GA4GH File Encryption Standard

27 Sep 2018

The master version of this document can be found at `https://github.com/samtools/hts-specs`.
This printing is version bf08b64-dirty from that repository, last modified on the date shown above.

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 described here can be used to encrypt any underlying file format. It also allows for seeking on the encrypted data. In particular indexes on the plain text version can also be used on the encrypted file without modification.

Contents

1 Introduction

1.1 Purpose

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.

1.2 Requirements

The key words "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].

Ed25519 An Edwards-curve Digital Signature Algorithm (EdDSA) over Curve25519 [RFC8032].

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].

cipher-text
The encrypted version of the data.

plain-text
The unencrypted version of the data.

2 Encrypted Representation Overview

The encrypted file consists of three parts:

- An unencrypted header, containing a magic number, version number, and parameters needed to decode the encrypted header.
- An encrypted header, which is encrypted using an asymmetric encryption algorithm. It lists the encryption key and other parameters needed to decrypt the encrypted data section.
- The encrypted data. This is the actual application data. It is encrypted using a symmetric encryption algorithm as described in the encrypted header.

3 Detailed Specification

3.1 Overall Conventions

3.1.1 Hexadecimal Numbers

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".

3.1.2 Byte Ordering

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

3.1.3 Integer Types

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

3.1.4 Multi-byte Integer Types

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

3.1.5 Structures

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'.

3.1.6 Enumerated Types

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;
};
```

3.1.7 Variants

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.

3.2 Header

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

```
struct Header {
  byte                          magic_number[8];
  le_uint32                     version;
  le_uint32                     header_length;
  Header_encryption_parameters  header_encryption;
  byte[]                        encrypted_header
};
```

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.

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 =====
```

`header_length` is the combined length of the `header_encryption` and `encrypted_header` sections.

3.2.1 Header Encryption Parameters

The 'Header_encryption_parameters' type is defined as:

```
struct Header_encryption_parameters {
  enum Header_encryption_method<le_uint32> header_encryption_method;

  select (header_encryption_method) {
    case X25519_chacha20_ietf_poly1305:
      byte                          public_key[32];
      byte                          nonce[12];
  };
};
```

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

`public_key` is the writer's public key. Treated as a concatenation of eight 32-bit little-endian integers.

`nonce` is a unique initialization vector. In chacha20_ietf_poly1305 it is 12 bytes long. This value must be unique for each file shared between the same reader and writer. The best way to ensure this is to generate a value with a cryptographically-secure random number generator.

3.2.2 Encrypted Header Format

The encrypted header encodes the parameters used to read the data part of the file. The plain-text is defined as:

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

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

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

`key` is the symmetric key used to decode the data section.

3.2.3 X25519_chacha20_ietf_poly1305 Encryption

This method uses Diffie-Hellman key exchange with additional hashing to generate a shared key. The shared key is then used along with a randomly-generated nonce to encrypt the data using the chacha20_ietf_poly1305 construction.

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

The writer's secret key and the reader's public key are used to generate a Diffie_Hellman shared key using the X25519 algorithm described in section 5 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].

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

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

The header data is then encrypted using the method described in the ChaCha20_IETF_Poly1305 Encryption section 3.3.1.

Finally, the writer's public key K_{pw}, the nonce N and the encrypted header data are combined into the file header.

3.2.4 X25519_chacha20_ietf_poly1305 Decryption

To decrypt the header, the reader needs the writer's public key K_{pw} and the nonce from the file header. 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):

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

The resulting key K_s and nonce N are then used to decrypt the remainder of the header.

3.3 Encrypted Data

3.3.1 ChaCha20_IETF_Poly1305 Encryption

ChaCha20 is a stream cipher which maps a 256-bit key, nonce and counter to a 512-bit keystream 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 keystream block. The cipher-text is the plain-text message combined with the keystream using the bitwise 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, the AAD is zero bytes long.

3.3.2 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:
      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.

4 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, 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 ciphertext (possibly fewer of it was the last segment), and finally the MAC are read.

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 MUST be reported if the ciphertext is not authenticated.

The key and nonce 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 plaintext 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.

5 Security Considerations

5.1 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 modification of segments.

5.2 Selection of Key

The security of the format depends on attackers not being able to guess the encryption key. The encryption key MUST be generated using a cryptographically-secure pseudo-random number generator. This makes the chance of guessing a key vanishingly small.

5.3 Nonce selection

All segments encrypted with the same key MUST use a unique nonce. One way to ensure this is to choose a random starting point and then use a counter or linear-feedback shift register to generate the nonce for each segment. This method guarantees a unique value even for very long files.

For a 96-bit nonce, it may be acceptable to generate each one using a cryptographically-secure pseudo-random number generator. Care should be taken to ensure that the random number generator is capable of generating a long enough stream of unique values. Due to the birthday problem, this method will have a non-zero (but very small) probability of failing. For example, a file of 24 Terabytes will have a reused nonce with probability of approximately $10^{-}12$

5.4 Message Forgery

Using Curve25519 and Ed25519 authenticates the content of the encrypted file header. Using ChaCha20-ietf-Poly1305 authenticates the content of each segment of the encrypted cipher-text.

5.5 No File Updates Permitted

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

6 References

[RFC2119] Bradner, S.,
"Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119,
`https://www.rfc-editor.org/info/rfc2119`,
March 1997

[RFC7693] Saarinen, M.J., Aumasson, J.P.
"The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC)", RFC7693,
`https://tools.ietf.org/html/rfc7693`,
November 2015

[RFC7748] A. Langley, Google, M. Hamburg, Rambus Cryptography Research, S. Turner, sn3rd,
"Elliptic Curves for Security", RFC7748,
`https://tools.ietf.org/html/rfc7748`,
January 2016

[RFC8032] S. Josefsson, SJD AB, I. Liusvaara,
"Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC8032,
`https://tools.ietf.org/html/rfc8032`,
January 2017

[RFC8439] Y. Nir, Dell EMC, A. Langley, Google, Inc.,
"ChaCha20 and Poly1305 for IETF Protocols", RFC8439,
`https://tools.ietf.org/html/rfc8439`,
June 2018

[RFC5246] Dierks, T., Rescorla, E.,
"The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246,
`https://www.rfc-editor.org/info/rfc5246`,
August 2008.