perfect forward secrecy (PFS), confidentiality, integrity, and authenticity:
  - X25519 for the key exchange, it is an ECDH algorithm.
  - HSalsa20 for the KDF function used to derive the symmetric key from the X25519 secret.
  - XSalsa20 for the encryption algorithm, it is a stream cipher initialized by a 192 bits nonce.
  - Poly1305 for the MAC algorithm.

  The request of a password requires a user validation. The validation can be set permanently to automatically allow the future retrieval of the same entry.

Man-in-the-Middle

Given an already running KeePassXC instance, a passive eavesdropping cannot obtain plain text passwords due to the encryption and the PFS. Active interception requires the modification of either the `KeePassXC` program, the `keepassxc-proxy` or the file containing the path of the proxy `%localAppData%\KeePassXC\org.keepassxc.keepassxc_browser_brave.json`. In the latter case, the file would contain another program dropped by an attacker for example:

```
{
   "allowed_origins": [
       "chrome-extension://pdffhmdngciaglkoonimfcmckehcpafo/",
       "chrome-extension://oboonakemofpalcgghocfoadofidjkkk/"
   ],
   "description": "KeePassXC integration with native messaging support",
   "name": "org.keepassxc.keepassxc_browser",
   "path": "C:\\Windows\\System32\\calc.exe",
   "type": "stdio"
}
```

Such modification would permit to perform the active interception and retrieve the passwords requested by the extension user. However, it requires killing the current `keepassxc-proxy`, waiting for the user to force a reload to make the browser re-reads the JSON file and run the new program. Killing the current `keepassxc-proxy` process results in a failure being displayed in the browser:

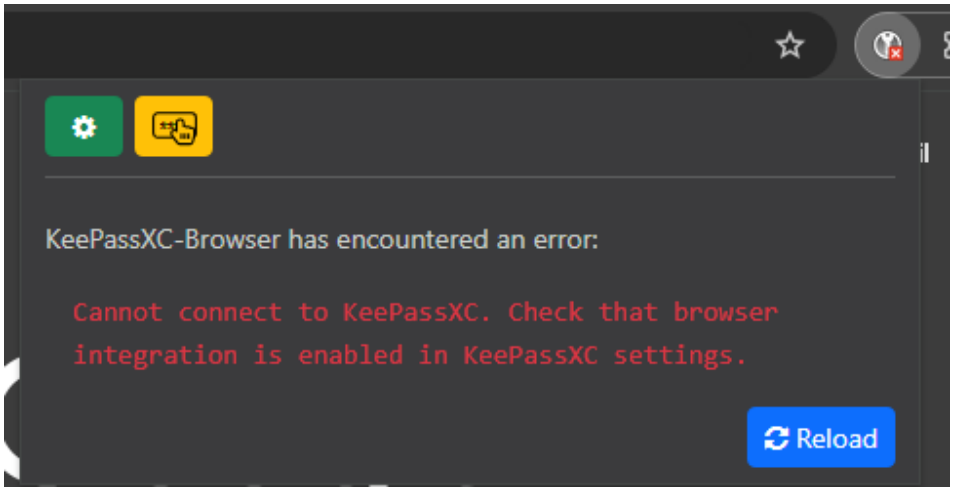


Illustration 23: The error triggered when the proxy process is killed.

Systematic prompt

When the browser extension requests a password, it sends the URL and the target URL of the HTML form. Upon receiving the request and finding a matching entry, the KeePassXC process prompts a dialogue window to the user to validate the operation:

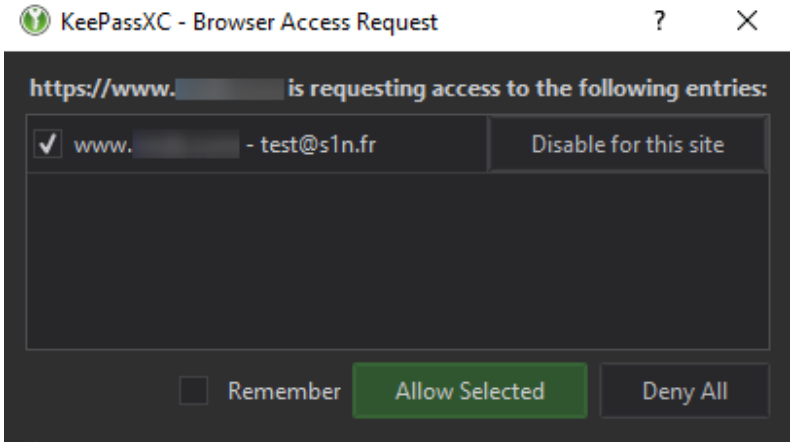


Illustration 24: KeePassXC prompts to allow the retrieval of an entry.

The user can either allow or deny the request and for both cases they may choose to remember the decision and instruct KeePassXC to automatically choose the same decision for future requests.

This decision is stored in the entry's properties:

Illustration 25: Behaviour policy stored in the entry properties.

Synacktiv experts studied the KeePassXC source code to understand how the entries were matched and whether the decision policy could be bypassed to allow retrieving arbitrary entries.

The KeePassXC process expects two arguments to look for an entry: `url` and `submitUrl`, the latter being optional:

```
{'action': 'get-logins',
'id': 'chrome-laptop',
'keys': [{'id': 'chrome-laptop',
'key': 'Yhpl+HJk99J2AXEyVYL1+df+LAeyNab+pSMEKznV6Qw='}],
'submitUrl': 'https://www.cesti.lab/ap/signin',
'url': 'https://www.cesti.lab/ap/signin?[...]'}
```

The request lands in the `BrowserAction::handleGetLogins` function that perform various initializations before calling the service `BrowserService::findEntries`

```
// src/browser/BrowserAction.cpp:223

QJsonObject BrowserAction::handleGetLogins(const QJsonObject& json, const QString&
action)
{
    [...]
    const auto browserRequest = decodeRequest(json);
    [...]
    const auto keyList = getConnectionKeys(browserRequest);
    EntryParameters entryParameters;
    entryParameters.dbid = id;
    entryParameters.hash = browserRequest.hash;
```

```
    entryParameters.siteUrl = siteUrl;
    entryParameters.formUrl = formUrl;
    entryParameters.httpAuth = httpAuth;
    bool entriesFound = false;
    const auto entries = browserService()->findEntries(entryParameters, keyList,
&entriesFound);
```

This function will call another function to obtain the matching entries and then will resolve the stored policy to either prompt, automatically allow or deny a request:

```
// src/browser/BrowserService.cpp:326

QJsonArray
BrowserService::findEntries(const EntryParameters& entryParameters, const
StringPairList& keyList, bool* entriesFound)
{
    if (entriesFound) {
        *entriesFound = false;
    }
    const bool alwaysAllowAccess = browserSettings()->alwaysAllowAccess();
    const bool ignoreHttpAuth = browserSettings()->httpAuthPermission();
    const QString siteHost = QUrl(entryParameters.siteUrl).host();
    const QString formHost = QUrl(entryParameters.formUrl).host();
    // Check entries for authorization
    QList<Entry*> entriesToConfirm;
    QList<Entry*> allowedEntries;
    for (auto* entry : searchEntries(entryParameters.siteUrl, entryParameters.formUrl,
keyList)) {
/* checking entry policy /* [...]
    }
```

The `BrowserService::searchEntries` only takes as argument the provided URL and form URL to look up the matching entries. This function iterates over all the entries and perform the following checks:

```
// src/browser/BrowserService.cpp:926

QList<Entry*> BrowserService::searchEntries(const QSharedPointer<Database>& db,
                                            const QString& siteUrl,
                                            const QString& formUrl,
                                            const QStringList& keys,
                                            bool passkey)
{
    QList<Entry*> entries;
    auto* rootGroup = db->rootGroup();
```

```
    if (!rootGroup) {
        return entries;
    }
    for (const auto& group : rootGroup->groupsRecursive(true)) {
    [...]
            for (auto* entry : group->entries()) {
            [...]
```

- Is the group/entry configured to be hidden from the extension? In this case, the group/entry is discarded from the search.
- Is the group/entry forbidden to be forwarded to a specific extension (identified by a key)? In this case the group/entry is discarded from the search. It seems that this setting is not available from the KeePassXC user interface.
- Is the entry in the recycle bin? If so, the entry is discarded from the search.

For the remaining entries, the provided URL are submitted to several checks:

- If the URL begins by the special URL scheme **`keepassxc://`**, then the entries are either matched against their path or their **`UUID`** according to the URL value:
  - **`keepassxc://by-uuid/032e8e0732244d928ed20e033e22e470`**
  - **`keepassxc://by-path/INTERNET/SITE`**

```
// src/browser/BrowserService.cpp:1336

bool BrowserService::shouldIncludeEntry(Entry* entry,
                                        const QString& url,
                                        const QString& submitUrl,
                                        const bool omitWwwSubdomain)
{
    // Use this special scheme to find entries by UUID
    if (url.startsWith("keepassxc://by-uuid/")) {
        return url.endsWith("by-uuid/" + entry->uuidToHex());
    } else if (url.startsWith("keepassxc://by-path/")) {
        return url.endsWith("by-path/" + entry->path());
    }
```

- If the URL begins with the special URL scheme **`file://`**, then a direct comparison is performed between the entry URL and the requested URL, (it is only check relying on the form URL):

```
// src/browser/BrowserService.cpp:1461
```

```
    // Make a direct compare if a local file is used
    if (siteUrl.startsWith("file://")) {
        return entryUrl == formUrl;
    }
```

- If the URL does not contain any scheme, then it is assumed that the requested scheme is for `https://`.

```
// src/browser/BrowserService.cpp:1444

   QUrl entryQUrl;
   if (entryUrl.contains("://")) {
       entryQUrl = entryUrl;
   } else {
       entryQUrl = QUrl::fromUserInput(entryUrl);
       if (browserSettings()->matchUrlScheme()) {
           entryQUrl.setScheme("https");
       }
   }
```

- Finally, the port, scheme and the presence of illegal characters are verified while the base domains and subdomains are compared:

```
// src/browser/BrowserService.cpp:1470

   // Match port, if used /* 1 */
   QUrl siteQUrl(siteUrl);
   if (entryQUrl.port() > 0 && entryQUrl.port() != siteQUrl.port()) {
       return false;
   }
   // Match scheme  /* 2 */
   if (browserSettings()->matchUrlScheme() && !entryQUrl.scheme().isEmpty()
       && entryQUrl.scheme().compare(siteQUrl.scheme()) != 0) {
       return false;
   }
   // Check for illegal characters /* 3 */
   QRegularExpression re("[<>\\^`{|}]");
   if (re.match(entryUrl).hasMatch()) {
       return false;
   }
   // Match the base domain /* 4 */
   if (urlTools()->getBaseDomainFromUrl(siteQUrl.host()) != urlTools()-
>getBaseDomainFromUrl(entryQUrl.host())) {
       return false;
   }
   // Match the subdomains with the limited wildcard /* 5 */
```

```
    if (siteQUrl.host().endsWith(entryQUrl.host())) {
        return true;
    }
```

To summarize, the following URL component are checked:

```
scheme://[2]subdomain[5].domain.tld[4]:port[1]
```

The check #3 ensures that the entry URL is well-formed.

The check #4 also distinguishes the top level domain from the second level domain to avoid that the requested URL only contains a top level domain to exploit the #5 check. This check is performed by identifying the top level domain using the public suffix list embed in the Qt Library:

```
// src/core/UrlTools.cpp:57

QString UrlTools::getBaseDomainFromUrl(const QString& url) const
{
    [...]
    const auto tld = getTopLevelDomainFromUrl(qUrl.toString());
    if (tld.isEmpty() || tld.length() + 1 >= host.length()) {
        return host;
    }
    // Remove the top level domain part from the hostname, e.g.
https://another.example.co.uk -> https://another.example
    host.chop(tld.length() + 1);
    // Split the URL and select the last part, e.g. https://another.example -> example
    QString baseDomain = host.split('.').last();
    // Append the top level domain back to the URL, e.g. example -> example.co.uk
    baseDomain.append(QString(".%1").arg(tld));
    return baseDomain;
}

QString UrlTools::getTopLevelDomainFromUrl(const QString& url) const
{
    auto host = QUrl::fromUserInput(url).host();
    if (isIpAddress(host)) {
        return host;
    }

    const auto numberOfDomainParts = host.split('.').length();
    static const auto dummy = QByteArrayLiteral("");

    // Only loop the amount of different parts found
```

```
    for (auto i = 0; i < numberOfDomainParts; ++i) {
        // Cut the first part from host
        host = host.mid(host.indexOf('.') + 1);

        QNetworkCookie cookie(dummy, dummy);
        cookie.setDomain(host);

        // Check if dummy cookie's domain/TLD matches with public suffix list
        if (!QNetworkCookieJar{}.setCookiesFromUrl(QList{cookie},
QUrl::fromUserInput(url))) {
            return host;
        }
    }

    return host;
}
```

Under the hood the Qt function **`setCookiesFromUrl`** validates whether the top level domain is present in the Public Suffix List (PSL) embedded in the Qt Library. This allows identifying dotted top level domain such as **`.co.uk`**.

To complement the source code analysis, Synacktiv experts performed dynamic tests to check whether it was possible to retrieve arbitrary entries without the knowledge of their configured URL. The following tests were performed:

- Ask for incomplete IP address (**`1.1`**)
- Specify different ports and schemes
- Check the specific scheme **`keepassxc://`**
- Try different top level domain and subdomains

None of the tests allowed to retrieve unexpected entries.

As described before, if an entry matches, a prompt is displayed to the user for them to accept or deny the password forward. This behaviour is implemented in the function **`BrowserService::findEntries`**. First, the policy stored in the entries properties are retrieved and checked:

```
// src/browser/BrowserService.cpp:363

        switch (checkAccess(entry, siteHost, formHost, entryParameters.realm)) {
        case Denied:
            continue;
        case Unknown:
            if (alwaysAllowAccess) {
```

```
                allowedEntries.append(entry);
            } else {
                entriesToConfirm.append(entry);
            }
            break;
        case Allowed:
            allowedEntries.append(entry);
            break;
        }
    }
```

The **`BrowserService::checkAccess`** loads the properties and check if the request URL is in the allowed list or in the denied list.

```
// src/browser/BrowserService.cpp:1195

BrowserService::Access
BrowserService::checkAccess(const Entry* entry, const QString& siteHost, const QString&
formHost, const QString& realm)
{
    if (entry->isExpired() && !browserSettings()->allowExpiredCredentials()) {
        return Denied;
    }
    BrowserEntryConfig config;
    if (!config.load(entry)) {
        return Unknown;
    }
    if ((config.isAllowed(siteHost)) && (formHost.isEmpty() ||
config.isAllowed(formHost))) {
        return Allowed;
    }
    if ((config.isDenied(siteHost)) || (!formHost.isEmpty() &&
config.isDenied(formHost))) {
        return Denied;
    }
    if (!realm.isEmpty() && config.realm() != realm) {
        return Denied;
    }
    return Unknown;
}
```

If this function returns **`Unknown`**, then the confirmation will be requested for this entry. This is handled by the **`BrowserService::confirmEntries`**. By default, an entry is allowed solely if it is selected by the user and if they click on “Allow Selected”. In all other cases, the entries are denied:

```
// src/browser/BrowserService.cpp:413

QList<Entry*> BrowserService::confirmEntries(QList<Entry*>& entriesToConfirm,
                                             const EntryParameters& entryParameters,
                                             const QString& siteHost,
                                             const QString& formUrl,
                                             const bool httpAuth)
{
   if (entriesToConfirm.isEmpty() || m_dialogActive) {
       return {};
   }
   [...]
   auto ret = accessControlDialog.exec();
   auto remember = accessControlDialog.remember();
   // All are denied
   if (ret == QDialog::Rejected && remember) {
       for (auto& entry : entriesToConfirm) {
           denyEntry(entry, siteHost, formUrl, entryParameters.realm);
       }
   }
   // Some/all are accepted
   if (ret == QDialog::Accepted) {
       auto selectedEntries = accessControlDialog.getEntries(SelectionType::Selected);
       for (auto& item : selectedEntries) {
           auto entry = entriesToConfirm[item->row()];
           allowedEntries.append(entry);
           if (remember) {
               allowEntry(entry, siteHost, formUrl, entryParameters.realm);
           }
       }
       // Remembered non-selected entries must be denied
       if (remember) {
           auto nonSelectedEntries =
accessControlDialog.getEntries(SelectionType::NonSelected);
           for (auto& item : nonSelectedEntries) {
               auto entry = entriesToConfirm[item->row()];
               denyEntry(entry, siteHost, formUrl, entryParameters.realm);
           }
       }
   }
   // Handle disabled entries (returned Accept/Reject status does not matter)
   auto disabledEntries = accessControlDialog.getEntries(SelectionType::Disabled);
   for (auto& item : disabledEntries) {
       auto entry = entriesToConfirm[item->row()];
       denyEntry(entry, siteHost, formUrl, entryParameters.realm);
   }
   // Re-hide the application if it wasn't visible before
   hideWindow();
```

```
    m_dialogActive = false;
    return allowedEntries;
}
```

Overall, this security feature is correctly implemented and protects the user's password as best as it can be according to what KeePassXC can do. Even though it is possible to set up a Man-in-the-Middle between the browser extension and the KeePassXC, an attacker would not be able to extract arbitrary entries.

SF6: KeeShare segregation

```
SF6: Imports using KeeShare are correctly segregated and do not permit to obtain any
information from the receiving database.
```

KeePassXC allows for database export or import through the KeeShare feature. Importing a database consists in adding a new subtree in a group that will contain the imported database entries. The export feature will save the given group as a new database on disk. For both cases, changes are synchronized in the corresponding way. It is also possible to synchronize a database which is equivalent to importing and exporting.

This feature is enabled by the build option **`-DWITH_XC_KEESHARE`** which is included in the default build.

The study of this security features aims to confirm that neither the import nor the export mechanisms leaked extra information related to the current database.

The import

The overall KeeShare mechanism is initialized during the creation of the **`MainWindow`**:

```
// src/gui/MainWindow.cpp:212

#if defined(WITH_XC_KEESHARE)
   KeeShare::init(this);
   m_ui->settingsWidget->addSettingsPage(new SettingsPageKeeShare(m_ui->tabWidget));
   connect(KeeShare::instance(),
           SIGNAL(sharingMessage(QString, MessageWidget::MessageType)),
           SLOT(displayGlobalMessage(QString, MessageWidget::MessageType)));
#endif
```

Then, when the database is open and the **`DatabaseWidget::replaceDatabase`** method is called, the **`KeeShare::connectDatabase`** instantiates and attaches an observer to the database:

```
// src/keeshare/ShareObserver.cpp:39

ShareObserver::ShareObserver(QSharedPointer<Database> db, QObject* parent)
   : QObject(parent)
   , m_db(std::move(db))
{
   connect(KeeShare::instance(), &KeeShare::activeChanged, this,
&ShareObserver::handleDatabaseChanged);
   connect(m_db.data(), &Database::groupDataChanged, this,
&ShareObserver::handleDatabaseChanged);
```

```
    connect(m_db.data(), &Database::groupAdded, this,
&ShareObserver::handleDatabaseChanged);
    connect(m_db.data(), &Database::groupRemoved, this,
&ShareObserver::handleDatabaseChanged);
    connect(m_db.data(), &Database::modified, this,
&ShareObserver::handleDatabaseChanged);
    connect(m_db.data(), &Database::databaseSaved, this,
&ShareObserver::handleDatabaseSaved);
    handleDatabaseChanged();
}
```

In most cases, the **`ShareObserver::handleDatabaseChanged`** function is called, followed by the method **`ShareObserver::reinitialize`** when KeeShare is indeed enabled. This function iterates over the database groups to identify those containing KeeShare settings:

```
// src/keeshare/ShareObserver.cpp:67

void ShareObserver::reinitialize()
{
    QList<QPair<QPointer<Group>, KeeShareSettings::Reference>> shares;
    for (Group* group : m_db->rootGroup()->groupsRecursive(true)) {
        auto oldReference = m_groupToReference.value(group);
        auto newReference = KeeShare::referenceOf(group);
        if (oldReference == newReference) {
            continue;
        }
        const auto oldResolvedPath = resolvePath(oldReference.path, m_db);
        m_groupToReference.remove(group);
        m_shareToGroup.remove(oldResolvedPath);
        m_fileWatchers.remove(oldResolvedPath);
        if (newReference.isValid()) {
            m_groupToReference[group] = newReference;
            const auto newResolvedPath = resolvePath(newReference.path, m_db);
            m_shareToGroup[newResolvedPath] = group;
        }
        shares.append({group, newReference});
    }
```

Then, for each identified share, if it is configured as an export, it does nothing as export is only triggered when saving the database. However, if it is an import, the **`ShareObserver::importShare`** is executed to read the target database and import it in the KeePassXC entries tree.

```
// src/keeshare/ShareObserver.cpp:97
```

```
    for (const auto& share : shares) {
        auto group = share.first;
        auto& reference = share.second;
        // Check group validity, it may have been deleted by a merge action
        if (!group) {
            continue;
        }
        [...]
        if (reference.isExporting()) {
            exported[reference.path] << group->name();
            // export is only on save
        }
        if (reference.isImporting()) {
            imported[reference.path] << group->name();
            // import has to occur immediately
            const auto result = this->importShare(reference.path);
            if (!result.isValid()) {
                // tolerable result - blocked import or missing source
                continue;
            }
```

Some checks are performed before the actual read to ensure that the share is active and not export only:

```
// src/keeshare/ShareObserver.cpp:214

ShareObserver::Result ShareObserver::importShare(const QString& path)
{
    if (!KeeShare::active().in) {
        return {};
    }
    const auto changePath = resolvePath(path, m_db);
    auto shareGroup = m_shareToGroup.value(changePath);
    if (!shareGroup) {
        qWarning("Group for %s does not exist", qPrintable(path));
        return {};
    }
    const auto reference = KeeShare::referenceOf(shareGroup);
    if (reference.type == KeeShareSettings::Inactive) {
        // changes of inactive references are ignored
        return {};
    }
    if (reference.type == KeeShareSettings::ExportTo) {
        // changes of export only references are ignored
        return {};
```

```
    }
    Q_ASSERT(shareGroup->database() == m_db);
    Q_ASSERT(shareGroup == m_db->rootGroup()->findGroupByUuid(shareGroup->uuid()));
    const auto resolvedPath = resolvePath(reference.path, m_db);
    return ShareImport::containerInto(resolvedPath, reference, shareGroup);
}
```

Finally, the insertion in the tree is performed by the method **`ShareImport::containerInto`**:

```
// src/keeshare/ShareImport.cpp:46

ShareObserver::Result ShareImport::containerInto(const QString& resolvedPath,
                                                 const KeeShareSettings::Reference&
reference,
                                                 Group* targetGroup)
{
    QByteArray dbData;
    auto uf = unzOpen64(resolvedPath.toLatin1().constData());
    if (uf) {
        // Open zip share, extract database portion, ignore signature file
        char zipFileName[256];
        auto err = unzGoToFirstFile(uf);
        while (err == UNZ_OK) {
            unzGetCurrentFileInfo64(uf, nullptr, zipFileName, sizeof(zipFileName),
nullptr, 0, nullptr, 0);
            if (QString(zipFileName).compare(KeeShare::containerFileName()) == 0) {
                dbData = readZipFile(uf);
            }
            err = unzGoToNextFile(uf);
        }
        unzClose(uf);
    } else {
        // Open KDBX file directly
        QFile file(resolvedPath);
        if (!file.open(QIODevice::ReadOnly)) {
            qCritical("Unable to open file %s.", qPrintable(reference.path));
            return {reference.path, ShareObserver::Result::Error, file.errorString()};
        }
        dbData = file.readAll();
        file.close();
    }
    QBuffer buffer(&dbData);
    buffer.open(QIODevice::ReadOnly);
    KeePass2Reader reader;
    auto key = QSharedPointer<CompositeKey>::create();
    key->addKey(QSharedPointer<PasswordKey>::create(reference.password));
```

```
    auto sourceDb = QSharedPointer<Database>::create();
    sourceDb->setEmitModified(false);
    if (!reader.readDatabase(&buffer, key, sourceDb.data())) {
        qCritical("Error while parsing the database: %s",
qPrintable(reader.errorString()));
        return {reference.path, ShareObserver::Result::Error, reader.errorString()};
    }
    sourceDb->setEmitModified(true);
    qDebug("Synchronize %s %s with %s",
           qPrintable(reference.path),
           qPrintable(targetGroup->name()),
           qPrintable(sourceDb->rootGroup()->name()));
    Merger merger(sourceDb->rootGroup(), targetGroup);
    merger.setForcedMergeMode(Group::Synchronize);
    merger.setSkipDatabaseCustomData(true);
    auto changelist = merger.merge();
    if (!changelist.isEmpty()) {
        return {reference.path, ShareObserver::Result::Success,
ShareImport::tr("Successful import")};
    }
    return {};
}
```

First, it obtains a handle on the database file to import, if it is compressed, it decompresses it and ignores the attached signature. Then, it reads the data using the **`KeePass2Reader`** class and decrypts it with the referenced password. Once the database has been read, KeePassXC adds it in the designated group. In import mode, any modification of this group is not back-ported to the database.

No modification of the target database is performed, during this loading process, as no write operation is performed.

Saving the overall database occurs through the event **`Database::databaseSaved`**, observed in the **`ShareObserver`** constructor:

```
// src/keeshare/ShareObserver.cpp:39
ShareObserver::ShareObserver(QSharedPointer<Database> db, QObject* parent)
    : QObject(parent)
    , m_db(std::move(db))
{
    [...]
    connect(m_db.data(), &Database::databaseSaved, this,
&ShareObserver::handleDatabaseSaved);
    [...]
}
```

The `handleDatabaseSaved` method first ensures that the KeeShare is enabled on this database and allowed to export parts of its tree:

```
// src/keeshare/ShareObserver.cpp:299
void ShareObserver::handleDatabaseSaved()
{
   if (!KeeShare::active().out) {
       return;
   }
   QStringList error;
   QStringList warning;
   QStringList success;
   const auto results = exportShares();
   [...]
}
```

Similarly, as the import process, the `ShareObserver::exportShares` method iterates over the database groups to find KeeShare settings, performs some checks on these settings (for example, whether the same target path is already used for the export of another group) and finally calls the `ShareExport::intoContainer`:

```
// src/keeshare/ShareObserver.cpp:246

QList<ShareObserver::Result> ShareObserver::exportShares()
{
   [...]
   QMap<QString, QList<Reference>> references;
   const auto groups = m_db->rootGroup()->groupsRecursive(true);
   for (const auto* group : groups) {
       const auto reference = KeeShare::referenceOf(group);
       if (!reference.isExporting()) {
           continue;
       }
       references[reference.path] << Reference{reference, group};
   }
   for (auto it = references.cbegin(); it != references.cend(); ++it) {
       if (it.value().count() != 1) {
           const auto path = it.value().first().config.path;
           QStringList groupnames;
           for (const auto& reference : it.value()) {
               groupnames << reference.group->name();
           }
           results << Result{
               path, Result::Error, tr("Conflicting export target path %1 in
%2").arg(path, groupnames.join(", "))};
```

```
        }
    }
    if (!results.isEmpty()) {
        // We need to block export due to config
        return results;
    }
    for (auto it = references.cbegin(); it != references.cend(); ++it) {
        [...]
        results << ShareExport::intoContainer(resolvedPath, reference.config,
reference.group);
        if (watcher) {
            watcher->start(resolvedPath, FileWatchPeriod, FileWatchSize);
        }
    }
    return results;
}
```

The `ShareExport::intoContainer` method creates a new database with a call to `ShareExport::extractIntoDatabase` and writes the database on the disk. In the case where the filename has the extension `.kdbx.share`, the written file corresponds to an archive containing a cryptographic signature and the database. The database is signed using the key defined within the KeePassXC program in order to be able to identify the KeePassXC instance responsible for exporting the resulting database:

```
// src/keeshare/ShareExport.cpp:160

ShareObserver::Result ShareExport::intoContainer(const QString& resolvedPath,
                                                 const KeeShareSettings::Reference&
reference,
                                                 const Group* group)
{
    QScopedPointer<Database> targetDb(extractIntoDatabase(reference, group));
    if (resolvedPath.endsWith(".kdbx.share")) {
        // Write database to memory and sign it
        QByteArray dbData, signatureData;
        QBuffer buffer;
        buffer.setBuffer(&dbData);
        buffer.open(QIODevice::WriteOnly);
        KeePass2Writer writer;
        if (!writer.writeDatabase(&buffer, targetDb.data())) {
            qWarning("Serializing export database failed: %s.",
writer.errorString().toLatin1().data());
            return {reference.path, ShareObserver::Result::Error, writer.errorString()};
        }
        buffer.close();
```

```
        // Get Own Certificate for signing
        const auto own = KeeShare::own();
        Q_ASSERT(!own.isNull());
        // Sign the database data
        KeeShareSettings::Sign sign;
        sign.certificate = own.certificate;
        signData(dbData, own.key, sign.signature);
        signatureData = KeeShareSettings::Sign::serialize(sign).toLatin1();
        auto zf = zipOpen64(resolvedPath.toLatin1().data(), 0);
        if (!zf) {
            return {reference.path, ShareObserver::Result::Error, ShareExport::tr("Could
not write export container.")};
        }
        writeZipFile(zf, KeeShare::signatureFileName().toLatin1().data(),
signatureData);
        writeZipFile(zf, KeeShare::containerFileName().toLatin1().data(), dbData);
        zipClose(zf, nullptr);
    } else {
        QString error;
        if (!targetDb->saveAs(resolvedPath, Database::Atomic, {}, &error)) {
            qWarning("Exporting database failed: %s.", error.toLatin1().data());
            return {resolvedPath, ShareObserver::Result::Error, error};
        }
    }
}
```

However, as previously seen, the zip archive is correctly opened during the import process, but the signature is not checked.

The written database is created in the **`ShareExport::extractIntoDatabase`**, which only exports entries, flattening the structure of the exported group:

```
// src/keeshare/ShareExport.cpp:65

Database* extractIntoDatabase(const KeeShareSettings::Reference& reference, const
Group* sourceRoot)
{
    [...]

    // Copy the source root as the root of the export database, memory manage the old
root node
    [...]
    const auto sourceEntries = sourceRoot->entriesRecursive(false);
    for (const Entry* sourceEntry : sourceEntries) {
        auto* targetEntry = sourceEntry->clone(Entry::CloneIncludeHistory);
        const bool updateTimeinfoEntry = targetEntry->canUpdateTimeinfo();
```

```
        targetEntry->setUpdateTimeinfo(false);
        targetEntry->setGroup(targetRoot);
        targetEntry->setUpdateTimeinfo(updateTimeinfoEntry);
        const auto iconUuid = targetEntry->iconUuid();
        if (!iconUuid.isNull() && !targetMetadata->hasCustomIcon(iconUuid)) {
            targetMetadata->addCustomIcon(iconUuid, sourceEntry->database()-
>metadata()->customIcon(iconUuid));
        }
    }
    auto key = QSharedPointer<CompositeKey>::create();
    key->addKey(QSharedPointer<PasswordKey>::create(reference.password));
    targetDb->setKey(key);

    auto obsoleteRoot = targetDb->setRootGroup(targetRoot);
    delete obsoleteRoot;

    targetDb->metadata()->setName(sourceRoot->name());
    [...]
    return targetDb;
}
```

The for loop is responsible for flattening the entries inside the target group. The name of the root is taken from the overall group from the source database.

This static analysis and a dynamic analysis confirmed that no information is written to the imported database from the receiving database. However, it has been identified that during an export, the root of the new database takes the name of the exported group. It can leak some information regarding the organization of the receiving database when KeeShare is configured for synchronizing a database in a specific group. In this case, both import and export process run and the root of the synchronized database will thus change to take the name of the group configured in the receiving database.

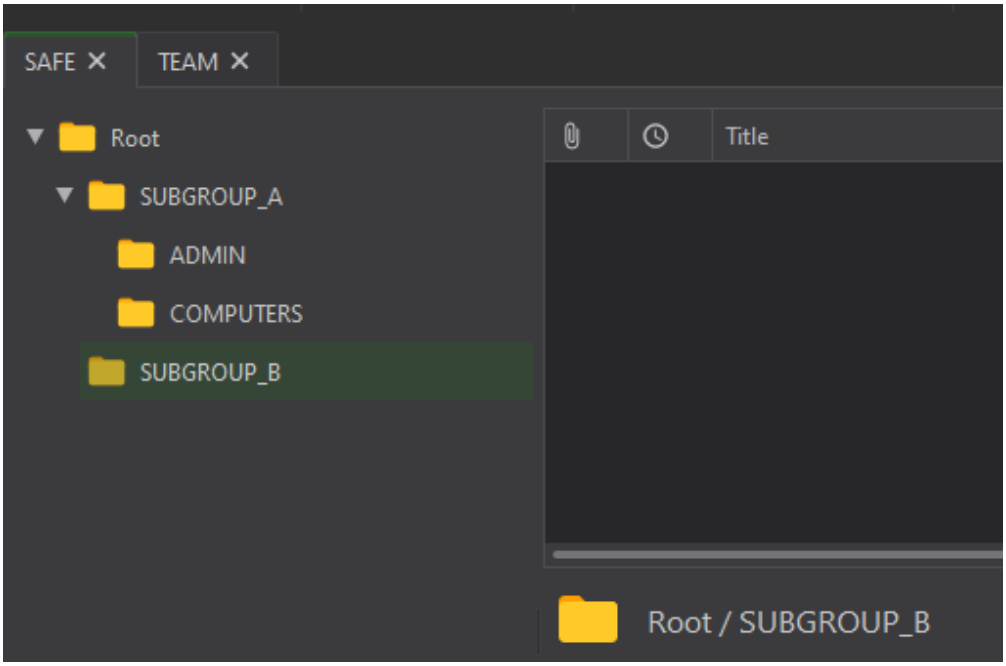


Illustration 26: Personal database that will synchronize another database in the **SUBGROUP_B** group.

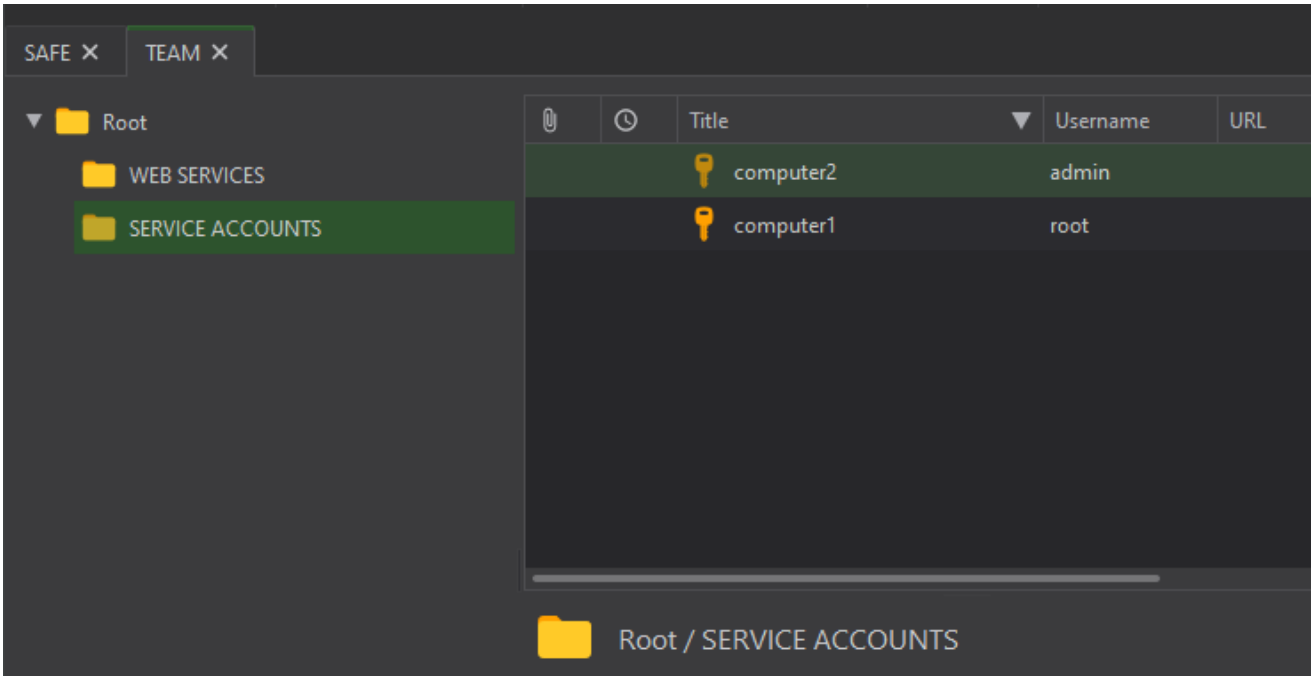


Illustration 27: TEAM database target of the synchronization.

The **SUBGROUP_B** has to be configured as an import first, otherwise, the export mechanism within the synchronized will empty override the content of the TEAM database. Then, the synchronization can be performed.

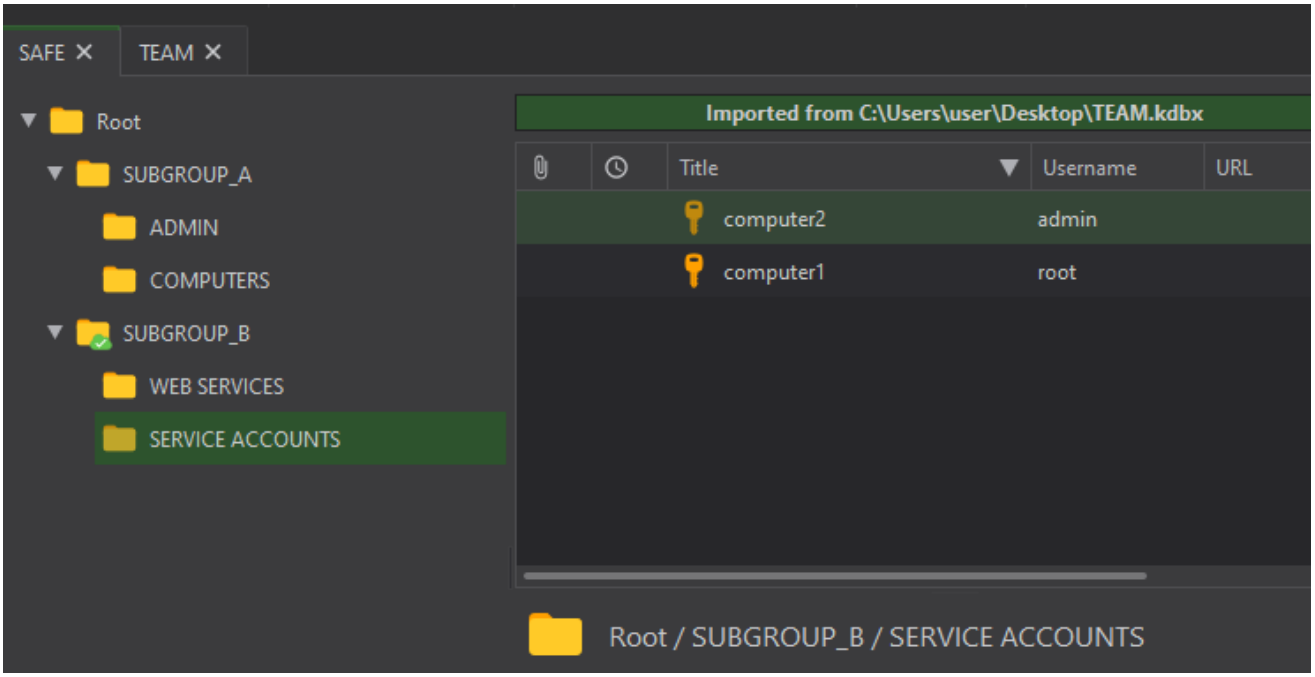


Illustration 28: The synchronization is performed from the SAFE database on the **SUBGROUP_B** group.

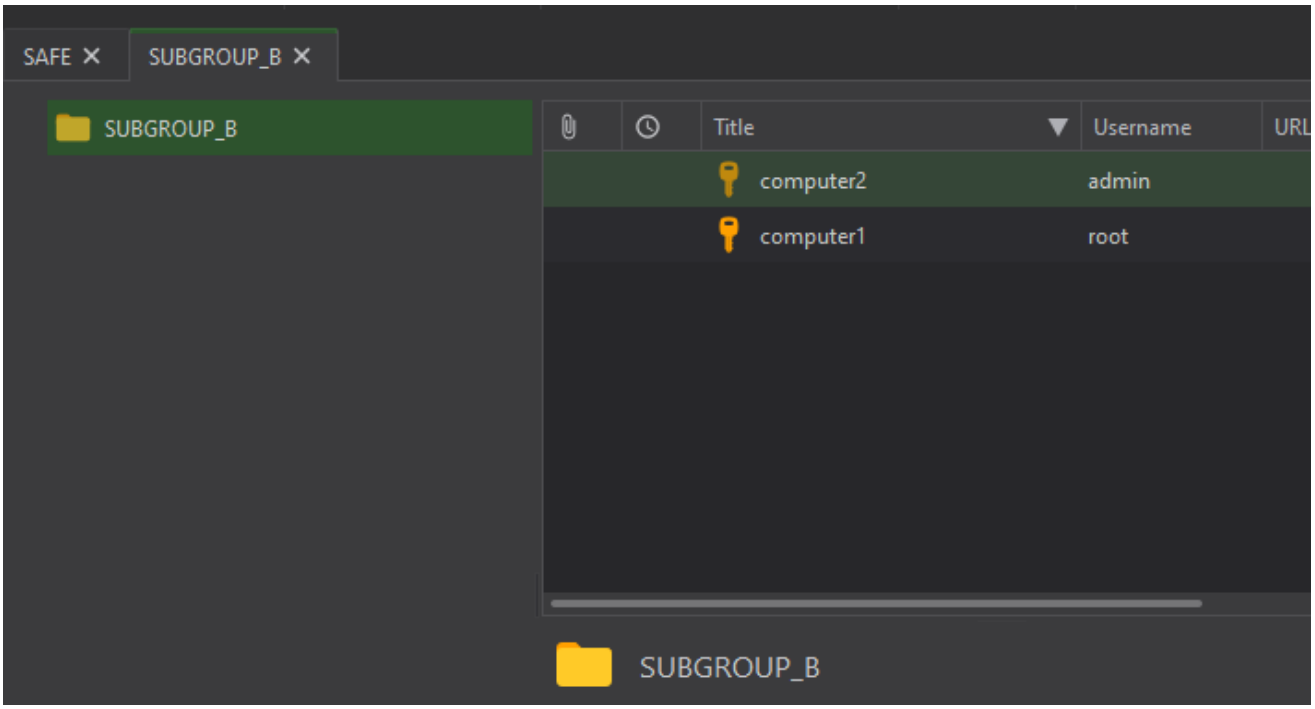


Illustration 29: When the SAFE database is saved, an export will be performed to the TEAM database that involves renaming the root group and flattening all the entries.

Overall, even though the feature does not ensure the complete integrity of the exported data regarding its structure, no security issue has been found in the KeeShare mechanisms.

SF7: ssh-agent interaction

```
SF7: ssh-agent interaction

KeePassXC securely interacts with ssh-agent/pageant to import private keys and
submitting unlocking passphrases.
```

KeePassXC provides a feature to store ssh private keys and load them automatically in an ssh-agent when unlocked. KeePassXC needs to be compiled with the flag `-DWITH_XC_SSHAGENT=On` in order to make this feature work. It is included in default builds.

To test this feature, the default OpenSSH program installed on Windows 10 has been used. The SSH version is OpenSSH_for_Windows_9.5p1, LibreSSL 3.8.2. By default, the `ssh-agent` is not started on Windows. The following PowerShell commands can be issued to manually start the `SSHAgent` service:

```
Get-Service -Name ssh-agent | Set-service -StartupType Manual
ssh-agent.exe
```

KeePassXC provides an option for the SSH integration that is not enabled by default. The SSH-Agent integration option is located in the settings, under the tab “SSH Agent” and in the “Enable SSH Agent integration” tick box. OpenSSH has been selected here for the purpose of this test.

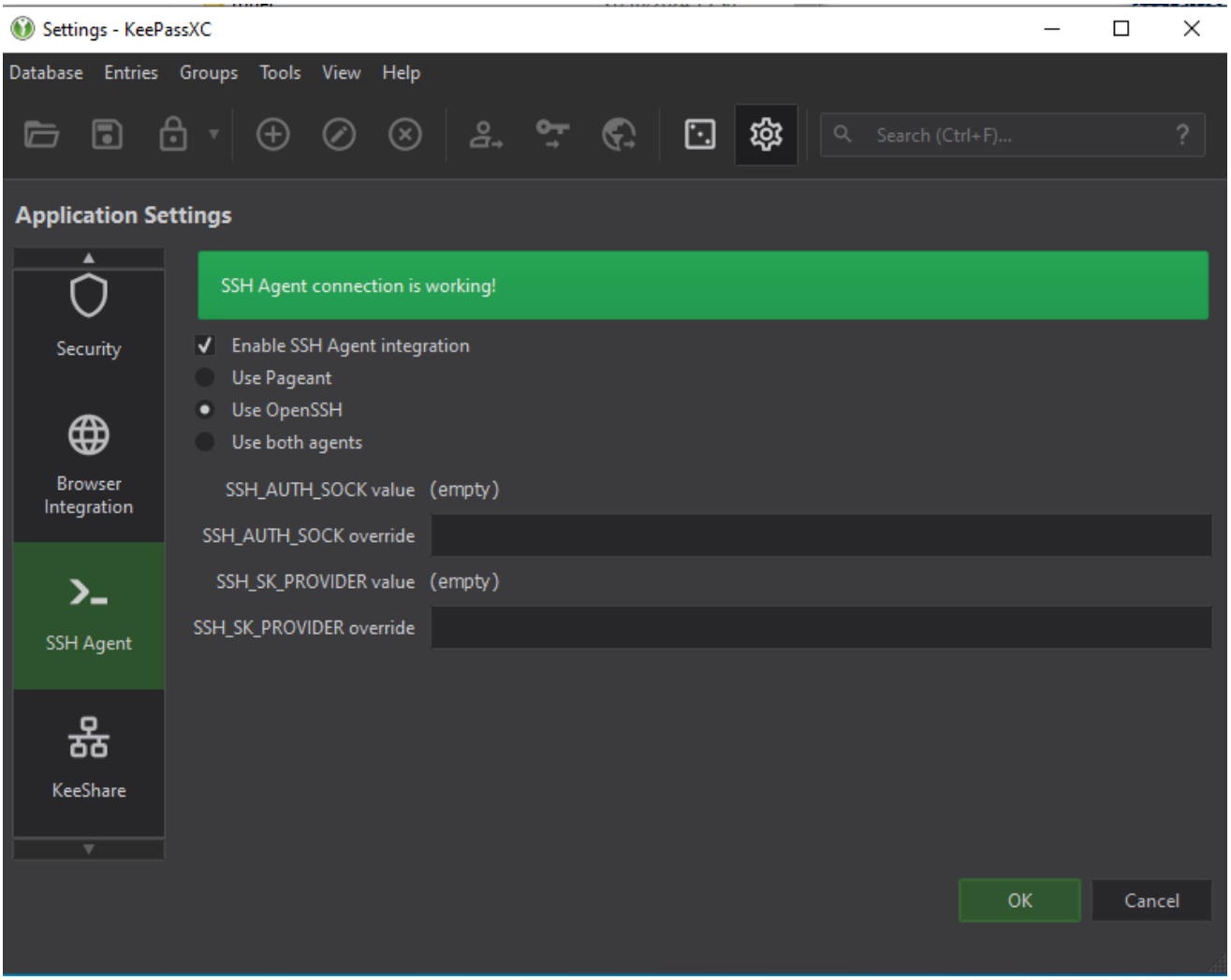


Illustration 30: SSH-Agent activation in KeePassXC settings.

If a user wishes to add an SSH private key, they need to create an entry in the database, and add the private key as an attachment.

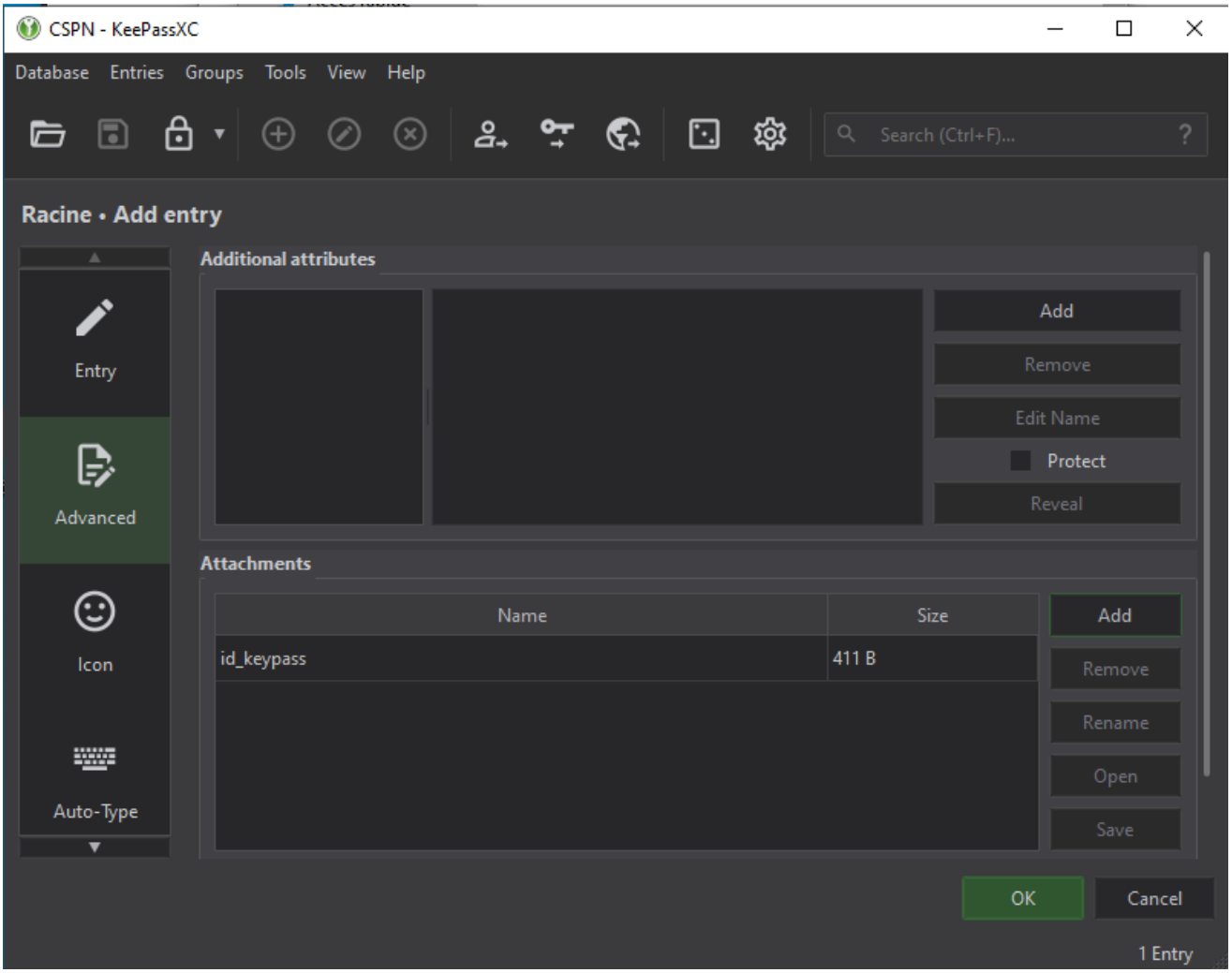


Illustration 31: Add ssh private key as attachment in KeePassXC entry.

Then, the user should select the SSH Agent tab, in the left panel menu of entry, and toggle the wanted options. In the Private Key section, the attachment that was selected before must be selected in order for the key to be added to an ssh-agent.

The following screenshot shows the options that were selected in this test.

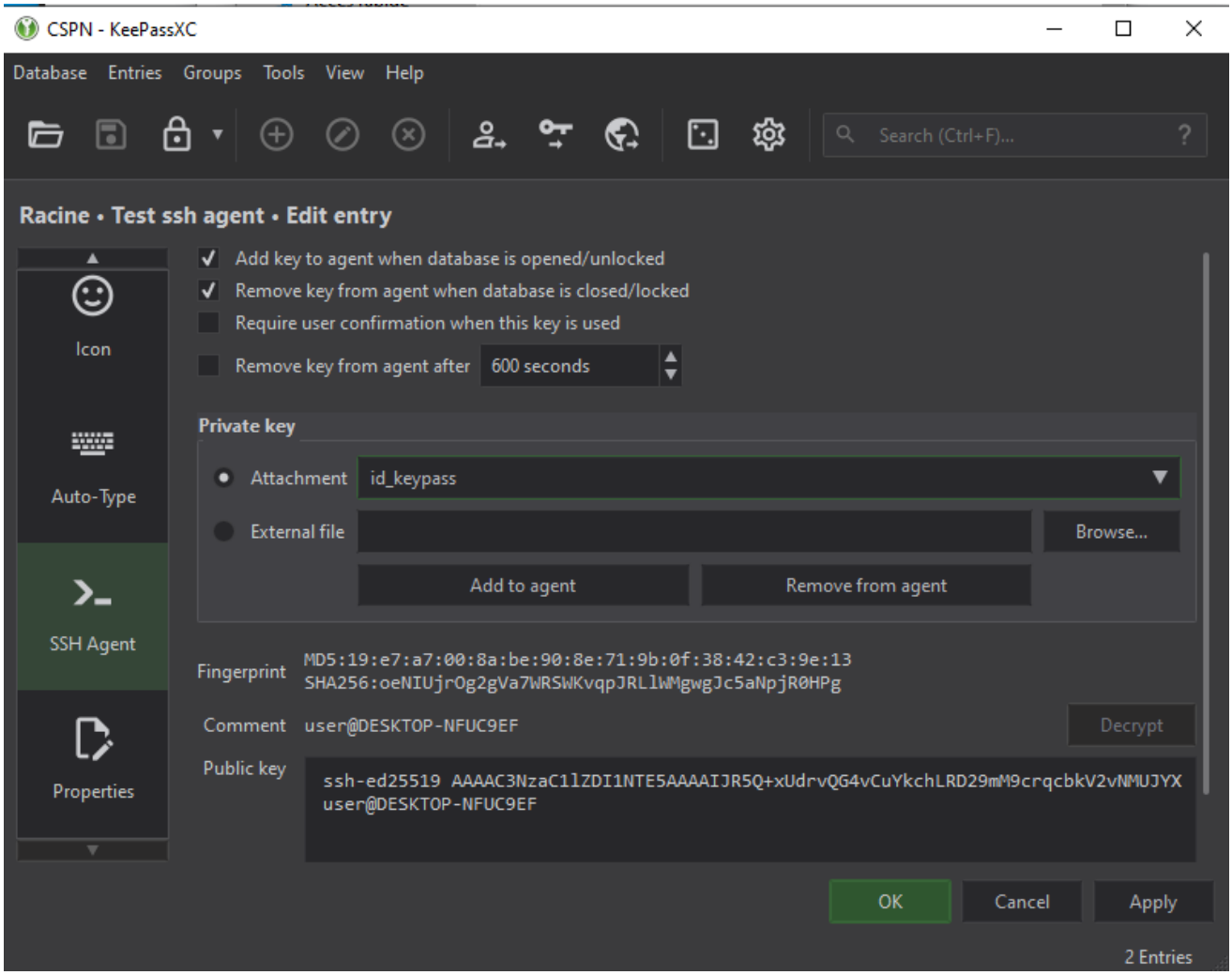


Illustration 32: KeePassXC ssh key option selection.

When an SSH key is added to an entry, and attachment is created. Attachment encryption is described in SF3: Database protection. When an entry `ssh-agent's` options are selected, another attachment called "KeeAgent.settings" is also added. This file contains an XML describing every selected options. For example, the following code contains the XML of the options selected on the picture above.

```
<?xml version="1.0" encoding="UTF-16"?>
<EntrySettings xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <AllowUseOfSshKey>true</AllowUseOfSshKey>
  <AddAtDatabaseOpen>true</AddAtDatabaseOpen>
  <RemoveAtDatabaseClose>true</RemoveAtDatabaseClose>
  <UseConfirmConstraintWhenAdding>false</UseConfirmConstraintWhenAdding>
  <UseLifetimeConstraintWhenAdding>false</UseLifetimeConstraintWhenAdding>
  <LifetimeConstraintDuration>600</LifetimeConstraintDuration>
  <Location>
    <SelectedType>attachment</SelectedType>
    <AttachmentName>id_keypass</AttachmentName>
```

```
    <SaveAttachmentToTempFile>false</SaveAttachmentToTempFile>
    <FileName/>
  </Location>
</EntrySettings>
```

If the private key is password protected, then the password should be inserted in the “password” field of the entry.

After the key has been loaded, the database must be locked and unlocked for the key to be inserted in the **`ssh-agent`**.

```
PS > ssh-add.exe -L
The agent has no identities.

PS > ssh-add.exe -L
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIJR5Q+xUdrvQG4vCuYkchLRD29mM9crqcbkV2vNMUJYX
user@DESKTOP-NFUC9EF
```

When the database is locked, the SSH key is successfully removed from the **`ssh-agent`**.

Code review

In the code, this behaviour is triggered by the call to **`DatabaseWidget::loadDatabase`**. When the database is unlocked, KeePassXC will iterate over every entry in the database to search for an entry that is not in the Recycle Bin folder and that has the option “Add key to agent when database is opened/unlocked”. If an entry is not configured with an ssh key, it has no **`Keepass.setting`** attachment, so the default value of **`settings.allowUseOfSshKey()`** is **`False`**.

```
// src/gui/DatabaseWidget.cpp:1172
void DatabaseWidget::loadDatabase(bool accepted)
{
//[...]
#ifdef WITH_XC_SSHAGENT
        sshAgent()->databaseUnlocked(m_db);
#endif


// src/sshagent/SSHAgent.cpp:511
void SSHAgent::databaseUnlocked(const QSharedPointer<Database>& db)
{
    if (!db || !isEnabled()) {
```

```
        return;
    }

    for (auto entry : db->rootGroup()->entriesRecursive()) {
        if (entry->isRecycled()) {
            continue;
        }

        KeeAgentSettings settings;

        if (!settings.fromEntry(entry)) {
            continue;
        }

        if (!settings.allowUseOfSshKey() || !settings.addAtDatabaseOpen()) {
            continue;
        }
```

When an entry verifying this condition is found, the attached private key is loaded by the method **`toOpenSSHKey`**.

```
// src/sshagent/SSHAgent.cpp:532
        OpenSSHKey key;

        if (!settings.toOpenSSHKey(entry, key, true)) {
            continue;
        }

        // Add key to agent; ignore errors if we have previously added the key
        bool known_key = m_addedKeys.contains(key);
        if (!addIdentity(key, settings, db->uuid()) && !known_key) {
            emit error(m_error);
        }
    }
}
```

The **`toOpenSSHKey`** function will parse the private key file format. The **`parsePKCS1PEM`** is called on the private key PEM data. It will first extract the PEM format in the **`extractPEM`** function. This function uses a regular expression to find the base64 payload of the key. Then, the PEM data is decoded from the base64. The internal structure of OpenSSH key is parsed inside the function **`parsePKCS1PEM`**. The parsing is done using **`QByteArray`** structures and calls to **`readString`**, which is a safe way to operate. Indeed, the length is checked inside the **`readString`** function before writing to **`QByteArray`**.

```
//src/sshagent/KeeAgentSettings.cpp:405
bool KeeAgentSettings::toOpenSSHKey(const Entry* entry, OpenSSHKey& key, bool decrypt)
{
    return toOpenSSHKey(
        entry->username(), entry->password(), entry->database()->filePath(), entry-
>attachments(), key, decrypt);
}


//src/sshagent/KeeAgentSettings.cpp:424
bool KeeAgentSettings::toOpenSSHKey(const QString& username,
                                    const QString& password,
                                    const QString& databasePath,
                                    const EntryAttachments* attachments,
                                    OpenSSHKey& key,
                                    bool decrypt)
{
    QString fileName;
    QByteArray privateKeyData;

    if (m_selectedType == "attachment") {
// [...]
        fileName = m_attachmentName;
        privateKeyData = attachments->value(fileName);
// [...]
  }
    if (privateKeyData.isEmpty()) {
        m_error = QCoreApplication::translate("KeeAgentSettings", "Private key is
empty");
        return false;
    }

    if (!key.parsePKCS1PEM(privateKeyData)) {
        m_error = key.errorString();
        return false;
    }

    if (key.encrypted() && (decrypt || key.publicKey().isEmpty())) {
        if (!key.openKey(password)) {
            m_error = key.errorString();
            return false;
        }
    }
// [...]
    return true;
}

// src/sshagent/OpenSSHKey.cpp:222
```

```
bool OpenSSHKey::parsePKCS1PEM(const QByteArray& in)
{
    QByteArray data;

    if (!extractPEM(in, data)) {
        return false;
    }
// [...]
    } else if (m_rawType == TYPE_OPENSSH_PRIVATE) {
        BinaryStream stream(&data);

        QByteArray magic;
        magic.resize(15);

        if (!stream.read(magic)) {
            m_error = tr("Key file way too small.");
            return false;
        }

        if (QString::fromLatin1(magic) != "openssh-key-v1") {
            m_error = tr("Key file magic header id invalid");
            return false;
        }

        stream.readString(m_cipherName);
        stream.readString(m_kdfName);
        stream.readString(m_kdfOptions);

        quint32 numberOfKeys;
        stream.read(numberOfKeys);

        if (numberOfKeys == 0) {
            m_error = tr("Found zero keys");
            return false;
        }

        for (quint32 i = 0; i < numberOfKeys; ++i) {
            QByteArray publicKey;
            if (!stream.readString(publicKey)) {
                m_error = tr("Failed to read public key.");
                return false;
            }

            if (i == 0) {
                BinaryStream publicStream(&publicKey);
                if (!readPublic(publicStream)) {
                    return false;
```

```
                }
            }
        }

        // padded list of keys
        if (!stream.readString(m_rawData)) {
            m_error = tr("Corrupted key file, reading private key failed");
            return false;
        }
    } else {
        m_error = tr("Unsupported key type: %1").arg(m_rawType);
        return false;
    }

    // load private if no encryption
    if (!encrypted()) {
        return openKey();
    } else {
        m_comment = tr("(encrypted)");
    }

    return true;
}

// src/sshagent/BinaryStream.cpp:113
bool BinaryStream::readString(QByteArray& ba)
{
    quint32 length;

    if (!read(length)) {
        return false;
    }

    ba.resize(length);

    if (!read(ba.data(), ba.length())) {
        return false;
    }

    return true;
}
```

If the key is not encrypted, it is directly loaded into the **`OpenSSHKey`** object with the **`openKey`** method.

If the key is encrypted, it is decrypted using the entry's password in the OpenSSHKey::`openKey(password)`. The `openKey` function can parse 3 types of SSH keys: RSA key, DSA key and OpenSSH key.

If the key is encrypted, it will decrypt it by deriving the OpenSSH encryption key from the password. This cryptographic part is out of the scope of the KeePassXC CSPN, but the parsing is an interesting surface. The parsing of the key file is done using structures that checks the length of the data parsed compared to the length written in the data header. It rightfully fails if the lengths differ.

```
// src/sshagent/OpenSSHKey.cpp:300

bool OpenSSHKey::openKey(const QString& passphrase)
{
    QScopedPointer<SymmetricCipher> cipher(new SymmetricCipher());
// [...]
    QByteArray rawData = m_rawData;

    if (m_cipherName != "none") {
  // [...]
        // Initialize the cipher using the processed key and iv data
        if (!cipher->init(cipherMode, SymmetricCipher::Decrypt, keyData, ivData)) {
            m_error = tr("Failed to initialize cipher: %1").arg(cipher->errorString());
            return false;
        }
        // Decrypt the raw data, we do not use finish because padding is handled
separately
        if (!cipher->process(rawData)) {
            m_error = tr("Decryption failed: %1").arg(cipher->errorString());
            return false;
        }
    }

    if (m_rawType == TYPE_DSA_PRIVATE) {
  // [...]
    } else if (m_rawType == TYPE_RSA_PRIVATE) {
  // [...]
    } else if (m_rawType == TYPE_OPENSSH_PRIVATE) {
        BinaryStream keyStream(&rawData);

        quint32 checkInt1;
        quint32 checkInt2;

        keyStream.read(checkInt1);
        keyStream.read(checkInt2);

        if (checkInt1 != checkInt2) {
            m_error = tr("Decryption failed, wrong passphrase?");
```

```
            return false;
        }

        return readPrivate(keyStream);
    }

    m_error = tr("Unsupported key type: %1").arg(m_rawType);
    return false;
}
```

After all the parsing is done, if **sshKey** is a valid key, it is transmitted to the ssh agent.

The **addIdentity** method firstly checks that the ssh agent is running. To ensure that the SSH agent service is running, it checks whether the socket to the agent exists.

```
// src/sshagent/SSHAgent.cpp:271
bool SSHAgent::addIdentity(OpenSSHKey& key, const KeeAgentSettings& settings, const
QUuid& databaseUuid)
{
    if (!isAgentRunning()) {
        m_error = tr("No agent running, cannot add identity.");
        return false;
    }

// src/sshagent/SSHAgent.cpp:142
 bool SSHAgent::isAgentRunning() const
{
#ifndef Q_OS_WIN
    QFileInfo socketFileInfo(socketPath());
    return !socketFileInfo.path().isEmpty() && socketFileInfo.exists();
// [...]
}
```

The socket path depends on the operating system. Ffor windows the **SSH_AUTH_SOCK** environment variable points to the socket connected to the default **ssh-agent** of the system.

```
// src/sshagent/SSHAgent.cpp:101
QString SSHAgent::socketPath(bool allowOverride) const
{
    QString socketPath;

#ifndef Q_OS_WIN
    if (allowOverride) {
```

```
        socketPath = authSockOverride();
    }

    // if the overridden path is empty (no override set), default to environment
    if (socketPath.isEmpty()) {
        socketPath = QProcessEnvironment::systemEnvironment().value("SSH_AUTH_SOCK");
    }
#else
    Q_UNUSED(allowOverride)
    socketPath = "\\\\.\\pipe\\openssh-ssh-agent";
#endif

    return socketPath;
}
```

The message to the **ssh-agent** contains all the options that were provided in the **KeeAgent.settings** file such as the lifetime for example. It also contains the SSH private key that should be added to the agent. The request built within this function is then sent to the **ssh-agent**.

```
// src/sshagent/SSHAgent.cpp:283

    QByteArray requestData;
    BinaryStream request(&requestData);
    bool isSecurityKey = key.type().startsWith("sk-");

    request.write(
        (settings.useLifetimeConstraintWhenAdding() ||
settings.useConfirmConstraintWhenAdding() || isSecurityKey)
            ? SSH_AGENTC_ADD_ID_CONSTRAINED
            : SSH_AGENTC_ADD_IDENTITY);
    key.writePrivate(request);

    if (settings.useLifetimeConstraintWhenAdding()) {
        request.write(SSH_AGENT_CONSTRAIN_LIFETIME);
        request.write(static_cast<quint32>(settings.lifetimeConstraintDuration()));
    }

    if (settings.useConfirmConstraintWhenAdding()) {
        request.write(SSH_AGENT_CONSTRAIN_CONFIRM);
    }

    if (isSecurityKey) {
        request.write(SSH_AGENT_CONSTRAIN_EXTENSION);
        request.writeString(QString("sk-provider@openssh.com"));
        request.writeString(securityKeyProvider());
```

```
    }

    QByteArray responseData;
    if (!sendMessage(requestData, responseData)) {
        return false;
    }
```

The **sendMessage** method wraps the specificity of each **ssh_agent**. For OpenSSH, it will connect to the socket and send the stream built previously.

The following screenshot shows the structure of the data sent to the ssh agent. The bottom part of the image contains the data written to the socket in the **sendMessageOpenSSH** function.

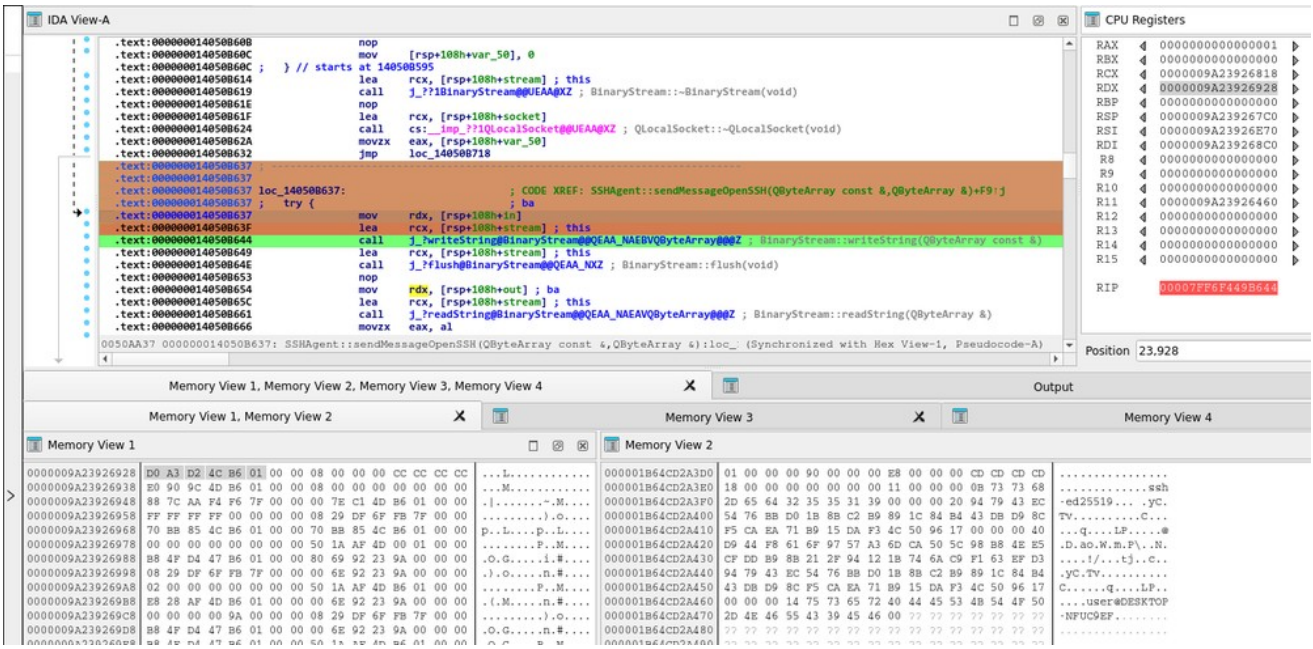


Illustration 33: Message sent to the OpenSSH agent.

```
// src/sshagent/SSHAgent.cpp:160
bool SSHAgent::sendMessage(const QByteArray& in, QByteArray& out)
{
#ifdef Q_OS_WIN
    if (usePageant() && !sendMessagePageant(in, out)) {
        return false;
    }
    if (useOpenSSH() && !sendMessageOpenSSH(in, out)) {
        return false;
```

```
    }
    return true;
#else
    return sendMessageOpenSSH(in, out);
#endif
}

// src/sshagent/SSHAgent.cpp:175
bool SSHAgent::sendMessageOpenSSH(const QByteArray& in, QByteArray& out)
{
    QLocalSocket socket;
    BinaryStream stream(&socket);

    socket.connectToServer(socketPath());
    if (!socket.waitForConnected(500)) {
        m_error = tr("Agent connection failed.");
        return false;
    }

    stream.writeString(in);
    stream.flush();

    if (!stream.readString(out)) {
        m_error = tr("Agent protocol error.");
        return false;
    }

    socket.close();
    return true;
}
```

After checking the return code of the ssh agent, if everything went smoothly, the **`privateKey`** is deleted from the memory and the **`publicKey`** is stored in a global array **`m_addedKeys`**. The settings regarding the removal at the database closing is also stored in the array.

```
// src/sshagent/SSHAgent.cpp:333
    OpenSSHKey keyCopy = key;
    keyCopy.clearPrivate();
    m_addedKeys[keyCopy] = qMakePair(databaseUuid, settings.removeAtDatabaseClose());
    return true;
}
```

The **`m_addedKeys`** array will be used in the **`databaseLocked`**, in order to remove all the identities that need to be removed on database lock.

The `removeIdentity` function will send the remove request to the `sshAgent` with the `sendMessage` function studied earlier.

```
// src/sshagent/SSHAgent.cpp:489
void SSHAgent::databaseLocked(const QSharedPointer<Database>& db)
{
    if (!db) {
        return;
    }

    auto it = m_addedKeys.begin();
    while (it != m_addedKeys.end()) {
        if (it.value().first != db->uuid()) {
            ++it;
            continue;
        }
        OpenSSHKey key = it.key();
        if (it.value().second) {
            if (!removeIdentity(key)) {
                emit error(m_error);
            }
        }
        it = m_addedKeys.erase(it);
    }
}

// src/sshagent/SSHAgent.cpp:345
bool SSHAgent::removeIdentity(OpenSSHKey& key)
{
    if (!isAgentRunning()) {
        m_error = tr("No agent running, cannot remove identity.");
        return false;
    }

    QByteArray requestData;
    BinaryStream request(&requestData);

    QByteArray keyData;
    BinaryStream keyStream(&keyData);
    key.writePublic(keyStream);

    request.write(SSH_AGENTC_REMOVE_IDENTITY);
    request.writeString(keyData);

    QByteArray responseData;
    return sendMessage(requestData, responseData);
}
```

Accessing attachment from KeePassXC

When a user opens an attachment from the entry view in KeePassXC, a temporary file is created on the filesystem as shown in the screenshot.

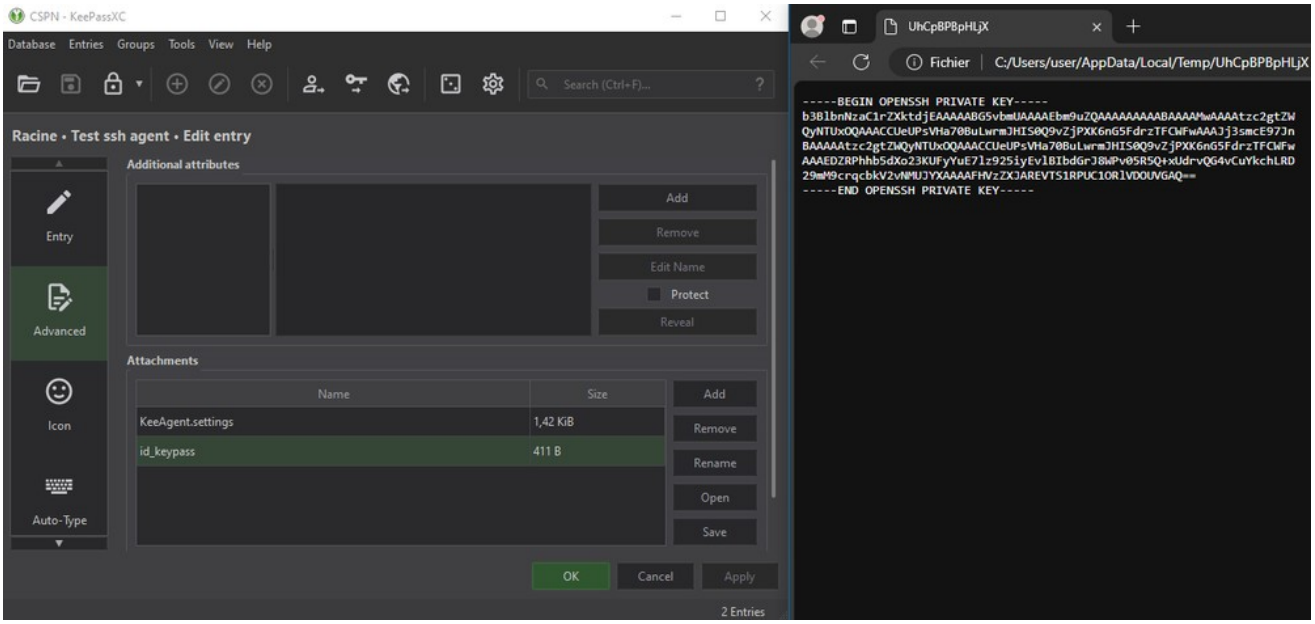


Illustration 34: Temporary file creation when opening an attachment.

The file stays until the database is locked, then it is deleted. Before the deletion, the file is overwritten with random values.

```
// src/core/EntryAttachments.cpp:158
void EntryAttachments::disconnectAndEraseExternalFile(const QString& path)
{
   // [..]
    QFile f(path);
    if (f.open(QFile::ReadWrite)) {
        qint64 blocks = f.size() / 128 + 1;
        for (qint64 i = 0; i < blocks; ++i) {
            f.write(randomGen()->randomArray(128));
        }
        f.close();
    }
    f.remove();
}
```

To verify the efficiency of this measure, the private key attachment was added the `.txt` extension, saved to the disk by KeePassXC. Then, after locking the database, the forensic tool, PhotoRec version 7.1 (https://www.cgsecurity.org/wiki/photoRec), was run in order to try to retrieve the private key, but it did not manage to retrieve it, confirming the correct shredding of the key.

Conclusion

The communication with the SSH socket is not encrypted, and KeePassXC does interact with the socket as securely as possible. Indeed, the ssh-agent does not provide any mechanism to encrypt these IPC.

However, KeePassXC does protect the ssh key files by storing them encrypted in the database. The private keys are decrypted in-memory and never lies unencrypted on the hard drive.

Cryptographic validation of used algorithms

KeePassXC relies on the Botan C++ implementation https://botan.randombit.net/. The source code is available, as well as some tests on GitHub: https://github.com/randombit/botan/tree/master/src/tests. These tests also include cryptographic test vectors. Synacktiv used these tests to validate the implementation of different algorithms used by KeePassXC.

In some cases, described below, some test vectors were added to the main test routine to validate the corresponding algorithm.

The KeePassXC program downloaded from the official website is shipped with the Botan dynamic library: **`botan-3.dll`**.

The used DLL has the following SHA-256 digest:

```
C:\[...]\botan> certutil.exe -hashfile 'C:\Program Files\KeePassXC\\botan-3.dll' SHA256
SHA256 hash of C:\Program Files\KeePassXC\\botan-3.dll:
274f811adb29c820dfe2cd96aad8e02dc8e84366fafd59821432028bef85cbce
CertUtil: -hashfile command completed successfully.
```

To build the Botan test program, the source code of Botan were cloned on a Windows machine:

```
C:\> git clone https://github.com/randombit/botan
```

Then, the Git repository was checked out to the exact version used by KeePassXC in the 2.7.9 version: 3.1.1, visible in the **`vcpkg.json`** file in the KeePassXC repository:

```
{
  "name": "keepassxc",
  "version-string": "2.8.0",
  "builtin-baseline": "2a6371b01420d8820d158b4707e79931feba27aa",
  "dependencies": [
    {
      "name": "argon2",
      "version>=": "20190702"
    },
    {
      "name": "botan",
      "version>=": "3.1.1"
    },
C:\> git checkout tags/3.1.1
```

```
C:\> python.exe configure.py
C:\> nmake
```

Then, the library was built using the provided script to create `botan-test.exe`. This program was used with the `botan-3.dll` shipped by KeePassXC to import the cryptographic functions and run its tests, as described above.

Key derivation with Argon2

Argon2 has been standardized by the IETF in RFC9106 in 2021.

```
C:\[...]\botan> botan-test.exe argon2 argon2_pass
Testing Botan 3.1.1 (unreleased, revision unknown, distribution vcpkg x64-windows)
Properties:
  CPU flags: rdtsc sse2
  drbg_seed: 0000AB8B92056AD0
argon2:
Argon2d ran 12 tests in 93.32 msec all ok
Argon2i ran 15 tests in 161.77 msec all ok
Argon2id ran 330 tests in 174.81 msec all ok
argon2_pass:
Argon2 password hash ran 9 tests in 136.80 msec all ok
Tests complete ran 366 tests in 610.33 msec all tests ok
```

The test cases include the test vector from the RFC:

```
// src/tests/data/argon2.vec:5
[Argon2d]

M = 32
T = 3
P = 4
Passphrase = 0101010101010101010101010101010101010101010101010101010101010101
Salt = 02020202020202020202020202020202
AD = 040404040404040404040404
Secret = 0303030303030303
Output = 512b391b6f1162975371d30919734294f868e3be3984f3c1a13a4db9fabe4acb
```

The Security Recommendations [ANSSI-PG-083] states that key derivation algorithm should be used in centralized systems, which is the case for KeePassXC. The storage of the master key is not handled by KeePassXC and the sole user is responsible for the passphrase/key file.

SHA-256 and SHA-512

KeepassXC uses SHA256 to verify the header integrity and SHA512 in order to compute the derivedkey to decrypt the database. The implementation relies on the Botan library. The unit tests have been run using the following command line:

```
> .\\botan-test.exe hash_algos
```

The test vectors are provided in the files :

- src/tests/data/hash_mc.vec containing the NIST Monte Carlo test vectors,
- src/tests/data/hash_rep.vec containing the test vectors from the DI-Management,
- src/tests/data/hash/truncated.vec containing truncated hash test vectors,
- src/tests/data/hash/sha3.vec containing the NIST CAVS 19.0 test vectors

The following code snippets displays the output of the tests filtered to display only SHA512 and SHA256 results. All the tests passed successfully.

```
Parallel(SHA-256,SHA-512) ran 19 tests in 0.03 msec all ok
SHA-256 ran 14025 tests in 8.11 msec all ok
SHA-3(256) ran 2398 tests in 5.91 msec all ok
SHA-3(512) ran 2398 tests in 5.98 msec all ok
SHA-512 ran 3278 tests in 3.12 msec all ok
SHA-512-256 ran 42 tests in 0.04 msec all ok
Truncated(SHA-256,1) ran 9 tests in 0.02 msec all ok
Truncated(SHA-256,11) ran 9 tests in 0.01 msec all ok
Truncated(SHA-256,15) ran 9 tests in 0.01 msec all ok
Truncated(SHA-256,16) ran 9 tests in 0.01 msec all ok
Truncated(SHA-256,256) ran 9 tests in 0.01 msec all ok
Truncated(SHA-256,9) ran 9 tests in 0.01 msec all ok
NIST Monte Carlo SHA-512 ran 1 tests in 50.56 msec all ok
NIST Monte Carlo SHA-512-256 ran 1 tests in 27.94 msec all ok
NIST Monte Carlo SHA-256 ran 1 tests in 9.46 msec all ok
Long input SHA-256 ran 1 tests in 0.48 msec all ok
Long input SHA-3(256) ran 1 tests in 2.55 msec all ok
Long input SHA-3(512) ran 1 tests in 5.47 msec all ok
Long input SHA-512 ran 1 tests in 1.79 msec all ok
// ...
Tests complete ran 79006 tests in 325.10 msec all tests ok
```

SHA-256-HMAC

The KeePassXC database header is protected with a SHA256-HMAC. Moreover, each block of 1 MB has their own SHA256-HMAC as well.

For the header, the HMAC value corresponds to:

```
hmac_header = MAC_SHA256(headerHmacKey)(headerData)
```

For every block *i*, the HMAC value corresponds to:

```
hmac_block_i = MAC_SHA256(blockHmacKey_i)(block_index_i . block_size_i .
encrypted_block_i)
```

- The **block_index_i** is equal to 0 for the first block and incremented for each following block. This parameter is necessary in order to avoid the reordering of blocks being unnoticed.

The headerHmacKey is equal to:

```
headerHmacKey = SHA512(0xFFFFFFFFFFFFFFFF . SHA512(masterSeed . derivedKey . "\x01"))
```

And for each block i, the blockHmacKey_i is equal to :

```
blockHmacKey_i = SHA512(block_index_i . SHA512(masterSeed . derivedKey . "\x01"))
```

There is a case where the blockHmacKey_i is equal to the headerHmacKey: if the KeepassXC has 0xFFFFFFFFFFFFFFFF blocks. This would mean that the database file is at least 18446 petabytes, which is highly unlikely and would make the database unusable anyway.

The HMAC_SHA256 is implemented using the **Botan::MessageAuthenticationCode** class. The test vectors for this feature are located in the file **botan/src/tests/data/mac/hmac.vec**, line 82. Some test vectors described in the RFC were not present in the file, so the missing one were added for the purpose of this test:

```
Key = 4a6566654a6566654a6566654a6566654a6566654a6566654a6566654a656665
In = 7768617420646F2079612077616E7420666F72206E6F7468696E673F
Out = 167f928588c5cc2eef8e3093caa0e87c9ff566a14794aa61648d81621a2a40c6

Key = aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
In =
dddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
dddddddddddd
Out = cdcb1220d1ecccea91e53aba3092f962e549fe6ce9ed7fdc43191fbde45c30b0
```

```
Key = 0102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f20
In =
cdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcd
dcdcdcdcdcdcd
Out = 372efcf9b40b35c2115b1346903d2ef42fced46f0846e7257bb156d3d7b30d3f
```

The tests were then performed using the process described above:

```
C:\[...]\botan> botan-test.exe mac
Testing Botan 3.1.1 (unreleased, revision unknown, distribution vcpkg x64-windows)
Properties:
 CPU flags: rdtsc sse2
 drbg_seed: 0000ABCB277F7F6C
mac_algos:
[...]
HMAC(SHA-256) ran 180 tests in 0.91 msec all ok
[...]
```

All the tests ran successfully.

Rule	Compliant	Rationale
RègleHash1 RègleHash2	Yes	SHA256-HMAC outputs 256-bit hashes. Argon2d outputs 256-bit hashes.
RecommandationHash-1	Yes	No partial attacks are known for SHA256-HMAC and Argon2d.

Symmetric cryptography

The data inside the KBDX file is encrypted using the AES256 algorithm with the CBC mode.

The passphrase and/or key file provided by the used are derived using the `Argon2d` key derivation function, so the decryption key is 32 bytes long, and the bytes are pseudo-random thanks to the properties of the KDF function.

For this mode, only one test vector is specified:

```
// botan/src/tests/data/modes/cbc.vec
[AES-256/CBC/NoPadding]
Key = 603DEB1015CA71BE2B73AEF0857D77811F352C073B6108D72D9810A30914DFF4
```

```
Nonce = 000102030405060708090A0B0C0D0E0F
In =
6BC1BEE22E409F96E93D7E117393172AAE2D8A571E03AC9C9EB76FAC45AF8E5130C81C46A35CE411E5FBC11
91A0A52EFF69F2445DF4F9B17AD2B417BE66C3710
Out =
F58C4C04D6E5F1BA779EABFB5F7BFBD69CFC4E967EDB808D679F777BC6702C7D39F23369A9D9BACFA530E26
304231461B2EB05E2C39BE9FCDA6C19078C6A9D1B
```

The following test vectors from the NIST were added for the purpose of the tests, and everything passed correctly.

```
Key = 603DEB1015CA71BE2B73AEF0857D77811F352C073B6108D72D9810A30914DFF4
Nonce = F58C4C04D6E5F1BA779EABFB5F7BFBD6
In = AE2D8A571E03AC9C9EB76FAC45AF8E51
Out = 9CFC4E967EDB808D679F777BC6702C7D

Key = 603DEB1015CA71BE2B73AEF0857D77811F352C073B6108D72D9810A30914DFF4
Nonce = 9CFC4E967EDB808D679F777BC6702C7D
In = 30C81C46A35CE411E5FBC1191A0A52EF
Out = 39F23369A9D9BACFA530E26304231461

Key = 603DEB1015CA71BE2B73AEF0857D77811F352C073B6108D72D9810A30914DFF4
Nonce = 39F23369A9D9BACFA530E26304231461
In = F69F2445DF4F9B17AD2B417BE66C3710
Out = B2EB05E2C39BE9FCDA6C19078C6A9D1B
```

The trace of the execution of the Botan tests is shown below.

```
C:\[...]\botan> botan-test.exe cipher_modes
Testing Botan 3.1.1 (unreleased, revision unknown, distribution vcpkg x64-windows)
Properties:
  CPU flags: rdtsc sse2
  drbg_seed: 0000AC2886488AE0
cipher_modes:
[...]
AES-256/CBC/NoPadding ran 173 tests in 1.62 msec all ok
[...]
```

The AES key is 256 bits long, the block size is 128 bits and the which is compliant with the Security Recommendations [RGS_B1], page 10.

The padding to be used with CBC is the Botan default padding. Referring to Botan documentation, the padding can be specified in the first parameter of `Botan::Cipher_Mode::create_or_throw`. In the

default case, only the string **"AES-256/CBC"** is provided, thus the default padding of Botan will be used, which is **PKCS7** according to the source:

```
// botan/src/lib/modes/cipher_mode.cpp:117
  #if defined(BOTAN_HAS_MODE_CBC)
  if(spec.algo_name() == "CBC") {
     const std::string padding = spec.arg(1, "PKCS7");
```

The [ANSSI-PG-083] does not provide any recommendation regarding the padding algorithm to use, but PKCS7 is a commonly used padding.

Chacha20

KeePassXC protects the password inside the encrypted XML with a stream cipher. For the latest version, KDBX4, Chacha20 is used as a stream cipher.

The stream cipher key is stored in the **innerHeader** of the block containing the password. This header is encrypted, like the data, by the symmetric cipher. It is generated when the database is created using the random generator **Botan::RandomNumberGenerator**.

```
// src/format/Kdbx4Writer.cpp:52
QByteArray protectedStreamKey = randomGen()->randomArray(64);

// src/crypto/Random.cpp:55
QByteArray Random::randomArray(int len)
{
    QByteArray ba(len, '\0');
    randomize(ba);
    return ba;
}

// src/crypto/Random.cpp:50
void Random::randomize(QByteArray& ba)
{
    m_rng->randomize(reinterpret_cast<uint8_t*>(ba.data()), ba.size());
}

// src/crypto/Random.h:50
QSharedPointer<Botan::RandomNumberGenerator> m_rng;
```

The encryption and decryption routines also use the Botan implementation, abstracted by the **SymmetricCipher** class in file **src/crypto/SymmetricCipher.cpp**.

The test vectors are in the file **botan/src/tests/test_sodium.cpp** line 709. The tests include one test vector from the RFC, and the other one's origin is not specified. Both of the tests runs correctly:

```
C:\[...]\botan> botan-test.exe sodium
Testing Botan 3.1.1 (unreleased, revision unknown, distribution vcpkg x64-windows)
Properties:
 CPU flags: rdtsc sse2
 drbg_seed: 0000ACA27B64E0A4
sodium:
[...]
crypto_stream_chacha20 ran 4 tests all ok
[...]
```

The stream cipher key for the hashed password correspond to the first 32 bytes of the **SHA512(StreamKey_in_innerheader)**, and the nonce is the following 12 bytes. The size of the parameters complies to the RFC7539.

The Security Recommendations [ANSSI-PG-083] does not encourage the use of stream ciphers. However, in this case, the stream cipher encrypts data that are then encrypted again by a block cipher. Therefore, its use here is compliant as the whole confidentiality of the data does not rely solely on the stream cipher.

Rule	Compliant	Rationale
RègleCléSym RecommandationCléSym	Yes	The encryption key sizes for AES256 and CHACHA20 are 32 bytes.
RègleBlocSym1 RègleBlocSym2 RecommandationBlocSym	Yes	AES256 uses 128-bits blocs. The key is renewed at each database save, which lessen the probability of using the key more than 2^{125} times.
RègleAlgoBloc RecommandationAlgoBloc	Yes	No attacks are known for AES-256
RègleModeChiff RecommandationModeChiff	Yes	No attacks are known for AES-256 CBC
RègleChiffFlot RecommandationChiffFlot	Yes	No attacks are known for CHACHA20

Sodium's cryptobox

The interaction between the KeePassXC program and the keepassxc-browser extension is encrypted using Sodium's cryptobox. This mechanism relies on the following algorithms:

- X25519 for the key exchange, it is an ECDH algorithm.
- HSalsa20 for the KDF function used to derive the symmetric key from the X25519 secret.
- XSalsa20 for the encryption algorithm, it is a stream cipher initialized by a 192 bits nonce.
- Poly1305 for the MAC algorithm.

These algorithms put together ensure perfect forward secrecy, confidentiality, and authenticity of the exchanged data.

The botan-test.exe program also offers to test cryptobox implementations:

```
C:\[...]\botan> botan-test.exe sodium
Testing Botan 3.1.1 (unreleased, revision unknown, distribution vcpkg x64-windows)
Properties:
 CPU flags: rdtsc sse2
 drbg_seed: 0000055BB02CB274
sodium:
[...]
crypto_box_curve25519xsalsa20poly1305 ran 14 tests all ok
[...]
```

However, the curve25519 is compliant with the Security Recommendations [ANSSI-PG-083] due to its subgroup being closer to 252 bits and thus higher than 250 bits. The other algorithms are compliant.

Rule	Compliant	Rationale
RègleECp RecommandationECp	Yes	X25519 has a order of 252 bits, the order is prime number and is based on the discreet logarithm but defined on GF(2^{255-19}).
RègleHash RecommandationHash	Yes	HSalsa20 outputs 256 bits hashes. No partial attacks are known for this algorithm.
RègleChiffFlot RecommandationChiffFlot	Yes	No attacks are known for XSalsa20.
RègleIntegSym		No attacks are known for Poly1305.

Random generation

Random generators are used to create the **masterSeed**, **encryptionIV** and **protectedStreamKey**.

A broken RNG would weaken the key derivation function, so the confidentiality of the database would be compromised.

```
//src/format/Kdbx4Writer.cpp:50
QByteArray masterSeed = randomGen()->randomArray(32);
QByteArray encryptionIV = randomGen()->randomArray(ivSize);
QByteArray protectedStreamKey = randomGen()->randomArray(64);
```

KeepassXC uses the Botan random generator, either **Botan::System_RNG** or **Botan::Autoseeded_RNG**.

```
//src/crypto/Random.h:53
static inline QSharedPointer<Random> randomGen()
{
    return Random::instance();
}

//src/crypto/Random.cpp:26
QSharedPointer<Random> Random::m_instance;

QSharedPointer<Random> Random::instance()
{
    if (!m_instance) {
        m_instance.reset(new Random());
    }
    return m_instance;
}

Random::Random()
{
#ifdef BOTAN_HAS_SYSTEM_RNG
    m_rng.reset(new Botan::System_RNG);
#else
    m_rng.reset(new Botan::Autoseeded_RNG);
#endif
}
```

According to Botan source code and documentation, the **System_RNG** accesses the process global instance of the system PRNG (using interfaces such as **/dev/urandom**, **getrandom**, **arc4random**, **BCryptGenRandom**, or **RtlGenRandom**). If the latter is not available, the **Autoseeded_RNG** is the 'best available' alternative. It uses HMAC_DRBG with either SHA-384 or SHA-256. The initial seed is

generated either by the system PRNG (if available) or a default set of entropy sources. The following entropy sources are currently used:

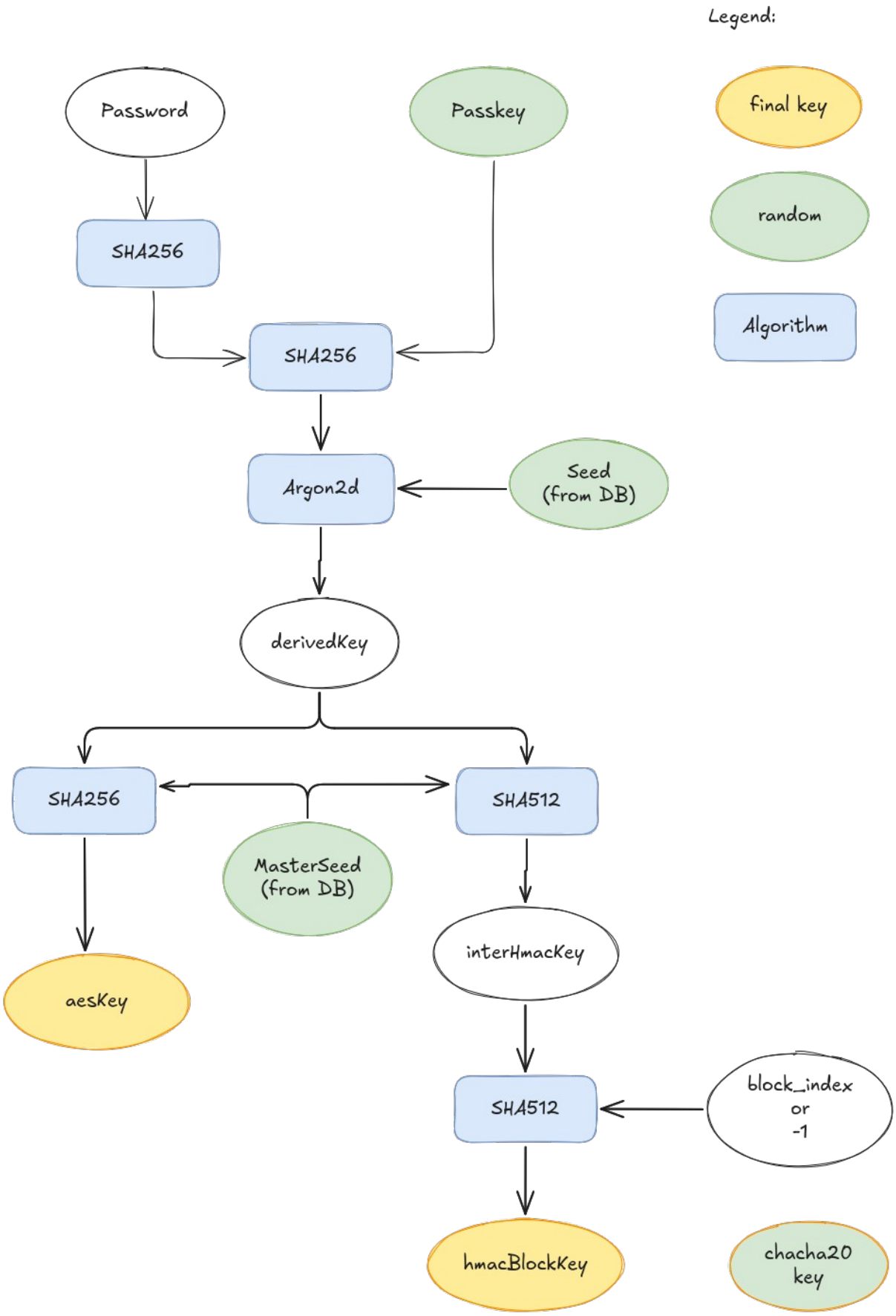
- The system RNG (`/dev/urandom`, `getrandom`, `arc4random`, `BCryptGenRandom`, or `RtlGenRandom`).
- Processor provided RNG outputs (RDRAND, RDSEED, DARN) are used if available, but not counted as contributing entropy
- The `getentropy` call is used on OpenBSD, FreeBSD, and macOS
- `/proc` walk: read files in `/proc`. Last ditch protection against flawed system RNG.
- Win32 stats: takes snapshot of current system processes. Last ditch protection against flawed system RNG.

HMAC-DRBG is a random number generator designed by NIST and specified in SP 800-90A.

Summary

Algorithm	Usage	Parameters sizes	Compliant with [ANSSI-PG-083]
HMAC-SHA512	Header integrity & authenticity verification	Key of 512 bits	Yes
SHA256	Header integrity verification	/	Yes
Argon2d	Key derivation function		Yes
SHA512	HMAC key computation	/	Yes
AES256-CBC	Database encryption	Key of 128 bits Derived with a KDF	Yes
Chacha20	Password encryption in the decrypted database	Key of 16 bytes Nonce of 12 bytes	Yes
curve25519xsalsa20poly1305	Interaction with the keepassxc-browser extension.	Key of 32 bytes Nonce of 24 bytes	Yes

The following schematics represents the keys dependancies to generate the AESKey (used for database encryption), the hmacBlockKey (used for database integrity of each block of 1M) and Chacha20 Key (used for obfuscation).



SYNACKTIV

Expert opinion and identified potential vulnerabilities

Overall, all the security features were compliant with the security target as well as with security best practices. The only identified issue is outside the scope of the security target as it concerns the content of KeePassXC process memory which is not accessible from the selected threat agents. It has been observed that when a database's passphrase is pasted in the corresponding text field instead of being manually typed, its value lingers in memory even after the database is locked or closed.

Identification of generic vulnerabilities

Standards used for the analysis

The rules and recommendations of [ANSSI-PG-083] were used as a reference when auditing the cryptographic mechanisms involved in the product.

Since the TOE is developed in C++, the vulnerability research was focused on:

- Usual vulnerabilities affecting programs coded in C++:
  - Out-of-bounds memory read or write (stack/heap based buffer overflow/underflow, etc.).
  - Integer underflow and overflow.
  - Bad handling of the heap (use-after-free, double free, use-after-realloc, etc.).
  - Etc.
- Logic vulnerabilities bounded to the security features
  - Passphrase verification
  - Passphrase or password leak
  - Etc.
- Vulnerabilities on the implementation of the cryptographic mechanisms
  - Missing parameters validation
  - Wrong implementation
- Misusage of external libraries, in particular when dealing with user inputs, such as the database XML documents, or key files.

Expert opinion and identified potential vulnerabilities

No vulnerability was identified during this analysis.

Vulnerabilities analysis

V-01: Database's master password lingers in memory when pasted

As part of the SF4 analysis regarding the protection of data present within the process memory, Synacktiv experts looked for sensitive data such as passphrases (master passwords) and passwords stored in memory. KeePassXC documentation already states that entries' password are stored in plaintext in the memory and therefore implement different counter-measure to prevent the unprivileged users to retrieve the process memory. Regarding passphrase, it was confirmed by studying the source code that KeePassXC discarded it when cryptographic keys are derived. It was also confirmed by looking for the passphrase in a memory dump of the KeePassXC process. However, when the passphrase is pasted in the corresponding text field, it was observed that in this case the passphrase remains findable in a memory dump even after the corresponding database was closed or locked.

Vulnerability analysis

In the example below, a memory dump of the KeePassXC process was performed, and the passphrase searched inside it:

```
PS Z:\dumps> C:\Users\user\Desktop\Tools\SysinternalsSuite\procdump.exe -ma
keepassxc.exe
ProcDump v11.0 - Sysinternals process dump utility
Copyright (C) 2009-2022 Mark Russinovich and Andrew Richards
Sysinternals - www.sysinternals.com
[16:11:00] Dump 1 initiated: Z:\dumps\KeePassXC.exe_241115_161100.dmp
[16:11:00] Dump 1 writing: Estimated dump file size is 325 MB.
[16:11:01] Dump 1 complete: 325 MB written in 1.0 seconds
[16:11:01] Dump count reached.
$ strings -e l KeePassXC.exe_241115_161100.dmp | grep -P
'vohthaehohseth2ohh4geNgaagoac8th'
vohthaehohseth2ohh4geNgaagoac8th
```

Using WinDBG, it was observed that passphrase was present in the heap part of the KeePassXC process:

```
0:000> s -u 000001661E223350 000002661E223350 L?800000000000
"vohthaehohseth2ohh4geNgaagoac8th"
00000266`1db2f100  0076 006f 0068 0074 0068 0061 0065 0068  v.o.h.t.h.a.e.h.
0:000> d 00000266`1db2f100
00000266`1db2f100  76 00 6f 00 68 00 74 00-68 00 61 00 65 00 68 00  v.o.h.t.h.a.e.h.
00000266`1db2f110  6f 00 68 00 73 00 65 00-74 00 68 00 32 00 6f 00  o.h.s.e.t.h.2.o.
```

```
00000266`1db2f120  68 00 68 00 34 00 67 00-65 00 4e 00 67 00 61 00  h.h.4.g.e.N.g.a.
00000266`1db2f130  61 00 67 00 6f 00 61 00-63 00 38 00 74 00 68 00  a.g.o.a.c.8.t.h.
00000266`1db2f140  00 00 dd dd dd dd dd dd-81 e8 cb 23 00 b0 02 8c  ...........#....
00000266`1db2f150  10 8b 2c 1d 66 02 00 00-60 1f 16 1e 66 02 00 00  ..,.f...`...f...
00000266`1db2f160  00 00 00 00 00 00 00 00-00 00 00 00 01 00 00 00  ................
00000266`1db2f170  10 00 00 00 00 00 00 00-a7 f7 0d 00 fd fd fd fd  ................
```

Nevertheless, if the password is manually entered, it is not retrievable from a memory dump. This behaviour can therefore be exploited in a case where shared databases are used, and the corresponding passphrases are stored in a personal one.

For example, given a database with the passphrase **`vohthaehohseth2ohh4geNgaagoac8th`**, entered manually, containing the passphrase of another database (**`ieZoox0toh4mohlaegaethaPh1airooc`**). When this second passphrase is copied and pasted to open the second database, it is still present in memory even though the database is locked or completely closed:

```
0:012> s -u 0 L?80000000000 ieZoox0toh4mohlaegaethaPh1airooc
00000205`10793c68  0069 0065 005a 006f 006f 0078 0030 0074  i.e.Z.o.o.x.0.t.
0:012> d 00000205`10793c68
00000205`10793c68  69 00 65 00 5a 00 6f 00-6f 00 78 00 30 00 74 00  i.e.Z.o.o.x.0.t.
00000205`10793c78  6f 00 68 00 34 00 6d 00-6f 00 68 00 6c 00 61 00  o.h.4.m.o.h.l.a.
00000205`10793c88  65 00 67 00 61 00 65 00-74 00 68 00 61 00 50 00  e.g.a.e.t.h.a.P.
00000205`10793c98  68 00 31 00 61 00 69 00-72 00 6f 00 6f 00 63 00  h.1.a.i.r.o.o.c.
00000205`10793ca8  00 00 fd fd fd fd ab ab-ab ab ab ab ab ab ab ab  ................
00000205`10793cb8  ab ab ab ab ab ab ee fe-00 00 00 00 00 00 00 00  ................
00000205`10793cc8  00 00 00 00 00 00 00 00-ee fe ee fe ee fe ee fe  ................
00000205`10793cd8  2e 3c 31 5e 47 66 0c 34-20 5b 0b 0e 05 02 00 00  .<1^Gf.4 [......
```

This behaviour cannot be explained from KeePassXC source code and seems to come from a side effect of the Qt library handling the clipboard event. The content of the clipboard being stored somewhere in the memory and not accessible from KeePassXC source code that merely retrieve the passphrase through the text field content.

Synacktiv experts confirmed that this behaviour was also reproducible using the Linux build of KeePassXC.

Conclusion

As stated earlier, this vulnerability is outside the scope of this assessment as it requires a threat agent able to retrieve the content of the KeePassXC process memory. The analysis of the SF4: Memory protection page 59 confirmed that only users with administrators' privileges on the Windows system would be able to retrieve it. Then, to be exploited, it also requires for the victim to paste the

passphrase to open the corresponding database which is unlikely for personal databases. Nonetheless, for shared databases this prerequisite is more likely. And even though, the shared database's passphrase would already be found in the process memory due to existing as a password entry in the personal database, this vulnerability exposes it after this database is locked or closed.

Summary of the evaluation

Summary of the product security

KeePassXC is a mature product, built around security and with a limited attack surface. All the analysis features are well-implemented and robust against common attacks.

Only one security issue has been identified but requiring a more privileged threat agent that those considered in the security target.

Assets or threats	Attack scenario	Exploitability
S3 Threat agent with administration privileges on the host system.	An attacker with administration privileges performs a memory dump of the KeePassXC process and may find databases' master passwords in it.	Exploited

Duration of work

Phase	Duration of work (in days*Person)
Need and environment analysis	1
Product application analysis	1
Design and development analysis	1
Compliance and robustness – Compliance, robustness, and vulnerability analysis	20
Compliance and robustness – Cryptography analysis	5
Results exploitation	2
Report redaction	5

Expert opinion

KeepassXC's development is performed with security in mind: the code base is clean, well documented and good coding practices are applied.

The code provides AFL++ integration in order to fuzz the code [FUZZING]. While the software may seem simple at first, it relies on many cryptographic primitives in order to achieve the database

protection. State-of-the-art cryptography is used, and no outdated or broken algorithm can be used in latest version.

During the allotted time of the audit, Synacktiv experts did not find any flaws or weaknesses preventing KeePassXC to obtain the CSPN.

Notes and remarks

Synacktiv experts recommend investigating the behaviour observed and described in the V-01: Database's master password lingers in memory when pasted page 131.

References

[CEM]	Common Methodology for Information Technology Security Evaluation : Evaluation Methodology, version in effect.
[ANSSI-PG-083]	Guide des mécanismes cryptographiques : Règles et recommandations concernant le choix et le dimensionnement des mécanismes cryptographiques.
[CSPN]	Certification de sécurité de premier niveau des produits des technologies de l'information, ANSSI-CSPN-CER-P-01, version in effect
[CRITERES]	Critères pour l'évaluation en vue d'une certification de sécurité de premier niveau, ANSSI-CSPN-CER-P-02, version in effet.
[Synacktiv-KeePassXC-cible_cspn]	Security target document of KeepassXC version 1.7
[Synacktiv-KeePassXC-specs_crypto]	Cryptographic specifications of KeepassXC version 1.3
[FUZZING]	Fuzz-Testing KeePassXC - https://fossies.org/linux/keepassxc/docs/FuzzTest.md – visité le 08/01/2025

Annexe 1: Security function compliance analysis sheets

SF1: Anti-screenshot/recording

Objective of the analysis	KeePassXC version 2.7.9	
Security function: Anti-screenshot/recording	Ref.: TEST-SF1-SCREENSHOT	Author: Caroline Leman
	Test subject: Protection against screenshot	
Test scenario: Race condition of screenshot protection of KeePassXC running in GUI and unlocked.		
Operations to be carried out	**Expected results**	
- Take a screenshot using Windows screen capture tool - Open/close keepassXC while taking screenshot to trigger a race condition between screenshot protection and screenshot,	KeepassXC windows should disappear or lock the database	
Observed results	**Conclusion**	
The windows does not appear on the screenshot	Correct result, the KeePassXC window does not appear on the screen. The screenshot protection seems to be done before the appearance of the window.	

Objective of the analysis	KeePassXC version 2.7.9	
Security function: Anti-screenshot/recording	Ref.: TEST-SF1-SCREENRECORD	Author: Caroline Leman
	Test subject: Protection against screen recording	
Test scenario: KeePassXC is running in GUI and unlocked.		
Operations to be carried out	**Expected results**	
- Record a video of the screen using OBStudio - Restart KeepassXC multiple times to trigger a race condition between protection setup and screen record.	KeepassXC windows should disappear or lock the database	
Observed results	**Conclusion**	
The windows does not appear on the recording	Correct result, the KeePassXC window does not appear on the screen.	

SF2: Clipboard and auto-type protection

Objective of the analysis:	KeePassXC version 2.7.9	
Security function: Clipboard and auto-type protection	Ref.: TEST-CLIPBOARD	Author: Caroline Leman
	Test subject: Protection against clipboard leak	
Test scenario: KeePassXC is unlocked.		
Operations to be carried out	**Expected results**	
- A password is copied in the clipboard. A script reads the content of the clipboard every second.	KeepassXC should erase the clipboard after a 10 seconds delay (default). The script should stop printing the password after 10 seconds.	
Observed results	**Conclusion**	
After 10 seconds, the script stops printing the password.	Correct result	

Objective of the analysis	KeePassXC version 2.7.9	
Security function: Clipboard and auto-type protection	Ref.: TEST-AUTOTYPE	Author: Caroline Leman
	Test subject: Auto-type feature protection	
Test scenario: KeePassXC is unlocked, auto-type is enabled.		
Operations to be carried out	**Expected results**	
Open a notepad Enable the auto-type of a password in notepad Verify that the password is not in the clipboard	The password should be written character by character in the notepad windows. The clipboard should be empty.	
Observed results	**Conclusion**	
Password is filled, and clipboard is empty	Correct result	

SF3: Database protection

Objective of the analysis	KeePassXC version 2.7.9	
Security function: Database protection	Ref.: TEST-WRONGPWD	Author: Caroline Leman
	Test subject: Test password protection	
Test scenario: KeePassXC is locked.		
Operations to be carried out	**Expected results**	
Type a password that is not correct	The database should not be opened	
Observed results	**Conclusion**	
A message telling HMAC mismatch is displayed. The database is not unlocked.	Correct result	

Objective of the analysis	KeePassXC version 2.7.9	
Security function: Database protection	Ref.: TEST-CORRUPTFILE	Author: Caroline Leman
	Test subject: Test corrupt database protection	
Test scenario: Open a corrupted database.		
Operations to be carried out	**Expected results**	
Modifiy some bytes in the KDBX4 file - Modification in the magic - Modification in the header without fixing the SHA256 - Modification of the header and the SHA256 in the header - Modification of the encrypted data	The database should not be opened, and an error should be displayed.	
Observed results	**Conclusion**	
Error messages are displayed. The database is not unlocked. - Error while reading the database: Not a keepass database. - Error while reading the database: Header SHA256 mismatch. - Error while reading the database: Invalid credentials were provided, please try again. If this reoccurs, then your database file may be corrupt (HMAC mismatch). - Error while reading the database: Invalid inner header id size	All messages are expected except the last one which should trigger an “invalid hmac” message.	

SF4: Memory protection

Objective of the analysis	KeePassXC version 2.7.9	
Security function: Memory protection	Ref.: TEST-MEMPROTEC-1	Author: Florian Guilbert
	Test subject: Memory of the KeePassXC process	
Test scenario: KeePassXC cleans the master password from its memory when the database is open		
Operations to be carried out	**Expected results**	
Open KeePassXC Unlock the database by entering the master password Then dump the process memory using Sysinternals' procdump utility. Run `strings` on the dumpfile and then `grep` the master password.	The master password should not be present within the process memory	
Observed results	**Conclusion**	
The master password value is not found in the process memory dump.	KeePassXC successfully cleans the master password from memory when the database is open.	

Objective of the analysis	KeePassXC version 2.7.9	
Security function: Memory protection	Ref.: TEST-MEMPROTEC-2	Author: Florian Guilbert
	Test subject: Memory of the KeePassXC process	
Test scenario: KeePassXC cleans the master password (that is pasted in the text area) from its memory when the database is open		
Operations to be carried out	**Expected results**	
Open KeePassXC Unlock the database by pasting the master password from another KeePassXC's database for example. Then dump the process memory using Sysinternals' procdump utility. Run `strings` on the dumpfile and then `grep` the master password.	The master password should not be present within the process memory	
Observed results	**Conclusion**	
The master password value is found in the process memory dump.	KeePassXC does not correctly clean the master password when it is copied and pasted.	

Objective of the analysis	KeePassXC version 2.7.9	
Security function: Memory protection	Ref.: TEST-MEMPROTEC-3	Author: Florian Guilbert
	Test subject: Memory of the KeePassXC process	
Test scenario: KeePassXC cleans the entry passwords when it is closed.		
Operations to be carried out	**Expected results**	
Open KeePassXC Unlock the database by entering the master password. Close the database. Then dump the process memory using Sysinternals' procdump utility. Run `strings` on the dumpfile and then `grep` a plaintext password.	Entry passwords should not be present within the process memory	
Observed results	**Conclusion**	
No entry passwords are found in the dump file.	KeePassXC successfully cleans from its memory plaintext passwords.	

Objective of the analysis	KeePassXC version 2.7.9	
Security function: Memory protection	Ref.: TEST-MEMPROTEC-4	Author: Florian Guilbert
	Test subject: Memory of the KeePassXC process	
Test scenario: KeePassXC cleans the entry passwords when it is locked.		
Operations to be carried out	**Expected results**	
Open KeePassXC Unlock the database by entering the master password. Lock the database. Then dump the process memory using Sysinternals' procdump utility. Run `strings` on the dumpfile and then `grep` a plaintext password.	Entry passwords should not be present within the process memory	
Observed results	**Conclusion**	
No entry passwords are found in the dump file.	KeePassXC successfully cleans plaintext passwords from its memory when the database is locked	

Objective of the analysis	KeePassXC version 2.7.9	
Security function: Memory protection	Ref.: TEST-MEMPROTEC-5	Author: Florian Guilbert
	Test subject: Memory of the KeePassXC process	
Test scenario: KeePassXC cleans the cryptograhic keys when it is closed.		
Operations to be carried out	**Expected results**	
Open KeePassXC Unlock the database by entering the master password. Close the database. Then dump the process memory using Sysinternals' procdump utility. Open the dump file and search for the bytes stream corresponding to the computed cryptographic keys.	Cryptographic keys should not be present within the process memory	
Observed results	**Conclusion**	
No entry passwords are found in the dump file.	KeePassXC successfully cleans cryptographic keys from its memory when closing.	

Objective of the analysis	KeePassXC version 2.7.9	
Security function: Memory protection	Ref.: TEST-MEMPROTEC-6	Author: Florian Guilbert
	Test subject: Memory of the KeePassXC process	
Test scenario: KeePassXC cleans the cryptograhic keys when it is locked.		
Operations to be carried out	**Expected results**	
Open KeePassXC Unlock the database by entering the master password. Close the database. Then dump the process memory using Sysinternals' procdump utility. Open the dump file and search for the bytes stream corresponding to the computed cryptographic keys.	Cryptographic keys should not be present within the process memory	
Observed results	**Conclusion**	
No cryptographic keys are found in the dump file.	KeePassXC successfully cleans cryptographic keys from its memory when locking.	

SF5: Prompt of each password access from a browser extension

Objective of the analysis	KeePassXC version 2.7.9	
Security function: Prompt of each password access from a browser extension	Ref.: TEST-BROWSER1	Author: Florian Guilbert
	Test subject: Entry passwords	
Test scenario: Send malformed URL to retrieve arbitrary passwords.		
Operations to be carried out	**Expected results**	
Run the script provided in annex: Script to interact with the KeePassXC browser extension Named Pipe page 149 and modify it with malformed URL: • Incomplete IP address (e.g `1.1`) • Different ports and schemes • The specific scheme `keepassxc://` • Different top level domains and subdomains	Only submitting the exact URL allows retrieving the corresponding entry.	
Observed results	**Conclusion**	
No malformed URL allows getting arbitrary entries.	The matching algorithm efficiently restricts the URL pattern that return passwords.	

SF6: KeeShare segregation

Code auditing was preferred over testing for this security feature.

SF7: SSH-agent interaction

Objective of the analysis	KeePassXC version 2.7.9	
Security function: ssh-agent interaction	Ref.: TEST-SSHAGENT1	Author: Caroline Leman
	Test subject: Attached ssh private key in temp folder	
Test scenario: KeePassXC does not leave ssh private keys on disk when locked.		
Operations to be carried out	**Expected results**	
Open the private key of an entry Close the file Lock the database	The file should be deleted and not be recoverable	
Observed results	**Conclusion**	
File is deleted, and photorec cannot find it.	When a private key is open, a temporary file is created. But when the database is locked, the file is successfully removed and overwritten so that recovery tools cannot recover the private key.	

Objective of the analysis	KeePassXC version 2.7.9	
Security function: ssh-agent interaction	Ref.: TEST-SSHAGENT2	Author: Caroline Leman
	Test subject: Attached ssh private key is encrypted in database	
Test scenario: Recover an ssh private key from a kbdx4 file without knowing the password.		
Operations to be carried out	**Expected results**	
- Store a private key in a database - Verify that the key is not stored in clear in the KDBX4 file	The private key should be encrypted and not readable in the KDBX4 file.	
Observed results	**Conclusion**	
Private key is encrypted and stored as an 'attachment' of an entry. It is encrypted in AES-CBC-256 like the rest of the database.	The ssh private keys are stored encrypted in the database.	

Objective of the analysis	KeePassXC version 2.7.9	
Security function: ssh-agent interaction	Ref.: TEST-SSHAGENT3	Author: Caroline Leman
	Test subject: Attached ssh private key in temp folder	
Test scenario: Keys are successfully removed from agent when database is locked.		
Operations to be carried out	**Expected results**	
- Store a private key in a database - Enable the " add key to agent when database is opened/unlocked" feature - Lock /Unlock the database multiple time very rapidly to race the removal of the key in the agent. - Verify that the key has been removed	The private key should be removed from ssh-agent when database is locked.	
Observed results	**Conclusion**	
The ssh key does not stay in ssh-agent when keepassXC is killed.	Ssh-agent and KeePassXC are synchronous regarding ssh-key handling.	

Annexe 2: Scripts

Interception script for the browser extension communications

```
#!/usr/bin/env python3

from win32pipe import (
    CreateNamedPipe,
    ConnectNamedPipe,
    PIPE_ACCESS_DUPLEX,
    PIPE_TYPE_MESSAGE,
    PIPE_READMODE_MESSAGE,
    PIPE_WAIT,
    PeekNamedPipe,
)
from win32file import (
    CloseHandle,
    ReadFile,
    CreateFile,
    WriteFile,
    GENERIC_READ,
    GENERIC_WRITE,
    OPEN_EXISTING,
)
from time import sleep
from json import dumps, loads
from base64 import b64encode, b64decode
from nacl.public import PrivateKey, Box, PublicKey
from nacl.encoding import Base64Encoder
from pprint import pprint
from colorama import Fore, init as colorama_init
from sys import stdout

NAMED_PIPE_NAME = r"\\.\\pipe\\org.keepassxc.KeePassXC.BrowserServer_user"
SERVER = f"{NAMED_PIPE_NAME}"
CLIENT = f"{NAMED_PIPE_NAME}2"
BUFFER_SIZE = 2**16


my_keypair = PrivateKey.generate()
my_pubkey = my_keypair.public_key
my_privkey = my_keypair

keys = {
    "Browser": None,
```

```
    "KeePassXC": None,
}

color = {"Browser": Fore.BLUE, "KeePassXC": Fore.GREEN}


def process_msg(msg, actor):
    if actor == "Browser":
        other = "KeePassXC"
    else:
        other = "Browser"

    if msg["action"] == "change-public-keys":
        keys[actor] = PublicKey(b64decode(msg["publicKey"]))
        new_msg = msg
        new_msg["publicKey"] = my_pubkey.encode(encoder=Base64Encoder).decode("utf8")
        return new_msg
    elif "message" in msg:
        nonce = b64decode(msg["nonce"])
        box = Box(my_privkey, keys[actor])
        decrypted = box.decrypt(b64decode(msg["message"]), nonce)
        stdout.write(color[actor])
        pprint(loads(decrypted.decode("utf8")))
        stdout.write(Fore.RESET)
        box = Box(my_privkey, keys[other])
        encrypted = box.encrypt(decrypted, nonce)
        new_msg = msg
        new_msg["message"] = b64encode(encrypted.ciphertext).decode("utf8")
        return new_msg


def process_incoming_data(pipe_in, pipe_out, avail, length, label):
    if len(avail) > 0 or length > 0:
        err, data = ReadFile(pipe_in, BUFFER_SIZE, None)
        if err != 0:
            print(f"{label} disconnected: error while reading")
            return False
        msg = loads(data.decode("utf8"))
        new_msg = process_msg(msg, label)
        err, _ = WriteFile(pipe_out, dumps(new_msg).encode("utf8"))
        if err != 0:
            print(f"{label} disconnected: error while writing")
            return False

    return True
```

```
if __name__ == "__main__":
    colorama_init()
    client = CreateFile(
        CLIENT,
        GENERIC_READ | GENERIC_WRITE,
        0,
        None,
        OPEN_EXISTING,
        0,
        None,
    )

    server = CreateNamedPipe(
        SERVER,
        PIPE_ACCESS_DUPLEX,
        PIPE_TYPE_MESSAGE | PIPE_READMODE_MESSAGE | PIPE_WAIT,
        1,
        BUFFER_SIZE,
        BUFFER_SIZE,
        0,
        None,
    )
    ConnectNamedPipe(server, None)
    print(f"Someone is connected to {SERVER}")
    while True:
        avail, length, _ = PeekNamedPipe(server, 0)
        if not process_incoming_data(server, client, avail, length, "Browser"):
            break
        avail, length, _ = PeekNamedPipe(client, 0)
        if not process_incoming_data(client, server, avail, length, "KeePassXC"):
            break
        else:
            sleep(0.1)
    CloseHandle(client)
    CloseHandle(server)
```

Script to interact with the KeePassXC browser extension Named Pipe

```
#!/usr/bin/env python3

from win32file import (
    CloseHandle,
    ReadFile,
    CreateFile,
    WriteFile,
    GENERIC_READ,
```

```
    GENERIC_WRITE,
    OPEN_EXISTING,
)
from json import dumps, loads
from base64 import b64encode, b64decode
from nacl.public import PrivateKey, Box, PublicKey
from nacl.encoding import Base64Encoder
from nacl.utils import random
from pprint import pprint

from colorama import Fore, init as colorama_init
from sys import stdout, argv

NAMED_PIPE_NAME = r"\\.\\pipe\\org.keepassxc.KeePassXC.BrowserServer_user"
CLIENT = f"{NAMED_PIPE_NAME}2"
BUFFER_SIZE = 2**16
CLIENT_ID = "ptQyCNSfesotfI1gK8srCkqa6HiZlvlg"


my_keypair = PrivateKey.generate()
my_pubkey = my_keypair.public_key
my_privkey = my_keypair


def get_nonce():
    return random(Box.NONCE_SIZE)


def send_message(pipe, msg):
    data = dumps(msg).encode("utf8")
    stdout.write(Fore.BLUE)
    print(f">> Sending >> {data}")
    stdout.write(Fore.RESET)
    err, _ = WriteFile(pipe, data)
    if err != 0:
        print("Error while writing a message")
        exit(1)


def receive_message(pipe):
    err, data = ReadFile(pipe, BUFFER_SIZE)
    stdout.write(Fore.GREEN)
    print(f">> Receiving >> {data}")
    stdout.write(Fore.RESET)
    if err != 0:
        print("Error while reading a message")
        exit(1)
```

```
    return loads(data.decode("utf8"))


def encrypt_message(box, nonce, message):
    return b64encode(
        box.encrypt(dumps(message).encode("utf8"), nonce).ciphertext
    ).decode("utf8")


def decrypt_message(box, nonce, msg):
    return loads(
        box.decrypt(b64decode(msg["message"]), b64decode(msg["nonce"])).decode("utf8")
    )


if __name__ == "__main__":
    # print(my_privkey.encode(Base64Encoder))
    # print(my_pubkey.encode(Base64Encoder))
    colorama_init()
    print(argv)
    pipe = CreateFile(
        CLIENT,
        GENERIC_READ | GENERIC_WRITE,
        0,
        None,
        OPEN_EXISTING,
        0,
        None,
    )

    msg = {
        "action": "change-public-keys",
        "publicKey": my_pubkey.encode(encoder=Base64Encoder).decode("utf8"),
        "nonce": b64encode(get_nonce()).decode("utf8"),
        "clientID": CLIENT_ID,
    }
    send_message(pipe, msg)
    msg = receive_message(pipe)

    keepassxc_public_key = PublicKey(b64decode(msg["publicKey"]))
    if msg["success"] != "true":
        print("Obtain 'success': 'false'")
        exit(1)
    box = Box(my_privkey, keepassxc_public_key)

    # GET-DATABASEHASH
    nonce = get_nonce()
```

```
msg = {
    "action": "get-databasehash",
    "nonce": b64encode(nonce).decode("utf8"),
    "clientID": CLIENT_ID,
}
message = {
    "action": "get-databasehash",
}
msg["message"] = encrypt_message(box, nonce, message)
send_message(pipe, msg)
msg = receive_message(pipe)
message = decrypt_message(box, nonce, msg)
pprint(message)
################

# TESTASSOCIATE
nonce = get_nonce()
msg = {
    "action": "test-associate",
    "nonce": b64encode(nonce).decode("utf8"),
    "clientID": CLIENT_ID,
}

message = {
    "action": "test-associate",
    "id": "chrome-laptop",
    "key": "Yhpl+HJk99J2AXEyVYL1+df+LAeyNab+pSMEKznV6Qw=",
}
msg["message"] = encrypt_message(box, nonce, message)
send_message(pipe, msg)
msg = receive_message(pipe)
message = decrypt_message(box, nonce, msg)
pprint(message)
################

# GET-LOGINS
nonce = get_nonce()
msg = {
    "action": "get-logins",
    "nonce": b64encode(nonce).decode("utf8"),
    "clientID": CLIENT_ID,
}

message = {
    "action": "get-logins",
    "url": f"{argv[1]}",
    # "submitUrl": "",
```

```
        "keys": [
            {
                "id": "chrome-laptop",
                "key": "Yhpl+HJk99J2AXEyVYL1+df+LAeyNab+pSMEKznV6Qw=",
            }
        ],
    }
    msg["message"] = encrypt_message(box, nonce, message)
    pprint(message)
    send_message(pipe, msg)
    msg = receive_message(pipe)
    message = decrypt_message(box, nonce, msg)
    pprint(message)
    ###############

    CloseHandle(pipe)
```

Script to decrypt KDBX4 file with default parameters

```
#!/usr/bin/env python3

from kdbx import Kdbx
from pprint import pprint
from argparse import ArgumentParser
from Cryptodome.Hash import SHA256, SHA512, HMAC
from Cryptodome.Cipher import AES
import argon2
from argon2 import PasswordHasher, Type # pip3 install argon2-cffi
from base64 import b64encode, b64decode

Crypto_uuid = {
    bytes.fromhex("61ab05a1-9464-41c3-8d74-3a563df8dd35".replace("-", "")) :
"CIPHER_AES128",
    bytes.fromhex("31c1f2e6-bf71-4350-be58-05216afc5aff".replace("-", "")) :
"CIPHER_AES256",
    bytes.fromhex("ad68f29f-576f-4bb9-a36a-d47af965346c".replace("-", "")) :
"CIPHER_TWOFISH",
    bytes.fromhex("d6038a2b-8b6f-4cb5-a524-339a31dbb59a".replace("-", "")) :
"CIPHER_CHACHA20",
    bytes.fromhex("c9d9f39a-628a-4460-bf74-0d08c18a4fea".replace("-", "")) : "KDF_AES",
    bytes.fromhex("7c02bb82-79a7-4ac0-927d-114a00648238".replace("-", "")) : "KDF_AES",
    bytes.fromhex("ef636ddf-8c29-444b-91f7-a9a403e30a0c".replace("-","")) :
"KDF_ARGON2D",
    bytes.fromhex("9e298b19-56db-4773-b23d-fc3ec6f0a1e6".replace("-","")) :
"KDF_ARGON2ID"}


parser = ArgumentParser()
parser.add_argument("-f", "--filename", help="KDBX file")
parser.add_argument("-p", "--password", help="Password for the decryption")
parser.add_argument("-o", "--output", help="Path to write decrypted databse",
default="decrypted.xml")

args = parser.parse_args()

print(f"Opening {args.filename}")
db = Kdbx.from_file(args.filename)

def field_print(data):
    if isinstance(data, bytes):
        print(data.hex(" "))
    elif isinstance(data, int):
        print(hex(data))
    else:
        print(data)
```

```
for field in db.header.SEQ_FIELDS:
    print("-", field, end=' : ')
    field_print(getattr(db.header, field))

kdf_parameters = dict()
for dyntype in db.dynamic_header_entries:
    if dyntype.type.name != "kdf_parameters":
        value = dyntype.data
        kdf_parameters[dyntype.type.name] = value
        if not isinstance(value, bytes):
            value = dyntype.data.type
        else:
            value = value.hex(" ")
        if dyntype.type.name == "cipher_id":
            print(f"- {dyntype.type.name} : {value} ({Crypto_uuid.get(value, '')})")
            continue
        print(f"- {dyntype.type.name} : {value}")


    else:
        print(f"- {dyntype.type.name} : ")
        for item in dyntype.data.parameters.serialized_items.items:
            if item.key_name == "$UUID":
                print(f"    {item.key_name} : {item.key_value.hex(' ')}
({Crypto_uuid.get(item.key_value, '')})")
                kdf_parameters[item.key_name] = item.key_value
                continue
            try:
                print(f"    {item.key_name} : {item.key_value.hex(' ')}")
            except Exception:
                print(f"    {item.key_name} : {hex(item.key_value)}")
            kdf_parameters[item.key_name] = item.key_value
        pass


header_data = open(args.filename, "rb").read(db.payload._debug['header_checksum']
['start'])
print("- Header's SHA256:", db.payload.header_checksum.hex(" "))
print("- Header's HMAC", db.payload.hmac.hex(" "))

print("[+] Header checksum is correct ?", SHA256.new(header_data).digest() ==
db.payload.header_checksum)

hashedPassword = SHA256.new(SHA256.new(bytes(args.password,
"utf-8")).digest()).digest()
print("[+] Hashed password", hashedPassword)
```

```
derivedKey = argon2.low_level.hash_secret_raw(hashedPassword, kdf_parameters["S"],
time_cost=kdf_parameters["I"], memory_cost=0x10000, parallelism=kdf_parameters["P"],
hash_len=32, type=argon2.low_level.Type.D)

print("[+] Derived key from password", derivedKey)

def get_hmac_key_block(hmacKey, block_index):
    return SHA512.new(int.to_bytes(block_index, 8, "little") + hmacKey).digest()

hmacKey = SHA512.new(kdf_parameters["master_seed"] + derivedKey + b"\x01").digest()
headhmacKey = get_hmac_key_block(hmacKey, 0xFFFFFFFFFFFFFFFF)
print("[+] Hmac key from password:", headhmacKey)

hmac = HMAC.new(headhmacKey, msg=header_data, digestmod=SHA256)


print("[+] HMAC verification is OK ?", hmac.digest() == db.payload.hmac)


databaseKey = SHA256.new(kdf_parameters["master_seed"] + derivedKey).digest()
print("[+] Decryption database key is (WIP):", databaseKey)

decryption = AES.new(databaseKey, AES.MODE_CBC, iv=kdf_parameters["encryption_iv"])


for block in db.payload.encrypted_blocks:
    if block.len_data == 0:
        break
    data = decryption.decrypt(block.data)

if kdf_parameters["compression_flags"].type.value == 1: #gzip
    import zlib
    data = zlib.decompress(data, 31)

print(f"[+] Writing decrypted database to {args.output}")
open(args.output, "wb").write(data)
```

Kaitai generated Kdbx class (used in decryption script)

```
# This is a generated file! Please edit source .ksy file and use kaitai-struct-compiler
to rebuild

import kaitaistruct
from kaitaistruct import KaitaiStruct, KaitaiStream, BytesIO
from enum import Enum
import collections


if getattr(kaitaistruct, 'API_VERSION', (0, 9)) < (0, 9):
    raise Exception("Incompatible Kaitai Struct Python API: 0.9 or later is required,
but you have %s" % (kaitaistruct.__version__))

class Kdbx(KaitaiStruct):
    """composite_key = sha256(sha256(password) + composites)
    aes = new AES(128, CBC, iv=0x0 *16, key TRANSFORMSEED
    transformed_key=sha256(for i in TRANSFORMROUNDS: aes.encrypt(transformed_key)))
    master_key=sha256(CONCAT(MASTERSEED,transformed_key))

    Kaitai drafty imprementation by Alexandre LEVAVASSEUR <alexandre+oss@13x.fr> (2017,
ISC license)
    """

    class DynamicHeaderType(Enum):
        end = 0
        comment = 1
        cipher_id = 2
        compression_flags = 3
        master_seed = 4
        transform_seed = 5
        transform_rounds = 6
        encryption_iv = 7
        protected_stream_key = 8
        stream_start_bytes = 9
        inner_random_stream_id = 10
        kdf_parameters = 11
        public_custom_data = 12

    class DynamicInnerHeaderType(Enum):
        end = 0
        stream_cipher = 1
        stream_key = 2
        binary = 3
    SEQ_FIELDS = ["header", "dynamic_header_entries", "payload"]
```

```
    def __init__(self, _io, _parent=None, _root=None):
        self._io = _io
        self._parent = _parent
        self._root = _root if _root else self
        self._debug = collections.defaultdict(dict)
        self._read()

    def _read(self):
        self._debug['header']['start'] = self._io.pos()
        self.header = Kdbx.Header(self._io, self, self._root)
        self._debug['header']['end'] = self._io.pos()
        self._debug['dynamic_header_entries']['start'] = self._io.pos()
        self.dynamic_header_entries = []
        i = 0
        while True:
            if not 'arr' in self._debug['dynamic_header_entries']:
                self._debug['dynamic_header_entries']['arr'] = []
            self._debug['dynamic_header_entries']['arr'].append({'start':
self._io.pos()})
            _ = Kdbx.DynamicHeaderEntry(self._io, self, self._root)
            self.dynamic_header_entries.append(_)
            self._debug['dynamic_header_entries']['arr']
[len(self.dynamic_header_entries) - 1]['end'] = self._io.pos()
            if _.type == Kdbx.DynamicHeaderType.end:
                break
            i += 1
        self._debug['dynamic_header_entries']['end'] = self._io.pos()
        self._debug['payload']['start'] = self._io.pos()
        self.payload = Kdbx.Payload(self._io, self, self._root)
        self._debug['payload']['end'] = self._io.pos()

    class Payload(KaitaiStruct):
        SEQ_FIELDS = ["header_checksum", "hmac", "encrypted_blocks"]
        def __init__(self, _io, _parent=None, _root=None):
            self._io = _io
            self._parent = _parent
            self._root = _root if _root else self
            self._debug = collections.defaultdict(dict)
            self._read()

        def _read(self):
            self._debug['header_checksum']['start'] = self._io.pos()
            self.header_checksum = self._io.read_bytes(32)
            self._debug['header_checksum']['end'] = self._io.pos()
            self._debug['hmac']['start'] = self._io.pos()
            self.hmac = self._io.read_bytes(32)
            self._debug['hmac']['end'] = self._io.pos()
            self._debug['encrypted_blocks']['start'] = self._io.pos()
```

```
            self.encrypted_blocks = []
            i = 0
            while True:
                if not 'arr' in self._debug['encrypted_blocks']:
                    self._debug['encrypted_blocks']['arr'] = []
                self._debug['encrypted_blocks']['arr'].append({'start':
self._io.pos()})
                _ = Kdbx.EncryptedBlock(self._io, self, self._root)
                self.encrypted_blocks.append(_)
                self._debug['encrypted_blocks']['arr'][len(self.encrypted_blocks) - 1]
['end'] = self._io.pos()
                if _.len_data == 0:
                    break
                i += 1
            self._debug['encrypted_blocks']['end'] = self._io.pos()


    class EncryptedBlock(KaitaiStruct):
        SEQ_FIELDS = ["block_hmac", "len_data", "data"]
        def __init__(self, _io, _parent=None, _root=None):
            self._io = _io
            self._parent = _parent
            self._root = _root if _root else self
            self._debug = collections.defaultdict(dict)
            self._read()

        def _read(self):
            self._debug['block_hmac']['start'] = self._io.pos()
            self.block_hmac = self._io.read_bytes(32)
            self._debug['block_hmac']['end'] = self._io.pos()
            self._debug['len_data']['start'] = self._io.pos()
            self.len_data = self._io.read_u4le()
            self._debug['len_data']['end'] = self._io.pos()
            self._debug['data']['start'] = self._io.pos()
            self.data = self._io.read_bytes(self.len_data)
            self._debug['data']['end'] = self._io.pos()


    class VariantDictionary(KaitaiStruct):
        SEQ_FIELDS = ["version", "serialized_items", "terminator"]
        def __init__(self, _io, _parent=None, _root=None):
            self._io = _io
            self._parent = _parent
            self._root = _root if _root else self
            self._debug = collections.defaultdict(dict)
            self._read()

        def _read(self):
```

```
            self._debug['version']['start'] = self._io.pos()
            self.version = self._io.read_u2le()
            self._debug['version']['end'] = self._io.pos()
            self._debug['serialized_items']['start'] = self._io.pos()
            self._raw_serialized_items = self._io.read_bytes((self._parent._parent.size
- 3))
            _io__raw_serialized_items =
KaitaiStream(BytesIO(self._raw_serialized_items))
            self.serialized_items =
Kdbx.VariantDictionary.SerializedItems(_io__raw_serialized_items, self, self._root)
            self._debug['serialized_items']['end'] = self._io.pos()
            self._debug['terminator']['start'] = self._io.pos()
            self.terminator = self._io.read_bytes(1)
            self._debug['terminator']['end'] = self._io.pos()
            if not self.terminator == b"\x00":
                raise kaitaistruct.ValidationNotEqualError(b"\x00", self.terminator,
self._io, u"/types/variant_dictionary/seq/2")

        class SerializedItems(KaitaiStruct):
            SEQ_FIELDS = ["items"]
            def __init__(self, _io, _parent=None, _root=None):
                self._io = _io
                self._parent = _parent
                self._root = _root if _root else self
                self._debug = collections.defaultdict(dict)
                self._read()

            def _read(self):
                self._debug['items']['start'] = self._io.pos()
                self.items = []
                i = 0
                while not self._io.is_eof():
                    if not 'arr' in self._debug['items']:
                        self._debug['items']['arr'] = []
                    self._debug['items']['arr'].append({'start': self._io.pos()})

self.items.append(Kdbx.VariantDictionary.SerializedItems.SerializedItem(self._io, self,
self._root))
                    self._debug['items']['arr'][len(self.items) - 1]['end'] =
self._io.pos()
                    i += 1

                self._debug['items']['end'] = self._io.pos()

            class SerializedItem(KaitaiStruct):

                class SerializedItemType(Enum):
                    uint32 = 4
                    uint64 = 5
```

```
                    bool = 8
                    int32 = 12
                    int64 = 13
                    string = 24
                    byte_array = 66
                SEQ_FIELDS = ["type", "key_name_len", "key_name", "key_value_len",
"key_value"]
                def __init__(self, _io, _parent=None, _root=None):
                    self._io = _io
                    self._parent = _parent
                    self._root = _root if _root else self
                    self._debug = collections.defaultdict(dict)
                    self._read()

                def _read(self):
                    self._debug['type']['start'] = self._io.pos()
                    self.type =
KaitaiStream.resolve_enum(Kdbx.VariantDictionary.SerializedItems.SerializedItem.Seriali
zedItemType, self._io.read_u1())
                    self._debug['type']['end'] = self._io.pos()
                    self._debug['key_name_len']['start'] = self._io.pos()
                    self.key_name_len = self._io.read_u4le()
                    self._debug['key_name_len']['end'] = self._io.pos()
                    self._debug['key_name']['start'] = self._io.pos()
                    self.key_name =
(self._io.read_bytes(self.key_name_len)).decode(u"UTF-8")
                    self._debug['key_name']['end'] = self._io.pos()
                    self._debug['key_value_len']['start'] = self._io.pos()
                    self.key_value_len = self._io.read_u4le()
                    self._debug['key_value_len']['end'] = self._io.pos()
                    self._debug['key_value']['start'] = self._io.pos()
                    _on = self.type
                    if _on ==
Kdbx.VariantDictionary.SerializedItems.SerializedItem.SerializedItemType.uint64:
                        self.key_value = self._io.read_u8le()
                    elif _on ==
Kdbx.VariantDictionary.SerializedItems.SerializedItem.SerializedItemType.int64:
                        self.key_value = self._io.read_s8le()
                    elif _on ==
Kdbx.VariantDictionary.SerializedItems.SerializedItem.SerializedItemType.int32:
                        self.key_value = self._io.read_s4le()
                    elif _on ==
Kdbx.VariantDictionary.SerializedItems.SerializedItem.SerializedItemType.bool:
                        self.key_value = self._io.read_u1()
                    elif _on ==
Kdbx.VariantDictionary.SerializedItems.SerializedItem.SerializedItemType.string:
                        self.key_value =
(self._io.read_bytes(self.key_value_len)).decode(u"UTF-8")
                    elif _on ==
Kdbx.VariantDictionary.SerializedItems.SerializedItem.SerializedItemType.uint32:
```

```
                        self.key_value = self._io.read_u4le()
                    else:
                        self.key_value = self._io.read_bytes(self.key_value_len)
                    self._debug['key_value']['end'] = self._io.pos()



    class DynamicHeaderEntry(KaitaiStruct):

        class DynamicHeaderTypeCompressionList(Enum):
            none = 0
            gzip = 1

        class DynamicHeaderTypeInnerRandomStreamIdList(Enum):
            none = 0
            arc4_variant = 1
            salsa20 = 2
            chacha20 = 3
        SEQ_FIELDS = ["type", "len", "size", "data"]
        def __init__(self, _io, _parent=None, _root=None):
            self._io = _io
            self._parent = _parent
            self._root = _root if _root else self
            self._debug = collections.defaultdict(dict)
            self._read()

        def _read(self):
            self._debug['type']['start'] = self._io.pos()
            self.type = KaitaiStream.resolve_enum(Kdbx.DynamicHeaderType,
self._io.read_u1())
            self._debug['type']['end'] = self._io.pos()
            if self._root.header.file_version < 262144:
                self._debug['len']['start'] = self._io.pos()
                self.len = self._io.read_u2le()
                self._debug['len']['end'] = self._io.pos()

            if self._root.header.file_version >= 262144:
                self._debug['size']['start'] = self._io.pos()
                self.size = self._io.read_u4le()
                self._debug['size']['end'] = self._io.pos()

            self._debug['data']['start'] = self._io.pos()
            _on = self.type
            if _on == Kdbx.DynamicHeaderType.transform_rounds:
                self._raw_data = self._io.read_bytes(self.size)
                _io__raw_data = KaitaiStream(BytesIO(self._raw_data))
```

```
                self.data =
Kdbx.DynamicHeaderEntry.DynamicHeaderTypeTransformRounds(_io__raw_data, self,
self._root)
            elif _on == Kdbx.DynamicHeaderType.inner_random_stream_id:
                self._raw_data = self._io.read_bytes(self.size)
                _io__raw_data = KaitaiStream(BytesIO(self._raw_data))
                self.data =
Kdbx.DynamicHeaderEntry.DynamicHeaderTypeInnerRandomStreamId(_io__raw_data, self,
self._root)
            elif _on == Kdbx.DynamicHeaderType.cipher_id:
                self._raw_data = self._io.read_bytes(self.size)
                _io__raw_data = KaitaiStream(BytesIO(self._raw_data))
                self.data =
Kdbx.DynamicHeaderEntry.DynamicHeaderTypeCipherid(_io__raw_data, self, self._root)
            elif _on == Kdbx.DynamicHeaderType.compression_flags:
                self._raw_data = self._io.read_bytes(self.size)
                _io__raw_data = KaitaiStream(BytesIO(self._raw_data))
                self.data =
Kdbx.DynamicHeaderEntry.DynamicHeaderTypeCompression(_io__raw_data, self, self._root)
            elif _on == Kdbx.DynamicHeaderType.comment:
                self._raw_data = self._io.read_bytes(self.size)
                _io__raw_data = KaitaiStream(BytesIO(self._raw_data))
                self.data =
Kdbx.DynamicHeaderEntry.DynamicHeaderTypeComment(_io__raw_data, self, self._root)
            elif _on == Kdbx.DynamicHeaderType.kdf_parameters:
                self._raw_data = self._io.read_bytes(self.size)
                _io__raw_data = KaitaiStream(BytesIO(self._raw_data))
                self.data =
Kdbx.DynamicHeaderEntry.DynamicHeaderTypeKdfParameters(_io__raw_data, self, self._root)
            else:
                self.data = self._io.read_bytes(self.size)
            self._debug['data']['end'] = self._io.pos()

        class DynamicHeaderTypeCompression(KaitaiStruct):
            SEQ_FIELDS = ["type"]
            def __init__(self, _io, _parent=None, _root=None):
                self._io = _io
                self._parent = _parent
                self._root = _root if _root else self
                self._debug = collections.defaultdict(dict)
                self._read()

            def _read(self):
                self._debug['type']['start'] = self._io.pos()
                self.type =
KaitaiStream.resolve_enum(Kdbx.DynamicHeaderEntry.DynamicHeaderTypeCompressionList,
self._io.read_u4le())
                self._debug['type']['end'] = self._io.pos()
```

```
        class DynamicHeaderTypeTransformRounds(KaitaiStruct):
            SEQ_FIELDS = ["count"]
            def __init__(self, _io, _parent=None, _root=None):
                self._io = _io
                self._parent = _parent
                self._root = _root if _root else self
                self._debug = collections.defaultdict(dict)
                self._read()

            def _read(self):
                self._debug['count']['start'] = self._io.pos()
                self.count = self._io.read_u8le()
                self._debug['count']['end'] = self._io.pos()


        class DynamicHeaderTypeInnerRandomStreamId(KaitaiStruct):
            SEQ_FIELDS = ["type"]
            def __init__(self, _io, _parent=None, _root=None):
                self._io = _io
                self._parent = _parent
                self._root = _root if _root else self
                self._debug = collections.defaultdict(dict)
                self._read()

            def _read(self):
                self._debug['type']['start'] = self._io.pos()
                self.type =
KaitaiStream.resolve_enum(Kdbx.DynamicHeaderEntry.DynamicHeaderTypeInnerRandomStreamIdL
ist, self._io.read_u4le())
                self._debug['type']['end'] = self._io.pos()


        class DynamicHeaderTypeCipherid(KaitaiStruct):
            SEQ_FIELDS = ["type"]
            def __init__(self, _io, _parent=None, _root=None):
                self._io = _io
                self._parent = _parent
                self._root = _root if _root else self
                self._debug = collections.defaultdict(dict)
                self._read()

            def _read(self):
                self._debug['type']['start'] = self._io.pos()
                self.type = self._io.read_bytes(self._parent.size)
                self._debug['type']['end'] = self._io.pos()
```

```
        class DynamicHeaderTypeComment(KaitaiStruct):
            SEQ_FIELDS = ["content"]
            def __init__(self, _io, _parent=None, _root=None):
                self._io = _io
                self._parent = _parent
                self._root = _root if _root else self
                self._debug = collections.defaultdict(dict)
                self._read()

            def _read(self):
                self._debug['content']['start'] = self._io.pos()
                self.content =
(self._io.read_bytes(self._parent.size)).decode(u"ascii")
                self._debug['content']['end'] = self._io.pos()


        class DynamicHeaderTypeKdfParameters(KaitaiStruct):
            SEQ_FIELDS = ["parameters"]
            def __init__(self, _io, _parent=None, _root=None):
                self._io = _io
                self._parent = _parent
                self._root = _root if _root else self
                self._debug = collections.defaultdict(dict)
                self._read()

            def _read(self):
                self._debug['parameters']['start'] = self._io.pos()
                self.parameters = Kdbx.VariantDictionary(self._io, self, self._root)
                self._debug['parameters']['end'] = self._io.pos()



    class Header(KaitaiStruct):

        class Version(Enum):
            keepass1 = 101
            keepass2_prerelease = 102
            keepass2 = 103
        SEQ_FIELDS = ["primary_identifier", "version", "secondary_identifier",
"file_version"]
        def __init__(self, _io, _parent=None, _root=None):
            self._io = _io
            self._parent = _parent
            self._root = _root if _root else self
            self._debug = collections.defaultdict(dict)
            self._read()

        def _read(self):
```

```
            self._debug['primary_identifier']['start'] = self._io.pos()
            self.primary_identifier = self._io.read_bytes(4)
            self._debug['primary_identifier']['end'] = self._io.pos()
            if not self.primary_identifier == b"\x03\xD9\xA2\x9A":
                raise kaitaistruct.ValidationNotEqualError(b"\x03\xD9\xA2\x9A",
self.primary_identifier, self._io, u"/types/header/seq/0")
            self._debug['version']['start'] = self._io.pos()
            self.version = KaitaiStream.resolve_enum(Kdbx.Header.Version,
self._io.read_u1())
            self._debug['version']['end'] = self._io.pos()
            self._debug['secondary_identifier']['start'] = self._io.pos()
            self.secondary_identifier = self._io.read_bytes(3)
            self._debug['secondary_identifier']['end'] = self._io.pos()
            if not self.secondary_identifier == b"\xFB\x4B\xB5":
                raise kaitaistruct.ValidationNotEqualError(b"\xFB\x4B\xB5",
self.secondary_identifier, self._io, u"/types/header/seq/2")
            self._debug['file_version']['start'] = self._io.pos()
            self.file_version = self._io.read_u4le()
            self._debug['file_version']['end'] = self._io.pos()
```