FreeMessage

Secure Messaging by GMX and WEB.DE

Encryption White Paper
September 2016

Contents

1 Introduction and Intention

Nowadays, billions of smartphone users share their most personal information online using private messaging apps. However, most messaging apps fail at protecting the content shared by users.

It is our firm belief this content shared on messaging apps is worthy of protection against unauthorized access by 3rd parties as well as protection against loss of data. Adding extra layers of data protection in messaging apps has strong implications on user experience. We are convinced, that for end-to-end encrypted messaging products to become ubiquitous, the friction generated by the use of encryption protocols needs to be eliminated. It is imperative, that encryption happens almost unconsciously, as we cannot assume users to have or acquire either any technological knowledge or tolerate inconvenience or lack of functionality imposed by encryption.

FreeMessage was designed with simplicity and strong protection of user data in mind. The FreeMessage apps implement a custom encryption protocol designed to satisfy both users' needs for privacy and a seamless user experience. The protocol's design prevents 3rd parties as well as 1&1 Mail & Media as the operator of FreeMessage from ever being able to access message contents in plaintext.

This paper explains the technical procedure to achieve end-to-end encryption on FreeMessage and describes the underlying trade-offs regarding privacy and functionality.

2 Terms and Nomenclature

Key block: Package used for securely transmitting encrypted secret keys
FreeMessage key store: Server-side component storing and distributing users' public-keys.
Message directory: Server-side component storing users' encrypted message history.
Document manager: Blob store to enable efficient exchange of files such as images and videos.
JID: Internal ID identifying the user account on FreeMessage.

3 Trust

Public keys obtained from our FreeMessage keystore will be signed by FreeMessage and implicitly considered as trustable. Users of FreeMessage have the possibility to verify each other's public key to rule out a man-in-the-middle attack. FreeMessage obtains the correct public key for a given user ID. If a user communicates without authenticating her chat partner's public key, she has to trust the FreeMessage key store to hand out the right key. In all cases, users have to trust the hardware and operating system they are using. Specifically there are Android and iOS clients available for FreeMessage, thus users have to trust Google and Apple that

- the app executables deployed via the respective app store are authentic.
- any decrypted or displayed content is not stored anywhere else outside the app's sandbox without user consent (this specifically concerns functionality like "Google Now on Tap").
- keys stored on the device are not re-distributed by the OS.

The user also has to trust 1&1 Mail & Media that the apps uploaded to Apple and Google do not exhibit malicious behaviour such as unsafely handling keys or other private data. See section 10 for information on code reviews.

4 Keys

Key Generation

The key-pairs are created on the client app. The client generates its key-pair by choosing a private key at random and then calculates the corresponding public key over the Elliptic Curve (Curve25519). To encrypt a message the client needs its own private key together with the receiver's public key. For decrypting the receiving client needs the sender's public key and its own private key.

Key Distribution & Verification

The only keys that are valid on FreeMessage, are the ones created by FreeMessage clients and distributed over FreeMessage key store. This transaction is protected by SSL and the server's certificates which are already trusted by the client through the operating system's key chain. The client will only download keys from the FreeMessage key store over this secure communication channel. This prevents keys from being changed on the way to the client.
Additionally users are able to verify key integrity themselves directly with their chat partner. This happens by comparing the key hashes over a second communication channel, ideally in person. A checked key can be marked "verified" on the local device. There is no chain of trust. Only this one key from this user will be shown as "verified". The name shown to the user in the chat and during key validation is from her local phone book. Therefore the other person has no way of changing the name from the server side.

5 Message Exchange

To encrypt messages, the clients creates a 256 bit random secret key `k` and a 128 bit random initial vector `IV`. Then the message is encrypted using `k` and `IV` with `AES256` in cipher block chaining mode: `c = AES256-CBC(k, IV, message)`. To allow validation of this message by the receiving client the sending client will also generate another random secret key `mk`. With this key it computes the message authentication code `mac = HMAC-SHA256(IV || c, mk)`. In this case `a||b` means concatenation `a` and `b`.
Then the random key `k` and the key for the message authentication code `mk` will be encrypted with Curve25519 for the receiver. This encryption works by generating the derived key from the users private key and the receivers public key as follows: Alice and Bob both have generated a 256 bit private key `dA` and `dB` as well as one public key `QA = dA * G` and `QB = dB * G` respectively on their device[1]. To derive a master key to exchange messages between Alice and Bob, Alice calculates: `QB * dA = dB * G * dA` and Bob calculates: `QA * dB = dA * G * dB`. After that they will both have a shared secret from which the master key can be derived. To prevent an attacker from being able to simply replay the sent message to the sender, the communication direction is added to the master key. To generate the master key and the message authentication code key this shared secret and the communication direction are appended and used as input to an `PBKDF2` with a fixed salt and an output length of 512 bit. The message authentication code key is being used to ensure the keyblock package is not being altered. We generate a 256 bit master key `kek = PBKDF2(JID1|JID2|Base64(shared_secret))` and a 256 bit message authentication code key `mk2`.

After generating the master key, Alice also creates a 128 bit initial vector `IV2`. Then the encryption key and the message authentication code are encrypted using the master key and this initial vector: `ck = AES256-CBC(kek, IV2, k +`

[1] https://cr.yp.to/ecdh/curve25519-20060209.pdf - Note that the operation denoted by * is an operation on the elliptic curve as described in the Bernstein's paper and must not be confused with a standard multiplication.

`mk`). The new message authentication code `mack` is computed as follows: `mack = HMAC-SHA256(IV2 || ck, mk2).`

From this data the client needs to send: `IV`, `IV2`, `mac`, `mack`, `c`, `ck`. This is split in two actual messages. Message one will contain `IV2`, `mack` and `ck`. Message two will contain a reference to message one (the so called keyblock id) and also `IV`, `mac` and `c`. With this data and its private key and the senders public key, the client will verify the authentication of the message by checking the `mack`. If the message authentication code is valid the client can decrypt the original message. To save storage the client sends all messages `Base64` encoded instead of using hex encoded messages. The message format is as follows:

```
<message to="receiver" id="...">
 <keyblock id="3c964b439..." xmlns="urn:1and1:xmpp:encrypted">
  <key jid="receiver" senderKeyId="0" receiverKeyId="1">
   <ciphertext>50cbc4c29...</ciphertext>
    <iv>2f7e6bc1e...</iv>
    <mac>8174cac91...</mac>
   </key>
  </keyblock>
 <nopush xmlns="urn:1and1:xmpp"/>
</message>

<message to="receiver" id="..." type="chat">
 <text keyblock="3c964b439..." xmlns="urn:1and1:xmpp:encrpyted">
  <ciphertext>9e615c66b...</ciphertext>
  <iv>f80b5ad4b...</iv>
  <mac>41582af32...</mac>
 </text>
</message>
```

The splitting into two messages allows to save space in the message directory and also to reduce performance impact on the sending and receiving clients. The first message containing the keyblock can be referenced by the sender for consecutive messages. For this the sending client only sends the second message, omitting the first one. The second message needs to contain a previously sent keyblock id. With this keyblock id the receiving client can use its local history or query the server history for the matching keyblock and use it to decrypt the message. Since keyblocks are not actual messages that are presented to the user they can not be deleted by the user. The client should send a new keyblock when one of the following conditions is met:

- time since last key block is more than 24 hours
- the receiving user(s) have changed
- one or more of the receiving users have changed their private key or the user has sent more than 100 messages using the same keyblock

The client will generate a different initial vector `IV` for each message that it sends out.

The FreeMessage encryption protocol provides *limited deniablity*: The sender and the receiver can generate the same initial key - no one else can generate that exact key. So a particular message encrypted with this key can only be from one of the chat partners. But there is no proof that it was actually sent by any one of them. Theoretically both of them have access to the same key, and both of them could have written the message. Because of this, no chat partner can prove that the other person wrote the message. The server-side message directory component knows which of the users is the actual sender. Yet this is no cryptographic proof that the message was sent by this person.

In order to mitigate risks from packet drop or replay attacks, the entire communication between the client and the server is be encrypted using TLS.

6 Group Messages

Groups are created by a single user. This user becomes the admin of the group. All groups are invite only and the admin is the only person that can invite other users into a group. Every user can query a list of his joined groups from the server. There is no functionality to query for open groups, since a user can not join without invitation. Joining is not optional, each user that is added to a group by the admin immediately becomes a member of the group. The admin is also the only person who can remove people from the group. All users have the opportunity to leave a group. All messages send to a group will be distributed to all participants. This is achieved by a server-side module iterating over all participants and sending copies of the original message to each member.
For group messaging the same approach as for Message encryption is used, only that there is one key entry in the keyblock per user. The key package for the user can be found by taking the key entry that contains the user's JID. The sending client can decrypt the message by taking one key entry at random. It can generate the key with each of them.

Implications of encryption to group chats:

- When a user is added to a group she will not get any old messages - akin to a real conversation, participants who joined later can not know what was spoken before.
- Even though the client constantly validates members and keys, there is the possibility, that the message can not be decrypted bye one or more clients.
- There is no explicit trust level for a group of users.

The client will get notifications if another user joins or leaves the group. When the client receives one of those notifications, it needs to refresh the participants list and sent a new key package along with the next message. When someone is removed from the group or leaves the group, the server will immediately stop routing messages to this user.

7 File Exchange

All files sent on FreeMessage are encrypted symmetrically with 256 bit encryption key `k` and a 128 bit initial vector `IV` generated at random by the client. Once encrypted, the file is uploaded into a document manager. The encryption key k is sent to the receiving client together with a reference and a store ID to generate a download link. The symmetric key is encrypted like the key of a regular message. For files in group chats the key will be encrypted once for every user in the group just like a regular message in a group. Since the key for the file is sent over a message and all the keyblocks are on the message, there is no way to tell who is the receiver or sender just by looking at the file.

There is also a meta field for each file in the message. This meta field contains the meta data of the file. This allows the client to “display the file” in the user interface to the user without downloading and decrypting the actual file. This field will be encrypted just like the file itself with the symmetric key. The reference ID and store IDs are later used to get a valid download URL from our server. These IDs will not be encrypted due to the fact that the server needs to clean up files when messages are deleted from the message directory.

Images and videos are encrypted just like any other file types - the only difference here is that the client will also send the thumbnails along with the actual file. This is needed because FreeMessage can not calculate the thumbnails on the server from the original since the server would need access to the (encrypted) file’s content. Downloading only thumbnails of sent videos and images instead of originals allows for a more responsive UI and less unnecessary data traffic. The message will contain one file URL for each thumbnail size we support, and one URL for the original file. The information about the contained thumbnails and the sizes are encrypted in the meta data. The client should be able to decrypt all meta data and decide which thumbnail will fit its screen size. All sent files (thumbnails and original) are encrypted using different random IVs.

The structure of such a file message containing an images would look as following:

```
<message to="receiver" id="...">
 <keyblock id="0573143c4..." xmlns="urn:1and1:xmpp:encrypted">
  <key jid="receiver" senderKeyId="0" receiverKeyId="1">
   <ciphertext>d3ac1af6b...</ciphertext>
   <iv>5d4e5529a...</iv>
   <mac>ade8c3aa6...</mac>
  </key>
 </keyblock>
 <nopush xmlns="urn:1and1:xmpp"/>
</message>

<message  to="receiver" id="...">
 <filetransfer keyblock="0573143c4..." xmlns="urn:
1and1:xmpp:encrypted">

   <meta>
    <ciphertext>9122d4e4e...</ciphertext>
    <iv>acd6f4ae4...</iv>
    <mac>70b2ccfd1...</mac>
   </meta>

   <file>
    <store>bf7d</store>
    <fileref>0e57</fileref>
    <iv>ba06ae9ea...</iv>
    <mac>6c7a84f89...</mac>
    <meta>
     <ciphertext>9122d4e4e...</ciphertext>
     <iv>acd6f4ae4...</iv>
     <mac>70b2ccfd1...</mac>
    </meta>
   </file>

   <file>
    <store>7529</store>
    <fileref>d7ba</fileref>
    <iv>31af3d64b4e...</iv>
    <mac>bf1827c8d4738...</mac>
    <meta>
     <ciphertext>186d9c16eb92...</ciphertext>
     <iv>26fb3d64b4e...</iv>
     <mac>bf1827c8d4738...</mac>
    </meta>
   </file>

   <file>
    <store>8f8b</store>
    <fileref>d78f</fileref>
    <iv>c55afc702...</iv>
    <mac>b6da99fca...</mac>
    <meta>
     <ciphertext>b29d6f1d3...</ciphertext>
     <iv>eca96d84e...</iv>
     <mac>230979195...</mac>
    </meta>
   </file>
  </filetransfer>
</message>
```

8 Client Implementation

Key Derivation

Android: For `AES`, `HMAC` and `PBKDF2` FreeMessage uses *Spongy Castle,* a *Bouncy Castle* fork for Android[2]. For compatibility reasons the Donna C implementation for `curve 25519`[3] is being used.

[2] https://github.com/rtyley/spongycastle

[3] https://code.google.com/p/curve25519-donna/

iOS: For `AES-256` and `HMAC-SHA256` FreeMessage uses the open source *Common Crypto* library provided by Apple[4]. For the elliptic curve the Donna C implementation of `curve 25519`[5] is being used.

Local Key and Data Storage

For storing sensible data on the clients, data is protected by an additional encryption layer. This layer is not really a bullet-proof security measure against stealing the private-key or the data if an attacker gets hold of the physical device. It is in place to make decrypting sensible user data in such a scenario significantly harder and more time consuming.

Android

For this encryption of local data FreeMessage uses a access key that can only be generated on the client. The access key can be created on the device without user interaction. For this a random secret is generated automatically on the device when the app is installed for the first time. After that on every launch the access key will be created with a key derivation function (KDF) - note that the KDF generates the same access key on every launch: The random secret and a fixed salt as well as a reasonably high number of iterations e.g. 10000 are used as inputs to the KDF. This way the access key can be recalculated each time the app is launched so that the access key is never stored in permanent memory. The application data will become unreadable when the app is uninstalled and the random secret is lost. Data is stored in a SQLite database located in the private data directory. Android enforces that other apps can not access this directory (if the device is not rooted).
Additionally the app uses SQLCipher[6] to apply a transparent `AES-256` encryption on top of the database. All media files are also `AES-256` encrypted

[4] http://www.opensource.apple.com/source/CommonCrypto

[5] https://code.google.com/p/curve25519-donna/

[6] https://github.com/sqlcipher/android-database-sqlcipher

using the access key mentioned above together with random initial vectors. Files are stored in the private directory.
The private key will also be protected by the random secret. Most Android phones offer full disk encryption and most recent models have it turned on by default. Therefore the data is also protected by an additional layer of encryption when the phone is turned off or locked. Explicit user input is not needed to decrypt sensible user data.
Random data on Android is generated using SecureRandom.

iOS

The private key is stored in the iOS Keychain which will grant access only to our application and only after the device was unlocked at least once after rebooting. The protection level will also make sure that the private key is not synced into the iCloud backup. The keychain protection level is called `kSecAttrAccessibleAfterFirstUnlockThisDeviceOnly` . All files and the core data database are stored in the private directory of the application. This private directory is only accessible to the FreeMessage application (unless the device is jailbroken). In addition to this FreeMessage uses the iOS Data Protection feature with the `NSFileProtectionCompleteUntilFirstUserAuthentication` setting. This will transparently encrypt and decrypt the files in the private application directory (and the database). All files will be unavailable when the iPhone restarts, since the users passcode is also used for encrypting and decrypting. The files can only be used after the user has entered his passcode for the first time after starting the device. Also all of the files will be excluded from iCloud backup by using the `NSURLIsExcludedFromBackupKey` attribute.
Random data on iOS is generated using /dev/urandom.

9 Limitations

FreeMessage uses a central message directory in which all messages for a certain account are stored in ciphertext which can only be decrypted with the the intended recipient’s private key (which in turn is stored on the device only). Storing messages anywhere else than on the client brings along drawbacks regarding the encryption protocol specifically excluding perfect forward secrecy for message encryption keys.

Forward secrecy implies that keys that are compromised in the future can’t be used to decrypt messages sent in the past. While forward secrecy of keys is an important attribute of a secure messaging system, it has strong negative implications on a server-side message history and multi-device usage (i.e. would effectively make those features impossible to build the way we want them to behave).

We explicitly made this trade-off decision and store encrypted messages in a server-side directory to account for user demand and behavior. The content shared over mobile messaging apps is important for users hold on to even in case of device loss and users expect messaging to work across different devices and platforms. The message directory allows to implement seamless and secure multi-device and backup solutions in the future that do not depend on the unencrypted message history being uploaded by the client to a potentially insecure cloud storage. Note, that the keys used to secure the transport layer are still perfectly forward secret.

Messages sent on FreeMessages are secure in the sense that neither 1&1 Mail & Media nor a third party can read the content. However, FreeMessage is not designed to offer anonymous messaging in the sense that all meta-data regarding the communication on the system is protected from the provider or 3rd parties.

10 Source Code

Protecting users' privacy and keeping FreeMessage secure is our biggest priority. This is why the source code of both clients (as well as the server components) is being reviewed by external experts on a regular basis. We are open to further 3rd party audits of our source code. Potential reviewers can request access by sending an email to review@freemessage.com.

Summary reports of past audits will be made available on freemessage.com shortly.

11 Conclusion

All content (regular messages, and files) shared on FreeMessage between any number of users is protected by state-of-the-art end-to-end encryption. Messages are protected, such that no 3rd party (1&1 Mail & Media as the provider, government institutions or attackers) can access the messages' contents. The relevant keys to decrypt the content are generated on the device. They never leave the device and are not accessed by 1&1 Mail & Media in any way. Users have the possibility to verify the integrity of their communication on FreeMessage.

FreeMessage is a service by GMX and WEB.DE. GMX and WEB.DE are operated by 1&1 Mail & Media GmbH.