GA4GH File Encryption Standard

Robert Davis, Sanger   Alexander Senf, EBI
Frédéric Haziza, CRG

January 3, 2019 Version: 0.4

Abstract

This document describes the format for Global Alliance for Genomics and Health (GA4GH) encrypted and authenticated files. Encryption helps to prevent accidental disclosure of confidential information. Allowing programs to directly read and write data in an encrypted format reduces the chance of such disclosure. The format can be used to encrypt any underlying file format and allows for seeking on the encrypted data. In particular, indexes on the plain text version can also be used on the encrypted file without modifications.

Contents

1 Introduction

By its nature, genomic data can include information of a confidential nature about the health of individuals. It is important that such information is not accidentally disclosed. One part of the defence against such disclosure is to, as much as possible, keep the data in an encrypted format.

This document describes a file format that can be used to store data in an encrypted and authenticated state. Existing applications can, with minimal modification, read and write data in the encrypted format. The choice of encryption also allows the encrypted data to be read starting from any location, facilitating indexed access to files.

1.1 Encrypted Representation Overview

The encrypted file consists of two parts: the header and the encrypted data portion.

The header encapsulates its data and prepends a magic number and a version number. We describe its construction in Section 2.

The encrypted data portion is the actual application data, described in Section 3. It is encrypted using a symmetric encryption algorithm, as specified in the header. The data is encrypted in 64 kilobytes segments. For each encrypted segment, a 12 bytes nonce is prepended and a 16 bytes MAC is appended. The encrypted data portion may include a checksum of the unencrypted data, if the checksum algorithm is listed in the header.

1.2 Requirements

The keywords “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in [RFC2119].

1.3 Terminology

Elliptic-curve cryptography (ECC) An approach to public-key cryptography based on the algebraic structure of elliptic curves over finite fields.

Curve25519 A widely used FIPS-140 approved ECC algorithm not encumbered by any patents [RFC7748].

ChaCha20-IETF-Poly1305 ChaCha20 is a symmetric stream cipher built on a pseudo-random function that gives the advantage that one can efficiently seek to any position in the key stream in constant time. It is not patented. Poly1305 is a cryptographic message authentication code (MAC). It can be used to verify the data integrity and the authenticity of a message [RFC8439].

ciphertext The encrypted version of the data.

plaintext The unencrypted version of the data.

1.4 Notations

Hexadecimal values are written using the digits 0-9, and letters A-F for values 10-15. Values are written with the most-significant digit on the left, and prefixed with "0x".

The basic data size is the byte (8 bits). All multi-byte values are stored in least-significant byte first ("little-endian") order, called the byte ordering. For example, the value 1234 decimal (0x4d2) is stored as the byte stream 0xd2 0x04.

Integers can be either signed or unsigned. Signed values are stored in two's complement form.

Name	**Byte Ordering**	**Integer Type**	**Size (bytes)**
byte		unsigned	1
le_int32	little-endian	signed	4
le_uint32	little-endian	unsigned	4
le_int64	little-endian	signed	8
le_uint64	little-endian	unsigned	8
le_uint96	little-endian	unsigned	12

Structure types may be defined (in C-like notation) for convenience.

```
struct demo {
  byte string[8];
  le_int32 number1;
  le_uint64 number2;
};
```

When structures are serialized to a file, elements are written in the given order with no padding between them. The above structure would be written as twenty bytes - eight for the array `string`, four for the integer `number1`, and eight for the integer `number2`.

Enumerated types may only take one of a given set of values. The data type used to store the enumerated value is given in angle brackets after the type name. Every element of an enumerated type must be assigned a value. It is not valid to compare values between two enumerated types.

```
enum Animal<le_uint32> {
  cat    = 1;
  dog    = 2;
  rabbit = 3;
};
```

Parts of structures may vary depending on information available at the time of decoding. Which variant to use is selected by an enumerated type. There must be a case for every possible enumerated value. Cases have limited fall-through. Consecutive cases, with no fields in between, all contain the same fields.

```
struct AnimalFeatures {
  select (enum Animal) {
    case cat:
    case dog:
      le_uint32 hairyness;
      le_uint32 whisker_length;

    case rabbit:
      le_uint32 ear_length;
  };
};
```

For the `cat` and `dog` cases, `struct AnimalFeatures` is eight bytes long and contains two unsigned four-byte little-endian values. For the `rabbit` case it is four bytes long and contains a single four-byte little-endian value.

If the cases are different lengths (as above), then the size of the overall structure depends on the variant chosen. There is no padding to make the cases the same length unless it is explicitly defined.

2 Header

2.1 Preamble

The file starts with a header, with the following structure:

```
struct Header {
  byte        magic_number[8];
  le_uint32   version;
  le_uint32   header_len;
  byte        header_data[header_len];
};
```

The `magic_number` is the ASCII representation of the string "crypt4gh".

The version number is stored as a four-byte little-endian unsigned integer. The current version number is 1.

`header_len` is the length of the *remainder* of the header, stored as a four-byte little-endian unsigned integer.

The current byte representation of the magic number and version is:

```
0x63 0x72 0x79 0x70 0x74 0x34 0x67 0x68 0x01 0x00 0x00 0x00
============= magic_number============= ===== version =====
```

2.2 Header Data

This section describes how the parameters, used to encrypt the application data, are themselves serialized and encrypted. The `header_data` sequence of bytes is organized using the type `HeaderData`, as follows:

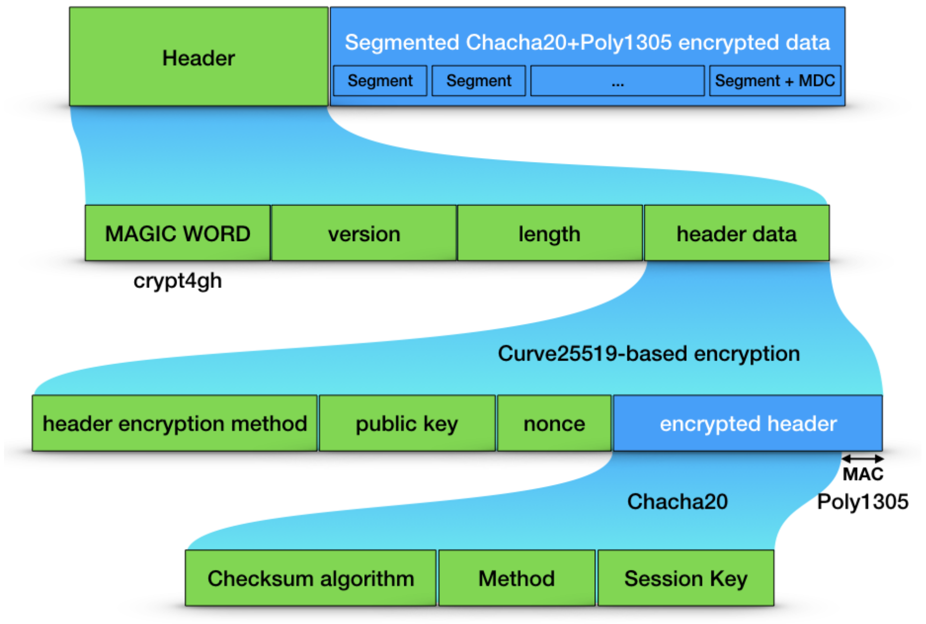

Figure 1: Structure of the Header

```
enum EncryptionMethod<le_uint32> {
  chacha20_ietf_poly1305 = 0;
};

struct HeaderData {
  enum EncryptionMethod<le_uint32> header_encryption_method;
  select (header_encryption_method) {
    case chacha20_ietf_poly1305:
      byte                           public_key[32];
      byte                           nonce[12];
  };
  byte[]                             encrypted_header;
};
```

`header_encryption_method` is an enumerated type that describes how the `encrypted_header` is generated.

The plaintext of the `encrypted_header` encodes, using the following type `EncryptionParameters`, the parameters used for the encryption of the data portion. The plaintext is then encrypted as stipulated by the `header_encryption_method` (See Section 2.3).

```
enum EncryptionMethod<le_uint32> {
  chacha20_ietf_poly1305 = 0;
};

enum ChecksumAlgorithm<le_uint32> {
  none = 0;
  md5 = 1;
  sha256 = 2;
};

struct EncryptionParameters {
  enum ChecksumAlgorithm<le_uint32> checksum_algorithm;

  enum EncryptionMethod<le_uint32>  method;
  select (method) {
    case chacha20_ietf_poly1305:
      byte                          key[32];
  };
};
```

`method` is an enumerated type that describes the type of encryption to be used.

`key` is a secret encryption key. In the case of `chacha20_ietf_poly1305`, it is a sequence of 32 bytes.

`checksum_algorithm` is an enumerated type that describes how the checksum over the *unencryted* data content is computed. If a checksum algorithm is chosen, 'sha256' SHOULD be prefered and 'md5' SHOULD only be used for backwards compatibility. Moreover, the checksum value is of the following form and appended at the end of the encrypted data portion (see Section 3). Nothing is appended if `checksum_algorithm` is `none`.

```
select (checksum_algorithm) {
  case md5:
    byte       key[16];
  case sha256:
    byte       key[32];
  };
```

2.3 Encrypted Header

The header data is encrypted using a Curve25519-based asymmetric encryption. Informally, Curve25519-based asymmetric encryption uses the Currve25519 ECC function to generate a shared encryption key from the encrypter's private key and the recipient's public key, which can be re-created by the recipient using its secret key and the encrypter's public key.

The `encrypted_header` is generated by encrypting the header data, following the `header_encryption_method`. In the case of `chacha20_ietf_poly1305`, it is encrypted using ChaCha20 and authenticated using Poly1305 [RFC8439], using the same algorithms described in Section 3.1. The public key of the encrypter and the randomly-generated nonce, used in the encryption, are prepended, and the Poly1305 authentication tag is appended.

The shared key, used by Chacha20, is calculated using X25519 ECC function as it is described in [RFC7748] (section 5).

The public key can be used as proof of origin of the encrypted file.

3 Encrypted Data

3.1 ChaCha20-Poly1305 Encryption

Informally, ChaCha20 works like a block cipher over blocks of 64 bytes. It is initialized with a 256-bit key, a nonce, and a counter. It produces a succession of blocks of 64 bytes, where the counter is incremented by one for each successive block, forming a keystream. The counter usually starts at 1. The ciphertext is the message combined with the output of the keystream using the XOR operation. The ciphertext does not include any authentication tag. In IETF mode, the nonce is 96 bits long and the counter is 32 bits long.

ChaCha20-Poly1305 uses ChaCha20 to encrypt a message and uses the Poly1305 algorithm to generate a 16-byte MAC over the ciphertext, which is appended at the end of the ciphertext. The MAC is generated for the whole ciphertext that is provided, and appended to the ciphertext. It is not possible to authenticate partially decrypted data.

3.2 Segmenting the input

While ChaCha20 allows to decrypt individual blocks (using the appropriate nonce and counter values), the authentication tag is calculated over the whole ciphertext. To retain streaming and random access capabilities, it is necessary to ensure that segments of the data can be authenticated, without having to read and process the whole file or stream. In this format, the plaintext is divided into 64 kilobytes segments, and each segment is encrypted using ChaCha20-Poly1305, and a randomly-generated 96-bit nonce. The last segment must be smaller. The nonce is prepended to the encrypted segment. Recall that ChaCha20-Poly1305 appends a 16-bytes MAC to the ciphertext. This expands the data by 28 bytes, so a 65536 byte plaintext input will become a 65564 byte encrypted and authenticated ciphertext output.

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

The encrypted data portion is composed of a sequence of segements of at most 65564 bytes. If a segment has a predecessor, the predecessor must be of size 65564 bytes. Note that the plaintext data encrypted in the last segment of the sequence might contain a checksum value of the unencrypted file content, as specified in the header.

3.3 Decryption

The plaintext is obtained by authenticating and decrypting the encrypted 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.

$$\text{seg\_start} = \text{header\_len} + \text{floor}(P/65536) * 65564$$

For an encrypted segment starting at position seg_start, 12-bytes are read to obtain the nonce, then the 65564 bytes of ciphertext (possibly fewer of it was the last segment), and finally the 16 bytes MAC.

An authentication tag is calculated over the ciphertext from that segment, and bitwise compared to the MAC. The ciphertext is authenticated if and only if the tags match. An error should be reported if the ciphertext is not authenticated.

The key and nonce are used to produce a keystream, using ChaCha20 as above, and combined with the ciphertext using the XOR function to obtain the plaintext segment.

Successive segments are decrypted, until the last segment for the range $[P; Q]$ starting at position seg_end, where

$$\text{seg\_end} = \text{header\_len} + \text{floor}(Q/65536) * 65564$$

Plaintext segments are concatenated to form the resulting output, granted that $P \% 65536$ bytes are discarded from the beginning of the first segment, and only $Q \% 65536$ bytes are retained from the last segment.

Implementation details for ChaCha20-Poly1305 (ietf mode) are described in [RFC8439].

Message digest over the unencrypted data Finally, recall that the plaintext header stipulates which checksum algorithm was used to compute the message digest over the unencrypted data. If the chosen algorithm was not `none`, the bytes representing the checksum value are discarded from the end of the resulting output to the recover the plaintext data. Naturally, the checksum value is bitwise compared to a newly computated message digest over the plaintext data. An error MUST be reported if the values do not match.

It is not possible to compare checksums when decrypting a given range of the file. It is only used when decrypting the entire file.

4 Security Considerations

4.1 Threat Model

This format is designed to protect files at rest and in transport from accidental disclosure. Using authenticated encryption in individual segments mirrors solutions like Transport Layer Security (TLS) as described in [RFC5246] and prevents undetected modifications of segments.

4.2 Selection of Key and Nonce

The security of the format depends on attackers not being able to guess the encryption key (and to a lesser extent the nonces). The encryption key and nonce MUST be generated using a cryptographically-secure pseudo-random number generator. This makes the chance of guessing a key vanishingly small. Additional security can be provided by using 'Associated Data' when encrypting a file. This data must be used to decrypt the data, although it is not part of the encrypted file [RFC8439].

4.3 Message Forgery

The encrypted header part is authenticated using the Ed25519 signature scheme. Using ChaCha20-ietf-Poly1305, the Poly1305 algorithm produces an authentication tag for each encrypted segment.

4.4 No File Updates Permitted

Implementations MUST NOT update encrypted files. Once written, a section of the file must never be altered.

References

[RFC2119] Scott O. Bradner. Key words for use in RFCs to Indicate Requirement Levels. RFC 2119, March 1997.

> **Abstract:** In many standards track documents several words are used to signify the requirements in the specification. These words are often capitalized. This document defines these words as they should be interpreted in IETF documents. This document specifies an Internet Best Current Practices for the Internet Community, and requests discussion and suggestions for improvements.

[RFC5246] Eric Rescorla and Tim Dierks. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, August 2008.

Abstract: This document specifies Version 1.2 of the Transport Layer Security (TLS) protocol. The TLS protocol provides communications security over the Internet. The protocol allows client/server applications to communicate in a way that is designed to prevent eavesdropping, tampering, or message forgery. [STANDARDS-TRACK]

[RFC7748] Adam Langley, Mike Hamburg, and Sean Turner. Elliptic Curves for Security. RFC 7748, January 2016.

Abstract: This memo specifies two elliptic curves over prime fields that offer a high level of practical security in cryptographic applications, including Transport Layer Security (TLS). These curves are intended to operate at the ˜128-bit and ˜224-bit security level, respectively, and are generated deterministically based on a list of required properties.

[RFC8439] Yoav Nir and Adam Langley. ChaCha20 and Poly1305 for IETF Protocols. RFC 8439, June 2018.

Abstract: This document defines the ChaCha20 stream cipher as well as the use of the Poly1305 authenticator, both as stand-alone algorithms and as a ”combined mode”, or Authenticated Encryption with Associated Data (AEAD) algorithm.