GA4GH File Encryption Standard

Robert Davies

The master version of this document can be found at `https://github.com/samtools/hts-specs`.
This printing is version 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 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 encryted data. In particular indexes on the plain text version can also be used on the encrypted file without modification.

Copyright Notice

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 defense 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 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 [**?**].

1.3 Terminology

Advanced Encryption Standard (AES)
: A FIPS-approved algorithm that can be used to protect data, defined in [**?**]. AES encrypts data in fixed-sized blocks of 16 bytes.

cipher-text
: The encrypted version of the data.

counter (CTR) mode
: A method of using a cipher with a fixed block size to encrypt data streams that are longer than a single block, converting it to a pseudo-random block of bytes. The counter is incremented for each block, leading to the generation of a repeatable byte stream. This stream can be XORed with the plain-text to generate the cipher-text. Because XOR is reversible, the same operation can also be used to convert the cipher-text back into the plain-text.

  CTR mode and its use with AES are described in [**?**].

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 the length of the encrypted header
- An encrypted header. This is encrypted using an asymmetric encryption algorithm. It lists the encryption keys and initialization vectors 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). Multi-byte values need to be stored in a defined order. This order will be one of the following:

Least-significant byte first ("little-endian")
The value 1234 decimal (0x4d2) is stored as the byte stream 0xd2 0x04.

Most-significant byte first ("big-endian")
The value 1234 decimal (0x4d2) is stored as the byte stream 0x04 0xd2.

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
be_uint128	big-endian	unsigned	16

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 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.

hdr_len is the sum of the lengths of the unencrypted and encrypted headers. It is stored as a four-byte little-endian unsigned integer. As it includes the unencrypted header, hdr_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 =====
```

3.3 Encrypted Header

3.3.1 Encryption Method

The encrypted header is encoded in the OpenPGP message format [**?**].

3.3.2 Plain-text Format

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

```
struct Encrypted_header {
  le_uint32                     number_records;
  struct Encryption_parameters  records[number_records];
};
```

'number_records' gives the number of 'Encryption_parameters' records that follow. Each record corresponds to a range of bytes in the plain-text version of the file. This allows chunks of encrypted data to be rapidly concatenated together without the need to decrypt and re-encrypt the whole file.

The 'Encryption_parameters' type is defined as:

```
enum Encryption_method<le_uint32> {
  AES-256-CTR = 0;
};

struct Encryption_parameters {
  le_uint64                         plaintext_start;
  le_uint64                         plaintext_end;
  le_uint64                         ciphertext_start;
  le_int64                          counter_offset;

  enum Encryption_method<le_uint32> method;

  select (method) {
    case AES-256-CTR:
      byte       key[32];
      be_uint128 iv;
  };
};
```

'plaintext_start' and 'plaintext_end' define the range of plain-text bytes over which this record is valid. The positions are indexed counting from zero. plaintext_end is the byte one past the end of the range. The value

of plaintext_end is allowed to be beyond the real end of the plain-text data. This means it can be set to `0xffffffffffffffff` when the final length of the file is unknown.

'ciphertext_start' is the location in the encrypted data where the given plain-text data starts. This location counts from 0 at the start of the encrypted data section. header_len from the unencrypted header should be added to this to find the location in the encrypted file.

'counter_offset' is used to adjust the counter value used in counter mode (CTR) encryption. This allows the correct counter value to be calculated for files that have been concatenated together.

The regions [plaintext_start..plaintext_end) MUST NOT overlap between records. Likewise, the corresponding cipher-text blocks MUST NOT overlap. To facilitate streaming, they SHOULD be in order, and the order of blocks in the cipher-text SHOULD match the order in the plain-text.

The union of regions over all the records does not have to cover the entire plain-text. This allows selective access to parts of a file to be granted. When preparing files, inaccessible parts SHOULD NOT be included in the cipher-text. It wastes the recipient's storage, and readers still may be able to gain access if they are encrypted with the same key as other parts of the file that are permitted.

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

'key' is a secret encryption key.

'iv' is an initialization vector.

3.4 Encrypted Data

3.4.1 AES-256-CTR Mode Encryption

AES-CTR mode generates a byte stream by AES encrypting a sequential counter. The counter starts at the given IV and increments by 1 for each block of 16 bytes. This byte stream is XORed with the plain-text to produce the cipher-text.

The cipher-text is decrypted by XORing it with the same byte stream, which can be generated if the IV and encryption key are known.

In this format, the counter value C for a particular plain-text file offset P can be calculated as follows:

```
C = (P + counter_offset) / 16
```

where counter_offset is given the encrypted header record. The sum (P + counter_offset) should always be positive.

The counter value is then combined with the IV by addition. For compatibility with OpenSSL, both C and IV are treated as 16-byte big-endian values.

The sum C + IV and the key are used to generate 16 bytes of key stream E:

```
E[0..15] = AES(key, C + IV)
```

This can be used to encrypt the byte B at file offset P:

```
B' = B XOR E[(P + counter_offset) % 16]  where % is the modulo operator.
```

3.4.2 Partial Blocks

It is likely that the end of the file, or the boundaries of a region listed in the encrypted header, will not occur at an exact multiple of the cipher block length. In such cases, the unused bytes of the key stream are simply discarded. There is no need to pad the encrypted data.

4 Security Considerations

4.1 Threat Model

This format is only designed to protect files at rest from accidental disclosure. In particular, it is not designed to protect files during transmission over insecure networks - existing solutions like Transport Layer Security (TLS) as described in [**?**] should be used for this.

4.2 Selection of Keys and IVs

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

When using CTR mode encryption, it is vital never to reuse an (IV + CTR) value for the same cryptographic key. Doing so could lead to two blocks being encrypted with identical key streams. If this happens, then an attacker can learn the XOR of the plain-text of the two blocks by simply XORing the cipher-text blocks (as key_stream XOR key_stream = 0). Values of (IV + CTR) MUST NOT be reused for the same key. This can be achieved by:

- Choosing a random IV using a cryptographically-secure random number generator.
- Always incrementing the counter for each block (as happens in this format).

If both the cryptographic key and IV are chosen randomly for each file, the chance of reusing a combination of (key, IV + CTR) becomes vanishingly small. Implementations SHOULD choose the IV using a cryptographically secure random number generator.

4.3 Message Forgery

Forging a message encrypted in CTR mode is trivial, especially if the attacker can guess the plain-text of an encrypted message. They simply XOR the cipher-text with a chosen byte stream. On decoding, the result will be the original plain-text XORed with the attacker's byte stream.

If an attacker knows that part of an encrypted file is, for example, a gzip header, then they know the plain-text bytes and can replace that part with any data they choose.

This format is currently not resistant to forgery. Care should be taken to protect files from unauthorised tampering. This is also why extra precautions should be taken when transmitting files over networks.

4.4 No File Updates Permitted

While it is possible to update parts of a file written in CTR mode, it violates the rule about not reusing the value of (IV + CTR) for a given key. An attacker who compares the before and after versions of the file will learn the XOR of the before and after plain-texts.

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

5 References

References

[AESSTD] National Institute of Standards and Technology,
"Announcing the ADVANCED ENCRYPTION STANDARD (AES)", FIPS 197,
`http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`,
November 26, 2001.

[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.

[RFC4880] Callas, J., Donnerhacke, L., Finney, H., Shaw D., Thayer, R.,
"OpenPGP Message Format", RFC4880,
`https://www.rfc-editor.org/info/rfc4880`,
November 2007.

[CTRMODE] Dworkin, M, National Institute of Standards and Technology,
"Recommendation for Block Cipher Modes of Operation", SP800-38A,
`http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf`,
December 2001.

[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.