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 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 four parts:

- An unencrypted header, containing a magic number, version number, length of public key (may be zero), public key, a header signature indicator, and the length of the encrypted header.
- An encrypted header, which is encrypted using an asymmetric encryption algorithm. It lists the encryption key and nonce needed to decrypt the encrypted data section.
- A 64-byte Ed25519 signature validating the encrypted header.
- The encrypted data. This is the actual application data. It is encrypted using a symmetric encryption algorithm as described in the encrypted header. The data is encrypted in 64K segments with a 16 byte MAC appended at the end of each segment.

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 Vectors

A vector is a stream of elements of the same type (which may be a structure). The number of items may be specified either as a constant value or in reference to a known integer value (for example, a variable previously read from the file).

```
le_in32   num     // Number of v2 array elements
le_int32  v1[8]   // Eight four-byte little-endian integers
le_int64  v2[num] // 'num' eight-byte little-endian integers
```

When vectors are serialized to a file, the elements are written with no padding between them.

3.1.6 Structures

Structure types may be defined for convenience. The syntax for definition is similar to that of C.

```
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. So the structure above would be written as twenty bytes - eight for the array 'string', four for the integer 'number1', and eight for the integer 'number2'.

3.1.7 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;
};
```

3.1.8 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 Unencrypted Header

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

```
struct Unencrypted_header {
  byte      magic_number[8];
  le_uint32 version;
  le_uint32 public_key_length;
  byte      public_key[public_key_length];
  le_uint32 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.

The public key length may be zero. If it is greater than zero then it is the length of the public key component of the private key used to sign the header (this is not the same as the public key used to encrypt the header).

header_len is the sum of the lengths of the unencrypted and encrypted headers. If a public key is specified then it also includes the 64-byte header signature field appended to the encrypted header. It is stored as a four-byte little-endian unsigned integer. As it includes the unencrypted header, header_len will always have a value of at least 16.

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 =====
```

Possible header configurations:

If no public key is specified:

```
[magic number][version][0][header len][encrypted header]
```

If a public key is specified:

```
[magic number][version][pub key len][public key][header len][encrypted header][header signature]
```

3.3 Encrypted Header

3.3.1 Encryption Method

The encrypted header is encoded in the Curve25519 message format [RFC7748].

3.3.2 Header Signature

The header signature is generated using the Ed25519 algorithm [RFC8032].

3.3.3 Plain-text Format

The plain-text data encoded in the encrypted header has the following overall structure:

The 'Encryption_parameters' type is defined as:

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

struct Encryption_parameters {
  byte checksum[32];
  enum Encryption_method<le_uint32> method;

  select (method) {
    case chacha20_ietf_poly1305:
      byte      key[32];
      le_uint96 nonce;
  };
};
```

'checksum' is an optional 32-byte SHA-256 checksum of the (unencrypted) data in the file.

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

'key' is a secret encryption 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.

3.4 Encrypted Data

3.4.1 ChaCha20 Mode Encryption

Internally, ChaCha20 works like a block cipher used in counter mode. It includes an internal block counter to avoid incrementing the nonce after each block. The cipher-text is the message combined with the output of the stream cipher using the XOR operation, and doesn't 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 encrypted data, which is appended at the end of the cipher-text. The MAC is generated for the whole cipher-text that is provided. It is not possible to authenticate partially decrypted data.

While it is possible to decrypt individual blocks in ChaCha20, to authenticate the content it is necessary to decrypt the entire cipher-text, with the authentication tag at the end. 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. In this format the plain-text is divided into 64KiB segments and encrypted to 64KiB of cipher-text. A MAC is generated for each encrypted segment individually and appended to each 64KiB block of cipher-text. This expands the data by 16 bytes, so a 65536 byte plain-text input will become a 65552 byte encrypted and authenticated output.

A dual counter method is used, where the nonce is incremented after each 64K segment, and the counter is incremented within each segment, producing a unique IV (nonce+counter) for each cipher block even if the input is very large. The initial value used for the counter when encrypting each 64KiB block is 1.

The cipher-text is decrypted by authenticating and decrypting the segment(s) enclosing the requested byte range. To decrypt starting from a plain-text position P, first the location of the segment containing that position must be found.

```
seg_start = header_len + floor(P/65536) * 65552
```

The 65552 bytes (possibly fewer at the end of the file) starting at this position are read to obtain the encrypted data and 16 byte MAC. Next the nonce for the segment is calculated.

```
seg_nonce = nonce + floor(P/65536)
```

The key, nonce, encrypted data and MAC are then used to obtain the plain text. Finally (P mod 65536) bytes at the start are discarded from the plain text to give the desired output.

Implementation details for ChaCha20-ietf-Poly1305are described in [RFC8439].

3.4.2 Partial Segments

It is likely that the end of the file will not occur at an exact multiple of the cipher block or 64K segment length. In such cases the short block will be encrypted in the way described above, apart from the block size will now be the number of remaining bytes. This will produce an encrypted output which will be exactly 16 bytes longer than the input.

If the last block is found to be shorter than 65552 bytes on reading, the last 16 bytes should be taken as the MAC, and the remaining bytes as the short block to decrypt. If the last block is found to be shorter than 16 bytes the file is corrupt and an error should be reported.

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 prevent undetected modification 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 nonce). 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

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.

4.4 No File Updates Permitted

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

5 References

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.

[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.