PrivMX Architecture

Łukasz Dudziński, Kamil Kaproń, Paweł Łazarski, Robert Łucarz, Maciej Muszytowski, Sebastian Smyczyński [1]*

Abstract
The PrivMX architecture is being developed as a tool for creating decentralized client-server applications that use client-side data encryption. The document presents the basic assumptions of the architecture and the general description of its key components. It describes an independently secured data transmission channel being a variant of the TLS protocol, the most important server API functions, solutions used in standard client library and independent PKI infrastructure for publishing and verifying public keys.

Keywords
Client-server — client-side encryption — zero-knowledge servers — sending and storing data — Cryptography — ECC — Keys as identifiers — PKI — CONIKS — Merkle trees — append-only structures

[1] *Simplito Sp. z o.o., Software R&D Dep, Toruń, Polska*
*__Contact__: www.simplito.com/contact

Contents

1. General information

The PrivMX architecture is a set of specifications and components serving the creation of client-server systems based on the concept of client-side encryption as well as servers with zero knowledge.

The systems created on the basis of PrivMX allow for secure transfer, storage and sharing of data.

The purpose of this document is to generally present the main elements of the PrivMX Architecture:

- encrypting and decrypting data on the client side, thus limiting access to data for the server and reducing of its complexity.
- Control of data access rights based on cryptographic methods.
- Work in the request-response scheme and possibility of use of various transport protocols, including non-duplex protocols such as i.e. HTTP.
- The PrivMX client and server independently secure their own communication, regardless of the security offered by the transport protocol.
- Own infrastructure for storage and verification of public keys (PKI) independent of third-party certification authorities.
- Ability to build decentralized systems.
- Addressing scheme which allows to identify users in decentralized systems.

- The possibility to share data by "distribution of access keys to specific data" instead of "distribution of user's right(s)".

Depending on your specific applications and requirements, the system using PrivMX architecture can cooperate with any other software components, also with ones which do not encrypt data and work on the side of the server.

In PrivMX it is possible to find ideas coming from different technologies and use known and proven algorithms:

- SHA256, PBKDF2, HMAC-SHA512;
- ECC (Elliptic Curve Cryptography) - asymmetric encryption - private and public keys generation as well as the ECC-related algorithms and schemes of ECDSA, ECDH(E), ECIES;
- AES (Advanced Encryption Scheme) - symmetric encryption;
- Bitcoin related ideas such as the Blockchain structure, Bitcoin addresses, BIP-32 (extended keys and their derivation, chaincodes), BIP-39 (Mnemonics for generating of keys)
- CONIKS;
- TLS (Transport Layer Security) - handshake procedures, tickets, frames;
- SRP (Secure Remote Passwords);
- OpenPGP (RFC 4880) - PGP packets.

2. Client-server communication

2.1 PrivMX addresses

The PrivMX architecture assumes that the servers are identified through the hostname (domain name) and the users of the individual servers - through the address of the form of user#hostname.

2.2 Service Discovery procedure

Calling the API methods of the PrivMX Server is possible after obtaining the exact address for sending requests (API endpoint) for a given hostname. This is done by the Service Discovery (SD) Procedure which attempts to establish connection with the target server and asks it for configuration for the selected transport protocol. For example, the host "simplito.com" for the "http" protocol can report the configuration:

```
defaultEndpoint =
  http://s1.simplito.com:3333/pmx/server
```

Such a configuration, in addition to defaultEndpoint, may also contain other fields, i.e. TTL (time-to-live that is the maximum storage time of the endpoint address in cache), or Redirect (order to search for an endpoint address on another server).

The SD procedure while looking for a configuration, checks fixed locations on the target server - asks for specific URL addresses, e.g. http://hostname/privmx/privmx-configuration.json, and checks the TXT records in the DNS entry for the hostname domain. An example entry in DNS might have the form:

```
V=privmx;t=http;ttl=20000;
  defaultEndpoint=
  http://hostname/privmx/server/api
```

After performing the SD procedure and obtaining the address "defaultEndpoint", the client module is ready to establish a connection with the server by means of the PrivMX TLS protocol.

2.3 PrivMX TLS protocol

API calls are implemented using the secure PrivMX TLS communications protocol being a modified and simplified variant of the TLS 1.2 protocol.

PrivMX TLS inherits from TLS the overall manner of the protocol implementation including among others the standard division into the frames (CHANGE_CIPHER_SPEC, ALERT, HANDSHAKE, APPLICATION_DATA) and handshake procedures.

The PrivMX TLS protocol, in addition to creating and maintaining connection sessions, and encrypting of data sent to both sides, also provides a mechanism for users' logging in and controlling the sessions for logged users. The characteristic features of PrivMX TLS are:

- operation on any transport (also non-duplex), in the request-response schema.
- Sessions control and their resuming based on the utilization of tickets lists used as a nonce - this means the lack of sending the fixed session ID and preventing observers from grouping frames. This is a different use of tickets than it is described in RFC 5077.
- Simplified handshake procedures, which, among others, do not negotiate encryption algorithms.
- Two levels of connection security:
  - **standard** - available to all clients. Connection security is achieved by using a common secret received from a handshake procedure utilizing ECDHE (or ECDH when the client has an authorized public key of the server).
  - **logged in** - the level available to the account holders on a given server. Logging into PrivMX is a procedure in which a client after the initial establishing of a "standard connection" asks the server for additional handshake procedure based on SRP-protocol. The PrivMX TLS module here, has access to the data stored on the server in the course of creating a user account.
- Once a connection has been established, each party may at any time force a change of the connection encryption keys.

PrivMX provides clients with different sets of API methods depending on the PrivMX TLS connection level used. This topic is often referred to in the following chapters, when describing API methods.

2.4 PrivMX Proxy

PrivMX provides the `API.proxy` method, which allows the client programs to use "their own" server module as an intermediary in connections with other PrivMX servers. The operation of this service can be configured as needed. In particular, it depends on the manner of implementation of the client's application whether and how it uses such a possibility.

Establishing PrivMX TLS connections with a "foreign" server via PrivMX Proxy service does not interfere with the security of connections. Encryption keys are set (handshake) with the target server in such a way that the proxy server does not have access to them.

3. PrivMX server

3.1 Main API functions of the server

The PrivMX server module provides API methods to client programs allowing for implementation of the following functions:

- **data storage** - functions of saving and reading of data blocks and descriptors;
- **handling of communication boxes** and functions for receiving and reading of messages;
- **storage and verification of public keys** - PrivMX PKI functions are described separately in the last chapter of this document;
- **creation of accounts, login, proxy, users' profiles**;
- **administration** - description of API function connected with the PrivMX server management is omitted in this document.

In this chapter we focus on the first two issues.

3.2 ECC keys - identifiers and access rights

The PrivMX architecture assumes that the IDs for newly created objects are given in most cases by the client program.

- The identifiers of the most important objects are (serialized) ECC public keys or the 160 bit bitcoin addresses counted from them. They are created based on ECC private keys randomly chosen by the client program;
- the server does NOT allow the client programs to enumerate objects that it stores.
- access to "normal", "unprotected" operations on the data requires the client program to have their identifier, i.e. the ECC public key. However the access to specific "protected" operations, i.e. modifying an object or removing it, requires to possess additionally an appropriate ECC private key;

Creating objects on the PrivMX server is always done only and exclusively at the wish of the client program. Due to the lack of possibility of enumeration of objects, the ECC keys pair (privkey, pubkey) randomly chosen by the client module becomes the only way of access to the object. Storage of this pair is the responsibility of the client program - PrivMX architecture does not specify the specific way to store these keys, but the standard client library contains some ready-to-use solutions (see further chapters).

3.3 Data blocks

PrivMX Server accepts and serves data like disk drivers in files systems - in form of blocks of maximum size of up to 128KB (this size can be configured).

The blocks are "raw data portions" with assigned identifiers - BIDs. Blocks are created by the client module, which sends them to the server where they are stored without modifying the content.

- The server does not encrypt the data - the client program is responsible for it.
- The client program determines the BID of the block so that it is equal to SHA256 of the content of the block sent.

In respect of the block design (BID of the block = hash of its contents):

- client and server modules can benefit from the natural control of the consistency of transmitted data - block identifier is also a checksum of data;
- it is not possible to create two blocks on the server for which the same data was sent;
- blocks are unmodifiable (immutable) - "change of block data" can only be implemented through creating and submitting of a new block - new data means a new BID.

Blocks do not exist independently - they can be created and read by the client module always in conjunction with the operation on the object, which uses these blocks - see descriptors and messages below. Each block can be used by several such objects. The PrivMX architecture assumes that blocks that are not assigned to any object should be automatically removed by the server.

3.4 Descriptors

Descriptors enable to store portions of data on the server larger than 128KB - descriptors "are grouping the blocks" and are access points to them for the client program. They are also facilities implementing a basic data access rights policy (in line with the previously mentioned manner of use of public and private ECC keys). Below we describe it more accurately.

The creation of descriptors (`API.descriptorCreate*`) and their modification (`API.descriptorUpdate*`) typically requires the transmission of one or more data blocks through single or multiple use of the corresponding API method. For

this purpose, a "transfer sessions" mechanism is used - a temporary unique data transfer identifier is created and used in subsequent calls of the `API.blockCreate`:

1. The client program (user logged in) calls the `API.descriptorCreateInit` and obtains the TransferID.
2. Then it sends the blocks using `API.blockCreate` - giving the TransferID and BID and the data of each block. If the client program wants to use blocks that already exist on the server, then it uses another function - `API.blockUseExisting`.
3. It randomly creates a new dpriv private key and generates the dpub public key for it.
4. Finally, it creates a new descriptor (`API.descriptorCreateFinish`) on the server. To this end it sends to it:
    - DID (160 bit) - bitcoin address ("main" network) for dpub;
    - TransferID, Blocks - TransferID used and BIDs list of blocks sent during the session;
    - Extra - space for additional, arbitrary descriptor data to be used by the client application.
    - dpub - public key of the descriptor;
    - signature of the above data with the dpriv key - so that the server will know that the client has the appropriate private key, i.e. it is the owner of the descriptor.

Descriptors and blocks, combined with client-side data encryption (encryption of block data and Extra fields) enable for the creation of data-storing mechanisms that hide the contents of files and directory structure from the server - see the chapter about the client module.

Reading descriptors is possible for each client program that has the corresponding DIDs. Then through calling of `API.descriptorGet` the client can obtain a full set of the above information (Blocks, Extra, dpub, etc.) and through subsequent calls of `API.blockGet` the client can retrieve all data blocks. Calling of `API.blockGet` requires to give BID and DID - it is not sufficient to know only the block ID in order to obtain access to its data.

Modification of the descriptor (`API.descriptorUpdate`) is an operation similar to creating a descriptor - it requires the request to be signed with the dpriv key and it requires sending new blocks (or using the existing ones). The descriptor removal request (`API.descriptorDelete`) also requires the signature by a private key. Removal of the descriptor does not cause automatic removal of blocks - this should happen only if they are not used by other descriptors or messages.

3.5 Mailboxes

The mechanism of the operation of mailboxes and messages between PrivMX client programs (users) can generally be described as follows:

1. **to send the message**, the sender's client program must sign it with the sender key and establish a standard PrivMX TLS connection with the recipient server (this can be done using a proxy). It calls the `API.messagePut` method on this server to drop a message into a specific mailbox. Boxes are identified by public keys.
2. **The recipient receives a message** from their server when the client program checks the status of their mailboxes (`API.sinkGetMessages`, `API.messageGet`). Corresponding requests must be signed with private keys of the boxes.

The manner of receiving messages by a mailbox, defined in the PrivMX API terminology as "sink", is determined at the time of its creation. Then the client program, after randomly creating a new pair of keys (spriv, spub), must convey the following data to the `API.sinkCreate` method:

- SID, which is the serialized spub public key.
- WriteMode - the value "private", "public" or "anonymous" specifying who may send messages to this mailbox:
    - **private** - only the owner of the private key of the mailbox may leave messages;
    - **public** - any user of any PrivMX server may leave messages. The message signature is then verified by using PrivMX PKI.
    - **anonymous** - the mailbox accepts messages signed with any key, i.e. by such one which is randomly created before sending.
- Extra - a field for any use by the client program.
- Signature of the above data executed with spriv key.

The WriteMode and Extra fields can later be read by the client program (`API.sinkGetInfo`) and modified (`API.sinkUpdate`) and the box as a whole can be removed by calling of `API.sinkDelete`. All the operations listed here are available by default only in the "logged in" connection and must be signed with the appropriate private key.

3.6 Messages

Sending a message (i.e. creating and placing it in a specific target server's mailbox using the `API.messsagePut` method) is similar to creating a descriptor - it involves the creation of a "transfer session" and the sending of data blocks. Only messages with sizes of up to 1 MB can be sent without the use of blocks.

Assuming the client program has a pair of sender's keys (cpriv, cpub):

1. sending a message starts calling of `API.messagePutInit` with parameters:
    - SID of destination mailbox.

- SenderAddress - sender's address in the format "user#server.net". This field is not used in case of anonymous boxes.
- SenderPubKey - cpub public key belonging to the sender.
- ExtraAuth - place for optional additional data to be used by the sender verification server. This field is especially useful when the box is running in anonymous mode - e.g. it receives messages from web forms using captcha.
- Signature of the entire request with cpriv key.

2. When the client is not rejected (see below), it receives a new TransferID that can be used to send message data blocks. It does this through the appropriate calls of `API.blockCreate`.

3. Sending the message ends with the `API.messagePutFinish` call with parameters:

- Extra - field for any use by the client (max 1MB).
- TransferID, Blocks - transfer session confirmation - TransferID used and list of BIDs sent to the blocks
- Tags - list of any strings set by the sender. The server stores them with the message.
- Signature of the entire request done with cpriv key.

Rejecting a message may result from poor verification of the sender's address or key (PrivMX PKI) or because of an optional ExtraAuth verification failure. The server does not check the data sent in the ExtraAuth field by default, but it can be done by server extensions - depending on the specified application.

The mailbox numbers the received messages and provides the number of the last message (lastNumber) through the already mentioned `API.sinkGetInfo`. Reading the list of messages contained in the mailbox is available to the holder of the private key of the mailbox via a properly signed call of `API.sinkGetMessages`. This method allows the client to obtain the message IDs (MIDs) from the given numbering range and those that have set specific tags.

The client program can read the message by calling of `API.messageGet` using the appropriate MID and SID. It then obtains access to the SenderAddress, SenderPubKey, ExtraAnon, Blocks, Extra, Tags fields. The request to read the message must be signed with the private key of the mailbox. The `API.messageDelete` method requires that the similar data and signature are given.

3.7 Public and private users' data

In addition to the objects described in the previous chapters, the server also stores the data associated with specific users. Their scope depends on specific applications.

In default configuration, the PrivMX server, for each user, stores the following private data generated by the client program:

- login data - hashing parameters (salt, rounds number, hashing algorithm name used on the client side) and SRP verifier. The server does not store the user's password.
- a small PrivData data record encrypted on the client side.

By using the PrivMX PKI mechanism, the server, by default, makes available publicly, i.e. for any client program, the following user-related data:

- SID of the default mailbox - associated with the user's address;
- IdentityKeyPub, i.e. the public key of the user;
- User's profile data - optional data such as full name, description, avatar, etc.

The meaning of the above data and ways of accessing them are described further in the course of the document.

3.8 Final Notes

This sub-chapter contains a variety of notes regarding configuration, extensions, and implementation of the PrivMX server.

Storing of blocks, descriptors, mailboxes, and messages can be accomplished in the best way by the use of simple and fast key-value, document data bases.

Removing blocks - the requirement to automatically remove blocks that are not assigned to any object (descriptor, messages) can be accomplished, for example, by cyclical running of a garbage collector. It is also possible to extend the API with the `API.blockDelete` method and then the client program could store BIDs of blocks in (encrypted) Extra descriptor field.

There are plenty of options for configuring PrivMX servers for eligibility of connections - from blocking access to specific API methods within the framework of a standard connection, by limiting connections between servers, up to specifying specific IP addresses and users which can contact the server.

The default implementation of the server assumes that all mailbox API functions are available only in the logged-in connection. This means that it is possible to extend the API with mailbox methods "automatically" using the keys of the logged-in user. At that time, there would be no need to constantly sign requests and convey a public key.

Mailboxes number their messages and share their number - lastNumber. It is possible to implement additional "counters" that can optimize the process of checking boxes, reading and modifying messages.

4. Standard PrivMX client library

4.1 Preliminary information

Client module is the central part of systems based on PrivMX architecture. The server offers limited functionality and by default the client application implements the

entirety of the business logic of the system simultaneously ensuring proper data encryption.

The standard PrivMX client library provides basic components for building system logic based on:

- mechanisms for secure account creation and login;
- "high-level" objects such as directories, files, inboxes, outboxes etc;
- sharing of data;
- encryption of data before sending to the server, coupled with proper generation and storage of keys.

In this section we describe briefly the above functions.

4.2 Extended ECC keys

The standard PrivMX client module uses the so-called extended ECC keys (see BIP 32 specification) - these are ordinary 256bit privkey keys and pubkey 512bit (packaged to 257bit) expanded with an additional data element, so called chaincode with a length of 256 bit - "additional entropy source". The extended keys are written in this document with a plus sign - as privkey+chaincode and pubkey+chaincode.

Chaincodes are used by the PrivMX client in the process of derivation of new keys from existing keys (CKD function with BIP 32) and as keys to encrypt user data with symmetric algorithm (AES).

4.3 Creating accounts

The PrivMX TLS protocol allows for the client application to log in, i.e. to force conducting an additional handshake procedure based on SRP. To enable this, the server must be in possession of the SRP verifier. Ensuring this is one of the goals of the account creation procedure. Another goal of this procedure is to set and save the private and public key for the new user.

New users' accounts on the PrivMX server can be created by default by the client programs only on the basis of "invitations" - 32-byte random tokens. To generate tokens, the `API.generateNewUserToken` method is used, which is available in the logged-in connection for distinguished users ("administrators"). The first token for the first user is usually generated during the PrivMX server installation procedure.

Any connected client program can start the account creation procedure, if it has a valid token and has already asked the user to choose their name and password.

1. The client program randomly generates Salt (16 bytes) and selects NumberOfRounds (number from 4000-5000);
2. Determines MixedPassword = H(user password, Salt, NumberOfRounds), where H is a hashing algorithm set by the client program, PBKDF2-SHA512 for example;
3. Calculates the srpVerifier using hash = SHA512(MixedPassword) % 16 bytes;
4. Randomly generates MasterKey = extended private ECC key;
5. Sets privData = AES256Encrypt(MasterKey) with password equal to SHA256(MixedPassword);
6. Calculates identityKey = CKD(MasterKey, m/0'), then identityKeyPub = extended public key counted for identityKey;
7. Calls the `API.register` method to which it conveys:
   - the token received from the administrator;
   - new user name;
   - srpVerifier - will be used by the server during subsequent logins;
   - Salt, NumberOfRounds and hashing algorithm name - the server will provide this data to each client who will want to log in to that account;
   - privData, which will be safely downloadable for the client after login (see below);
   - identityKeyPub - a new user's public key that will be published by the server within the PrivMX PKI service;
   - signature - signature of the set of all the above data made with private identityKey.

The server saves the above data and uses them in subsequent attempts to log in to the newly created account.

4.4 Login and initialization of the client

A client application that uses the PrivMX module usually starts its operation with establishing a standard connection to the PrivMX server. For this purpose, it connects with a fixed API endpoint or at first uses the Service Discovery procedure for the specified hostname or user#hostname address.

If the application wants to obtain access to a particular user's data or wants to send a message on his behalf - it must perform the login procedure. Assuming that the client has already established the PrivMX TLS standard connection and has the user's ID and password provided by the user:

1. The client retrieves from the server the Salt and the NumberOfRounds saved for a given username (`API.getLoginParams`);
2. Using these data and the password of the user, it counts MixedPassword and srpVerifier - in the same way as during the creation of the account;
3. Asks, by calling the appropriate PrivMX TLS function, to perform the SRP handshake, as a result of which, the encryption keys for the connection are changed and the client obtains authorization as a user possessing the account, obtains access to the API methods for the logged users;
4. Retrieves from the server the user's privData (`API.getPrivData`);
5. Calculates MasterKey = AES256Decrypt(privData) with password equal to SHA256(MixedPassword);
6. Calculates identityKey and identityKeyPub - based on MasterKey, in the same way as during the setting up of the account.

The two keys obtained in result of performing the above procedure are the basis for the further operation of the standard client module:

- **The extended private MasterKey** is for the client program the "master handle" for user data stored on the server. Two additional keys are generated from it:
  - extended private key **HomeDir** = CKD(MasterKey, m/1') designating and giving full access to the "home directory" of the user;
  - extended private key **SinkList** = CKD(MasterKey, m/2') designating and giving full access to the file storing the private keys to the user mailboxes.
- **extended private IdentityKey** = CKD(MasterKey, m/0') is the user's private key which is used to decrypt messages addressed to the user and to confirm their identity (signatures) etc.

All of the above mentioned keys are generated by the client program and they are not stored on the server side.

After the first login, the standard client module cannot read the HomeDir directory and the SinkList file because they do not exist yet. They must be created and may be filled with initial data depending on the application. The default action is to create a blank home directory and the first mailbox which private key is placed in the SinkList file. The public key of the mailbox can be set as the default public SID of the user associated with address name#hostname.

4.5 Files and directories

The PrivMX Server does not offer API functions related to objects such as files or directories. If the client application wants to use such objects and store them on the PrivMX server, then it must simulate them using the available API methods. The standard client module does this by making the following assumptions:

- **file** is a descriptor with associated blocks. The file data is stored in blocks, and the file metadata in the descriptor's Extra field.
- **Directory** is a special file (in json format) that contains a list of names and keys of files and subdirectories that are to be located in a given directory. Descriptor's Extra field contains the directory metadata.

The HomeDirKey key obtained after login is an extended private key of the main directory. Blocks of that descriptor contain the list of all files and directories of the "root level" - their identifiers, public and private keys. In this way, the PrivMX client obtains the tree structure of the encrypted files and directories.

This structure and content of files are hidden from the server because the standard PrivMX client module encrypts the data which it places in the blocks and in the Extra descriptor fields. It uses symmetric encryption for that purpose (AES256) with chaincodes from extended ECC keys as passwords.

For example: in order to save a file in directory k, the client program must have an extended private key of this directory (kpriv+kchaincode) to prove that it has write/modify access rights. It then performs the procedure of creating a new descriptor with appropriately encrypted data and appending it to directory k:

1. Randomly creates a 256-bit KS key to encrypt the file data.
2. Divides the file into blocks, encrypts each of them with the KS key and assigns BID (hash of cryptogram) to them.
3. Randomly creates a new extended private key (dpriv+dchaincode) and generates an extended public key (dpub+dchaincode) from it - the new file identifier.
4. Prepares the Metadata structure that includes:
   - type = "file"
   - data = { filename, mimetype, size, date of creation and modification of the file }
   - blockskey = KS
5. Creates a new descriptor on the server, sends prepared blocks, dpub key, proper DID, and:
   - Extra = AES256Encrypt(data = Metadata, password = dchaincode)
   - Signature of the entire request made with the dpriv key
6. Using the private kpriv key, modifies the directory/descriptor k so that it adds to its data (json) a line containing:
   - name = filename
   - type = "file"
   - pub = (dpub+dchaincode)
   - encpriv = AES256Encrypt(data = dpriv, password = kpriv)

The possibility of reading the data connected with the file/directory descriptor depends on the keys that the client program possesses:

- **Possessing of only DID** does not give access to decrypted data. It allows to check whether a given descriptor exists, and thanks to access to the Blocks field, allows the downloading of blocks of encrypted data.
- **Possessing of extended public key (dpub+dchaincode)** allows the above and additionally decrypts the Extra field, that is, obtaining the file metadata and the "blockskey" key. Thanks to the latter one the client can decrypt the data blocks - they will then receive the decrypted file data. In case of a directory the client will obtain the list of the files and subdirectories that are located in it and thereby the possibility to read the directory "inwards".

- **Possessing of an extended private key (dpriv+dchaincode)** allows to read all the data as above and additionally enables the client program to modify the descriptor and remove it. In case it is a directory - it is possible to decrypt the private keys of files and subdirectories and consequently to modify and delete them.

4.6 Sending and receiving messages

After conducting the login procedure, the standard PrivMX client library obtains the SinkList key to the file containing the user's mailboxes list. For each box it is possible to find here the name, description and extended private key. This file is updated accordingly after creating a new mailbox or removing an unwanted one.

The client program must know the recipient's mailbox identifier (SID) in order to send the message to the recipient. Distribution of SIDs can be implemented in various ways. By default a decentralized system configuration is assumed and therefore the standard client module creates the first mailbox right after the first login, and adds it to the SinkList and public user's record in the PrivMX PKI service. Thanks to PrivMX PKI, message senders can retrieve (and verify) the default SID of the user. More on this topic can be read in the last chapter of the document.

The standard client module uses `API.messagePut` in order to leave its message in the right mailbox on the target server. However, it uses message blocks and the Extra field in a specific way:

- data blocks are mainly used for transferring files - attachments to messages. For each attachment, a new key is generated and used to encrypt blocks of this attachment (AES256Encrypt).
- The main structure of the message is placed in the Extra field and is encrypted with a common secret generated according to the ECIES scheme for the sender's IdentityKey private key and the SID public key of the mailbox. The standard structure of the message includes, among others:
  - title and contents of the message - text fields formatted freely by the client;
  - senderName - sender's name (e.g. "John Smith")
  - attachments - list of attachments; for each one there are specified:
    * name and mimetype;
    * blocks - BIDs of this attachement's blocks;
    * key - the key encrypting these blocks.

The use of the public key of the mailbox (SID) in the ECIES schema causes that the content of the transmitted data will be able to be retrieved, decrypted and read only by the private key holder of the target mailbox. If this is for example the default box of the user who has not shared any of their keys with anybody (which is the most common case), the message will be read only by that user.

4.7 Data sharing

The standard way to share data within the framework of the PrivMX architecture consists in distributing appropriate keys to specific objects stored by the server. For example, if you want to share a read-only file with another user, you must give them a public key to it.

Public keys are calculated from private keys, so for example, by distributing a private key to the descriptor, we give the recipient the possibility to modify and also to read it. In the case of mailboxes, obtaining the right to read (private key) also gives the right to put new messages in it.

The way of distributing the keys depends entirely on the client application, its purpose, way of operation and the user's interface solutions. Keys to common objects (most often to files and directories) are most frequently sent through messages to default or special users' mailboxes. Such an approach also enables data sharing between users from different servers, although the condition is that the configuration of servers should allow access to data also for "foreign users" also.

4.8 Notes on client implementation and modification

All the keys used by the PrivMX architecture based system are randomly generated by the client module, therefore one of the most important elements affecting the security of such a system is the high quality of the random number generator. Each implementation of the PrivmX client module should take care of this.

Account setup and login procedures can be extended by using the 2-factor authentication scheme.

Grouping of requests is done by PrivMX TLS protocol optimization, that improves conducting operations on files and directories that require making multiple API queries. Of course, such optimization also requires server-side support.

Derived by default, after login, the SinkList key to the file containing the list of private mailboxes can be replaced with a regular file in the HomeDir directory.

It is also possible to store the private keys of mailboxes inside themselves - in their Extra field. This can be a useful optimization when the server API is extended with functions that can list all user's mailboxes. The private keys of mailboxes should then be encrypted using the chaincode of the HomeDir key as a password (for example).

By default, through the procedure of CKD (keys derivation) only the "master mountpoints" are obtained for the user, but the client application can create for own needs any larger hierarchies of private keys.

Client programs can create messages that can be decrypted only by specific users (and not by any owner of the private key of the mailbox). For this purpose, in the content of the message there may be included a portion encrypted with ECIES scheme using the sender's private key and the public key of the recipient (and not the mailbox).

Server configuration parameters that are relevant to client programs (e.g. maximum data block size) are available for

reading by the `API.getServerConfig` method.

5. PrivMX PKI

5.1 PrivMX Public Key Infrastructure

PrivMX PKI are elements of PrivMX architecture that perform publication and verification of public keys for users and servers. They are important primarily in case of decentralized systems and enable, inter alia, verification of messages senders - checking whether a just received message contains the correct sender's signature.

PrivMX does not use classical PKI solutions for this purpose which assume the use of external certification authorities. It also does not use PGP solutions based on mutual the signing of keys published on external servers.

The PrivMX Infrastructure of public keys is based on the ideas connected with CONIKS (https://coniks.cs.princeton.edu/) thanks to which, one of the assumptions of the PrivMX secure communication can be fulfilled - eliminating or minimizing the role of a "trusted third party" controlled by the external companies/organizations in respect of the application.

In case of decentralized applications, in the default configuration, each PrivMX server maintains its own base of public keys (keystores) of its users and makes it available through the API to client programs.

PrivMX keys bases are private databases publishing hash (SHA256) of its entire content and history of changes of the hash using the append-only structure (i.e. blockchain). Thanks to this, these databases can be subject to audit and monitoring for consistency conducted by other PrivMX servers. Each retrieved public key (keystore) is supplied with a proof of correctness that is associated with the hash of the entire database. Verification of correctness of the downloaded keystore consists of two steps - checking the correctness of database hash based on its history (also performed together with other servers - audits, web-of-trust) and verification of the provided proof of keystore correctness.

The following subchapters describe in more detail the features and performance of PrivMX PKI and the default way to use these mechanisms in a decentralized application.

5.2 Private key base with public history of changes

The PrivMX architecture assumes the handling of two types of keystores:

- **pgpkeystore** - a PGP-compatible keystore containing a full history of changes. It allows to perform operations such as the annulment of keys (revoke) and the adding of subkeys (and other PGP packages i.e. public key, signature, userid). Keystore changes of this type can be made by the holder of the distinguished private key.
- **simplekeystore** - a simple structure keystore that contains a list of public keys. Such a keystore can be modified by owners of any matching private keys. The history of keystore changes of this type are not saved in it.

Keystores, in addition to key data and information on their type, can also contain attachments of any files.

Standard server and client implementations of PrivMX provide the appropriate functions for using both types of keystores. They allow to create, read and modify them using private keys. This document omits the description of these functions.

The set (database) of keystores stored by the PrivMX server has the structure of the binary Merkle tree. In simple terms it can be presented as such that the tree leaves are keystores and the nodes other than leaves consist of the SHA256 hashes of their descendants. The tree root hash is therefore the hash of the entire database of keystores - it is often called "tree revision" or "base revision".

Every change in the tree (i.e., adding or modifying of the keystore) causes changes to the corresponding tree branches and changes the hash of the entire base. In addition to the history of tree changes, PrivMX PKI also stores the history of changes of hash of the entire base in the form of the append-only list; each element of this history contains subsequent base hash and hash of the previous element in the history.

This list, similar to the blockchain structure, is publicly made available by the PrivMX server (`API.pkiGetHistory`). It does not contain any data (keys and attachments) in itself that are stored by PrivMX PKI, but it is sufficient to verify the keys retrieved from the given server and to perform external base audits.

5.3 Downloading and verifying public keys

Access to keystores at the server API level is achieved through the use of readable alphanumeric identifiers such as "server:simplito.com", "user:john#example.com". These identifiers are transformed by the server into zero-one paths in a binary tree using a verifiable random function (VRF). PrivMX PKI uses 256-bit paths-identifiers and compresses them in order to avoid the creation of trees that are too deep.

The parameters of the `API.pkiKeyStoreGet` function are:

- name - alphanumeric identifier containing a colon (it designates namespaces);
- IncludeAttachments - if equal true (1), the function, in addition to key data, also returns all attachments of the selected keystore;
- revision - revision (hash) of the database from which the keystore has to be retrieved.

The function returns keystore data (if the corresponding keystore exists in the database), correctness proof, and attachments, if required. "Correctness proof" is the VRF-function-generated string containing the corresponding binary path fragments and all hashes of Merkle tree nodes (including root) needed for the client program to verify that the downloaded keystore belongs to the base of the required revision, and

that there is no missing data in it. Appropriate functions for this verification are provided in the standard PrivMX client implementations.

Omitting the revision parameter when retrieving the keystore causes that the server will use the revision of the database at the time of the query. Then the verifying of the correctness requires additional retrieval of the database history in order to find in it the database hash-revision contained in the proof.

Downloading the base history and keystores is possible after establishing the "standard" PrivMX TLS connection - it is available by default for all client programs. The server also provides functions `API.pkiKeyStorePut`, `API.pkiKeyStoreModify`, `API.pkiKeyStoreDelete` the access to which, is limited and configured depending on the specific application.

5.4 Audits, consensus, web-of-trust

Despite the possibility of the verification described above, the client program must "believe" the server from which it downloads the keys. In the situation of decentralized and open application it is not an obvious issue when many servers belonging to different people and organizations are used. In order to avoid situations where one client is presented a history and keys other than those which are seen by the rest of the network - it is possible to check the server by performing an audit.

The audit consists in sending to selected (other) servers the requests for verification of the history of the database of the server being inspected. This involves establishing a connection and calling `API.pkiConfirmRevision` in selected servers giving:

- hostname - domain name of the server we want to check;
- revision - hash of the database the existence of which on the target server we want to confirm.

Servers that are asked for the audit check if the hostname server provides them with a consistent history containing the given revision – they make the appropriate calls of `API.pkiGetHistory` and use the previously saved hostname server database history. Each of the servers that are asked for help finally return information whether they also see the same revision on the hostname server or not.

After collecting this information, the client program can inform the user accordingly about the level of credibility of the server. If all answers from the auditors are positive or all are negative, then we can talk about reaching an agreement (consensus) about the reliability of the server being inspected. All intermediate situations where the responses are different or when some servers are not responding, need configuration and interface in the client application. Appropriate heuristics can be used to "calculate the level of the consensus" or special operations can be made available to the user – marking the server as untrusted, renewed attempt of verification, consultations with other servers users, etc.

The number and selection of server auditors are two of the most important issues when outsourcing audits. The application that performs them can use the established, trusted PrivMX servers and/or may allow users to build their own web-of-trust - networks of trusted servers. It can choose auditors at random or according to any of their own criteria. The PrivMX architecture does not determine the manner of selecting servers, it depends on the specific application.

5.5 Examples of using PrivMX PKI

Below we describe the actions taken by default by the PrivMX Server configured to work within the framework of a decentralized application (for example, the application PrivMX WebMail, available at the address https://privmx.com).

- During PrivMX server installation, the server's private key is set, and the corresponding public key is placed in the PrivMX PKI under the identifier "server".
- Every call of the `API.register` registering a new user inserts the public key of the user (IdentityKeyPub) into the database with the identifier "user:username". This key is accompanied by attachments that include the user's avatar, the SID of their public mailbox and other public profile data. This keystore is used by other users to verify messages signed by the user username#hostname.
- The first user creating an account on a new server obtains administrator rights. In the course of the first login of this user, new keystore "admin" is created (of the type of simplekeystore) which stores public keys of all administrators of the server. This keystore is made available to all users of the server who can then check whether, for example, a given message comes from the server's administrator.

The mentioned PrivMX WebMail application also allows server administrators to create their own webs-of-trust. Admins send special invitation messages to other admins, and the acceptance of such invitations causes that both sides add each other to their private lists of trusted servers. These lists are stored as attachments in the "admin:hostname" keystore and they are read by each user during each login. In this way the administrator publishes (suggests) lists of trusted servers to his users/clients. In addition, the operation of the `API.pkiRevisionConfirm` function is limited on the server so that it only performs an audit when it is requested by the server from the list of trusted servers.