BabelApp

Your secure communication platform

White Paper

Contents

1. PRODUCT SPECIFICATIONS

BabelApp is a platform for secure communication. It provides secure sending and storing of encrypted messages and documents. BabelApp is available on mobile (iOS, Android) and desktop (Windows, macOS) platforms.

BabelApp uses strong cryptography algorithms and protocols, based on international standards. The BabelApp platform is composed of servers and clients for mobile and desktop devices with different operating systems.

1.1 MAIN ASPECTS

BabelApp provides encrypted communication on supported devices and integrity checks among all the end-points involved in data transfer. As this document explains, BabelApp does not require the user to possess any digital certificates. The platform is based on servers (BabelApp servers) that securely communicate with client devices as well as among themselves through the internet.

1.1.1 SUPPORTED DEVICES

BabelApp client applications are available on all major mobile and desktop platforms.

- client for iOS
- client for Android
- client for BlackBerry
- client for PC with Windows
- client for Mac with OS-X

A user can have multiple registered devices, regardless of the types of registered devices.
Each device can be connected to multiple servers.

1.1.2 MAIN COMMUNICATION SERVICES

BabelApp servers provide central communication services:

- Central contact directory
- Distribution and synchronization of users' public keys
- Communication among multiple BabelApp servers for public key synchronization and cross-server communication
- Asynchronous delivery of messages and attachments to the recipient's devices
- Synchronization of sent messages to all of the sender's devices
- Temporary storage of encrypted attachments – until downloaded by all recipients
- Gateway for the sending of push notifications
- Distribution of Message sent, delivery and read receipts
- Distribution of "Undeliverable" notifications
- Communication gateway with REST API for easy integration with applications and programmable devices for encrypted message distribution and business processes automation

1.1.3 ADMINISTRATION FEATURES

BabelApp provides full control over the infrastructure to the company administrator.

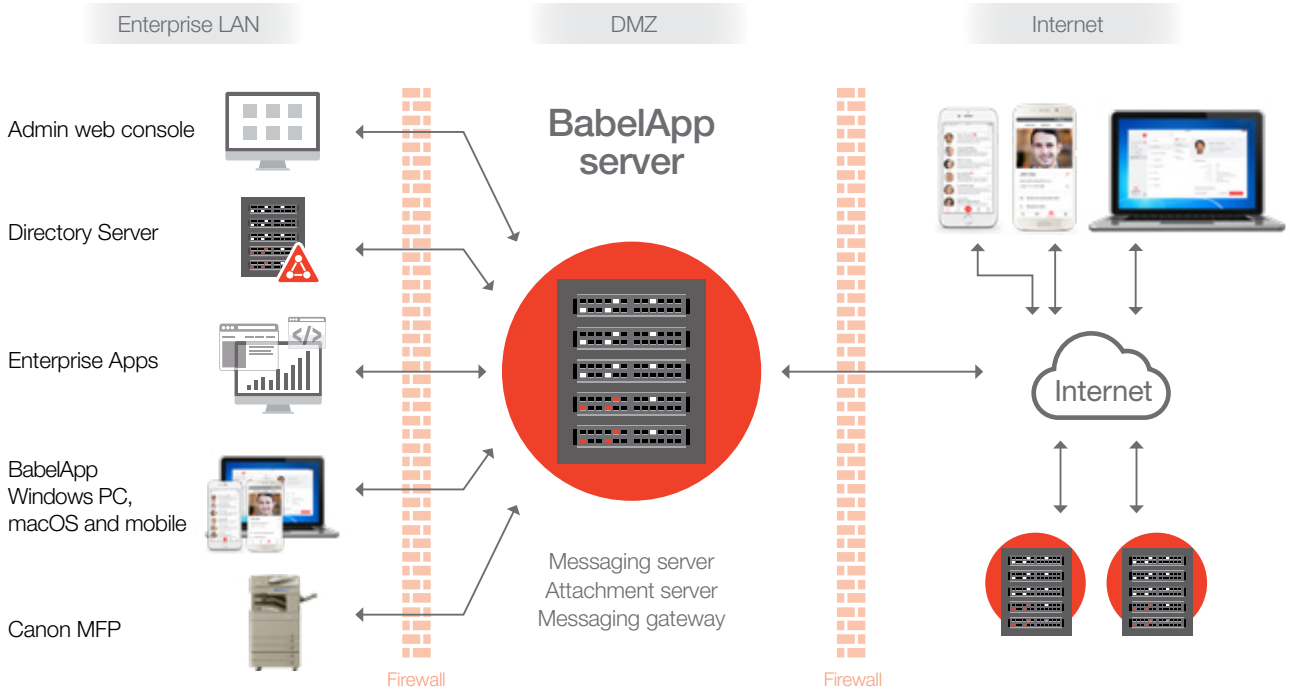
- Web console for system administration
- Import of User information from LDAP/AD
- Synchronization of user accounts changes directly from LDAP/AD
- Possibility to create and manage user accounts and groups for both internal and external users
- User device registration using a one-time password or LDAP authentication
- Removal of registered devices
- Key revocation
- User blocking
- Reset of server accounts on remote servers
- Deletion of a remote server from the local server database
- System logging of server and user events
- Traffic & usage statistics

2. SYSTEM ARCHITECTURE

BabelApp provides a robust business platform for encrypted communication between mobile devices and workstations including easy integration with business applications and multifunction printers/ scanners.

To be able to send messages to recipients who have accounts on different BabelApp servers than the sender, BabelApp implements communication among servers.

To achieve high server availability, the BabelApp platform supports cluster implementation resistant to failure.



2.1 SERVER COMPONENTS

The BabelApp server consists of a set of functional components:

- Asynchronous messaging service
- Service for attachment downloading
- Central directory of contacts
- Communication Gateway with API
- Gateway for alert distribution
- Scheduler of server-sent messages
- Platform management via web administration
- Personalized web pages for users

To support these services, the BabelApp server uses (via connectors) network directory services:

- LDAP connecter
- SSO connecter

2.1.1 ASYNCHRONOUS MESSAGING: MESSAGING SERVICE

The basic service of the BabelApp servers is the support of asynchronous messaging based on the XMPP protocol. Messages are in JSON format and contain all the metadata needed to process message delivery, integrity checking, transport of message encryption keys, attachment metadata etc.

Messages can be sent to multiple recipients and can have multiple parts, each part carries information about the protocol version for which the message is intended. The server checks recipient's addresses and protocol versions supported by the recipient's devices and delivers the suitable parts of the message with the same or the closest lower protocol version.

The server receives an asynchronous message and returns a confirmation receipt to the sender and stores the message until A) the server receives a delivery confirmation from the recipients' client application or B) until the message expiration date and time is reached. In either case, the message is deleted from the server right after.

Recipients can have multiple devices registered. The server will be trying to deliver messages to all of the recipient's devices. Messages are considered as delivered, if it has been delivered to at least one of the recipient's devices.

Once the recipient decrypts the message, the server sends a read receipt to the sender.

In case the message delivery was not successful, the server sends a „Message was not delivered" notification to the sender.

In case the sender has multiple devices registered to the account, the server always attempts to synchronize sent messages across all of the sender's devices.

2.1.2 ATTACHMENT DOWNLOAD SERVICE

The possibility to send documents in the form of attachments can be enabled or disabled by the BabelApp server administrator. Messages contain only attachment metadata. The attachments can be downloaded using a link sent along with the message. Attachments are encrypted in the same way as messages, please refer to 3.5.9. for more information. Administrators can setup the maximum size of attachments and maximum expiration period – time after which attachments are automatically deleted from the server, regardless of their delivery status.

2.1.3 ADDRESS BOOK

The server provides a central address book service and group management.

Contacts are senders and recipients of messages and their public keys are the basis of the cryptographic model for message encryption.

Roles are used to setup permissions to a group or multiple groups. Users have the permissions of all the roles assigned to them. Currently, two permissions are implemented:

There are two types of memberships:

- **List** – find contacts in a group with a partial match of their name.
- **Add** – add a group to the user's address book.

User's address books are created on the server and synchronized to all of the user's devices. Users can add contacts found through the "List" permission or contacts found using their BabelApp username (name#babelapp_server). User's address books eliminate the need to synchronize extensive amounts of contacts to all users, allowing effective adding of contacts and communication with them.

2.1.4 COMMUNICATION GATEWAY: API GATEWAY

Bi-directional distribution of encrypted messages and documents among all end-points requires a complete implementation of the BabelApp protocol, which is available as a part of BabelApp iOS, Android, Windows and macOS applications.

To allow for the distribution mechanism (one-way communication) of encrypted messages and documents even from a wide spectrum of applications and intelligent programmable devices, a communication gateway with a simple REST API has been developed as part of the BabelApp edition. This allows simple integration of the server with 3rd party applications. You can distribute encrypted messages and attachments from your company information systems and applications directly to devices with BabelApp client applications.

The integrated server application has a special account on the server, the so called API contact. The private AK key of the application is stored in an encrypted form in the communication gateway storage, which has the client part of the BabelApp protocol implemented and encrypts the data on behalf of the application. The application receives the API contact name and randomly generated authentication password which consists of alphabetical characters: {12346789ABCDEFGHJKMNPQRTUVWXY abcdefghjkmnpqrtuvwx}, with the length of 128 bits.

To store the D-H private key securely, the application generates a random salt and from the application password and the salt value the application generates the BK key, which is 128 bits long.

BK = PBKDF2[hmacWithSHA1]
(password, salt, iteration_number, 128)

Number of iterations: 1 000

The private AK key is encoded in the ASN.1 form with regards to the PKCS#8 (RFC 5208) specification and the final structure is aligned using PKCS#7 padding.

The private AK key of the application is encrypted using the BK key and the AES algorithm in the CBC mode. The initialization vector has a value of 0.

eAK = ENC-CBC[BK,0](AK)

A thusly encrypted eAK private key is stored along with the salt value in the database in the form of the ASN.1 structure.

The application communicates with the gateway using an SSL channel. The mechanism of successful private AK key decryption is used for authentication purposes. As soon as the gateway receives the decrypted private key from the AK application, it can encrypt messages using the same process as a BabelApp client application. Please refer to chapter 3.5.6.

2.1.5 MESSAGE SCHEDULER

The BabelApp server allows the administrator to schedule and automatically distribute messages to a selected user base. Messages, possibly with attachments, can be sent right away or at scheduled times.

Message distribution from the server uses the communication gateway technology, please refer to 2.1.4.

2.1.6 PUSH NOTIFICATION GATEWAY

Mobile devices with iOS and Android can receive push notifications, which inform users about messages that are waiting for delivery on the server. Push notifications can be received even if the application is running in the background.

Push notifications are distributed using the Apple/Google notification infrastructure. The BabelApp Push Notification Gateway only sends requests for a notification distribution to Apple/Google. Requests for push notification delivery must be electronically signed using the BabelApp Push Notification Gateway's private key and a certificate, which is registered with Apple's/Google's notification servers.

Use of push notifications can be enabled or disabled by the administrator.

Every BabelApp server has its own private key and certificate issued by the OKsystem a.s. certification authority and is used to sign all requests which are sent to the Push Notification Gateway. The Push Notification Gateway (after verifying the signature) creates and signs a new push notification request on behalf of the sender's device and sends it to an appropriate Apple/Google notification server.

Push notification requests and the notifications themselves do not contain any data about the actual messages.

2.1.7 WEB ADMIN CONSOLE

Administration of the BabelApp server is done using a web admin console, which is published on a web address, e.g. https://babel.domain:port. Each domain consists of a DNS organization name, where the BabelApp server is placed.

The default port is 9091. A different port address can be chosen during the installation. Each server must have a certificate installed for the domain. Certificates are issued by the OKsystem a.s. certification authority based on certificate requests created by BabelApp administrators.

2.1.8 CLIENT DASHBOARD

Users, who have an account registered with a BabelApp server, have their personal BabelApp web pages. The status of their devices can be viewed and new device registration requests can be created on this page.

All requests must be approved by the company administrator. Once approved, a QR code with an OTP is displayed on the personal web page for easy authentication of the new device.

Administrators can set up a URL for easy access to the personal web pages and enable authentication using SSO.

2.2 COMMUNICATION AMONG BABELAPP PRO SERVERS

BabelApp users can have accounts on more than one server, which allows them to directly communicate with all the contacts registered on such servers. Every user can decide which server will be used as default.

BabelApp cross-server communication has been developed to allow users to communicate also with contacts on different servers.

If a user is registered to multiple servers, one of them will facilitate communication with other servers.

2.2.1 BABELAPP NAME

Every user is identified in the BabelApp network, using a unique BabelApp address:

babelname#babelapp_server

Where: **Name – user (account) name in the BabelApp server account database.**

babelapp_server - fully qualified DNS name (FQDN) of the BabelApp server, for which a digital certificate has been issued by OKsystem.

A DNS type A record must exist for the babelapp_server name.

SRV records are used for the communication ports of the BabelApp servers. If non-existent, implicit ports are used. The implicit port for BabelApp client application connection is 5222.

Example:

BabelApp server with FQDN babel.oksystem.cz is accessible on the internet address 193.222.130.33

TCP port for client application connection to the _babel service is 5222

Corresponding DNS records are:

babel.oksystem.cz A 193.222.130.33

_babel._tcp.oksystem.cz. 86400 IN SRV 10 10 5222 babel.oksystem.cz

2.2.2 REGISTRATION AMONG MULTIPLE SERVERS

For cross-server communication, it is necessary to register servers' accounts (similarly to user account registration). Server administrators can deny the registration of one, more or all external servers and therefore not allow cross-server communication with such servers.

If it is desired for server S1 to be registered to server S2, server S1 sends a registration message with the SERVER type to server S2. Server S2 verifies that server S1 has a valid certificate signed by OKsystem and issued for the domain listed in the registration message. If the certificate is found to be valid, server S2 also verifies that server S1 is not on the S2 block list. Block lists are managed by BabelApp server administrators.

2.2.3 SYNCHRONIZATION OF CONTACTS AMONGST MULTIPLE SERVERS

A registered sever S1 can request Address Book updates from server S2 and obtain public keys from all server S2 contacts to enable server S1 to encrypt all the metadata about messages from S1 contacts intended for S2 recipients.

Server S1 can thereafter send metadata information to server S2, which then informs its recipients about messages that are available to be downloaded directly from server S1.

2.2.4 SENDING AND RECEIVING A MESSAGE

User device U1#S1 sends message Z to its implicit BabelApp server S1. Server S1 stores message Z and creates a new message M which only contains metadata information about the sender, conversation and message expiration.

Consequently, server S1 delivers the message M via Server S2 to the contact U2#S2. Once the message M is delivered, the contact U2#S2 is then able to retrieve the URL and message Z identification. Contact U2#S2 then connects to the given URL and authenticates itself by proving that it possesses a private key that can decrypt message Z. Once verified, message Z is downloaded to the device.

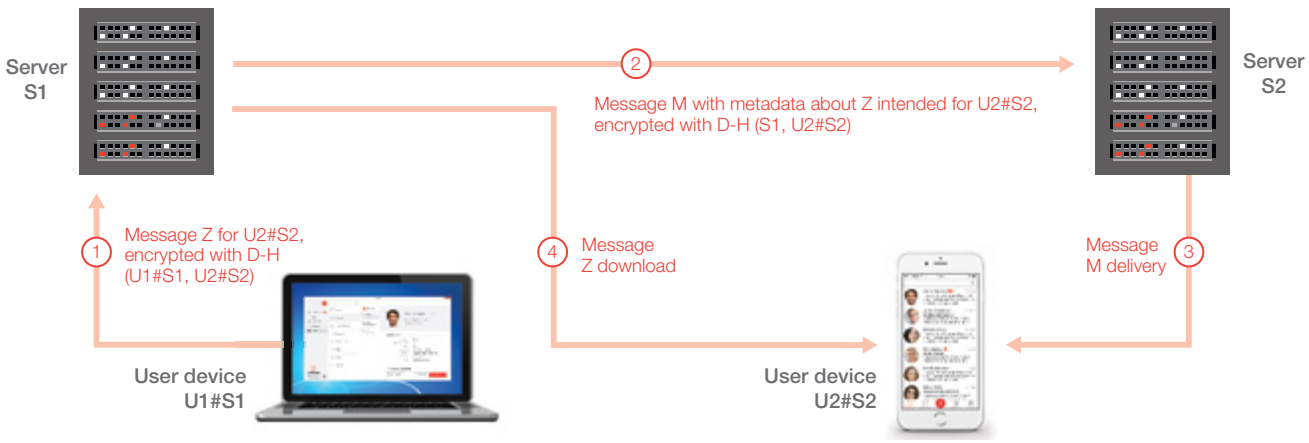
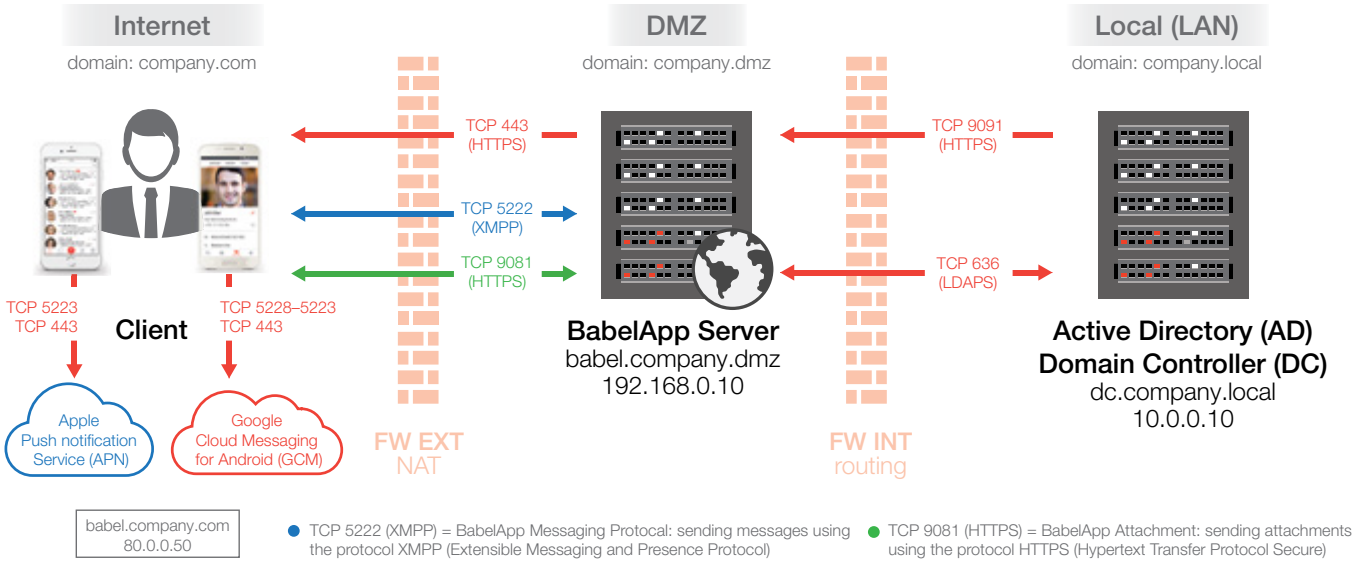


Diagram describing cross-server message delivery

2.3 NETWORK AND COMMUNICATION

BabelApp server is typically located within a so-called "Demilitarized zone" of corporate networks, which is connected to a router and firewall system with an internal network and internet. Routers can use Network Address Translation (NAT) to access the Internet. Firewall settings must allow connections for a combination of protocol/IP address/port between BabelApp server and an internal network or Internet. Addresses and ports listed in the following picture are only for demonstration purposes, specific installations may be different.



2.3.1 VIRTUAL BABELAPP SERVERS AND SNI

BabelApp servers and clients support SNI (Server Name Indication) technology within TLS communication – multiple virtual BabelApp servers / domains can run using the same public IP address. SNI is based on the mechanism of TLS expansion within which the BabelApp client provides the TLS server with a DNS name of the virtual BabelApp server to which the communication will be re-directed.

Support of the SNI technology allows for BabelApp server hosting in the cloud.

SNI technology is described in detail in RFC 4366 and RFC 6066.

2.3.2 BABELAPP CLIENTS

BabelApp clients are available on all major mobile and desktop platforms; please refer to 1.2.1 for more information. Each user can have more than one device registered under their account.

Every BabelApp client application has the complete application protocol for encrypted communication and encrypted storage of data implemented on the device.

BabelApp clients communicate with BabelApp servers using cellular data, wifi or company LAN/wifi. To communicate with a server, devices must be registered to a BabelApp server under a user's account and must have the DNS of the BabelApp server and communication port set correctly.

BabelApp clients can be registered to multiple servers, but one is always set as default.

Mobile BabelApp client applications can send and receive encrypted messages via SMS. In this case, it is not possible to send or receive documents. When sending SMS messages, there is no communication between the client application and BabelApp server. The length of such message is not limited to a maximum number of characters. Sending encrypted SMS messages is useful when no data connection is available or if using other than standard data communication channels is desired.

3. CRYPTOGRAPHY DESIGN

3.1 BASIS FOR THE CRYPTOGRAPHY DESIGN

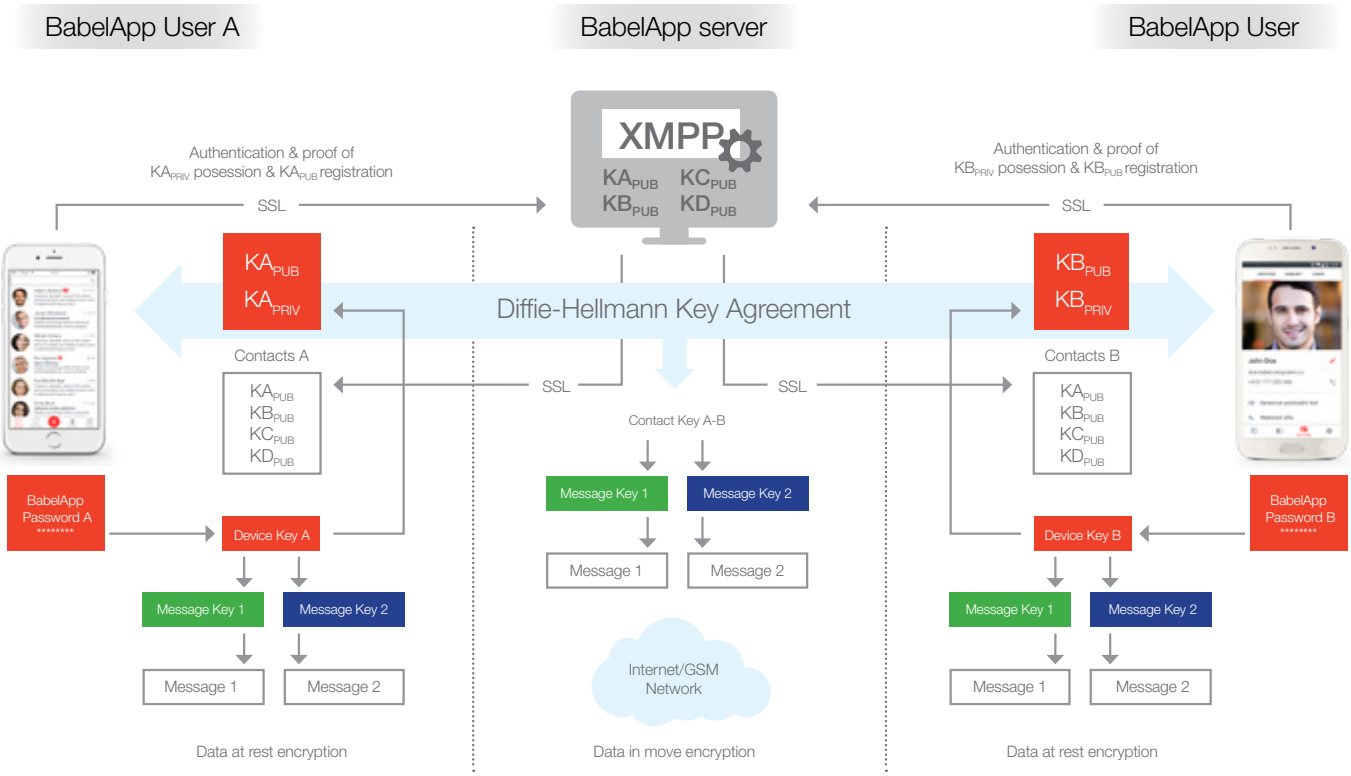
During the cryptography design, we worked with the following requirements:

- The major goal was to secure the content of the communication, not the fact that the communication actually took place.
- Application encryption will be used in between the end-points during the data transfer.
- Application encryption will be used for data storage on devices.
- User public key certificates or device certificates will not be used.
- The server will be used for:
  - user account administration
  - distribution and synchronization of public keys
  - asynchronous communication among devices with BabelApp application
- Server does not poses any keys that can be used to decrypt messages.
- Server can only access information about users, devices and message metadata
- Transported messages will not be stored on BabelApp servers for a longer period of time than it is necessary for successful message delivery
- Servers are under the organisations' own administrations
- Users can have more than one device (e.g. smartphone, tablet, PC, laptop…) – messages will always be sent from one device but synchronized to other devices under the account
- Standard strong cryptography algorithms and recommended parameters and operation modes will be used
- Techniques for elimination of active attacks will be used – checks of integrity, authenticity and message sequence – strictly before any attempt to decrypt messages

3.2 CRYPTOGRAPHY MODEL

The following diagram describes the conceptual cryptography model. The model describes the application encryption of data and does not contain integrity checks on the JSON level of the messages.

The TLS protocol is used for all communication between the clients and the server, as well as the application encryption. System services for data encryption are used for data storage on the user devices.



3.3 CRYPTOGRAPHY ALGORITHMS

BabelApp uses the following cryptography algorithms:

- Diffie-Hellman according to NIST SP 800-56A
- AES 128, AES 256 according to paragraph 5 FIPS PUB 197
- PBKDF2 according to PKCS#5 v2
- SHA-2 according to FIPS PUB 180-4, paragraph 6.2 with a 256 bit thumbprint
- HMAC according to RFC 2104 and extension for the use of hash algorithms SHA2 according to RFC 4868

The algorithms mentioned above were used in the hereunder described cryptography schemas and protocols.

3.4 CRYPTOGRAPHY PROTOCOLS

With the use of keys and cryptography algorithms the hereunder standard protocols and application encryption were implemented for communication and device storage.

3.4.1 COMMUNICATION AMONG BABELAPP CLIENT APPLICATIONS AND SERVERS

Communication among BabelApp client applications and BabelApp servers takes place within the Transport Layer Security protocol (TLS v1.2.) according to RFC5246. To establish TLS communication, the servers is equipped with a digital certificate issued by OKsystem. Such a certificate is part of the BabelApp application source code on all platforms to ensure server authenticity.

JSON messages with integrity checks are sent as part of the TLS connection (please refer to 3.5.7.) and transport (among other information) application encrypted messages and attached documents between the end-points (please refer to 3.5.6. and 3.5.9.).

3.4.2 CLIENT REGISTRATION TO BABELAPP SERVER USING OTP

The registration of client devices uses One Time Password Protocol authentication, which is based on a randomly generated password and password imprint derivation using PBKDF2 according to NIST sp800-132.

3.4.3 CLIENT REGISTRATION TO A BABELAPP SERVER USING AD SSO

BabelApp client allows for alternative registration of company users, who have an account in a company LDAP directory (typically Active Directory), based on the Single Sign On system.

3.4.4 PROOF OF THE D-H PRIVATE KEY POSSESSION

The proof of possession of the private key to the presented Public Key value is done during the user device registration to the BabelApp server based on the Diffie-Hellman Proof of Possession according to RFC 6955 protocol.

3.4.5 CLIENT AUTHENTICATION TO A BABELAPP SERVER

A random authentication password AH is generated on the BabelApp server, based on the client device's registration. The password's 160 bit long authentication imprint AS, derived using PBKDF2:

AS = PBKDF2[hmacWithSHA1]
(password_AH, salt, number_of_itterations, 160)

The authentication password is sent within a TLS session to the user's device to be used for future client-server authentications.

BabelApp clients automatically authenticate themselves to registered servers at the moment of launch, assuming that a connection to the server can be established. User authentication to the client is not required for the client to authenticate itself to the server.

3.4.6 KEY GENERATION

Generation of random cryptography material (encryption keys, authentication keys, DH key pairs) is based on random number generators dependent on client platforms.

3.4.7 DERIVATION OF KEYS FROM PASSWORDS

Keys can be derived from passwords using the PBKDF2 function according to PKCS#5 v2, using a pseudorandom HMAC function with SHA256 and a high number of iterations.

3.5 APPLICATION COMMUNICATION ENCRYPTION

Application encryption forms the basis of security of communication between BabelApp clients and local data storage on user devices. Application encryption is based on a combination of symmetrical cryptography algorithms with secret keys and non-symmetrical cryptography with public keys. Implementation of the cryptography model (3.2.) is described in detail below.

3.5.1 CONTACT KEY AGREEMENT

The Diffie-Hellman protocol (described in RFC 2631 and ISO standards 14883-3, ANSI X9.62 and NIST SP 800-56A) is used to reach a key agreement.
Implementation of the Diffie-Hellman protocol in BabelApp uses multiplicative integer group modulo p, with parameters p, g and q, according to RFC 5114, paragraph 2.3:
the bit length of multiplicative group of order p is 2048-bit
the bit length of prime number multiplicative subgroup generated q of order p is 256-bit

3.5.1.1 DOMAIN PARAMETERS

Values of domain parameters p, q, q (hexadecimal system):

p = 87A8E61DB4B6663CFFBBD19C651959998CEEF608660
DD0F25D2CEED4435E3B00E00DF8F1D61957D4FAF7DF45
61B2AA3016C3D91134096FAA3BF4296D830E9A7C209E0
C6497517ABD5A8A9D306BCF67ED91F9E6725B4758C022
E0B1EF4275BF7B6C5BFC11D45F9088B941F54EB1E59BB
8BC39A0BF12307F5C4FDB70C581B23F76B63ACAE1CAA
6B7902D52526735488A0EF13C6D9A51BFA4AB3AD834
7796524D8EF6A167B5A41825D967E144E5140564251CCAC-
B83E6B486F6B3CA3F7971506026C0B857F689962856DE
D4010ABD0BE621C3A3960A54E710C375F26375D7014103
A4B54330C198AF126116D2276E11715F693877FAD7EF09
CADB094AE91E1A1597

g=3FB32C9B73134D0B2E77506660EDBD484CA7B18F21
EF205407F4793A1A0BA12510DBC15077BE463FFF4FED-
4AAC0BB555BE3A6C1B0C6B47B1BC3773BF7E8C6F629
01228F8C28CBB18A55AE31341000A650196F931C77A57F2
DDF463E5E9EC144B777DE62AAAB8A8628AC376D282D6ED
3864E67982428EBC831D14348F6F2F9193B5045AF2767164
E1DFC967C1FB3F2E55A4BD1BFFE83B9C80D052B985D182
EA0ADB2A3B7313D3FE14C8484B1E052588B9B7D2BBD2DF-
016199ECD06E1557CD0915B3353BBB64E0EC377FD02837
0DF92B52C7891428CDC67EB6184B523D1DB246C32F6307
8490F00EF8D647D148D47954515E2327CFEF98C582664B-
4C0F6CC41659

q=8CF83642A709A097B447997640129DA299B1A47D1EB-
3750BA308B0FE64F5FBD3

3.5.1.2 DH KEY GENERATION

Every BabelApp user application generates a private key X (random number 1 < X < q - 1), which is kept in secret. A corresponding public key is calculated as Y = g^X mod p.

Comment 1:
Users generate a DH key pair on every device which is registered to their account on BabelApp servers.
In case the user registered a device to the server before, he is allowed to overwrite the original keys with a new set of keys, remove previously registered devices and revoke the original key or transfer the original DH key to the newly registered device.

For a detailed description of the secure key transfer between devices please refer to 3.5.11.

Comment 2:
Private key X is encrypted on the device using a device DK key, as described in chapter 3.6.5.

Example 1:
User A generates a random private key Xa and calculates a public key Ya using domain parametrs (p, q, g) a Ya = g^Xa mod p

Xa = 1973D625770FCD1058C9474213DD3E7905DEAA4
A2226D2FA26A08504D8700ABF

Ya = 3415B6FD8C2531D0DDC5E38A7FFD6E0D03EB5446
94B6A74D0B768B57D1CEC6A9D8B18F6BB920B6382ED
FCF113DDEB549E5BC272F031984514C2F87435A7B0668
F850A21B82E2C1EDADE3D2BE7358BB97859B0A22B0134
AB457C77EC6D3308188C197BD4D8FFE4D698DC4805715
E049043290B2EE11D9F8F834326F11460C56A743925D29
C1A862D51BF13556F1D9A44A1D1EDC80E78052AC97A4A
2D998045EFEDCF6C3463D88C69BA9CA904CD53BA5E0D7
6313A29FC1CB307385FE047B7FBD176C1BB35D57F9B7C4
AA63B670164BA59A4EB7F1ACB525C55F74C454E0F3888F
79152A7127E6976CCEF821391591005551064217193538
448317302609964318259122758009CEF82139159100555
106421719353844831730260996431825912275800 9

3.5.1.3 PUBLIC KEY VALUE SERVER REGISTRATION

Public DH key value registration is part of the first user's device registration to the server. The DH value is generated during the installation of BabelApp. Users need to be successfully authenticated for the registration to succeed. Based on the server settings, users can be authenticated using a name and One Time Password (typically scanned from a QR code), or using LDAP authentication.

After a successful sign in every device needs to cryptographically prove that is possesses the private part of the DH key which was sent to the server. This proof is based on Diffie-Hellman Proof of Possession according to RFC 6955.

After a successful registration the persistent authentication data is returned by the server to the client for future sign-ins.

Users' registered public keys are provided to all devices, which meet the requirements of user visibility in the address book and have requested synchronization of the contact.

3.5.1.4 FINDING THE SHARED SECRET Z BETWEEN USERS A AND B

Assumptions:
Users A and B share domain parameters (p, q, g) and obtain each other's public keys.

User A has a private key Xa and public key Ya
User B has a private key Xb and public key Yb

Calculations:
User A calculates the value of the shared secret Z = Yb^Xa mod p

User B calculates the value of the shared secret
Z = Ya^Xb mod p

Example 2:
User B calculates the value of the shared secret Z based on the private key Xb value and public Ya value using Z = Ya^Xb mod p

Xb = 7433D90B61EA231F2350ADE584EE047D-
DC8116D077BB6B6977CAAE2DDE399545

Z = 1D0D34D49D5F613892611DC620D66D80F2690249D
474248B4CA4863D2E5EA2F722E9C98B91D70BC0E791BC3
AFB5F105F518E749FDAC9A0374DD340B8D369409BAB061
EEE708672F6954883F4D21311A5331E6DABA2E4EB620DB
DCA8343F7033E8BB3C1929DD406250D4E4AAECE1B063
CABB5B82966B53AFBA82DEEFC969D24888B44CCE8ECCC
3F5F1BE3D4CE1AC5E10A9F38121ABDA08F55301E5495A6
AD78C80562E501DFA51DEACEEDFB5698722AC9FE7B971
6F60956C0B06EA069C91AFBC26C07831F7FF0764F2818
A54FFD849D4E679BD4F580E20CBD3FAB0BE831020C67CC
4350B996DDDCFC8D88021AF0804C4099C3A04676E52A
EAA13A7212A3640AF2

3.5.2 DERIVATION OF THE KEY MATERIAL FROM THE SHARED SECRET Z

To derive the key material (contact key and integrity check key), it is necessary to use a cryptography function for key generation with necessary length from value Z.
For this derivation we use the algorithm described in RFC 2631, chapter 2.1.2
It is possible to generate a practically unlimited amount of key material KM from the Z value using the following algorithm:
KM = H(Z || OTHER_INFO)
Where H is a hash function and OTHER_INFO is an ASN. 1/DER structure.

3.5.2.1 CONTACT KEY

Contact key CK has a length of 128 bit (AES).

The used hash function H is SHA1. From the 160 bit long hash value, we use 128 bits from the beginning as a CK key value.

Structure of OTHER_INFO is as follows:

```
static uint8_t DER_OTHER_INFO[] = {
0x30, 0x1B, // sequence length 27
 // keyInfo
0x30, 0x11, // sequence length 17
0x06, 0x09, 0x60, 0x86, 0x48, 0x01, 0x65, 0x03,
0x04, 0x01, 0x02,
// algorithm OID 2.16.840.1.101.3.4.1.2 AES 128 CBC
0x04, 0x04, 0x00, 0x00, 0x00, 0x00,// counter
0xA2, 0x06, 0x04, 0x04, 0x00, 0x00, 0x00, 0x80 //
suppPubInfo EXPLICIT
};
```

Example 3:
CK = sha1(Z || '30 1B 30 11 06 09 60 86 48 01 65 03 04 01 02 04 04 00 00 00 00 A2 06 04 04 00 00 00 80')

The CK key value for the shared secret Z mentioned in example 2:
CK = 5FAD22F98B27648BE55F0B40D58E4FA0

3.5.2.2 KEY FOR INTEGRITY CHECKS

Key for CK_{hmac} integrity checks has a length of 256 bit.

The algorithm for the message integrity control is HMAC with SHA256,

OID = {iso(1) member-body(2) us(840) rsadsi(113549) digestAlgorithm(2) hmacWithSHA256(9)}

The used hash function H is SHA256, the entire hash value is used for CK_{hmac} key value.

Structure OTHER_INFO is as follows:

```
static uint8_t DER_OTHER_INFO_HMAC[] = {
0x30, 0x1A, // sequence length 26
// keyInfo
0x30, 0x10, // sequence length 16
0x06, 0x08, 0x2A, 0x86, 0x48,  0x86, 0xF7, 0x0D,
0x02, 0x09,
// algorithm OID 1.2.840.113549.2.9 – PBKDF2 HMAC
with SHA256
0x04, 0x04, 0x00, 0x00, 0x00, 0x01, // counter
0xA2, 0x06, 0x04, 0x04, 0x00, 0x00, 0x01, 0x00 //
suppPubInfo EXPLICIT
};
```

Example 4:
CK_{hmac} key value for shared secret Z mentioned in example 2:

CK_{hmac} = F180BE4E4196A7FE9AA3090046F732DFC
12D2526F0CC4E3996D842F5230909F2

3.5.3 MESSAGE KEY (MK) GENERATION

For every message a random MK key is generated = RND (16) of 128 bit length. Platform based random number generators are used for key generation.

3.5.4 DATA PADDING

Data is padded to become an undivided multiple of 16 octets (byte padding) prior to the encryption process.

Byte padding is done according to PKCS#7, described in RFC 5652, chapter 6.3. – messages are padded by concatenation with (16 – (length_message mod 16)) octets with value (16 – (length_message mod 16)).

Example 5:
If the data is 1 byte long, it is concatenated with 15 bites '0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F '

If the data is 16 bytes long, it is concatenated with 16 bites '10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10'

If the data is 26 bytes long, it is concatenated with 6 bites '06 06 06 06 06 06'

3.5.5 KEY ENCRYPTION

For message MK key encryption of 128 bit length, an AES128 algorithm is used (FIPS PUB 197) in mode ECB according to NIST SP 800-38A, chapter 6.1.

Padding is not used before key encryption. Keys have a fixed length of 128 bits.

Initialization vector is not used in the ECB mode

eMK = ENC-ECB[CK](MK)

3.5.6 MESSAGE ENCRYPTION

3.5.6.1 MESSAGE PREPARATION

All text messages T are converted to the Unicode character set with UTF-8 codding according to RFC 3629, so that text compatibility is achieved among all platforms.

Before encryption, a time stamp TS is taken (number of milliseconds from midnight 1.1.1970 UTC). This timestamp is coded as TS', 4 octets with „big endian" bite organization to achieve compatibility among all platforms. TS' are added to the end of messages.

Messages with TS' stamps are padded according to PKCS#7.

3.5.6.2 ENCRYPTION

A random MK key is generated for every message T. MK = RND(16) of 128 bit length.

For message text encryption an AES-128 algorithm is used (FIPS PUB 197) in mode CBC according to NIST SP 800-38A, chapter 6.2.

Initialization vector IV has the value of 0 (IV = '00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00') for all messages. Use of an identical value IV for all encrypted messages is not a problem because every message is encrypted with a different, randomly generated key.

eT = ENC-CBC[MK,IV](utf8(T) || TS')

Example 6:

- T is an open message text
- MK is a message encryption key
- CK is a contact key calculated in example 3
- eMK is a key used for message encryption and is encrypted with CK contact key
- TS' is an encoded time stamp
- eT is an encrypted message

T = "Privacy exists"
utf8(T) = 507269766163792065786973737473123456780E0E0E0E0E0E0E0E0E0E0E0E0E
MK = 11121314212223243132333441424344 (hex string)
CK = 5FAD22F98B27648BE55F0B40D58E4FA0
eMK = 874B7E0985282C274F63161987BBC09D
TS' = 12345678
utf8(T) || TS' || padding = 507269766163792065786973737473123456780E0E0E0E0E0E0E0E0E0E0E0E0E0E
eT = B8EB1C3D985D2D92A8C574C85ABE49B7C1FF8BCAC2B60E1128AF0AB5E2BD0182

3.5.7 MESSAGE INTEGRITY

Two independent mechanisms are used to ensure integrity, authenticity and correct message sequencing, which are all important components in detection and elimination of active attacks.

- Cryptographic authentication HMAC code of the entire JSON message
- Sequence numbering of sent messages

3.5.7.1 AUTHENTICATION CODE HMAC

Encrypt-then-MAC mechanism is used. Encrypted data is provided with an authentication code calculated using the HMAC-SHA256 algorithm based on the HMAC definition according to RFC 2104 and the extension for the use of hash algorithms SHA2 according to RFC 4868.

MAC = H(K XOR opad, H(K XOR ipad, text))

where

H is a hash function SHA-256, according to FIPS PUB 180-4, chapter 6.2
Ipad is a sequence of 64 octets with the same value 0x36
Opad is a sequence of 64 octets with the same value 0x5C
K is a CK_{hmac} 256 bit key

Note: Original HMAC specification described in RFC 2104 uses hash algorithm MD5 or SHA1.

3.5.7.2 MESSAGE NUMBERING AND IDENTIFICATION

Every message M is identified within a XMPP/JSON message using 256 bit identification messageId, which is randomly generated.

Message sequence is numbered with a 128 bit sequenceId, which is randomly initialized and incremented with every sent message.

3.5.8 RECEIVING AND DECRYPTING A MESSAGE

3.5.8.1 INTEGRITY CHECK OF RECEIVED MESSAGES

Before messages can be decrypted, the integrity of a JSON message M is checked using an HMAC calculation and via comparison with the relevant value of the authentication message code.

HMAC' = HMAC-SHA256(CK_{hmac}, M)

Should the values of the sent authentication code and the value of the authentication code calculated by the recipient not match, an active attack would be detected.

Such an attack might be in a form of a changed or fake message. In that case the recipient's BabelApp application would not decrypt the message (to prevent a possible data leak via the side channels when displaying error messages) and would display a warning.

If the recipient receives a message with an incorrect sequence number, a warning is displayed.

3.5.8.2 MESSAGE DECRYPTION

Recipients first decrypt the message MK key with which the message was encrypted. It is done using the CK key, which is shared by the recipient and the sender.

$MK = DEC\text{-}ECB[CK](eMK)$

Recipients decrypt the message eT using MK, initializing vector IV = 0

$T' = DEC\text{-}CBC[MK,IV](eT)$

By cutting off the last 4 bites T'(time stamp) and decoding the UTF-8 you get the message T.

3.5.9 ATTACHMENT ENCRYPTION

BabelApp allows users to send encrypted attachments (documents of any type). Attachments are encrypted using the same MK key as the message itself, however, attachments and messages are sent separately. Attachment metadata is sent along with the message.

- Non-encrypted metadata – attachment identification, attachment hash and a length of the attachment in bites.
- Individually encrypted metadata – type, name and miniature (e.g. an attachment preview)

Metadata is encrypted with the same MK key as the message.

Messages can contain metadata about more than one attachment.

The sender generates a random initialization vector for attachment encryption:

$IV_{Data} = RND(16)$

The sender encrypts Data using the MK key and IV_{Data} and attaches the IV_{Data} in front of the encrypted data.

$eData = IV_{Data} \,||\, ENC\text{-}CBC[MK, IVData](Data)$

The sender calculates hash of eData for integrity check:

$hash = sha256(eData)$

The sender calculates the length of eData in bites :

$length = len(eData)$

The sender uploads the eData to the server and receives identificator ID_{file}.

The sender generates 3 random initialization vectors for the encrypted metadata:

$IV_{Name} = RND(16)$

$IV_{Type} = RND(16)$

$IV_{Thumbnail} = RND(16)$

The sender encrypts the type, name and attachment miniature using the MK key and the corresponding initialization vector IV_{Type}, IV_{Name}, $IV_{Thumbnail}$:

$eName = IV_{Name} \,||\, ENC\text{-}CBC[MK, IV_{Name}](utf8encode(Name))$

$eType = IV_{Type} \,||\, ENC\text{-}CBC[MK\ IV_{Type}](utf8encode(Type))$

$eThumbnail = IV_{Thumbnail} \,||\, ENC\text{-}CBC[MK, IV_{Thumbnail}](Thumbnail)$

The sender sends out attachment metadata ID_{file}, eName, eType, eThumbnail, hash and length via the JSON message with an integrity check (please refer to 3.5.7).

3.5.10 ATTACHMENT DECRYPTION

Should a message with attachment metadata be received, the attachment identificator IDfile is used to identify and download the encrypted attachment data eData from the server.

The recipient calculates the hash of eData and compares it with the corresponding hash value, which is part of the message metadata.

$hash' = sha256(eData)$

$hash' == hash$; Should the values vary, the recipient's application interrupts the processing, cancels the attachment and displays a warning

The recipient separates the initialization vector IVData from the first 16 bites of edata:

$IV_{Data} \,||\, eData' = eData$

and uses the MK key to decrypt the eData' and obtain the decrypted values of Data:

$Data = DEC\text{-}CBC[MK, IV_{Data}](eData')$

3.5.11 KEY TRANSFER BETWEEN USER'S DEVICES

Every BabelApp user can have multiple devices registered under their account. All the devices share the D-H value of the user's key pair.

For better understanding, the newly registered device will be referred to as the ND and the previously registered device as the OD.

The user generates DH key pair on all of their devices and registers them with his/her account on the BabelApp server.

If the server has any devices already registered under the account, the server returns a warning message along with a contact GUID and the public part of the DH key.

The user can choose whether they prefer to transfer the previous DH key to the newly registered device (ND), or to re-write the previous keys with new ones which would disable all previously registered devices and revoke the old keys.

3.5.11.1 KEY TRANSFER FROM THE OD TO THE ND

It is presumed that the already authenticated user has access to both the OD and ND and that both devices have internet

connection and are on-line. For the key transfer to happen it is necessary to type a 5-digit code displayed on the ND to the OD so that the potential attacker would need to possess not only the server authentication information but also both devices in order to perform a successful attack.

If the user chooses to transfer the old keys to the newly registered device (ND), an authentication PIN is generated on the ND using the shared secret Z calculated based on the private part of the DH key pair of the ND and the public part of the DH key pair of the OD sent by the server to the ND in the error message. The calculation of value Z is an analogical process described in chapter 3.5.1.4.

The 5-digit PIN value is calculated from the Z value:

PIN = SHA1(Z) mod 100000

The server requests the DH key pair transfer from the OD and as a part of the request also sends the public part of the DH key of the ND. The OD prompts the user to enter the authentication PIN, which was generated and displayed on the ND. The user enters the PIN to the OD, which independently calculates the value from the same data as the ND and compares it with the PIN value entered.
In case the values match (PIN´ = PIN), the following steps are taken by the OD:

a) Calculates the shared secret Z based on the private part of the DH key pair of the OD and the public part of the DH key of the ND. The calculation of the value Z is an analogical process described in chapter 3.5.1.4.
b) Generates a 256 bit AES encryption key for private key encryption.
c) Encodes the DH private key of the OD and its identification GUID into ASN.1 structure as specified in PKCS#8.
d) Analogically encrypts the DH key pair using the message key as described in chapter 3.5.6.
e) Provides it with an HMAC signature, analogically, as described in chapter 3.5.7.1.
f) Sends it out via the server to the ND.

The ND checks the integrity and authenticity of the message by verifying the HMAC signature validity and if successful, proceeds with the following:

a) Analogically decrypts the DH key pair, as described in chapter 3.5.8.2.
b) Compares the GUID with the value received in the server error message sent during the registration process.
c) Verifies that there is an exact match in between the public part of the DH key pair and the value used for PIN calculation.

In case the entire check process is successful, the ND deletes its generated DH key pair and substitutes it with the newly received DH key pair.

The ND sends a new registration request to the server, this time with the received (OD) DH key pair.

3.5.11.2 OVERWRITING THE KEYS IN USE WITH NEW ONES

If a user chooses to overwrite the keys in use with a new set of keys, the server deletes all their previously registered devices and revokes the keys.

3.6 APPLICATION ENCRYPTION OF DATA STORED ON BABELAPP DEVICES

Data is stored on BabelApp devices in an encrypted form. Data means messages, attachments, encrypted message keys and encrypted device keys – which are used to encrypt / decrypt message keys and private parts of DH key pairs.

3.6.1 SQLITE DATABASE

All conversations (sent or received) are stored in the SQLite database. All items are saved into the database in an encrypted form, as described in 3.6.9. All documents (sent and downloaded attachments) are stored on devices as encrypted files in the BabelApp sandbox file system, as described in 3.6.9.

3.6.2 USER PASSWORD

The foundation of the BabelApp application security is a strong user password. Recommendations on how to choose a password are in chapter 5.1.

The password is used for two things:

- Derivation of a key which is then used for encryption and decryption of the Device key
- User authentication in the BabelApp client application

The password needs to be typed in in the following situations:

Windows PC or macOS:

- For login to the application after startup, or application lockdown

Mobile devices with iOS and Android:

- During BabelApp application startup – either after a device restart or after the application has been removed from the device memory
- In case of repeated unsuccessful PIN entry or Fingerprint scan

3.6.3 DEVICE KEY

Device Key DK is unique for every device and generated as a random 256 bit AES key:

DK = RND(32)

In case the BabelApp application is not running, DK is encrypted (eDK) using a 256bit AES key PK, derived from the user's password and randomly generated salt value:

PK = PBKDF2[hmacWithSHA256](password_user, salt, numberof_itterations, 256)

eDK = ENC_ECB[PK](DK)

After BabelApp client startup and password entry the PK key is calculated. Using the PK it is now possible to decrypt the DK key:

DK= DEC-ECB[PK](eDK)

The decrypted DK key remains in the device memory as long as the application is running – even if in the background.

3.6.4 USER AUTHENTICATION TO THE BABELAPP CLIENT APPLICATION

User authentication to the BabelApp client is based on a password entry and comparison of the calculated value with the KCV value stored on the device. The KCV value is derived from device key DK.

Assuming the DK key has been decrypted as described in 3.6.3., the KCV value is then calculated by an encryption of a constant using the AES algorithm with DK in ECB mode:

KCV = ENC_ECB[DK](constant)

If the stored and the calculated KCV values match, the user is successfully authenticated and the application will unlock.

3.6.5 USER'S DH KEYS

Every user owns a DH key pair – private key X and corresponding public key Y, which is registered with the BabelApp server. DH keys are generated on the user's device and possibly shared among their devices, as described in chapter 3.5.11.

Private key X is stored on the user's device in an encrypted form. For the X key encryption an AES algorithm in ECB mode and the device DK key are used.

eX = ENC-ECB[DK](X)

Private key X is used to establish a key agreement on the contact key CK value.

3.6.6 CONTACTS' KEYS

Contacts' keys CK (and keys for integrity checks) are calculated based on the public part of the DH contact key and private DH key as described in 3.5.2.

Contact keys are stored on users' devices in an encrypted form. For CK key encryption an AES algorithm in an ECB mode and device DK key are used:

eCK = ENC-ECB[DK](CK)

CK keys are used for encryption and decryption of message keys MK. Message keys MK are used for encryption and decryption of the actual conversations and are unique for every contact.

3.6.7 UNLOCKING THE BABELAPP MOBILE CLIENT USING A PIN

After authenticating the user to the mobile client by entering their password, the DK key is decrypted and kept in memory while the application is running. For security reasons, it is advisable to either always lock the application manually when inactive or activate the automatic locking of the application after a set time period of inactivity.

It is not very comfortable to type in the strong password (chapter 5.1) every time when unlocking the client application. That is why we implemented the option to lock and unlock the application in an authenticated state (DK key decrypted) using a 4-digit PIN.

The PIN code can be enabled / disabled or changed in the application after successfully entering the password.

Unlocking the application is based on PIN entry and comparison of the calculated and stored PCV value, cryptographically derived from the PIN value and randomly chosen salt value generated using the PBKDF2 function:

PCV = PBKDF2[hmacWithSHA1]
(PIN, salt, number_of_itterations, 128)

If the stored and calculated PCV values match, the application will unlock.

Note:
Since typing on a PC with Windows and macOS is a lot easier, desktop applications do not have the PIN mechanism implemented and the strong password has to be entered when unlocking the application.

3.6.8 UNLOCKING OF THE BABELAPP MOBILE CLIENT USING A FINGERPRINT

With mobile devices that support fingerprint authentication, it is possible to use it as an alternative instead of a PIN number.

3.6.9 ENCRYPTION OF MESSAGES AND ATTACHMENTS STORED ON THE DEVICE

Every message T is encrypted using a random MK key. Key MK is protected (encrypted) by a contact key CK, as described in 3.5.6.2.

Before received messages can be stored, the MK key has to be decrypted as described in 3.5.8.2.

MK = DEC-ECB[CK](eMK)

Then the MK key is encrypted using the device key DK:

eMK = ENC-ECB[DK](MK)

Once completed, the encrypted message and encrypted eMK key are stored on the device.

Note:
As seen in the description, messages (and attachments) are kept encrypted with the same MK key both during the transport and when stored on the device. MK is protected by an additional encryption layer in the application.

3.6.10 DECRYPTION OF MESSAGES AND ATTACHMENTS STORED ON THE DEVICE

AES key PK is derived from the user's password and is used to decrypt the DK key, as described in 3.6.3.

Using the DK key, it is then possible to decrypt all the Message eMK keys:

MK = DEC-ECB[DK](eMK)

Which allows to analogically decrypt all the messages eT, as described in 3.5.8.2.:

T = DEC-CBC[MK,IV](eT), where IV=0

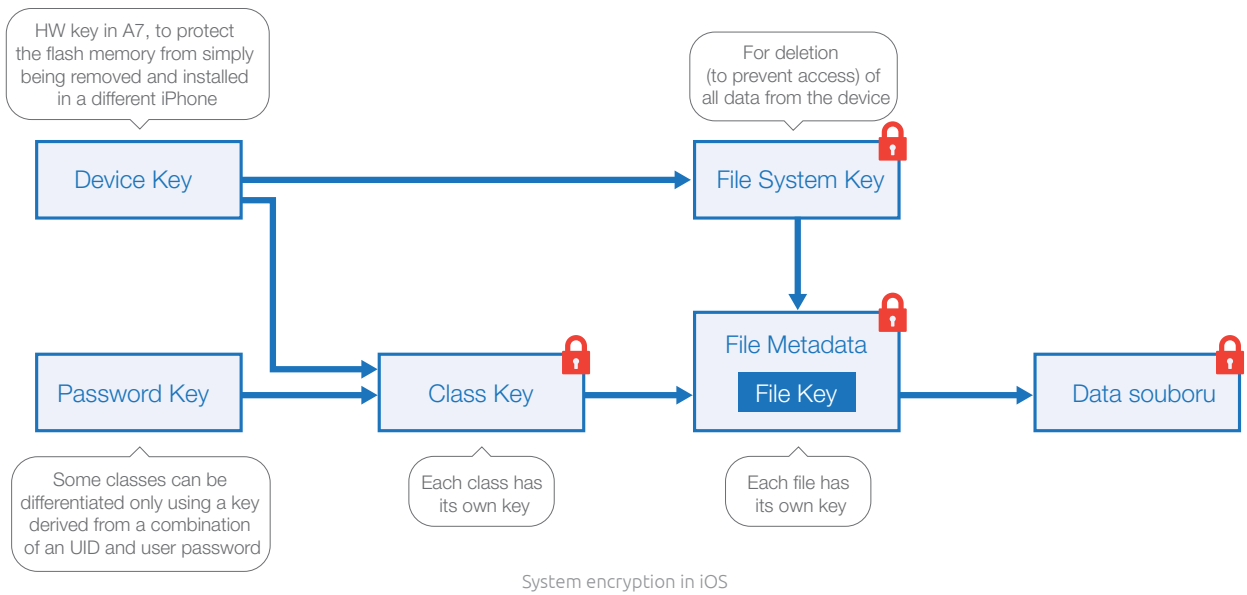
It is also possible to analogically decrypt attachment metadata and attached documents as described in 3.5.10.

3.6.11 SYSTEM ENCRYPTION

The core of BabelApp's security is the application encryption, which is independent from the operating system, however, some operating systems allow the developers to add a system encryption as an additional security layer.

3.6.11.1 iOS

All data on the iOS platform is encrypted in the device flash memory using a hierarchical key structure, described in the scheme below:



System encryption in iOS

A requirement for the use of system encryption on iOS devices is that a password is setup and used.

iOS is purposely slowing down password entry attempts (to aprox. 80ms between attempts) to protect users against brute force attacks. Brute force attacks are performed directly on the device since passwords are combined with UID in the processor.

Level of the system cryptography protection for messages stored in the SQLite database and for attachments stored in the application system file sandbox is set up to value NSFileProtectionCompleteUntilFirstUserAuthentication, which decrypts the Class Key as soon as the user's iOS password is typed in and keeps it in its memory even after the application is locked.

A good compromise between the discomfort of frequent password entry and security can be achieved by the use of fingerprint authentication (iPhone 5s, iPad 3 and higher). Relatively low FAR of the biometric authentication is compensated for by the necessity to type in the password after 5 unsuccessful fingerprint authentication attempts.

3.6.11.2 ANDROID

System encryption is not used.

3.6.11.3 WINDOWS

System encryption is not used.

3.6.11.4 macOS

System encryption is not used.

3.7 APPLICATION ENCRYPTION WHEN USING VOIP

3.7.1 PERFECT FORWARD SECRECY

Perfect Forward Secrecy technology is used to find and use shared encryption keys. It guarantees that no other session keys (past or future) are compromised in the event of a compromise of DH keys or session keys.

The calling party generates a new temporary DH key pair before each call and sends a public part of the key to the recipient in a sess-initiate signal message. After receiving the sess-initiate, the recipient generates its new temporary DH key-pair and sends the relevant public key part to the calling party in the sess-accept signal message.

Because new temporary DH keys are generated before every call (for both calling parties), there needs to be a guarantee for their authenticity and integrity - they must be checked to prevent a MiTM attack. For this purpose, a control mechanism based on the HMAC algorithm described in Article 3.5.7.1 is used in the signaling messages. The key used in HMAC to check the integrity and authenticity of signaling messages is derived from the long-term DH keys of both parties (see 3.5.2.2) and thus allows for checking the integrity and authenticity of temporary DH keys.

By the procedure set out in Article 3.5.1.4, both parties will first calculate the temporary shared secret (SS) from which they then generate a MasterKey MK key with a length of 256 bits following the procedure set out in 3.5.2.2.

The MK key is used as an input to the SRTP library, which derives from it the AES128 session key and the initialization vector (salt) to encrypt the RTP data stream.

All keys are located only in the memory of the client applications, are not saved, and are deleted immediately after the call is over.

3.7.2 EXTENSION OF THE CRYPTOGRAPHIC MODEL FOR VOIP

Unlike asynchronous message forwarding via BabelApp server, VoIP requires effective direct communication between clients (the exception being the use of a TURN server for relaying). SRTP protocol is used for communication, transmitting a stream of data blocks encrypted by application via UDP transport packets.

This transport mechanism requires an extended cryptographic model that uses existing secure transmission of BabelApp messages for VoIP signaling, existing keys to check message integrity to authenticate temporary DH keys, and AES encryption in counter mode for packet loss resistance.

3.7.3 RTP DATASTREAM ENCRYPTION

Within SRTP, the RTP stream is encrypted using the AES-128 algorithm (FIPS PUB 197) in counter mode (CTR mode) according to NIST SP 800-38A, Article 6.5. Datastream encryption is performed using the EK encryption key derived from the shared MK key.

The principle of counter mode is to encrypt a sequence of integers with a randomized start value of the counter. The reason for using CTR mode is the unreliability of the RTP data transmission within UDP, where packets may not be delivered properly. CTR mode allows you to encrypt and decrypt individual blocks when knowing counter values independently of each other, meaning that a failure of one part of the encrypted data stream does not affect the recipient's ability to decrypt the following data.

When using the counter mode, the critical initial value of the counter must be different for each EC encryption key value and, of course, known to both sides of the communication. So the initial value of the counter is derived from a combination of SSRC (a randomly selected 32-bit identifier that is unique within a specific RTP session) and salt that is derived from a shared MK key.

3.7.3.1 DERIVATION OF THE ENCRYPTION KEY (ENCRYPTION KEY EK)

To derive the EK encryption key from the MK Master Key value, AES128 encryption in ECB mode is used as follows: The first 16 bytes will be taken from MK, which will be the value of AES128 key K. Encrypted M1 data consists of the following 14 bytes supplemented from the right by zeros to the length of 16 bytes.

EK = ENC_ECB[K](M)

Example:

MK = f6c8fe882700ea330f416e9bf9e970381
cb56a2eb2ed87becdef48150a2d69ea

K = f6c8fe882700ea330f416e9bf9e97038

M1 = 1cb56a2eb2ed87becdef48150a2d0000

E_K = ENC_ECB[f6c8fe882700ea330f416e9bf9e97038]
(1cb56a2eb2ed87becdef48150a2d0000) =
ddb089593a0c8478670ae70f576780f4

IV_{Name} = RND(16)

IV_{Type} = RND(16)

IVThumbnail = RND(16)

3.7.3.2 SALT DERRIVATION (S)

Salt S is generated similarly to EK as follows: from MK, the first 16 bytes will be taken, which will form the value of AES128 key K. M2 encrypted data consists of the following 14 bytes supplemented from the right by zeros to the length of 16 bytes, on which XOR operation with constant 00000000000000020000000000000000 is performed.

Example:

Mk = f6c8fe882700ea330f416e9bf9e970381c-
b56a2eb2ed87becdef48150a2d69ea

K = f6c8fe882700ea330f416e9bf9e97038

M2 = 1cb56a2eb2ed87becdef48150a2d0000 XOR 0000
00000000000200000000000000000 = 1cb56a2eb2e
d87bccdef48150a2d0000

S = ENC_ECB[f6c8fe882700ea330f416e9bf9e97038]
(1cb56a2eb2ed87bccdef48150a2d0000) =
a8f746c53802b7391dd505113bbc13d7

3.7.3.3 COUNTER (CTR)

The counter structure is as follows:

- Byte 0 -3: filled from left 0Byte 4 - 7: SSRC
- Byte 8 - 13: packet counter
- Byte 14 -15: the encrypted blocks counter in the packet

SSRC (Synchronization source) is a randomly selected value of 4 bytes, which must be unique within an RTP session. The SSRC value is transmitted in the header of each RTP packet – see https://tools.ietf.org/html/ rfc3550.

Initialization of the CTR counter

When establishing an RTP session, the counter is initialized as follows:

- Byte 0 – 3: 0000H
- Byte 4 – 7: SSRC
- Byte 8 – 13: 000000H
- Byte 14 – 15: 00H

The initial value of the CTR counter is obtained by the XOR operation as follows:

CTR = CTR XOR S0

S0 is the adjusted value of salt S, obtained according to 3.7.2.2, where the last 2 bytes are zeroed.

Example:

SSRC = 048e3f1d

S = a8f746c53802b7391dd505113bbc13d7

S0 = a8f746c53802b7391dd505113bbc0000

CTR = 00000000048e3f1d0000000000000000 XOR
a8f746c53802b7391dd505113bbc0000 =
a8f746c53c8c88241dd505113bbc0000

3.7.3.4 ENCRYPTION OF DATA STREAM

SRTP generates a stream of output blocks by encrypting CTR counter values. Each block in the output block stream is created by encrypting the CTR counter value with an EC key.

The resulting encrypted data consists of an XOR operation of this output block and a block of open data in the packet. To encrypt the next block of open data in a packet, the byte value of 14-15 (block counter) in CTR is incremented. To encrypt the next packet, byte value 8-13 (packet counter) is incremented while zeroing the byte value of 14-15 (block counter) in CTR.

Example:

Example shows the incrementation of the CTR

First Block counter in the first packet

CTR = 00000000048e3f1d0000000000000000 XOR
a8f746c53802b7391dd505113bbc0000 =
a8f746c53c8c88241dd505113bbc0000

The second block in the first packet

CTR = 00000000048e3f1d0000000000000001 XOR
a8f746c53802b7391dd505113bbc0000 =
a8f746c53c8c88241dd505113bbc0001

The first block in the second packet

CTR = 00000000048e3f1d0000000000010000 XOR
a8f746c53802b7391dd505113bbc0000 =
a8f746c53c8c88241dd505113bbd0000

Example:

The example shows encrypting data in open form by using output blocks created by encrypting CTR

Data counter values in an open form:

M = 0800a0e72b6096e720ad92788189e6fa70395
fd8bb942ee8a54c7ea8

First block:

Message M is divided into 128-bit blocks M1 and M2

M1 = 0800a0e72b6096e720ad92788189e6fa

E_K = ddb089593a0c8478670ae70f576780f4

CTR = a8f746c53c8c88241dd505113bbc0000

OB = ENC_ECB[E_K](CTR)

OB1 = ENC_ECB[ddb089593a0c8478670ae70f576780f4]
(a8f746c53c8c88241dd
505113bbc0000) = 31e87f4515807246714bb245
efa57e60

Encrypted 1st block C1:

C1 = OB1 XOR M1

C1 = 31e87f4515807246714bb245efa57e60 XOR
0800a0e72b6096e720ad92788189e6fa =
39e8dfa23ee0e4a151e6203d6e2c989a

Second block:

M2 = 70395fd8bb942ee8a54c7ea8

EK = ddb089593a0c8478670ae70f576780f4

CTR = a8f746c53c8c88241dd505113bbc0001

OB = ENC_ECB[E_K](CTR)

OB2 = ENC_ECB[ddb089593a0c8478670ae70f576780f4]
(a8f746c53c8c88241dd505113bbc0001) =
c420698fdb98de7715721a50af0955f1

Encrypted 2nd block C2:

C2 = OB2 XOR M2

C2 = c420698fdb98de7715721a50 XOR 70395fd8
bb942ee8a54c7ea8 = b4193657600cf09fb03e64f8

Encrypted string M = M1 || M2

39e8dfa23ee0e4a151e6203d6e2c989
ab4193657600cf09fb03e64f8

3.7.4 RCP PACKET INTEGRITY

Each RTP packet is equipped with an HMAC integrity check with the SHA-1 hash function.

4. SERVER PLATFORM

The BabelApp server requires the following to run: (please refer to the BabelApp Implementation guide for details):

4.1 HARDWARE

PC server with CPU Intel x86/x64, clocked to at least 2 GHz

- Memory – at least 3 GB
- Disc – free space of at least 10 GB
- Network Interface Controller – 100 Mbps or higher

The server can be physical or virtual. Tested virtualization platforms are VMware and Hyper-V.

4.2 OPERATING SYSTEMS

BabelApp runs on Microsoft Windows Server or Linux:

Linux

- Oracle Linux Server 8.x
- Red Hat Enterprise 8.x
- CentOS/Almalinux 8.x

4.3 JAVA

BabelApp server requires Java installation 1.8, 32 bit or higher.

4.4 DATABASE

The BabelApp server uses a PostgreSQL database system, minimally version 9.6, preferably 13.

4.4.1 OPERATING SYSTEM ACCOUNT FOR POSTGRESQL

Linux: the PostgreSQL server must be started under a dedicated unix account with a strong password, not under the root account, or a different user account. Such a dedicated account should only own data that is managed by the server and cannot be shared with other services.

Windows: server service PostgreSQL must be started under a dedicated account, which has the rights to only access the data it manages. Such a dedicated account must have the rights to read in all directories that form the path to service directories and write permisison in the data folder.

4.4.2 SUPERUSER ACCOUNT

It is necessary to change the Superuser's PostgreSQL (postgres:postgres) implicit name and password during the installation to a strong password (command ALTER ROLE).

```
ALTER ROLE postgres WITH PASSWORD 'some_strong_password'
```

4.4.3 BABELAPP DATABASE SETUP

It is necessary to set up three databases during the installation:

- openfire
- babel
- babel_attachment

4.4.4 DATABASE ACCOUNT CREATION

Database accounts need to be created (user with a LOGIN privilege role to a relevant database) and strong passwords should be used in all instances. Database accounts will be named the same as the databases:

```
CREATE USER openfire WITH PASSWORD strong_password_1'
CREATE USER babel WITH PASSWORD strong_password_2'
CREATE USER babel_attachment WITH PASSWORD strong_password_2'
```

4.4.5 DATABASE ACCOUNT AUTHENTICATION TYPE

PostgreSQL supports a number of authentication methods, which are given by the pg_hba.conf file, placed in the root directory of the database server.

It is recommended to limit the connection to a local connection only and set the authentication method to authenticate to the local server operating of the system.

Peer Authentication (for local authentication only)
In case the BabelApp server is installed on the same HW (or the same virtual server) as PostgreSQL, it is possible to use an authentication to local accounts of the operating system. It is a preferred method called peer.

Database accounts are logically separated from user accounts in the operating system. If the same accounts are created both in the operating system and PostgreSQL, it is possible to use the peer authentication method for local connection.

Set up postgress.conf in
listen_addresses to an empty address list

Structure of pg_hba.conf for local connection:

# TYPE	DATABASE	USER	METHOD
local	sameuser	all	peer

Password Authentication (for network authentication)
Password authentication with the md5 option authenticates the client using a password, which is hashed twice, once after concatenation with the user name and for a second time with added salt.

In postgress.conf, set up

listen_addresses to an allowed network client address, for example (IPv4) 192.168.1.0/24

port to TCP port, implicitly 5432

Structure of pg_hba.conf for network connection:

It is assumed that the client (BabelApp server) runs on the following address 192.168.1.0/24

# TYPE	DATABASE	USER	CIDR-ADDRESS	METHOD
host	openfire	openfire	192.168.1.0/24	md5
host	babel	babel	192.168.1.0/24	md5
host	babel_attachment	babel_attachment	192.168.1.0/24	md5

4.5 OPENFIRE

The BabelApp server uses XMPP Openfire server http://www.igniterealtime.org/projects/openfire/, which is implemented in the Java environment and licensed under the Open Source Apache License. Openfire allows for feature expansion using plug-in modules (plugins). BabelApp features are realized through the plug in module babel.jar.

5. SECURITY REQUIREMENTS AND RECOMMENDATIONS

5.1 STRONG PASSWORD

A strong password should have a minimum entropy of 70 bits.

The following table shows a passwords' character's entropy, if the character is randomly chosen from a subset of equally probable characters:

- Numbers (10 characters), entropy 3,32 bites/character
- Lowercase letters without accents (26 characters), entropy 4,7 bits/character
- Uppercase letters without accents (26 characters), entropy 4,7 bits/character
- Lowercase and uppercase letters without accents (52 characters), entropy 5,7 bits/character
- Lowercase and uppercase letters without accents and numbers (62 characters), entropy 5,95 bits/character

An example of such a password is a password, which consists of a randomly chosen combination of lowercase and uppercase letters and numbers with a minimum length of 12 characters.

5.2 iOS

For mobile devices with iOS, the following security requirements and recommendations apply:

Requirements
- iOS version 7.0 or later
- jailbreak not applied
- device is locked using a PIN code
- a strong password is used to login to BabelApp

Recommendations
- activate data deletion after 10 unsuccessful PIN code entries
- use Touch-id to unlock the device and a strong password for a code lock
- BabelApp password is different from the device password
- Use touch-id for BabelApp login

5.3 ANDROID

For mobile devices with the Android operating system, the following security requirements and recommendations apply:

Requirements
- android operating system version 8 or later
- root not applied

Recommendations
- android version of 11 with activated system data encryption
- PIN code is activated
- do not use gestures for PIN entry
- activate remote data wipe (Google service)
- use Touch-id for unlocking the device
- BabelApp password is different from the device password

5.4 WINDOWS

For PC devices with the Windows operating system, the following security requirements and recommendations apply:

Requirements
- minimum version of Windows is Vista
- Device does not contain any malicious software and is protected against malicious software attacks
- A strong password is used
- A strong password is used for login to BabelApp

Recommendations
- BabelApp password is different from the system password
- Minimum version of Windows is 7

5.5 macOS

For devices with macOS, the following security requirements and recommendations apply:

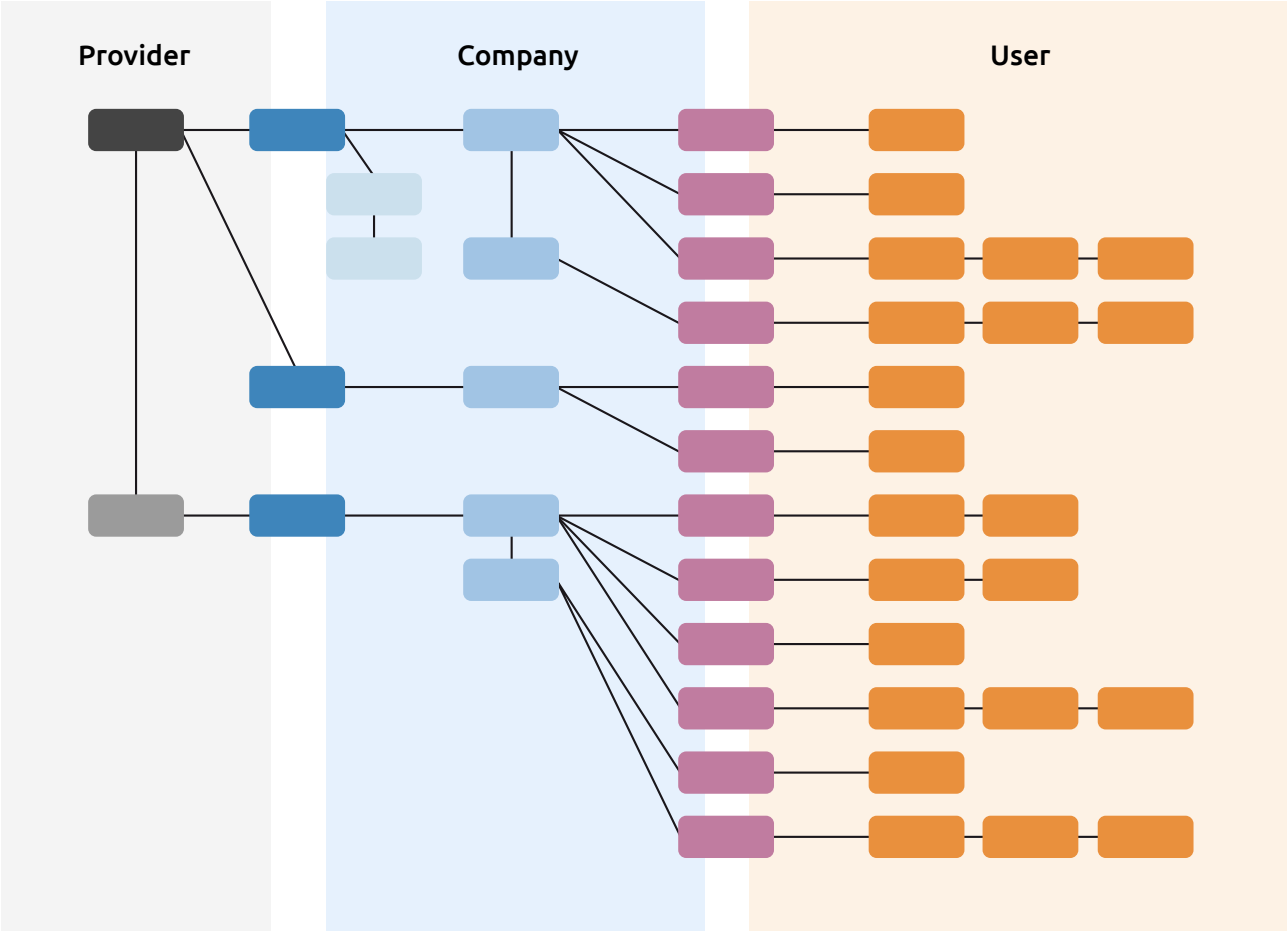
Requirements
- Minimum version of macOS is 10.11
- Device does not contain any malicious software and is protected against malicious software attacks
- A strong password is used
- A strong password is used for login to BabelApp

Recommendations
- BabelApp password is different from the system password

6. PROTECTION THROUGH THE BITCOIN NETWORK

When Bitcoin network protection is activated, one or more Bitcoin transactions are created, which are then sent to the Bitcoin network, where they are authenticated and included in the transaction block. Transactions written in this way can be read by anyone, but it is already very difficult (practically impossible) to change or delete such transactions.
Transactions in the Bitcoin network always only transfer the amount from one or more previous transactions to a new transaction. Electronic signatures ensure that the amount can be transferred.

In this way, a transaction tree can be built from Bitcoin transactions. The root of this tree is transaction TX0, which is created by OKsystem a.s. (hereinafter referred to as the provider) and which must be the ancestor of any other transaction belonging to BabelApp protection. Conversely, the leafs in this tree contain transactions that contain client data for key verification. This client data is already created and signed directly by the clients (BabelApp user applications on phones or desktops) and no one else can create it.



6.1 TRANSACTION

All Bitcoin transactions used for BabelApp protection have the same structure.

1. entry always refers to the previous transaction, which is part of the protection. This applies to TX0, which is the first BabelApp protection transaction and therefore no longer has any ancestor. Its entry is only a wallet to pay for the TX0 transaction.

2. entry is not mandatory and if there is, it is always a wallet used to pay for the transaction.

1. output always contains the data used for BabelApp protection. The output is realized by the script OP_RETURN, it does not contain any BTC amount and it is never possible to connect any other transaction to it.

The data always starts with two bytes that specify the type of transaction.

- bb0a - TX0
- bb0b - TX0'
- bb1a - TX1
- bb1b - TX1'
- bb1c - TX1_C
- bb2a - TX2
- bb3a - TX3

For example: RETURN PUSHDATA(2)[bb1b]

Depending on the type of transaction, the script may contain additional information.

For example: RETURN PUSHDATA(2)[bb1a] PUSHDATA(10)[54819c66332fa033f75c] PUSHDATA(10)[ca62a9103415ebe85fdd]

The other outputs that the transaction contains depend on its type.

6.1.1 Root transaction - TX0

Transaction TX0 was created when Bitcoin network protection was enabled. The transaction was included in block 000000000000000000177b0bf7514b0c08e49dff-00184362f2104a8820428d65 and its hash is 683db0ae-401581952a80b9672fb3b0abc404a558d2681e-c4490a78231c9f762f

Data: [bb0a]

Inputs:

1. entry: wallet

Outputs:

1. output: data (bb0a)
2.-11. output: P2PK for TX1 connection
Output 12.: P2PK for TX0 connection '
Output 13.: P2PKH for the rest of the amount spent

Hash TX0 and the block number in which it is located are contained in all BabelApp clients. When verifying the transaction tree, it must always end with this TX0. No transactions belonging to BabelApp protection are searched in blocks older than the block containing TX0.

6.1.2 Root transaction TX0'

If all outputs to which transaction TX1 (either in TX0 or in TX0') can be connected are already occupied (the output is spent), a new transaction TX0' is created. This transaction will again include several outputs for connecting the TX1', one output for connecting the TX0' and an optional output for the rest of the amount spent for the newly created TX0' and fees for miners for its insertion into the block in the Bitcoin network.

When validating whether the transaction belongs to the BabelApp network security mechanism, it is necessary to verify that the 1st input refers directly to TX0 or TX0' (then it is necessary to continue the validation up to TX0).

Data: [bb0b]

Inputs:

1. input: TX0 or TX0'
2. entry: wallet

Outputs:

1. output: data (bb0b)
2 - x. output: P2PK for TX1 connection
x + 1. output: P2PK for connection TX0'
x + 2. output: P2PKH optional output with the rest of the amount spent

6.1.3 TX1 corporate transactions

The transaction is created by the provider based on a request from the company that owns the BabelApp server and wants to use Bitcoin protection. This transaction gives the owner the exclusive possibility to establish any number of other transactions under the given transaction TX1. The provider no longer has access to TX1.

Data: [bb1a] [certificate hash] [chained hash domains (1-6)]

- SSL certificate hash: the first 10 bytes of the SHA1 thumbprint
- hash domains: the first 10 bytes of the hash HASH160 (SHA256 followed by RIPEMD160) UTF8 encoded domain name.

Example:

TX1 with one domain name:

RETURN PUSHDATA(2)[bb1a] PUSHDATA(10)[54819c66332fa033f75c] PUSHDATA(10)[ca62a9103415ebe85fdd]

TX1 with 2 domain names:

RETURN PUSHDATA(2)[bb1a] PUSHDATA(10)[cdadf3bae8ea0174c50a] PUSHDATA(20)[4f855e3cd1ae75b7de98129ac10b041c4ab209de

Inputs:

1. input: TX0 or TX0'
2. entry: wallet

Outputs:

1. data output (bb1a)
2. P2PKH output for TX1' connection, key owned by company
3. P2SH output for TX1_C connection for domain change
4. P2PKH output for the rest of the amount spent

6.1.4 Expanding corporate transactions TX1'

The transaction is created by the company that owns the BabelApp server. The transaction expands the number of outputs for connecting transactions with information about individual contacts.

Data: [bb1b]

Example: RETURN PUSHDATA(2)[bb1b]

Inputs:

1st input: TX1 or TX1'
2nd entry: wallet

Outputs:

1st output: data (bb1b)
2 - X output: P2PK for connection of TX2, or another TX1'
X + 1st P2PKH output for the rest of the amount spent

6.1.5 Transaction to change domain TX1_C

The transaction is connected either to TX1 or when the domain is changed repeatedly to the previous TX1_C. The connection script is P2SH (MULTISIG). This transaction needs to be signed by both the owner of the BabelApp server and the provider.

Data: [bb1c] [certificate hash] [chained hash domains (1-6)]

- SSL certificate hash: the first 10 bytes of the SHA1 thumbprint
- hash domains: the first 10 bytes of the hash HASH160 (SHA256 followed by RIPEMD160) UTF8 encoded domain name

Example:

TX1_C with one domain name:

RETURN PUSHDATA(2)[bb1a] PUSHDATA(10) [54819c66332fa033f75c] PUSHDATA(10) [ca62a9103415ebe85fdd]

TX1_C with 2 domain names:

RETURN PUSHDATA(2)[bb1a] PUSHDATA(10) [cdadf3bae8ea0174c50a] PUSHDATA(20) [4f855e3cd1ae75b7de98129ac10b041c4ab209de

Inputs:

1st input: TX1 or TX1'
2nd entry: wallet

Outputs:

1. data output (bb1c)
2. P2SH output for connection of another TX1_C for next domain change
3. P2PKH output for the rest of the amount spent

6.1.6 Transactions for transfer to TX2 user

Transaction TX2 is created by the owner of the BabelApp server and is connected to TX1'. This transaction makes it possible to create transactions for a specific user.

Data: [bb2a]

hash SSL certifikátu: prvních 10 bytů SHA1 thumbprintu

- hashe domén: prvních 10 bytů z hashe HASH160 (SHA256 následovaná RIPEMD160) UTF8 zakódovaného doménového jména

Example:

RETURN PUSHDATA(2)[bb2a]

Inputs:

1st entry: TX1'
2nd entry: optional wallet

Outputs:

1st output: data (bb2a)
2nd output: MULTISIG (owner + user) for TX3 connection

6.1.7 User transaction TX3

The TX3 transaction is signed by both the user and the company owning the server. TX3 contains information about BabelApp addresses and public keys used for encryption.

Data: [bb3a] [hash DH public key, or empty field] [0 to 3 fields of 20 bytes - hasel prefix babel address]

- The hash function is HASH160 (ie SHA256 followed by RIPEMD160).
- The DH public key is converted to bytes without a leading zero, if any.
- For BabelApp addresses, the prefix is first encoded using UTF-8.

Examples:

TX3 key + 1 BabelApp address

RETURN PUSHDATA(2)[bb3a] PUSHDATA(20) [11d953bea4fd10d9f0b3238e802f89b7fb7a84e7] PUSHDATA(20)[ec2d242427dfac4704043aaf8cffa03db-f791a3a]

only one BabelApp address (no key information)

RETURN PUSHDATA(2)[bb3a] 0[] PUSHDATA(20) [14b43a06e4d849dbaa7d836c5e34e2f445c116f4]

Inputs:

1st input: TX2, or previous TX3
2nd entry: optional wallet

Outputs:

1st output: data (bb3a)
2nd output: MULTISIG (owner + user) for TX3 or P2SH connection containing redeem script type FRIENDS.

6.2 WALLET, PAYMENTS, AMOUNTS

Each Bitcoin transaction means the transfer of an amount, which is the sum of all inputs, to individual outputs. If the amount transferred from the 1st input is not enough for all defined outputs + fees to miners, it is necessary to connect the 2nd input and increase the amount. If the amount on the inputs is too high, it is possible to add the last output, which returns the excess amount to the wallet. Transactions TX0, TX0' and TX1 are created and paid for by the provider. Transactions TX1', TX1_C and TX2 are created and paid for by the company owning the BabelApp server through the server wallet. The TX3 transaction is created by the client, but the TX3 server checks, co-signs it, and adds a second entry with the required amount if necessary. TX3 never includes an overcharge with a wallet.

6.3 PROVIDER (OKsystem a.s.)

The provider created TX0 and creates an extension TX0' as needed. If any BabelApp server wants to connect to the Bitcoin network, it will ask the provider and the provider will create a new TX1 based on the certificate that the provider has previously signed and all domain names. No domain names may ever be repeated, nor may a certificate.

6.4 OWNER OF THE BABELAPP SERVER

6.4.1 Connecting to a Bitcoin network

When activating protection via the Bitcoin network, the server first reads, validates and saves blocks from the Bitcoin network to the DB. Not all the data that the Bitcoin network contains (several hundred GB) is downloaded, but it is enough to retrieve data from the block 000000000000000000177b0bf7514b0c08e49dff-00184362f2104a8820428d65, which must contain TX0. When retrieving data from the Bitcoin network, a bloom filter is set, thanks to which only a fraction of all data is sent to the BabelApp server. The bloom filter contains identifiers of all types of transactions (bba0 - bb3a) together with the BLOOM_UPDATE_P2PUBKEY_ONLY flag. The BabelApp server also provides the data obtained in this way to all clients. The BabelApp server therefore has stored together with the blocks all BabelApp transactions, both transactions that concern the given server and all other BabelApp servers.

6.4.2 Activation of protection via Bitcoin network - creation of TX1

To use Bitcoin protection, the provider first creates TX1 transactions. TX1 contains the hash of the current certificate and the hash of domain names (common name, alternative name). The second output for TX1 'is secured by a P2PKH script with the address of the BabelApp server owner and the third output for TX1_C is secured by P2SH with the MULTISIG redee script (address of the provider and the owner of the BabelApp server). The provider inserts the transaction created in this way into the Bitcoin network and the server owner can connect TX1' and subsequently individual clients to it.

TX1 verification

Once the server obtains a transaction from the TX1 Bitcoin network, it verifies that it is created correctly

a. verifies that the 2nd output of TX1 is protected by a P2PKH script with the address of the BabelApp server owner
b. verify that the 3rd output of TX1 is protected by a P2SH script, where the redeem script used is MULTISIG 2 of 2 and one of the addresses belongs to the given server

6.4.3 Domain change

Transaction TX1 or TX1_C contains a hash of the current certificate and a hash of domain names (common name, alternative name). When the domain is changed, a new certificate is created that contains the new domain name (common name) and the old domain name is in the alternative name. The provider creates a TX1_C transaction that contains a new certificate hash and a new domain hash. TX1_C is signed by the provider and the owner of the BabelApp server. Therefore, the change of the BabelApp server domain is performed only with the cooperation of both the BabelApp server owner and the service provider.

6.4.4 Loss of key

If the TX1 protection key (TX1 ') is lost, you need to request the creation of a new TX1 with a different certificate and with new domains. All contacts must then be re-registered under the new domain. It is therefore very important to store and back up company keys securely.

6.5 CLIENTS

6.5.1 Transaction validation

Clients obtain individual blocks containing transactions from the server to which they are connected. Alternatively, they can obtain transactions from BabelApp.com. BabelApp server provides headers of all blocks (from TX0) and selected transactions in the merkle tree structure. Clients verify the data obtained in this way, obtain BabelApp transactions from them and store them in the DB. The data in the merkle tree structure cannot be changed or added, but the data can be concealed. Therefore, clients contact multiple servers (including BabelApp.com) for information about the number of transactions for a given block of data. The server only sends blocks that have at least 10 acknowledgments.

PREPROCESSING WHEN RECEIVING A TRANSACTION

a. Bitcoin validations - all blocks and transactions contained in them must comply with the rules defined by the Bitcoin network and are also validated accordingly.

b. BabelApp validation - first the transaction is parsed, it is verified that it is a BabelApp transaction and its type is determined

   i. TX0 - must be found in block 000000000000000000177b0bf7514b0c08e49dff-00184362f2104a8820428d65 and its hash must be 683db0ae401581952a80b9672fb3b0abc404a-558d2681ec4490a78231c9f762f

   ii. TX0' - must refer to TX0, or already verified TX0'

   iii. TX1 - must refer to TX0, or already verified TX0'

      1. The 1st hash of the certificate is unique, it does not contain any older verified TX1 or verified TX1_C

      2. the hash of all domains is unique, none of the domains contains any older verified TX1 or verified TX1_C

   iv. TX1' - must refer to an already verified TX1, or already verified TX1'

   v. TX1_C - must refer to already verified TX1, or already verified TX1_C

      1. The 1st hash of the certificate is unique, it does not contain any older verified TX1 or verified TX1_C

      2. the hash of all domains is unique, none of the domains contains any older verified TX1 or verified TX1_C, otherwise the BabelApp server (TX1 subtree) is marked as untrusted

   vi. TX2 - must refer to an already verified TX1'

   vii. TX3 - must refer to an already verified TX2, or an already verified TX3

      1. hash none of the prefix BabelApp addresses is already in the same TX1 subtree (BabelApp server), otherwise the BabelApp server is marked as untrustworthy.

FOREIGN USER VALIDATION

The contact is validated in the following order.

a. server discovery and validation (TX1)

   i. all domains are found according to all BabelApp addresses of the validated contact

   ii. RV obtains the certificates of all domains and verifies that they are identical

   iii. its hash is calculated from the certificate and the correct subtree (TX1) is found

   iv. it is verified that the certificate is not revoked or that the BabelApp server is not untrusted

   v. it is verified that the hashes of all domains are stored at TX1 or TX1_C

b. user discovery and validation (TX3)

   i. according to the hash of the BabelApp address prefix, information about the user can be found in the TX1 subtree (TX3 transactions)

   ii. it is verified that the last hash of the DH key in TX3 is the same as the hash of the DH key provided by the BabelApp server

   iii. it is verified that all prefixes of BabelApp addresses are stored in the given TX3 transactions and that none are missing or remaining

   iv. possible that this is not an attack, but that a transaction with a new change has not yet been inserted into block in the Bitcoin network (or the block still does not have 10 authentications). In this case, the client contacts the server that co-signed the transaction and has this transaction sent directly to him. Subsequently, this transaction is verified in the same way as any other TX3. If the authentication is OK, no contact attack is reported, but only a warning that disappears once the block receives the 10th authentication.

VALIDATION OF OWN CONTACT

This validation is performed in the same way as when validating a foreign contact. In addition, it is verified that the contact has access to the last TX3 transaction.

a. the 2. output of the last TX3 transaction is secured by a MULTISIG 2 of 2 script, of which one address belongs to the given contact

b. or the 2. output of the last TX3 transaction is secured by the FRIENDS script that created and signed the contact

6.6 REDEEM SCRIPT FRIENDS

The private keys used to create TX3 transactions are stored only on clients. Whenever other devices are registered, they are transferred together with the DH key. If this key is lost, it is not possible to create a new TX3 transaction and therefore it is not possible, for example, to change the DH encryption key. For this case, the user can set (as the 2nd output in a TX3 transaction) a P2SH script whose redeem script is of the FRIENDS type.

6.6.1 Properties

The script allows you to define the addresses of friends who have the authority to connect a new TX3 transaction even without the user's key. When defining, you can set a set of friends and how many of them are needed to connect a new TX3. It is always necessary to cooperate with the server, which must also sign the newly created TX3 transaction.

The script is created according to the following template **Ks && (Kc || (X-of-Y Kf1, Kf2, ...))**, where

- **Ks** - server key
- **Kc** - contact key
- **Kfx** - individual keys of friends
- **X** - The number of friends needed to connect
- **Y** - number of all friends

Kf1,..., Kfn must be lexicographically sorted (according to bip 67)

Script example:

PUSHDATA(33)[Ks] CHECKSIGVERIFY DEPTH 1 EQUAL IF PUSHDATA(33)[Kc] CHECKSIG ELSE [X] PUSHDATA(33) [Kf1] PUSHDATA(33)[Kf2] PUSHDATA(33)[Kf3] [Y] CHECK-MULTISIG ENDIF