Security Note

BBM Enterprise SDK

Published: 2017-06-14
SWD-20170614150814325

Contents

About this guide

1

This guide is intended for senior IT professionals responsible for evaluating the product and planning its deployment, as well as anyone who's interested in learning more about the BBM Enterprise SDK or the BBM Enterprise SDK security features. After you read this guide, you should understand how BBM Enterprise SDK adds additional protection to messages.

Security and encryption

2

The BBM Enterprise SDK was designed to comply with the following three security principles:

- Messages are digitally signed, so you're assured of who sends each message in your app
- Messages are encrypted, so you're assured that only the intended recipient can read the message
- Messages are subjected to integrity signature checks, so you're assured the message isn't modified in transit

In the BBM Enterprise SDK, messages are protected from being viewed or modified by anyone other than the sender and intended recipients. The encryption, signing, and symmetric keys used to protect communications are stored and distributed in a cloud storage system that you choose.

Encryption, signing, and symmetric keys are locally generated by your application's endpoint. You should store these keys in your infrastructure's cloud, or in a generic secure cloud storage solution.

Your app will generate two private and public key pairs for each user, using ECDSA secp521r1: one key pair for signing, and one key pair for encryption. The public signing and encryption keys are distributed to each user of your app, and these signing and encryption keys must be stored in a generic storage cloud.

All sent messages are encrypted using a per-message encryption key, and signed by the sender's private signing key. All received messages are verified using the sender's public signing key, and decrypted using a per-message encryption key.

Requirements: cloud key storage solution

- The cloud storage system must authenticate users that are using tokens or credentials managed by your application
- The cloud storage system must allow a user to read and write private data
- The cloud storage system must allow a user to publish or send public data that other users can read only

Private data

The cloud storage solution must include restricted read and write access (private data can only be read or written if the owner of that data is logged into the app). The following private data must be stored:

- Private encryption keys
- Private signing keys
- Symmetric encryption keys

Public data

The cloud storage solution must include public read access, and restricted write access (where private data can only be written if the owner of that data is logged into the app). The following public data must be stored:

- Public encryption keys
- Public signing keys

About key storage

In order for your app to function, exported encryption, signing, and symmetric keys must be stored and distributed in a generic storage cloud.

A common storage schema for the encryption, signing, and symmetric keys must be maintained across all product lines, so your app can use any combination of the SDKs to fulfill your needs. For example, both the Android and iOS versions of your app must be able to exchange encryption, signing, and symmetric keys, and users must be able to switch between Android and iOS without losing their encryption, signing, or symmetric keys.

For instance, you can use a cloud-hosted NoSQL real-time cloud database for encryption, signing, and symmetric key storage. The BBM Enterprise SDK includes a sample for using Firebase as the cloud key storage solution. For more information, see Appendix: Using Firebase to store and share keys.

Identity encryption

Identity messages are messages exchanged between two users outside of a chat (for instance, an invitation to join a chat from one user to another). Identity messages are encrypted using a per-message key generated by both the sender and recipient: the remote user's public encryption key and the local user's private encryption key are used to generate a ECDH secp521r1 528-bit shared secret. This shared secret is combined with the message counter and nonce to make a secret that is used to derive a 32-byte key using ANSI-X9.63-KDF.

The message counter is incremented for each message, while the nonce is randomly generated for each message. These values are sent in the plaintext portion of the message.

The derived key is used to encrypt or decrypt the identity message using AES CTR mode.

The following demonstrates the mathematical notation of identity encryption.

```
A calculates K_shared = EC-DH(K'enc_A, Kenc_B)
B calculates K_shared = EC-DH(K'enc_B, Kenc_A)
Headers = Nonce (64-bit) || Counter (32-bit) || <other_message_headers>
K_msg = KDF(Headers || K_shared)
Message = Enc_{K_msg, Nonce}(Payload)
```

Chat encryption

Chat messages are encrypted using a per-message key generated by both the sender and recipient. The chat key is sent to the user as a protected identity message, requesting that the user to join the chat. the remote user's public encryption key and the local user's private encryption key are used to generate a ECDH secp521r1 528-bit shared secret. The chat key is combined with the message counter and nonce to create the secret, which is used to derive a 32-byte key using ANSI-X9.63-KDF.

The message counter is incremented for each message, while the nonce is randomly generated for each message. These values are sent in the plaintext portion of the message.

The derived key is used to encrypt or decrypt the identity message using AES CTR mode.

The following demonstrates the mathematical notation of chat encryption.

```
K_Session = mailbox key (256-bit)
Headers = Nonce (64-bit) || Counter (32-bit) || <other_message_headers>
K_msg = KDF(Headers || K_Session)
Message = Enc_{K_msg, Nonce}(Payload)
```

Video and voice encryption

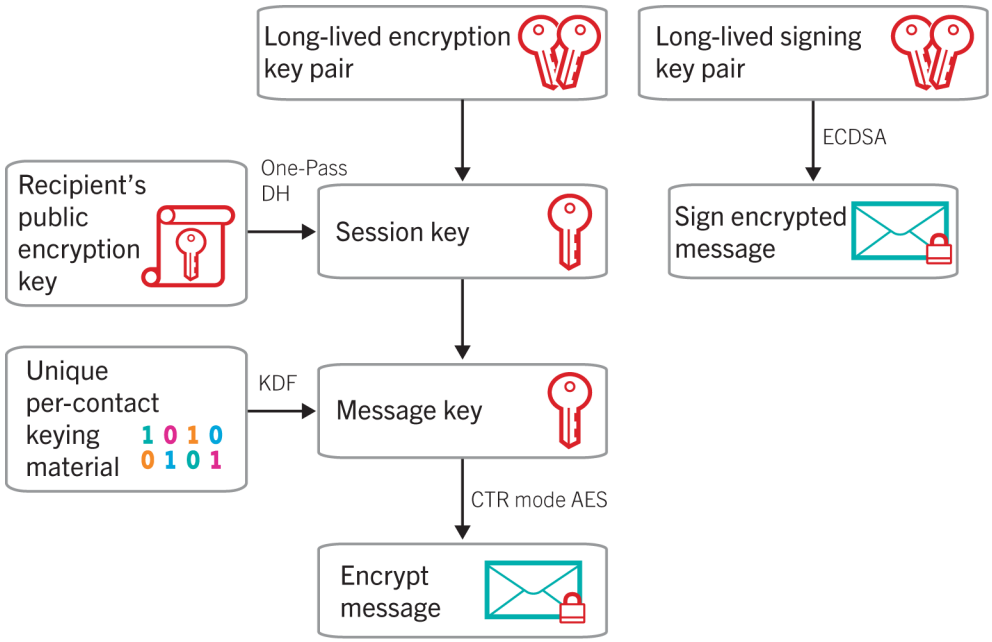
The BBM Enterprise SDK provides end-to-end security for a real-time media streaming channel, allowing users to make secure voice or video calls. Encryption keys are exchanged using DTLS-SRTP. Additionally, the DTLS fingerprints are encrypted and signed using the security keys of both participants in the video or voice call, adding another layer of security and identity verification. Subsequent encryption of the real-time media streaming channel follows SRTP specification, using AES 128 in CTR mode as a cipher.

Signing

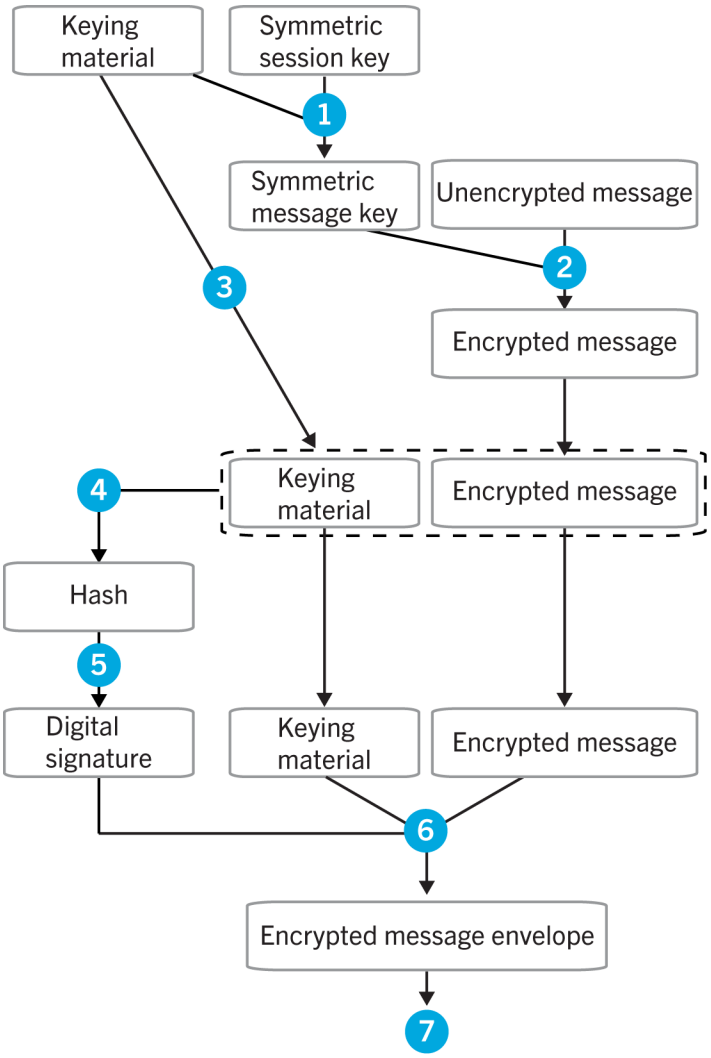
Before identity and chat messages are signed, the encrypted data is combined with the message counter, nonce, and identity information. It is then signed with the sender's private signing key, using ECDSA secp521r1 SHA-512. The signature information is sent in the plaintext portion of the protected message, to be verified by the recipient. The recipient combines the received encrypted data, message counter and nonce, and identity information to verify it using ECDSA secp521r1 SHA-512 with the sender's public signing key.

Security Diagrams

Encryption and Signing



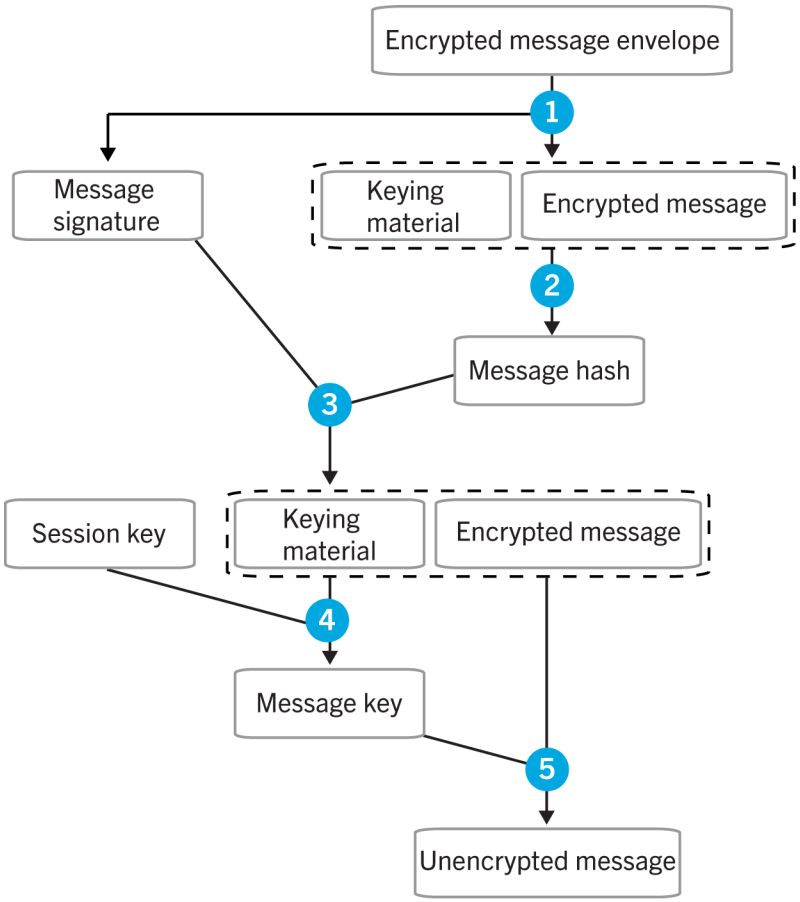
Data flow: Sending an encrypted message



When a user sends a message, the BBM Enterprise SDK performs the following actions:

1. Establishes a 256-bit AES message key from the session key and unique keying material
2. Encrypts the message with the symmetric key using AES in CTR mode
3. Includes the keying material to recreate the message key in the unencrypted portion of the message
4. Hashes the whole message using SHA-512
5. Signs the hash with the sender's private signing key (ECC-521) using ECDSA
6. Wraps the parts in a message envelope
7. Passes the message to the transport layer

Data flow: Decrypting a received message



When a user receives a message, the BBM Enterprise SDK performs the following actions:

1. Parses the envelope containing the encrypted message
2. Hashes the encrypted message using SHA2-512
3. Verifies the message signature using the sender's public key and the encrypted message hash; a pass indicates that the message is authentic
4. Derives the message key from the session key and the unencrypted keying material
5. Decrypts the message using AES in CTR mode

Glossary

3

AES	Advanced Encryption Standard
BES12	BlackBerry Enterprise Service 12
CTR	Counter
DH	Diffie-Hellman
ECC	Elliptic Curve Cryptography
ECDH	Elliptic Curve Diffie-Hellman
ECDSA	Elliptic Curve Digital Signature Algorithm
EC-SPEKE	Elliptic Curve – Simple Password Exponential Key Exchange
FIPS	Federal Information Processing Standards
HMAC	keyed-hash message authentication code
KDF	key derivation function
MAC	message authentication code
NIST	National Institute of Standards and Technology
SHA	Secure Hash Algorithm
SMS	Short Message Service
TLS	Transport Layer Security

Appendix: Using Firebase to store and share keys

4

The sample app included in the BBM Enterprise SDK uses Firebase as a storage cloud solution for storing and sharing encryption, signing, and symmetric keys between users of your app. Firebase offers a cloud-hosted NoSQL real-time database, located here: https://firebase.google.com/docs/database/. Firebase also offers a variety of authentication system integration options that allow you to be flexible in your design. For more information, visit https://firebase.google.com/docs/auth/.

The mechanism for storing private and public encryption, signing, and symmetric keys must adhere to the following rules:

Rules for a user's private key data:

- Only the user may read or write their private key data
- Other users have no read or write access to another user's private key data

Rules for a user's public key data:

- Only the user may read or write their public key data
- Other users only have read access to another user's public key data

This design also maintains an important separation between a user's regId, and the identity employed by the authentication system. Therefore, you can use any drop-in authentication solution that is compatible with Firebase. For example, you can use popular federated identity providers such as Google Sign-In and Facebook Login.

An alternative design, which can authoritatively link the user's regId and their system user ID, is not discussed beyond acknowledging that the data store design is made much simpler and rigorous when a custom authentication system is used to do either of the following:

- Set the regId as a user's OAuth UID
- Establish an authoritative claim in the OAuth token (which declares the user's ownership of a regId), that is guaranteed to have a 1:1 mapping with the UID by the authorization system issuing the token.

For more information on these rules, refer to the Firebase Realtime Database documentation for Security & Rules, located here: https://firebase.google.com/docs/database/security/.

Because the Firebase Realtime Database employs JSON as a means of data storage and transfer, it is recommended that all encryption, signing, and symmetric keys stored in this database be base64url-encoded, without padding.

It is also recommended that any keys retrieved from the public key store be monitored for changes, to ensure that the most up-to-date copy of a user's public encryption, signing, and symmetric keys are available to the client, and are cached locally. For more information on monitoring encryption, signing, and symmetric keys for changes, visit one of the following websites:

For Android developers:https://firebase.google.com/docs/reference/android/com/google/firebase/database/Query.html#addChildEventListener(com.google.firebase.database.ChildEventListener)

For iOS developers:https://firebase.google.com/docs/reference/ios/firebasedatabase/api/reference/Classes/FIRDatabaseReference#/Attaching%20observers%20to%20read%20data.

Security and data integrity rules

The following rules may be installed into a project's Realtime Database via the project's Firebase console, which is located here: https://console.firebase.google.com/.

```
// All keys used in Firebase are subject to the following limitations:
//https://firebase.google.com/docs/database/web/structure-data
//
//   If you create your own keys, they must be UTF-8 encoded, can be a maximum
//   of 768 bytes, and cannot contain ., $, #, [, ], /, or ASCII control
//   characters 0-31 or 127.
//
//   This will impose restrictions on the values that can be used as keys within
//    the Firebase database. The restrictions on the keys, if any, are described
//    in the key description that precedes the key as it is introduced in the
//    rules.
//
//    The following rules may be installed into a project's Firebase Realtime
//    Database via the project's Firebase console:
// https://console.firebase.google.com/
{
    "rules": {
        // The Private Key Store is used to store a user's private keys for both
        // encryption and signing purposes as well as the symmetric key for each
        // mailbox to which the user is subscribed.
        //
        // The private key store entry containing a user's regId must be the
first
        // user record created.
        //
        // The private key store of a user cannot be deleted until the regId
        // ownership claimed in the regIds index has been removed. This
        // restriction ensures that regId entries and public key store data for a
        // user cannot be orphaned.
        "privateKeyStore": {
            // Indexed by the Firebase user's OAuth ID. The UID is system
assigned
            // and requires no special treatment or encoding to be used as a key.
            "$uid": {
                // Reading is restricted to authenticated users whose UID matches
the
                // UID of the record.
                ".read": "auth != null && auth.uid == $uid",
                // The following restrictions are used to determine whether a write
to
                // the record is allowed:
                // 1) The user must be authenticated and their UID must match the
UID
                // of the record; and
                // 2) New data being written is always allowed (it must
additionally
                // pass verification checks); or
```

```
                // 3) Deleting a record that does not exist is always allowed;
or
                // 4) The deletion of a record can only proceed if:
                // * The claim in the regIds index associated with this record
                // does not exist; or
                // * The claim in the regIds index associated with this record
                // is not associated with the UID of the record being removed.
                ".write": "
                    auth != null && auth.uid == $uid &&
                    (newData.exists() || !data.exists() ||
                    !root.child('regIds').child(data.child('regId').val()).exists()
||
                    root.child('regIds').child(data.child('regId').val()).val() !=
$uid)",
                // When the private key store data is being set outright, it
must
                // always contain a regId member.

                ".validate": "newData.child('regId').exists()",
                // The regId is the BBM identifier for a user and it must be
                // authoritatively associated with an authorized user of the
system.
                // The first part of guaranteeing a 1:1 relationship between a
regId
                // and a UID is to declare that a UID 'owns' a given regId when
the
                // private key store data is created.
                //
                // Because claiming ownership of a regId is a two step process, it
is
                // possible to encounter races between different users who have
been
                // incorrectly assigned the same regId.  A user who is unable
to
                // complete the process to claim a regId must still be able to
change
                // the regId associated with their UID so that they can declare
their
                // intent to make a claim on another regId if they are somehow able
to
                // obtain one.
                //
                // In order for the regId to be written it must meet the
following
                // criteria:
                // 1) The new regId value must be a non-empty string; and
                // 2) The existing regId value must:
                // * Not exist (new record); or
                // * The existing regId must not yet be claimed in the regIds
                // index; or
                // * The existing regId must not be claimed by the current user;
or
                // * Be the same regId as the new value being claimed; and
                // 3) The new regId value must:
                // * Not yet be claimed in the regIds index; or
                // * Be already claimed by the current user
                //
                // The regId value is system assigned and numeric and may be
inserted
                // without any additional encoding.
                "regId": {
```

```
                    ".validate": "
                    newData.isString() && newData.val().length > 0 &&
                    (! data.exists() ||
                    ! root.child('regIds').child(data.val()).exists() ||
                    root.child('regIds').child(data.val()).val() != auth.uid
||
                    newData.val() == data.val()) &&
                    (! root.child('regIds').child(newData.val()).exists()
||
                    root.child('regIds').child(newData.val()).val() == auth.uid)"
                },
                // The encryptionKey must be a non-empty string when it is being
set.
                //
                // The encryptionKey value contains binary data and must be encoded
as
                // a base64url string without padding.
                "encryptionKey": {
                    ".validate": "newData.isString() && newData.val().length >
0"
                },
                // The signingKey must be a non-empty string when it is being
set.
                //
                // The signingKey value contains binary data and must be encoded
as
                // a base64url string without padding.
                "signingKey": {
                    ".validate": "newData.isString() && newData.val().length > 0"
                },

                // Each mailbox symmetric key must be a non-empty string when it
is
                // being set.
                "mailboxes": {
                        // The mailboxId assigned by the BBM infrastructure may
contain
                        // characters that are illegal for use as a Firebase key.  To
ensure
                        // that it can be used as a key, it must be encoded as a
base64url
                        // string without padding.
                        //
                        // The mailbox key value contains binary data and must be
encoded as
                        // a base64url string without padding.
                        "$mailboxId": {
                            ".validate": "newData.isString() && newData.val().length >
0"
                        }
                    }
                }
            },

// The mapping of regIds to the UIDs that have claimed them. This is used
// as an index to maintain data integrity and complete the 1:1 guarantee
// between UIDs and regIds employed within this key store.
//
// An entry in this index authoritatively declares the UID associated with
// the regId as the owner. Thus, this index is also used to enforce
// access control permissions on the public key store data (defined
```

```
// below).
//
// An entry in this index must be the second user record created.
//
// An entry in this index cannot be deleted until the public key store
// data associated with the regId has been removed. This restriction
// ensures that public key stores cannot be orphaned.
//
// The regId assigned by the BBM system is numeric an has no need of
// further encoding to be used as a key.
"regIds": {
    "$regId": {
        // No one is allowed to read/query this index.
        ".read": false,
        // The following restrictions are used to determine whether a write to
        // this record is allowed:
        // 1) A user must be authenticated to write; and
        // 2) An existing entry must be owned by the authenticated user
        // attempting the write; and
        // 3) New data being written is always allowed (it must additionally
        // pass verification checks); or
        // 4) Deleting a record that does not exist is always allowed; or
        // 5) The deletion of a record can only proceed if there is no
        // public key store data associated with the regId.
        ".write": "
            auth != null &&
            (!data.exists() || data.val() == auth.uid) &&
            (newData.exists() || !data.exists() ||
            !root.child('publicKeyStore').child($regId).exists())",
        // In order for the record to written, the UID being associated
with
        // the regId must meet the following criteria:
        // 1) Must be the UID of the current authenticated user; and
        // 2) The current authenticated user must have an entry in the private
        // key store; and
        // 3) The current authenticated user's private key store entry must
        // have a regId value declared; and
        // 4) The current authenticated user's regId value declared in their
        // private key store entry must match the regId or the record being
        // written.
        ".validate": "
            newData.val() == auth.uid &&
            root.child('privateKeyStore').child(auth.uid).exists() &&
            root.child('privateKeyStore').child(auth.uid).child('regId').exists()
&&
            root.child('privateKeyStore').child(auth.uid).child('regId').val() ==
$regId"
        }
    },

// The Public Key Store is used to store a user's public keys for
// both encryption and signing purposes.
//
// To create a public key store entry, a user must first have claimed the
// regId under which the public keys are to be advertised. This means
// they must first have:
// 1) a private key store entry with the regId value
// 2) a regIds entry associating the regId value with their UID
//
// Public key data may be deleted by the regId owner without restriction.
"publicKeyStore": {
```

```
    // The public key data is stored by regId. This allows other users to
    // find the public keys of any user for whom they have a regId.
    //
    // The regId assigned by the BBM system is numeric an has no need of
    // further encoding to be used as a key.
     "$regId": {
        // Any authenticated user can read the public key data.
        ".read": "auth != null",
        // The following restrictions are used to determine whether a write to
        // the record is allowed:
        // 1) The user must be authenticated; and
        // 2) The user must be the owner of the regId in the regIds index; or
        // 3) A record that does not exist is being deleted.
        ".write": "
            auth != null &&
            (root.child('regIds').child($regId).exists() &&
            root.child('regIds').child($regId).val() == auth.uid) ||
            (!newData.exists() && !data.exists())",
            // The encryptionKey must be a non-empty string when it is being
set.
            //
            // The encryptionKey value contains binary data and must be encoded
as
            // a base64url string without padding.
            "encryptionKey": {
                ".validate": "newData.isString() && newData.val().length > 0"
            },
            // The signingKey must be a non-empty string when it is being set.
            //
            // The signingKey value contains binary data and must be encoded as
            // a base64url string without padding.
            "signingKey": {
                ".validate": "newData.isString() && newData.val().length > 0"
                }
            }
        }
    }
}
```

Sample data

The key storage is separated into three separate sections, allowing the configuration of different access controls to the private and public data associated with each user.

```
{
    // The Private Key Store is used to store the private key data associated
    // with each user.  Only the logged in user whose UID matches the record may
    // read or write each the record.
    "privateKeyStore": {
        "uid1" : {
        "regId": "1234",
        "encryptionKey": "base64url encoded without padding private encryption key for
user uid1",
        "signingKey": "base64url encoded without padding encoded private signing key
for user uid1",
```

```
        // The mailboxId keys are also base64url encoded without padding to
ensure
        // the key values are compatible with firebase key restrictions.
        "mailboxes": {
            // mailboxId1
            "bWFpbGJveDE": "base64url encoded without padding key for
mailboxId1",
            // mailboxId2
            "bWFpbGJveDI": "base64url encoded without padding key for
mailboxId2",
            ...
            },
        "uid2": {
            "regId": "5678",
            ...
        }
    },

    // The regId index that, in combination with the regId a UID has declared in
    // their private key store, enforces a 1:1 relationship between a regId and
    // the UID.
    "regIds": {
        "1234": "uid1",
        "5678": "uid2"
    },

    // The Public Key store is used to store the public key data associated with
    // each user. Only the logged in user whose ownership of the regId is
    // confirmed by the regIds index may write to the record. Any logged in
    // user may read any record in this section.
    "publicKeyStore": {
        "1234": {
        "encryptionKey": "base64url encoded without padding public encryption key for
user uid1",
        "signingKey": "base64url encoded without padding public signing key for user
uid1"
        },
        "5678": {
            ...
        }
    }
}
```

Legal notice

5

REFERENCED HEREIN INCLUDING WITHOUT LIMITATION ANY OF THE FOLLOWING DAMAGES: DIRECT, CONSEQUENTIAL, EXEMPLARY, INCIDENTAL, INDIRECT, SPECIAL, PUNITIVE, OR AGGRAVATED DAMAGES, DAMAGES FOR LOSS OF PROFITS OR REVENUES, FAILURE TO REALIZE ANY EXPECTED SAVINGS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, LOSS OF BUSINESS OPPORTUNITY, OR CORRUPTION OR LOSS OF DATA, FAILURES TO TRANSMIT OR RECEIVE ANY DATA, PROBLEMS ASSOCIATED WITH ANY APPLICATIONS USED IN CONJUNCTION WITH BLACKBERRY PRODUCTS OR SERVICES, DOWNTIME COSTS, LOSS OF THE USE OF BLACKBERRY PRODUCTS OR SERVICES OR ANY PORTION THEREOF OR OF ANY AIRTIME SERVICES, COST OF SUBSTITUTE GOODS, COSTS OF COVER, FACILITIES OR SERVICES, COST OF CAPITAL, OR OTHER SIMILAR PECUNIARY LOSSES, WHETHER OR NOT SUCH DAMAGES WERE FORESEEN OR UNFORESEEN, AND EVEN IF BLACKBERRY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW IN YOUR JURISDICTION, BLACKBERRY SHALL HAVE NO OTHER OBLIGATION, DUTY, OR LIABILITY WHATSOEVER IN CONTRACT, TORT, OR OTHERWISE TO YOU INCLUDING ANY LIABILITY FOR NEGLIGENCE OR STRICT LIABILITY.

THE LIMITATIONS, EXCLUSIONS, AND DISCLAIMERS HEREIN SHALL APPLY: (A) IRRESPECTIVE OF THE NATURE OF THE CAUSE OF ACTION, DEMAND, OR ACTION BY YOU INCLUDING BUT NOT LIMITED TO BREACH OF CONTRACT, NEGLIGENCE, TORT, STRICT LIABILITY OR ANY OTHER LEGAL THEORY AND SHALL SURVIVE A FUNDAMENTAL BREACH OR BREACHES OR THE FAILURE OF THE ESSENTIAL PURPOSE OF THIS AGREEMENT OR OF ANY REMEDY CONTAINED HEREIN; AND (B) TO BLACKBERRY AND ITS AFFILIATED COMPANIES, THEIR SUCCESSORS, ASSIGNS, AGENTS, SUPPLIERS (INCLUDING AIRTIME SERVICE PROVIDERS), AUTHORIZED BLACKBERRY DISTRIBUTORS (ALSO INCLUDING AIRTIME SERVICE PROVIDERS) AND THEIR RESPECTIVE DIRECTORS, EMPLOYEES, AND INDEPENDENT CONTRACTORS.

IN ADDITION TO THE LIMITATIONS AND EXCLUSIONS SET OUT ABOVE, IN NO EVENT SHALL ANY DIRECTOR, EMPLOYEE, AGENT, DISTRIBUTOR, SUPPLIER, INDEPENDENT CONTRACTOR OF BLACKBERRY OR ANY AFFILIATES OF BLACKBERRY HAVE ANY LIABILITY ARISING FROM OR RELATED TO THE DOCUMENTATION.

Prior to subscribing for, installing, or using any Third Party Products and Services, it is your responsibility to ensure that your airtime service provider has agreed to support all of their features. Some airtime service providers might not offer Internet browsing functionality with a subscription to the BlackBerry® Internet Service. Check with your service provider for availability, roaming arrangements, service plans and features. Installation or use of Third Party Products and Services with BlackBerry's products and services may require one or more patent, trademark, copyright, or other licenses in order to avoid infringement or violation of third party rights. You are solely responsible for determining whether to use Third Party Products and Services and if any third party licenses are required to do so. If required you are responsible for acquiring them. You should not install or use Third Party Products and Services until all necessary licenses have been acquired. Any Third Party Products and Services that are provided with BlackBerry's products and services are provided as a convenience to you and are provided "AS IS" with no express or implied conditions, endorsements, guarantees, representations, or warranties of any kind by BlackBerry and BlackBerry assumes no liability whatsoever, in relation thereto. Your use of Third Party Products and Services shall be governed by and subject to you agreeing to the terms of separate licenses and other agreements applicable thereto with third parties, except to the extent expressly covered by a license or other agreement with BlackBerry.

The terms of use of any BlackBerry product or service are set out in a separate license or other agreement with BlackBerry applicable thereto. NOTHING IN THIS DOCUMENTATION IS INTENDED TO SUPERSEDE ANY EXPRESS WRITTEN AGREEMENTS OR WARRANTIES PROVIDED BY BLACKBERRY FOR PORTIONS OF ANY BLACKBERRY PRODUCT OR SERVICE OTHER THAN THIS DOCUMENTATION.

BlackBerry Limited
2200 University Avenue East

Waterloo, Ontario
Canada N2K 0A7

BlackBerry UK Limited
200 Bath Road
Slough, Berkshire SL1 3XE
United Kingdom

Published in Canada