2 Encrypted Format Overview

2.1 Keys

A number of cryptographic keys are required by the format. The type and function of each key is listed here, along with symbols (K_x where $_x$ is the key type) used to refer to the key in the rest of this specification.

2.1.1 Asymmetric Keys

This specification uses the term "secret key" rather than "private key" so that the symbol K_s can be used for secret keys and K_p for public keys.

Reader's secret key (K_{sr})
: This key is used by the reader when decrypting header packets and should be kept private. It is generated using a cryptographically-secure random number.

Reader's public key (K_{pr})
: This key is passed to the writer so that they can encrypt header packets (section 3.2.1) for the reader. It is derived from K_{sr} (see section 3.3.1).

Writer's secret key (K_{sw})
: This key is used by the writer to encrypt header packets. It should either be kept private, or deleted after use. It is generated using a cryptographically-secure random number.

Writer's public key (K_{pw})
: This key is included in the header packet (section 3.2.1) so that the reader can use it to derive the shared key (K_{shared}, see below) needed to decrypt header packet data. It is derived from K_{sw} (see section 3.3.1).

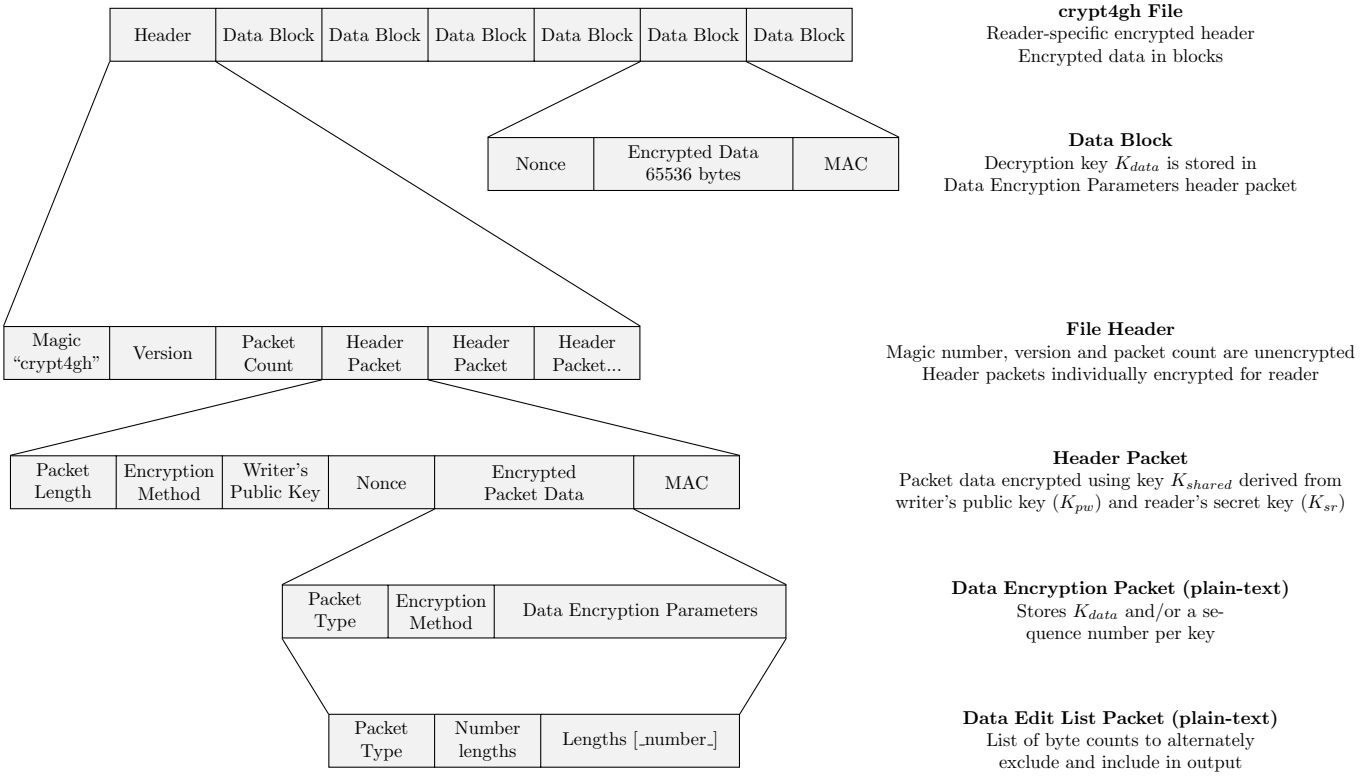
2.1.2 Symmetric keys

Diffie-Hellman key (K_{dh})
: This is generated as part of the derivation of K_{shared}.

Shared key (K_{shared})
: This key is used to encrypt header packet data. It can be derived either from (K_{sw} and K_{pr}) or from (K_{sr} and K_{pw}) - see section 3.3.1. The writer will use the first of these derivations and the reader will use the second.

Data key (K_{data})
: This key is used to encrypt the actual file data (section 3.4). It is generated using a cryptographically-secure random number. The data key SHOULD be generated uniquely for each file. This key is stored in a data_encryption_parameters header packet (see section 3.2.3). It is possible to encrypt parts of a file with different data keys, in which case each key will be stored in a separate data_encryption_parameters header packet.

2.2 File Structure



The encrypted file structure, shown in the diagram above, consists of the following parts:

- A header containing: a “magic” string, version number and header packet count.
  - A “magic” string for file type identification.
  - A version number.
  - The header packet count.
  - One or more header packets containing:
    - The packet length in bytes.
    - The method used to encrypt the header packet data.
    - The writer’s public key (K_{pw}) used to encrypt the header packet data. This is needed (along with the reader’s secret key K_{sr}) to calculate the shared key used to encrypt the header packet.
    - A random “nonce”, also required for decryption.
    - The encrypted data for the header packet.
    - A MAC calculated over the encrypted header packet data.

    The first item in the encrypted header packet data is a code indicating the packet type. This is followed by type-specific data, described in section 2.3.
- The encrypted data. This is the actual application data, stored in a sequence of blocks containing:
  - A random “nonce”, needed for decryption.
  - 64 kilobytes of encrypted data (the last block may contain less than this).
  - A MAC calculated over the encrypted data.

2.3 Header Packet Types

There are two types of header packet:

- Data encryption ~~key~~ parameters packets.

  ~~These describe the~~ They contain a code indicating the type of encryption and describe the list of parameters used to encrypt one or more of the data blocks. ~~They contain a code indicating the type of encryption, and the~~

  The list starts with the symmetric key (K_{data}) needed to decrypt the data.

  If parts of the data have been encrypted with different keys, more than one of this packet type will be present.

  In AEAD mode, an additional 8-bytes sequence number is appended to the parameter list. The sequence number forms part of the authenticated data used when encrypting each segment (See section 3.4.3). This mode ensures no encrypted segments can be lost or re-ordered.

- Data edit list packets.

  These packets allow parts of the data to be discarded after decryption. They can be used to avoid having to decrypt and re-encrypt files during splicing operations.

  For example, a user may want to extract the blocks corresponding to Chromosome X from a CRAM file and store them in a new file. If the start and end points of the extract do not correspond to a 64Kbyte data block boundary, they would normally have to decrypt all of the data blocks covering the region, discard a few bytes from the start and end, re-encrypt the remaining data and store it in a new file.

  The data edit list enables a simpler solution where the necessary encrypted data blocks are copied directly into the new file. On reading, the data blocks are decrypted and then the edit list is used to find out which parts of the unencrypted data should be discarded.

2.4 Encoding For Multiple Public/Secret Key Pairs

It is sometimes useful to encrypt files so that they can be accessed using more than one secret key (K_{sr}). For example, multiple members of a team may need to access to a file with their own key.

To allow this, the header packet data is encrypted using each reader's public key (K_{pr}) and stored in a separate header packet for each individual reader.

Where this is done, it is likely that anyone reading the file will only have the correct secret key (K_{sr}) for a subset of the header packets. Attempting to decode a header packet with the wrong key will result in a failure to verify the MAC stored in the file. When this happens, implementations should ignore the undecodable header packet and move on to the next one. Failing to decrypt a packet in this way SHOULD NOT cause an error to be reported; however an error MUST be raised if, on reaching the end of the header, it has not been possible to decrypt at least one data encryption key packet.

```
    case X25519_chacha20_ietf_poly1305:
      byte   writer_public_key[32];
      byte   nonce[12];
  };

  byte encrypted_payload[];

  select (packet_encryption) {
    case X25519_chacha20_ietf_poly1305:
      byte MAC[16];
  };
};
```

`packet_length` is the length of the entire header packet (including the `packet_length` itself). To prevent packet types from being guessed by looking at the size, it is permitted for the `packet_length` to be longer than strictly needed to encode all of the packet data. Any remaining space after the actual data should be padded in a suitable manner (for example by setting it to zero) and encrypted.

`packet_encryption` is the encryption method used for this header packet.

`writer_public_key` (K_{pw}) and `nonce` are parameters needed to decrypt the `encrypted_payload` in the packet.

`encrypted_payload` is the encrypted part of the header packet, for which the plain-text is described below.

`MAC` is a message authentication code calculated over the encrypted data.

Implementations should ignore any header packets that they cannot decrypt successfully, as these may have been intended for a different reader.

3.2.2 Header packet encrypted payload

The `encrypted_payload` part of the header packet contains the following plain-text:

```
enum Header_packet_type<le_uint32> {
  data_encryption_parameters = 0;
  data_edit_list = 1;
};

enum Data_encryption_method<le_uint32> {
  chacha20_ietf_poly1305 = 0;
  chacha20_ietf_poly1305_with_AEAD = 1;
};

struct Encrypted_header_packet {
  Header_packet_type<le_uint32> packet_type;

  select (packet_type) {
    case data_encryption_parameters:
      enum Data_encryption_method<le_uint32>  data_encryption_method;
      select (data_encryption_method) {
        case chacha20_ietf_poly1305:
          byte  data_key[32];
        case chacha20_ietf_poly1305_with_AEAD:
          byte      data_key[32];
          le_uint64 sequence_number;
      };
```

```
    case data_edit_list:
      le_uint32 number_lengths;
      le_uint64 lengths[number_lengths];
%DIF >
  };
};
```

`packet_type` defines what sort of data packet this is.

3.2.3 data_encryption_parameters packet

This packet contains the parameters needed to decrypt the data part of the file.

`data_encryption_method` is an enumerated type that describes the type of encryption used.

`data_key` is the symmetric key K_{data} used to decode the data section.

To allow parts of the data to be encrypted with different K_{data} keys, more than one of this packet type may be present. If there is more than one, the `data_encryption_method` MUST be the same for all of them to prevent problems with random access in the encrypted file. If the data encryption methods are mixed, the file MUST be rejected.

When `data_encryption_method` is `chacha20_ietf_poly1305_with_AEAD`, the AEAD mode is activated and each `data_key` is followed by an 8-bytes unsigned integer `sequence_number`, which forms part of the authenticated data used to encrypt part of the file. Application of the AEAD mode to the plain-text is described in section 3.4.3.

3.2.4 data_edit_list packet

This packet contains a list of edits that should be applied to the plain-text data following decryption.

`number_lengths` is the number of items in the `lengths` array.

`lengths` is an array of byte counts.

Application of the edit list to the plain-text is described in section 4.3.

It is not permitted to have more than one edit list. If more than one edit list is present, the file SHOULD be rejected.

3.3 Header packet encryption

3.3.1 X25519_chacha20_ietf_poly1305 Encryption

This method uses Elliptic Curve Diffie-Hellman key exchange with additional hashing to generate a shared key (K_{shared}). K_{shared} is then used along with a randomly-generated nonce to encrypt the header packet data using the ChaCha20-IETF-Poly1305 construction. The elliptic curve algorithm used is X25519, described in section 5 of [RFC7748].

Encryption requires the writer's public and secret keys (K_{pw} and K_{sw}), the reader's public key (K_{pr}) and a nonce (N).

The nonce is a unique initialisation vector. In ChaCha20-IETF-Poly1305 it is 12 bytes long. This value MUST be unique for each packet encrypted with the same reader's and writer's keys. The best way to ensure this is to generate a value with a cryptographically-secure random number generator.

The secret keys MUST be generated using a cryptographically-secure random number generator. The corresponding public keys are derived using the method in section 6.1 of [RFC7748].

$K_p = X25519(K_s, 9)$

The writer's secret key and the reader's public key are used to generate a Diffie-Hellman shared key as described in section 6.1 of [RFC7748].

$K_{dh} = X25519(K_{sw}, K_{pr})$

As the X25519 algorithm does not produce a completely uniform bit distribution, and many possible (K_{sw}, K_{pr}) pairs can produce the same output, the Diffie-Hellman key is hashed along with the two public keys to produce the final shared key. The hash function used to do this is Blake2b, as described in [RFC7693].

$K_{shared} = Blake2b(K_{dh}||K_{pr}||K_{pw})$

As ChaCha20 uses a 32-byte key, only the first 32 bytes of K_{shared} are used; the rest are discarded.

The header packet type, data and any padding is then encrypted using the method described in the chacha20_ietf_poly1305 Encryption section 3.4.2. Note that header packets are not segmented; they are always encrypted in a single block.

Finally, the packet length, encryption type, writer's public key K_{pw}, the nonce N and the encrypted header packet data are combined to make the header packet.

For extra security, writers MAY choose to discard the writer's secret key K_{sw} after use. Due to the symmetry of the Diffie-Hellman algorithm, the holder of either secret key can regenerate the shared key as long as the other public key is known. Deleting the writer's key K_{sw} ensures only the holder of the reader's secret key K_{sr} can decode the header packet. As long as the writer uses randomly-generated keys, it also makes accidental nonce reuse very unlikely.

3.3.2 X25519_chacha20_ietf_poly1305 Decryption

To decrypt the header packet, the reader obtains the writer's public key K_{pw} and the nonce from the beginning of the packet. Also needed are the reader's public and secret keys (K_{pr} and K_{sr}).

The Diffie-Helman key is obtained using:

$K_{dh} = X25519(K_{sr}, K_{pw})$

This is then hashed to obtain the shared key (again only the first 32 bytes are retained):

$K_{shared} = Blake2b(K_{dh}||K_{pr}||K_{pw})$

The resulting key K_{shared} and nonce N are then used to decrypt the remainder of the packet.

If the header packet was intended for a different reader, the reader will be unable to decode the header packet as the Poly1305 MAC will be incorrect. This should not be considered an error.

3.3.3 Reading the header

The reader should check that the `magic_number` and `version` in the header match the expected values.

It should then attempt to decode all of the header packets, ignoring any that do not decrypt successfully (detected by a failure to verify the MAC). At the end of this process the reader should have decoded at least one `data_encryption_parameters` packet. If no such packet was decoded, it SHOULD report an error. If more than one is present, they should all have the same `data_encryption_method`, otherwise the reader SHOULD report an error. The reader should store all of the keys that it has decoded in a list for use when decoding the encrypted data section.

If a `data_edit_list` packet is found, the reader should store it for use while processing the data blocks. If more than one `data_edit_list` packet is present, the file SHOULD be rejected.

3.4 Encrypted Data

3.4.1 Segmenting the input

To allow random access without having to authenticate the entire file, the plain-text is divided into 65536-byte (64KiB) segments. If the plain-text is not a multiple of 64KiB long, the last segment will be shorter. Each segment is encrypted using the method defined in the header. The nonce used to encrypt the segment is then stored, followed by the encrypted data, and then the MAC.

```
struct Segment {
  select (method) {
    case chacha20_ietf_poly1305:
    case chacha20_ietf_poly1305_with_AEAD:
      byte    nonce[12];
      byte[]  encrypted_data;
      byte    mac[16];
  };
};
```

The addition of the `nonce` and `mac` bytes will expand the data slightly. For chacha20_ietf_poly1305, this expansion will be 28 bytes, so a 65536 byte plain-text input will become a 65564 byte encrypted and authenticated cipher-text output.

3.4.2 chacha20_ietf_poly1305 Encryption

ChaCha20 is a stream cipher which maps a 256-bit key, nonce and counter to a 512-bit key-stream block. In IETF mode the nonce is 96 bits long and the counter is 32 bits. The counter starts at 1, and is incremented by 1 for each successive key-stream block. The cipher-text is the plain-text message combined with the key-stream using the bit-wise exclusive-or operation.

Poly1305 is used to generate a 16-byte message authentication code (MAC) over the cipher-text. As the MAC is generated over the entire cipher-text it is not possible to authenticate partially decrypted data.

ChaCha20 and Poly1305 are combined using the AEAD construction described in section 2.8 of [RFC8439]. This construction allows additional authenticated data (AAD) to be included in the Poly1305 MAC calculation. ~~For the purposes of this format~~In case the selected encryption method is `chacha20_ietf_poly1305`, the AAD is zero bytes long. In case the selected encryption method is `chacha20_ietf_poly1305_with_AEAD`, the AAD is a 8-bytes little-endian number (section 3.4.3).

3.4.3 ~~Segmenting the input~~

3.4.3 AEAD encrypting mode: chacha20_ietf_poly1305_with_AEAD

~~To allow random access without having to authenticate the entire file, the plain-text is divided into 65536-byte (64KiB)segments. If the plain-text is not a multiple of 64KiB long, the last segment will be shorter. Each segmentis encrypted using the method defined inthe header. The nonce used to encrypt the segment is then stored, followed by the encrypted data, and then the MAC.~~ The AEAD mode ensures no segments can be lost or re-ordered.

~~struct Segment {~~
~~ select (method) {~~
~~   case chacha20_ietf_poly1305:~~
~~   byte   nonce[12];~~
~~   byte[] encrypted_data;~~
~~   byte   mac[16];~~
~~ };~~
~~};~~

Consider the incrementing sequence of segment indexes, starting at 0, created when the file is read segment by segment, in order. When encrypting the plain-text segment, at index i, using the key k (as in 3.4.2), we

attach the number n as authenticated data. n is obtained by adding i to the sequence number paired with the encryption key k. Note that n is limited to 8-bytes, so it might eventually wrap around.

~~The addition of the `nonce` and `mac` bytes will expand the data slightly. For chacha20_ietf_poly1305, this expansion will be 28 bytes, so a 65536 byte plain-text input will become a 65564 byte encrypted and authenticated cipher-text output.~~ Additionally, in case the end of the file lands on a segment boundary, a final and empty encrypted segment is appended to the ciphertext. If not, the last segment is smaller then the segment maximum size and no extra encrypted segment is appended.

4 Decryption

4.1 chacha20_ietf_poly1305 Decryption

The cipher-text is decrypted by authenticating and decrypting the segment(s) enclosing the requested byte range $[P;Q]$, where $P < Q$. For a range starting at position P, the location of the segment `seg_start` containing that position must first be found. For the chacha20_ietf_poly1305 method, when no edit list is in use, this can be done using the formula:

```
seg_start = header_len + floor(P/65536) * 65564
```

For an encrypted segment starting at position `seg_start`, the nonce, then the 65536 bytes of cipher-text (possibly fewer of it was the last segment), and finally the MAC are read.

An authentication tag is calculated over the cipher-text from that segment, and bit-wise compared to the MAC. The cipher-text is authenticated if and only if the tags match. If more than one key (K_{data}) was included in the header, each should be tried in turn until either one authenticates correctly or no keys are left to try. An error MUST be reported if the cipher-text is not authenticated.

The key K_{data} and nonce N are then used to decrypt the cipher-text for the segment, returning the plain-text. Successive segments are decrypted, until the segment containing position Q is reached. The plain-text segments are concatenated to form the resulting output, discarding $P \,\%\, 65536$ bytes from the beginning of the first segment and retaining $Q \,\%\, 65536$ bytes of the last one.

If more than one key (K_{data}) is in use, readers can speed up decryption by trying the previous successful key first when attempting to authenticate each block. However, this does open up a possible timing attack where an observer watching the decoding process can find out where key changes occur due to the extra time needed to select the new key at these points. If this is unacceptable, readers could either try each key for every block (although this may still be vulnerable to timing attacks which try to detect which key was successful); or simply insist that only one key is used for the whole file.

4.2 AEAD mode

The encryption method MUST be `chacha20_ietf_poly1305_with_AEAD` and is `data_key` is paired with a `sequence_number`.

The n^{th} segment is decrypted, as in 4.1, using the sequence number incremented by n as authenticated data.

Finally, the presence of the eventual last empty segment must be checked according to 3.4.3.
Failing that check SHOULD consider the file as truncated, and reject it.

4.3 Edit List

The edit list is designed to assist splicing of encrypted files (for example to remove parts that are not needed for later analysis) without having to decrypt and re-encrypt the entire file. It is only possible to splice crypt4gh files at the 64K encryption block boundaries. The edit list can be used to work around this limitation by describing which parts of the unencrypted blocks should be discarded to give the final desired plain-text.

The following algorithm describes how to apply the edit list *edlist* to unencrypted text *input* to return the desired edited plain-text. In this algorithm, function IsEmpty returns true if a list is empty and false if not. Function RemoveFirst removes the first item from a list and returns it. length returns the length of a string. substr returns part of a string from a given zero-based position and with a given length (or shorter if the requested part extends beyond the end of the input string). StringConcatenate returns the string concatenation of its input parameters in order from left to right.

1: **function** ApplyEditList(*edlist*, *input*)
2: **if** IsEmpty(edlist) **then**