BDAP End-to-End Encryption/Decryption

This document describes how BDAP End-to-End (E2E) encryption and decryption are done.

Notations

We assume the following notations are used:

a_s	Sender's long-term Ed25519 private key
A_s	Sender's long-term Ed25519 public key
b_s	Sender's long-term Curve25519 private-key, derived from its Ed25519 private-key
B_s	Sender's long-term Curve25519 public-key, derived from its Ed25519 public-key
a_i	The i-th recipient's long-term Ed25519 private key
A_i	The i-th recipient's long-term Ed25519 public key
b_i	The i-th recipient's long-term Curve25519 private-key, derived from its Ed25519 private-key
B_i	The i-th recipient's long-term Curve25519 public-key, derived from its Ed25519 public-key
u	An ephemeral random Curve25519 private-key
U	An ephemeral random Curve25519 public-key
s	An ephemeral secret key
k_i	The i-th recipient derived secret
n_i	The i-th recipient derived nonce or initialization vector
$\mathrm{ECDH}(a, B)$	Diffie-Hellman key-exchange between private-key a and public-key B
$\mathrm{AESCTR}_{\mathrm{E}}(k, n, m)$	AES-CTR encryption of payload m using key k and initialization vector n
$\mathrm{AESCTR}_{\mathrm{D}}(k, n, c)$	AES-CTR decryption of ciphertext c using key k and initialization vector n
$\mathrm{AESGCM}_{\mathrm{E}}(k, n, m)$	AES-GCM encryption of payload m using key k and initialization vector n without additional authentication data
$\mathrm{AESGCM}_{\mathrm{D}}(k, n, c)$	AES-GCM decryption of ciphertext c using key k and initialization vector n without additional authentication data
$\mathrm{XOF}(m, L)$	Extendable-output function (XOF) operating on input m and produces L bytes
$X \mid\mid Y$	Concatenation of X and Y
$\lvert m \rvert$	The length of input m in bytes

It is assumed that the key k for AES-CTR and AES-GCM is 32 bytes, i.e. $256-$bit AES key, and that size of the initialization vector n is 16 and 12 bytes for AES-CTR and AES-GCM respectively.

Encryption

Assuming that a sender wants to encrypt a message or payload m to a group of N recipients, then BDAP E2E encryption works as follows:

1. The sender generates an ephemeral random Curve25519 public/private key-pair, i.e. U and u respectively. Note that $|U| = 32$ bytes.

2. The sender generates a random ephemeral 32 bytes secret key, s.

3. For each recipient i, for $i = \{1, 2, \ldots, N\}$, the sender does the following:

   (a) Derive a Curve25519 public-key B_i from the recipient's long-term Ed25519 public-key A_i.

   (b) Perform a Curve25519 key-exchange between the ephemeral private-key u and the i-th recipient public-key B_i, i.e. $Q_i = \mathrm{ECDH}(u, B_i)$.

   (c) Derive an encryption key as follows

   $$k_i \mid\mid n_i = \mathrm{XOF}(Q_i \mid\mid B_i \mid\mid U, 48)$$

   where $|k_i| = 32$ bytes and $|n_i| = 16$ bytes.

   Note that it's not sufficient just to use Q_i to derive k_i and n_i because the co-factor of Curve25519 is not 1, i.e. it's possible to have other combinations of u and B_i pair that produces the same Q_i value.

   (d) Performs AES-CTR encryption on the secret s using k_i and n_i, i.e. $c_i = \mathrm{AESCTR_E}(k_i, n_i, s)$ and $|c_i| = |s| = 32$ bytes[1].

4. Derive message encryption key k and nonce n from the ephemeral secret s, i.e. $k \mid\mid n = \mathrm{XOF}(s, 44)$, where $|k| = 32$ bytes and $|n| = 12$ bytes.

5. Encrypt the message or payload m using k and n, i.e. $c = \mathrm{AESGCM_E}(k, n, m)$ whereby $|c| = |m| + 16$ bytes and the additional 16 bytes are for AES-GCM tag.

6. Combine the overall ciphertext as follows

   $$c' = N \mid\mid U \mid\mid f_1 \mid\mid c_1 \mid\mid f_2 \mid\mid c_2 \mid\mid \ldots \mid\mid f_N \mid\mid c_N \mid\mid c$$

   where $|N| = 2$ bytes and f_i is the fingerprint of the i-th recipient public-key. In this specification, the first 7 bytes of the i-th recipient long-term Ed25519 public-key A_i are used as f_i.

   The overall size of c' therefore is given by

   $$\begin{aligned} |c'| &= |N| + |U| + N.(|c_i| + |f_i|) + |c| \text{ bytes} \\ &= 50 + 39N + |m| \text{ bytes.} \end{aligned}$$

In terms of ciphertext encoding, because the size of N, U, c_i and f_i is fixed, parsing of the ciphertext is relatively straightforward. In particular, if there is a version information for each ciphertext, it can be guaranteed that all of the aforementioned parameters will be fixed. Therefore, assuming 256−bit AES, Curve25519 and the number of recipients N is known, parsing of ciphertext could be done as follows:

1. Extract the first 2 bytes from c' to get N.

2. Extract the subsequent 32 bytes to get U.

[1] Whilst it will be good to use AES-GCM here, it is not necessary. If AES-GCM is used, additional 16 bytes will be added to each c_i, so there will be an expansion of $16N$ bytes in the overall ciphertext. Another alternative is to use AES-CBC, but this will also incur similar expansion (due to padding) on each ciphertext.

3. Extract the next $39N$ bytes to obtain a sequence of fingerprint and ciphertext pairs

$$T = f_1 \mid\mid c_1 \mid\mid f_2 \mid\mid c_2 \mid\mid \ldots \mid\mid f_N \mid\mid c_N.$$

   In order to extract the i-th recipient fingerprint and ciphertext pair $T_i = f_i \mid\mid c_i$, an offset of $34 + 39(i-1)$ bytes is required from the beginning of c'.

4. The remaining bytes are the actual ciphertext of the payload.

In general, the public-key and output ciphertext appear random, i.e. high-entropy, therefore compressing them may not result in any significant gain. If compression is mandatory, depending on the nature of the input plaintext data, it may be better to compress it prior to encryption.

Decryption

Given a piece of ciphertext c', a recipient can decrypt the ciphertext using BDAP E2E decryption as follows:

1. Parse the ciphertext c' to obtain:
   - The number of recipients N: the first 2 bytes from c';
   - Ephemeral public-key U: the next 32 bytes following N;
   - A sequence of fingerprint and ciphertext pairs $T = \{T_1, T_2, \ldots, T_N\}$ where $T_i = f_i \mid\mid c_i$, $|T_i| = 39$ bytes and $|T| = 39N$ bytes;
   - The payload ciphertext c: the remaining data block, i.e. skip c' by $34 + 39N$ bytes.
2. Compute the Ed25519 public-key A from private-key a.
3. Search through T for $T_i = f_i \mid\mid c_i$ where f_i is equal to A in the first 7 bytes.
4. Abort if no valid T_i is found, otherwise extract c_i from the valid T_i.
5. Derive a Curve25519 private-key b from the recipient's long-term Ed25519 private-key a.
6. Compute the Curve25519 public-key B from the private-key b.
7. Perform a Curve25519 key-exchange between the recipient private-key b and the ephemeral public-key U, i.e. $Q = \mathrm{ECDH}(b, U)$.
8. Derive a decryption key as follows

$$k_i \mid\mid n_i = \mathrm{XOF}(Q \mid\mid B \mid\mid U, 48)$$

   where $|k_i| = 32$ bytes and $|n_i| = 16$ bytes.
9. Performs AES-CTR decryption on the ciphertext c_i using k_i and n_i, i.e. $s = \mathrm{AESCTR_D}(k_i, n_i, c_i)$ and $|s| = |c_i| = 32$ bytes
10. Derive message decryption key k and nonce n from the ephemeral secret s, i.e. $k \mid\mid n = \mathrm{XOF}(s, 44)$, where $|k| = 32$ bytes and $|n| = 12$ bytes.
11. Decrypt the ciphertext c using k and n, i.e. $m = \mathrm{AESGCM_D}(k, n, c)$ whereby $|m| = |c| - 16$ bytes.

C++ Encryption Function Specification

```
typedef std::vector<unsigned char> CharVector;
typedef std::vector<CharVector> vCharVector;
bool EncryptBDAPData(const vCharVector& vchPubKeys,
                     const CharVector& vchData,
                     CharVector& vchCipherText,
                     std::string& strErrorMessage);
```

where:

- `vchPubKeys`: ed25519 hex encoded public keys vectors
- `vchData`: serialized data that will be encrypted
- `vchCipherText`: output ciphertext
- `strErrorMessage`: output string if an error occurs

Encryption pseudocode:

$u, U \leftarrow$ Random ephemeral Curve25519 key-pair
if Ephemeral key-pair cannot be generated **then**
    add error-message to `strErrorMessage`
    return `FALSE`
end if
$s \leftarrow$ Random ephemeral secret
if Ephemeral secret cannot be generated **then**
    add error-message to `strErrorMessage`, clear sensitive data from memory
    return `FALSE`
end if
$c' \leftarrow$ `vchPubKeys.size()` $||\ U$
for $0 \leq i <$`vchPubKeys.size()` **do**
    $B_i \leftarrow$ Derive Curve25519 public-key from `vchPubKeys[i]`
    if Curve25519 public-key derivation fails **then**
        add error-message to `strErrorMessage`, clear sensitive data from memory
        return `FALSE`
    end if
    $Q_i \leftarrow \mathrm{ECDH}(u, B_i)$
    if ECDH function fails **then**
        add error-message to `strErrorMessage`, clear sensitive data from memory
        return `FALSE`
    end if
    $k_i \ ||\ n_i \leftarrow \mathrm{XOF}(Q_i \ ||\ B_i \ ||\ U, 48)$
    $c_i \leftarrow \mathrm{AESCTR_E}(k_i, n_i, s)$
    if AES-CTR encrypt function fails **then**
        add error-message to `strErrorMessage`, clear sensitive data from memory
        return `FALSE`
    end if
    $f_i \leftarrow$ the first 7 bytes of B_i
    $c' \leftarrow c' \ ||\ f_i \ ||\ c_i$
end for
$k \ ||\ n \leftarrow \mathrm{XOF}(s, 44)$
$c \leftarrow \mathrm{AESGCM_E}(k, n,$ `vchData`$)$

if AES-GCM encrypt function fails **then**
  add error-message to `strErrorMessage`, clear sensitive data from memory
  return `FALSE`
end if
`vchCipherText` $\leftarrow c' \mid\mid c$
Clear sensitive data from memory
return `TRUE`

C++ Decryption Function Specification

```
bool DecryptBDAPData(const CharVector& vchPrivKeySeed,
                     const CharVector& vchCipherText,
                     CharVector& vchData,
                     std::string& strErrorMessage);
```

where:

- `vchPrivKeySeed`: input hex encoded ed25519 private key seed
- `vchCipherText`: input ciphertext created by the `EncryptBDAPData()` function
- `vchData`: output decrypted data
- `strErrorMessage`: output string if an error occurs

Decryption pseudocode:

$N, U, \{T_1, \dots, T_N\}, c \leftarrow$ Extract the number of recipients, ephemeral public-key, recipient fingerprint and ciphertext pairs and payload ciphertext from `vchCipherText`
if Parsing or extraction fails **then**
  add error-message to `strErrorMessage`
  return `FALSE`
end if
$A \leftarrow$ Compute Ed25519 public-key from Ed25519 private-key `vchPrivKeySeed`
if computation of A fails **then**
  add error-message to `strErrorMessage`, clear sensitive data from memory
  return `FALSE`
end if
Search for valid $T_i = f_i \mid\mid c_i$
if no valid T_i found **then**
  add error-message to `strErrorMessage`, clear sensitive data from memory
  return `FALSE`
end if
$b \leftarrow$ Derive Curve25519 private-key from Ed25519 private-key `vchPrivKeySeed`
if derivation of b fails **then**
  add error-message to `strErrorMessage`, clear sensitive data from memory
  return `FALSE`
end if
$B \leftarrow$ Compute Curve25519 public-key from b
$Q \leftarrow \mathrm{ECDH}(b, U)$
if ECDH function fails **then**
  add error-message to `strErrorMessage`, clear sensitive data from memory
  return `FALSE`

end if
$k_i \mid\mid n_i \leftarrow \text{XOF}(Q \mid\mid B \mid\mid U, 48)$
$s \leftarrow \text{AESCTR}_\text{D}(k_i, n_i, c_i)$
if AES-CTR decrypt function fails **then**
  add error-message to `strErrorMessage`, clear sensitive data from memory
  return `FALSE`
end if
$k \mid\mid n \leftarrow \text{XOF}(s, 44)$
`vchData` $\leftarrow \text{AESGCM}_\text{D}(k, n, c)$
Clear sensitive data from memory
if AES-GCM decrypt function fails **then**
  add error-message to `strErrorMessage`
  return `FALSE`
end if
return `TRUE`

Core C Implementation

The core crypto work shall be implemented in C language. The C++ encryption and decryption functions will call this core C implementation, i.e. a C++ wrapper. The C declaration of the encrypt and decrypt functions are shown below.

```
bool BDAP_encrypt(const char** pub_key_array,
                  const int num_pub_key,
                  const unsigned char* data,
                  unsigned char** ciphertext,
                  size_t* ciphertext_size,
                  char** error_message);
/*
 pub_key_array: an array of hex-encoded Ed25519 public-keys
 num_pub_key: the number of elements in pub_key_array
 data: the payload data to be encrypted
 ciphertext: the output ciphertext
 ciphertext_size: the number of bytes of the ciphertext
 error_message: output error message if an error occurs
 */

bool BDAP_decrypt(const char* priv_key_seed,
                  const unsigned char* ciphertext,
                  size_t ciphertext_size,
                  unsigned char** data,
                  size_t* data_size,
                  char** error_message);
/*
 priv_key_seed: hex-encoded Ed25519 private-key
 ciphertext: the input ciphertext
 ciphertext_size: the number of bytes of the ciphertext
 data: the output decrypted data
 data_size: the number of bytes of the decrypted data
 error_message: output error message if an error occurs
 */
```

Output Error Messages

While it is useful for both debugging and informational purposes, it is a bad practice in general. This is because the error messages leak information and in particular in the case of decryption, they may be exploitable by attackers. In general, one should treat cryptographic operations atomic, returning either success or fail.

Therefore, it is recommended that the functionality of outputting error messages to be removed.

BDAP E2E Encryption/Decryption Unit Tests

The original unit test specification is applicable to this updated specification. The only exception is Negative Test 2 since compression is not used. The updated specification is below.

Negative Test 2

(a) Create three random key seeds and use them to create the `vchPubKeys` variable.

(b) Create a random length string between 1000-5000 characters for the `vchData` variable.

(c) Call `EncryptBDAPData(vchPubKeys, vchData, vchCipherText, strErrorMessage)`. Make sure `EncryptBDAPData` returns `true` and `strErrorMessage` is empty.

(d) Parse `vchCipherText` and extract the payload ciphertext into variable `vchLastValue`.

(e) Try to decrypt `vchLastValue` using three new randomly generated Curve25519 private key.

(f) Unit test passes if `EncryptBDAPData` returns `true`, parsing and extraction is successful, and all attempts to decrypt `vchLastValue` fail.