Stream: Internet Engineering Task Force (IETF)
RFC: 9841
Updates: 7932
Category: Informational
Published: September 2025
ISSN: 2070-1721
Authors:
J. Alakuijala, *Google, Inc.*
T. Duong, *Google, Inc.*
E. Kliuchnikov, *Google, Inc.*
Z. Szabadka, *Google, Inc.*
L. Vandevenne, Ed., *Google, Inc.*

RFC 9841
Shared Brotli Compressed Data Format

Abstract

This specification defines a data format for shared brotli compression, which adds support for shared dictionaries, large window, and a container format to brotli (RFC 7932). Shared dictionaries and large window support allow significant compression gains compared to regular brotli. This document specifies an extension to the method defined in RFC 7932.

Status of This Memo

This document is not an Internet Standards Track specification; it is published for informational purposes.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Not all documents approved by the IESG are candidates for any level of Internet Standard; see Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc9841.

Copyright Notice

Table of Contents

1. Introduction

1.1. Purpose

The purpose of this specification is to extend the brotli compressed data format [RFC7932] with new abilities that allow further compression gains.

- Shared dictionaries allow a static shared context between encoder and decoder for significant compression gains.
- Large window brotli allows much larger back reference distances to give compression gains for files over 16 MiB.
- The framing format is a container format that allows storage of multiple resources and references dictionaries.

This document is the authoritative specification of shared brotli data formats and the backwards compatible changes to brotli. This document also defines the following:

- The data format of serialized shared dictionaries
- The data format of the framing format
- The encoding of window bits and distances for large window brotli in the brotli data format
- The encoding of shared dictionary references in the brotli data format

1.2. Intended Audience

This specification is intended for use by software implementers to compress data into and/or decompress data from the shared brotli dictionary format.

The text of the specification assumes a basic background in programming at the level of bits and other primitive data representations. Familiarity with the technique of LZ77 coding [LZ77] is helpful, but not required.

1.3. Scope

This specification defines a data format for shared brotli compression, which adds support for dictionaries and extended features to brotli [RFC7932].

1.4. Compliance

Unless otherwise indicated below, a compliant decompressor must be able to accept and decompress any data set that conforms to all the specifications presented here. Additionally, a compliant compressor must produce data sets that conform to all the specifications presented here.

1.5. Definitions of Terms and Conventions Used

Byte: 8 bits stored or transmitted as a unit (same as an octet). For this specification, a byte is exactly 8 bits, even on machines that store a character on a number of bits different from eight. See below for the numbering of bits within a byte.

String: A sequence of arbitrary bytes.

Bytes stored within a computer do not have a "bit order" since they are always treated as a unit. However, a byte considered as an integer between 0 and 255 does have a most significant bit (MSB) and least significant bit (LSB), and since we write numbers with the most significant digit on the left, we also write bytes with the MSB on the left. In the diagrams below, the bits of a byte are written so that bit 0 is the LSB, i.e., the bits are numbered as follows:

```
+--------+
|76543210|
+--------+
```

Within a computer, a number may occupy multiple bytes. All multi-byte numbers in the format described here are unsigned and stored with the least significant byte first (at the lower memory address). For example, the decimal 16-bit number 520 is stored as:

```
0        1
+--------+--------+
|00001000|00000010|
+--------+--------+
^        ^
|        |
|        + more significant byte = 2 x 256
+ less significant byte = 8
```

1.5.1. Packing into Bytes

This document does not address the issue of the order in which bits of a byte are transmitted on a bit-sequential medium, since the final data format described here is byte- rather than bit-oriented. However, the compressed block format is described below as a sequence of data elements of various bit lengths, not a sequence of bytes. Therefore, we must specify how to pack these data elements into bytes to form the final compressed byte sequence:

- Data elements are packed into bytes in order of increasing bit number within the byte, i.e., starting with the LSB of the byte.
- Data elements other than prefix codes are packed starting with the LSB of the data element. These are referred to here as integer values and are considered unsigned.
- Prefix codes are packed starting with the MSB of the code.

In other words, if one were to print out the compressed data as a sequence of bytes starting with the first byte at the **right** margin and proceeding to the **left**, with the MSB of each byte on the left as usual, one would be able to parse the result from right to left with fixed-width elements in the correct MSB-to-LSB order and prefix codes in bit-reversed order (i.e., with the first bit of the code in the relative LSB position).

As an example, consider packing the following data elements into a sequence of 3 bytes: 3-bit integer value 6, 4-bit integer value 2, 3-bit prefix code b'110, 2-bit prefix code b'10, and 12-bit integer value 3628.

```
  byte 2   byte 1   byte 0
+--------+--------+--------+
|11100010|11000101|10010110|
+--------+--------+--------+
 ^            ^ ^   ^   ^
 |            | |   |   |
 |            | |   |   +------ integer value 6
 |            | |   +---------- integer value 2
 |            | +-------------- prefix code 110
 |            +---------------- prefix code 10
 +---------------------------- integer value 3628
```

2. Shared Brotli Overview

Shared brotli extends brotli [RFC7932] with support for shared dictionaries, a larger LZ77 window, and a framing format.

3. Shared Dictionaries

A shared dictionary is a piece of data shared by a compressor and decompressor. The compressor can take advantage of the dictionary context to encode the input in a more compact manner. The compressor and the decompressor must use exactly the same dictionary. A shared dictionary is specially useful to compress short input sequences.

A shared brotli dictionary can use two methods of sharing context:

LZ77 dictionary: The encoder and decoder could refer to a given sequence of bytes. Multiple LZ77 dictionaries can be set.

Custom static dictionary: A word list with transforms. The encoder and decoder will replace the static dictionary data with the data in the shared dictionary. The original static dictionary is described in Section 8 in [RFC7932]. The original data from Appendices A and B of [RFC7932] will be replaced. In addition, it is possible to dynamically switch this dictionary based on the data compression context and/or include a reference to the original dictionary in the custom dictionary.

If no shared dictionary is set, the decoder behaves the same as in [RFC7932] on a brotli stream.

If a shared dictionary is set, then it can set LZ77 dictionaries, override static dictionary words, and/or override transforms.

3.1. Custom Static Dictionaries

If a custom word list is set, then the following behaviors of the decoder defined in [RFC7932] are overridden:

Instead of the Static Dictionary Data from Appendix A of [RFC7932], one or more word lists from the custom static dictionary data are used.

Instead of NDBITS at the end of Appendix A of [RFC7932], a custom SIZE_BITS_BY_LENGTH per custom word list is used.

The copy length for a static dictionary reference must be between 4 and 31 and may not be a value for which SIZE_BITS_BY_LENGTH of this dictionary is 0.

If a custom transforms list is set without context dependency, then the following behaviors of the decoder defined in [RFC7932] are overridden:

The "List of Word Transformations" from Appendix B of [RFC7932] is overridden by one or more lists of custom prefixes, suffixes, and transform operations.

The transform_id must be smaller than the number of transforms given in the custom transforms list.

If the dictionary is context dependent, it includes a lookup table of 64 word list and transform list combinations. When resolving a static dictionary word, the decoder computes the literal Context ID as described in Section 7.1 of [RFC7932]. The literal Context ID is used as the index in the lookup tables to select the word list and transforms to use. If the dictionary is not context dependent, this ID is implicitly 0 instead.

If a distance goes beyond the dictionary for the current ID and multiple word/transform list combinations are defined, then the next dictionary is used in the following order:

- If context dependent:
  - use the index matching the current context first, and then
  - use the same order as defined in the shared dictionary (excluding the current context) next.
- If not context dependent:
  - use the same order as defined in the shared dictionary.

3.1.1. Transform Operations

A shared dictionary may include custom word transformations to replace those specified in Section 8 and Appendix B of [RFC7932]. A transform consists of a possible prefix, a transform operation, a parameter (for some operations), and a possible suffix. In the shared dictionary format, the transform operation is represented by a numerical ID, which is listed in the table below.

ID	Operation
0	Identity
1	OmitLast1
2	OmitLast2
3	OmitLast3
4	OmitLast4
5	OmitLast5
6	OmitLast6
7	OmitLast7

ID	Operation
8	OmitLast8
9	OmitLast9
10	FermentFirst
11	FermentAll
12	OmitFirst1
13	OmitFirst2
14	OmitFirst3
15	OmitFirst4
16	OmitFirst5
17	OmitFirst6
18	OmitFirst7
19	OmitFirst8
20	OmitFirst9
21	ShiftFirst (by PARAMETER)
22	ShiftAll (by PARAMETER)

Table 1

Operations 0 to 20 are specified in Section 8 of [RFC7932]. ShiftFirst and ShiftAll transform specifically encoded SCALARs.

A SCALAR is a 7-, 11-, 16-, or 21-bit unsigned integer encoded with 1, 2, 3, or 4 bytes, respectively, with the following bit contents:

```
7-bit SCALAR:
+--------+
|0sssssss|
+--------+

11-bit SCALAR:
+--------+--------+
|110sssss|XXssssss|
+--------+--------+

16-bit SCALAR:
+--------+--------+--------+
|1110ssss|XXssssss|XXssssss|
+--------+--------+--------+

21-bit SCALAR:
+--------+--------+--------+--------+
|11110sss|XXssssss|XXssssss|XXssssss|
+--------+--------+--------+--------+
```

Given the input bytes matching the SCALAR encoding pattern, the SCALAR value is obtained by concatenation of the "s" bits, with the MSBs coming from the earliest byte. The "X" bits could have arbitrary value.

An ADDEND is defined as the result of limited sign extension of a 16-bit unsigned PARAMETER:

> At first, the PARAMETER is zero-extended to 32 bits. After this, 0xFF0000 is added if the resulting value is greater or equal than 0x8000.

ShiftAll starts at the beginning of the word and repetitively applies the following transformation until the whole word is transformed:

> If the next untransformed byte matches the first byte of the 7-, 11-, 16-, or 21-bit SCALAR pattern, then:
>
> > If the untransformed part of the word is not long enough to match the whole SCALAR pattern, then the whole word is marked as transformed.
> >
> > Otherwise, let SHIFTED be the sum of the ADDEND and the encoded SCALAR. The lowest bits from SHIFTED are written back into the corresponding "s" bits. The "0", "1", and "X" bits remain unchanged. Next, 1, 2, 3, or 4 untransformed bytes are marked as transformed according to the SCALAR pattern length.
>
> Otherwise, the next untransformed byte is marked as transformed.

ShiftFirst applies the same transformation as ShiftAll, but does not iterate.

3.2. LZ77 Dictionaries

If an LZ77 dictionary is set, the decoder treats it as a regular LZ77 copy but behaves as if the bytes of this dictionary are accessible as the uncompressed bytes outside of the regular LZ77 window for backwards references.

Let LZ77_DICTIONARY_LENGTH be the length of the LZ77 dictionary. Then word_id, described in Section 8 of [RFC7932], is redefined as:

```
word_id = distance - (max allowed distance + 1 +
LZ77_DICTIONARY_LENGTH)
```

For the case when LZ77_DICTIONARY_LENGTH is 0, word_id matches the [RFC7932] definition.

Let dictionary_address be:

LZ77_DICTIONARY_LENGTH + max allowed distance - distance

Then distance values of <length, distance> pairs [RFC7932] in range (max allowed distance + 1).. (LZ77_DICTIONARY_LENGTH + max allowed distance) are interpreted as references starting in the LZ77 dictionary at the byte at dictionary_address. If length is longer than (LZ77_DICTIONARY_LENGTH - dictionary_address), then the reference continues to copy (length - LZ77_DICTIONARY_LENGTH + dictionary_address) bytes from the regular LZ77 window starting at the beginning.

4. Varint Encoding

A varint is encoded in base 128 in one or more bytes as follows:

where the "x" bits of the first byte are the LSBs of the value and the "x" bits of the last byte are the MSBs of the value. The last byte must have its MSB set to 0 and all other bytes must have their MSBs set to 1 to indicate there is a next byte.

The maximum allowed amount of bits to read is 63 bits; if the 9th byte is present and has its MSB set, then the stream must be considered as invalid.

5. Shared Dictionary Stream

The shared dictionary stream encodes a custom dictionary for brotli, including custom words and/or custom transformations. A shared dictionary may appear as a standalone or as contents of a resource in a framing format container.

A compliant shared brotli dictionary stream must have the following format:

2 bytes: File signature in hexadecimal format (bytes 91 and 0).

varint: LZ77_DICTIONARY_LENGTH. The number of bytes for an LZ77 dictionary, or 0 if there is none. The maximum allowed value is the maximum possible sliding window size of brotli or large window brotli.

LZ77_DICTIONARY_LENGTH bytes: Contents of the LZ77 dictionary.

1 byte: NUM_CUSTOM_WORD_LISTS. May have a value in range 0 to 64.

NUM_CUSTOM_WORD_LISTS times a word list with the following format for each word list:

28 bytes: SIZE_BITS_BY_LENGTH. An array of 28 unsigned 8-bit integers, indexed by word lengths 4 to 31. The value represents log2(number of words of this length), with the exception of 0 meaning 0 words of this length. The max allowed length value is 15 bits. OFFSETS_BY_LENGTH is computed from this as OFFSETS_BY_LENGTH[i + 1] = OFFSETS_BY_LENGTH[i] + (SIZE_BITS_BY_LENGTH[i] ? (i << SIZE_BITS_BY_LENGTH[i]) : 0).

N bytes: Words dictionary data, where N is OFFSETS_BY_LENGTH[31] + (SIZE_BITS_BY_LENGTH[31] ? (31 << SIZE_BITS_BY_LENGTH[31]) : 0), with all the words of shortest length first, then all words of the next length, and so on, where there are either 0 or a positive power of two number of words for each length.

1 byte: NUM_CUSTOM_TRANSFORM_LISTS. May have a value in range 0 to 64.

NUM_CUSTOM_TRANSFORM_LISTS times a transform list with the following format for each transform list:

2 bytes: PREFIX_SUFFIX_LENGTH. The length of prefix/suffix data. Must be at least 1 because the list must always end with a zero-length stringlet even if it is empty.

NUM_PREFIX_SUFFIX times: Prefix/suffix stringlet. NUM_PREFIX_SUFFIX is the number of stringlets parsed and must be in range 1..256.

1 byte: STRING_LENGTH. The length of the entry contents. 0 for the last (terminating) entry of the transform list. For other entries, STRING_LENGTH must be in range 1..255. The 0 entry must be present and must be the last byte of the PREFIX_SUFFIX_LENGTH bytes of prefix/suffix data, else the stream must be rejected as invalid.

STRING_LENGTH bytes: Contents of the prefix/suffix.

1 byte: NTRANSFORMS. Number of transformation triplets.

NTRANSFORMS times the data for each transform listed below:

1 byte: Index of prefix in prefix/suffix data; must be less than NUM_PREFIX_SUFFIX.

1 byte: Index of suffix in prefix/suffix data; must be less than NUM_PREFIX_SUFFIX.

1 byte: Operation index; must be an index in the table of operations listed in Section 3.1.1.

If and only if at least one transform has operation index ShiftFirst or ShiftAll, then NTRANSFORMS times the following:

2 bytes: Parameters for the transform. If the transform does not have type ShiftFirst or ShiftAll, the value must be 0. ShiftFirst and ShiftAll interpret these bytes as an unsigned 16-bit integer.

If NUM_CUSTOM_WORD_LISTS > 0 or NUM_CUSTOM_TRANSFORM_LISTS > 0 (else implicitly NUM_DICTIONARIES is 1 and points to the brotli built-in and there is no context map):

1 byte: NUM_DICTIONARIES. May have a value in range 1 to 64. Each dictionary is a combination of a word list and a transform list. Each next dictionary is used when the distance goes beyond the previous. If a CONTEXT_MAP is enabled, then the dictionary matching the context is moved to the front in the order for this context.

NUM_DICTIONARIES times the DICTIONARY_MAP, which contains:

1 byte: Index into a custom word list or value NUM_CUSTOM_WORD_LISTS to indicate using the brotli [RFC7932] built-in default word list.

1 byte: Index into a custom transform list or value NUM_CUSTOM_TRANSFORM_LISTS to indicate using the brotli [RFC7932] built-in default transform list.

1 byte: CONTEXT_ENABLED. If 0, there is no context map. If 1, a context map used to select the dictionary is encoded as below.

If CONTEXT_ENABLED is 1, there is a context map for the 64 brotli [RFC7932] literals contexts:

64 bytes: CONTEXT_MAP. Index into the DICTIONARY_MAP for the first dictionary to use for this context.

6. Large Window Brotli Compressed Data Stream

Large window brotli allows a sliding window beyond the 24-bit maximum of regular brotli [RFC7932].

The compressed data stream is backwards compatible to brotli [RFC7932] and may optionally have the following differences:

In the encoding of WBITS in the stream header, the following new pattern of 14 bits is supported:

8 bits: Value 00010001 to indicate a large window brotli stream.

6 bits: WBITS. Must have value in range 10 to 62.

Distance alphabet: If the stream is a large window brotli stream, the maximum number of extra bits is 62 and the theoretical maximum size of the distance alphabet is (16 + NDIRECT + (124 << NPOSTFIX)). This overrides the value for the distance alphabet size given in Section 3.3 of [RFC7932] and affects the number of bits in the encoding of the Simple Prefix Code for distances as described in Section 3.4 of [RFC7932]. An additional limitation to distances, despite the large allowed alphabet size, is that the alphabet is not allowed to contain a distance symbol able to represent a distance larger than ((1 << 63) - 4) when its extra bits have their maximum value. It depends on NPOSTFIX and NDIRECT when this can occur.

A decoder that does not support 64-bit integers may reject a stream if WBITS is higher than 30 or a distance symbol from the distance alphabet is able to encode a distance larger than 2147483644.

7. Shared Brotli Compressed Data Stream

The format of a shared brotli compressed data stream without a framing format is backwards compatible with brotli [RFC7932] with the following optional differences:

- LZ77 dictionaries as described above are supported.
- Custom static dictionaries replacing or extending the static dictionary of brotli [RFC7932] with different words or transforms are supported.
- The stream may have the format of regular brotli [RFC7932] or the format of large window brotli as described in Section 6.

8. Shared Brotli Framing Format Stream

A compliant shared brotli framing format stream has the format described below.

8.1. Main Format

4 bytes: File signature in hexadecimal format (bytes 0x91, 0x0a, 0x42, and 0x52). The first byte contains the invalid WBITS combination for brotli [RFC7932] and large window brotli.

1 byte: Container flags that are 8 bits and have the following meanings:

bits 0 and 1: Version indicator that must be b'00. Otherwise, the decoder must reject the data stream as invalid.

bit 2: If 0, the file contains no final footer, may not contain any metadata chunks, may not contain a central directory, and may encode only a single resource (using one or more data chunks). If 1, the file may contain one or more resources, metadata, and a central directory, and it must contain a final footer.

multiple times: A chunk, each with the format specified in Section 8.2.

8.2. Chunk Format

varint: Length of this chunk excluding this varint but including all next header bytes and data. If the value is 0, then the chunk type byte is not present and the chunk type is assumed to be 0.

1 byte: CHUNK_TYPE

- 0: padding chunk
- 1: metadata chunk
- 2: data chunk
- 3: first partial data chunk
- 4: middle partial data chunk
- 5: last partial data chunk
- 6: footer metadata chunk
- 7: global metadata chunk
- 8: repeat metadata chunk
- 9: central directory chunk
- 10: final footer

If CHUNK_TYPE is not padding chunk, central directory, or final footer:

1 byte: CODEC:

- 0: uncompressed
- 1: keep decoder
- 2: brotli
- 3: shared brotli

If CODEC is not "uncompressed":

varint: Uncompressed size in bytes of the data contained within the compressed stream.

If CODEC is "shared brotli":

1 byte: Number of dictionary references. Multiple dictionary references are possible with the following restrictions: there can be 1 serialized dictionary and 15 prefix dictionaries maximum (a serialized dictionary may already contain one of those). Circular references are not allowed (any dictionary reference that directly or indirectly uses this chunk itself as dictionary).

Per dictionary reference:

1 byte: Flags:

bits 0 and 1: Dictionary source:

00:

Internal dictionary reference to a full resource by pointer, which can span one or more chunks. Must point to a full data chunk or a first partial data chunk.

01: Internal dictionary reference to single chunk contents by pointer. May point to any chunk with content (data or metadata). If a partial data chunk, only this part is the dictionary. In this case, the dictionary type is not allowed to be a serialized dictionary.

10: Reference to a dictionary by hash code of a resource. The dictionary can come from an external source, such as a different container. The user of the decoder must be able to provide the dictionary contents given its hash code (even if it comes from this container itself) or treat it as an error when the user does not have it available.

11: Invalid bit combination

bits 2 and 3: Dictionary type:

00: Prefix dictionary, set in front of the sliding window

01: Serialized dictionary in the shared brotli format as specified in Section 5.

10: Invalid bit combination

11: Invalid bit combination

bits 4-7: Must be 0

If hash-based:

1 byte: Type of hash used. Only supported value: 3, indicating 256-bit HighwayHash [HWYHASH].

32 bytes: 256-bit HighwayHash checksum to refer to dictionary.

If pointer based: Varint-encoded pointer to its chunk in this container. The chunk must come in the container earlier than the current chunk.

X bytes: Extra header bytes, depending on CHUNK_TYPE. If present, they are specified in the subsequent sections.

remaining bytes: The chunk contents. The uncompressed data in the chunk content depends on CHUNK_TYPE and is specified in the subsequent sections. The compressed data has following format depending on CODEC:

- uncompressed: The raw bytes.
- If "keep decoder", the continuation of the compressed stream that was interrupted at the end of the previous chunk. The decoder from the previous chunk must be used and its state it had at the end of the previous chunk must be kept at the start of the decoding of this chunk.
- brotli: The bytes are in brotli format [RFC7932].

- shared brotli: The bytes are in the shared brotli format specified in Section 7.

8.3. Metadata Format

All the metadata chunk types use the following format for the uncompressed content:

Per field:

2 bytes: Code to identify this metadata field. This must be two lowercase or two uppercase alpha ASCII characters. If the decoder encounters a lowercase field that it does not recognize for the current chunk type, non-ASCII characters, or non-alpha characters, the decoder must reject the data stream as invalid. Uppercase codes may be used for custom user metadata and can be ignored by a compliant decoder.

varint: Length of the content of this field in bytes, excluding the code bytes and this varint.

N bytes: The contents of this field.

The last field is reached when the chunk content end is reached. If the length of the last field does not end at the same byte as the end of the uncompressed content of the chunk, the decoder must reject the data stream as invalid.

8.4. Chunk Specifications

8.4.1. Padding Chunk (Type 0)

All bytes in this chunk must be zero except for the initial varint that specifies the remaining chunk length.

Since the varint itself takes up bytes as well, when the goal is to introduce a number of padding bytes, the dependence of the length of the varint on the value it encodes must be taken into account.

A single byte varint with a value of 0 is a padding chunk of length 1. For more padding, use higher varint values. Do not use multiple shorter padding chunks since this is slower to decode.

8.4.2. Metadata Chunk (Type 1)

This chunk contains metadata that applies to the resource whose beginning is encoded in the subsequent data chunk or first partial data chunk.

The contents of this chunk follows the format described in Section 8.3.

The following field types are recognized:

id (N bytes):

Name field. May appear 0 or 1 times. Has the following format: name in UTF-8 encoding, length determined by the field length. Treated generically but may be used as a filename. If used as a filename, forward slashes '/' should be used as directory separators, relative paths should be used, and filenames ending in a slash with 0-length content in the matching data chunk should be treated as an empty directory.

mt (8 bytes): Modification type. May appear 0 or 1 times. Has the following format: contains microseconds since epoch, as a little-endian, signed two's complement 64-bit integer.

custom user field: Any two uppercase ASCII characters.

8.4.3. Data Chunk (Type 2)

A data chunk contains the actual data of a resource.

This chunk has the following extra header bytes:

1 byte: Flags:

bit 0: If true, indicates this is not a resource that should be output implicitly as part of extracting resources from this container. Instead, it may be referred to only explicitly, e.g., as a dictionary reference by hash code or offset. This flag should be set for data used as dictionary to improve compression of actual resources.

bit 1: If true, hash code is given.

bits 2-7: Must be zero.

If hash code is given:

1 byte: Type of hash used. Only supported value: 3, indicating 256-bit HighwayHash [HWYHASH].

32 bytes: 256-bit HighwayHash checksum of the uncompressed data.

The uncompressed content bytes of this chunk are the actual data of the resource.

8.4.4. First Partial Data Chunk (Type 3)

This chunk contains partial data of a resource. This is the first chunk in a series containing the entire data of the resource.

The format of this chunk is the same as the format of a data chunk (Section 8.4.3) except for the differences noted below.

The second bit of flags must be set to 0 and no hash code given.

The uncompressed data size is only of this part of the resource, not of the full resource.

8.4.5. Middle Partial Data Chunk (Type 4)

This chunk contains partial data of a resource and is neither the first nor the last part of the full resource.

The format of this chunk is the same as the format of a data chunk (Section 8.4.3) except for the differences noted below.

The first and second bits of flags must be set to 0.

The uncompressed data size is only of this part of the resource, not of the full resource.

8.4.6. Last Partial Data Chunk (Type 5)

This chunk contains the final piece of partial data of a resource.

The format of this chunk is the same as the format of a data chunk (Section 8.4.3) except for the differences noted below.

The first bit of flags must be set to 0.

If a hash code is given, the hash code of the full resource (concatenated from all previous chunks and this chunk) is given in this chunk.

The uncompressed data size is only of this part of the resource, not of the full resource.

The type of this chunk indicates that there are no further chunk encoding this resource, so the full resource is now known.

8.4.7. Footer Metadata Chunk (Type 6)

This metadata applies to the resource whose encoding ended in the preceding data chunk or last partial data chunk.

The contents of this chunk follows the format described in Section 8.3.

There are no lowercase field types defined for footer metadata. Uppercase field types can be used as custom user data.

8.4.8. Global Metadata Chunk (Type 7)

This metadata applies to the whole container instead of a single resource.

The contents of this chunk follows the format described in Section 8.3.

There are no lowercase field types defined for global metadata. Uppercase field types can be used as custom user data.

8.4.9. Repeat Metadata Chunk (Type 8)

These chunks optionally repeat metadata that is interleaved between data chunks. To use these chunks, it is necessary to also read additional information, such as pointers to the original chunks, from the central directory.

The contents of this chunk follows the format described in Section 8.3.

This chunk has an extra header byte:

1 byte: Chunk type of repeated chunk (metadata chunk or footer metadata chunk).

This set of chunks must follow the following restrictions:

- It is optional whether or not repeat metadata chunks are present.
- If they are present, then they must be present for all metadata chunks and footer metadata chunks.
- There may be only 1 repeat metadata chunk per repeated metadata chunk.
- They must appear in the same order as the chunks appear in the container, which is also the same order as listed in the central directory.
- Compression of these chunks is allowed; however, it is not allowed to use any internal dictionary except an earlier repeat metadata chunk of this series, and it is not allowed for a metadata chunk to keep the decoder state if the previous chunk is not a repeat metadata chunk. That is, the series of metadata chunks must be decompressible without using other chunks of the framing format file.

The fields contained in this metadata chunk must follow the following restrictions:

- If a field is present, it must exactly match the corresponding field of the copied chunk.
- It is allowed to leave out a field that is present in the copied chunk.
- If a field is present, then it must be present in **all** other repeat metadata chunks when the copied chunk contains this field. In other words, if you know you can get the name field from a repeat chunk, you know that you will be able to get all names of all resources from all repeat chunks.

8.4.10. Central Directory Chunk (Type 9)

The central directory chunk along with the repeat metadata chunks allow quickly finding and listing compressed resources in the container file.

The central directory chunk is always uncompressed and does not have the codec byte. It instead has the following format:

varint: Pointer into the file where the repeat metadata chunks are located or 0 if they are not present.

per chunk listed:

varint: Pointer into the file where this chunk begins.

varint: Number of header bytes N used below.

N bytes: Copy of all the header bytes of the pointed at chunk, including total size, chunk type byte, codec, uncompressed size, dictionary references, and X extra header bytes. The content is not repeated here.

The last listed chunk is reached when the end of the contents of the central directory are reached. If the end does not match the last byte of the central directory, the decoder must reject the data stream as invalid.

If present, the central directory must list all data and metadata chunks of all types.

8.4.11. Final Footer Chunk (Type 10)

The final footer chunk closes the file and is only present if bit 2 of the initial container flags was set.

This chunk has the following content, which is always uncompressed:

reversed varint: Size of this entire framing format file, including these bytes themselves, or 0 if this size is not given.

reversed varint: Pointer to the start of the central directory, or 0 if there is none.

A reversed varint has the same format as a varint but its bytes are in reversed order, and it is designed to be parsed from the end of the file towards the beginning.

8.4.12. Chunk Ordering

The chunk ordering must follow the rules described below. If the decoder sees otherwise, it must reject the data stream as invalid.

Padding chunks may be inserted anywhere, even between chunks for which the rules below say no other chunk types may come in between.

Metadata chunks must come immediately before the data chunks of the resource they apply to.

Footer metadata chunks must come immediately after the data chunks of the resource they apply to.

There may be only 0 or 1 metadata chunks per resource.

There may be only 0 or 1 footer metadata chunks per resource.

A resource must exist out of either 1 data chunk or 1 first partial data chunk, 0 or more middle partial data chunks, and 1 last partial data chunk, in that order.

Repeat metadata chunks must follow the rules of Section 8.4.9.

There may be only 0 or 1 central directory chunks.

If bit 2 of the container flags is set, there may be only a single resource, no metadata chunks of any type, no central directory, and no final footer.

If bit 2 of the container flags is not set, there must be exactly 1 final footer chunk, and it must be the last chunk in the file.

9. Security Considerations

The security considerations for brotli [RFC7932] apply to shared brotli as well.

In addition, the same considerations apply to the decoding of new file format streams for shared brotli, including shared dictionaries, the framing format, and the shared brotli format.

The dictionary must be treated with the same security precautions as the content because a change to the dictionary can result in a change to the decompressed content.

The CRIME attack [CRIME] shows that it's a bad idea to compress data from mixed (e.g., public and private) sources -- the data sources include not only the compressed data but also the dictionaries. For example, if you compress secret cookies using a public-data-only dictionary, you still leak information about the cookies.

The dictionary can reveal information about the compressed data and vice versa. That is, data compressed with the dictionary can reveal contents of the dictionary when an adversary can control parts of the data to compress and see the compressed size. On the other hand, if the adversary can control the dictionary, the adversary can learn information about the compressed data.

The most robust defense against CRIME is not to compress private data, e.g., sensitive headers like cookies or any content with personally identifiable information (PII). The challenge has been to identify secrets within a vast amount of data to be compressed. Cloudflare uses a regular expression [CLOUDFLARE]. Another idea is to extend existing web template systems (e.g., Soy [SOY]) to allow developers to mark secrets that must not be compressed.

A less robust idea, but easier to implement, is to randomize the compression algorithm, i.e., adding randomly generated padding, varying the compression ratio, etc. The tricky part is to find the right balance between cost and security (i.e., on one hand, we don't want to add too much padding because it adds a cost to data, but on the other hand, we don't want to add too little because the adversary can detect a small amount of padding with traffic analysis).

Additionally, another defense is to not use dictionaries for cross- domain requests and to only use shared brotli for the response when the origin is the same as where the content is hosted (using CORS). This prevents an adversary from using a private dictionary with user secrets to

compress content hosted on the adversary's origin. It also helps prevent CRIME attacks that try to benefit from a public dictionary by preventing data compression with dictionaries for requests that do not originate from the host itself.

The content of the dictionary itself should not be affected by external users; allowing adversaries to control the dictionary allows a form of chosen plaintext attack. Instead, only base the dictionary on content you control or generic large scale content such as a spoken language and update the dictionary with large time intervals (days, not seconds) to prevent fast probing.

The use of HighwayHash [HWYHASH] for dictionary identifiers does not guarantee against collisions in an adversarial environment and is intended to be used for identifying the dictionary within a trusted, known set of dictionaries. In an adversarial environment, users of shared brotli should use another mechanism to validate a negotiated dictionary such as a cryptographically proven secure hash.

10. IANA Considerations

This document has no IANA actions.

11. References

11.1. Normative References

[HWYHASH] Alakuijala, J., Cox, B., and J. Wassenberg, "Fast keyed hash/pseudo-random function using SIMD multiply and permute", DOI 10.48550/arXiv.1612.06257, February 2017, <https://arxiv.org/abs/1612.06257>.

[RFC7932] Alakuijala, J. and Z. Szabadka, "Brotli Compressed Data Format", RFC 7932, DOI 10.17487/RFC7932, July 2016, <https://www.rfc-editor.org/info/rfc7932>.

11.2. Informative References

[CLOUDFLARE] Loring, B., "A Solution to Compression Oracles on the Web", The Cloudflare Blog, 27 March 2018, <https://blog.cloudflare.com/a-solution-to-compression-oracles-on-the-web/>.

[CRIME] CVE Program, "CVE-2012-4929", <https://www.cve.org/CVERecord?id=CVE-2012-4929>.

[LZ77] Ziv, J. and A. Lempel, "A Universal Algorithm for Sequential Data Compression", IEEE Transactions on Information Theory, vol. 23, no. 3, pp. 337-343, DOI 10.1109/TIT.1977.1055714, May 1977, <https://doi.org/10.1109/TIT.1977.1055714>.

[SOY] Google Developers, "Closure Tools", <https://developers.google.com/closure>.

Acknowledgments

The authors would like to thank Robert Obryk for suggesting improvements to the format and the text of the specification.

Authors' Addresses

Jyrki Alakuijala
Google, Inc.
Email: jyrki@google.com

Thai Duong
Google, Inc.
Email: thaidn@google.com

Evgenii Kliuchnikov
Google, Inc.
Email: eustas@google.com

Zoltan Szabadka
Google, Inc.
Email: szabadka@google.com

Lode Vandevenne (EDITOR)
Google, Inc.
Email: lode@google.com