Network Working Group
Internet-Draft
Intended status: Informational
Expires: September 6, 2018

V. Krasnov
Cloudflare, Inc.
Y. Weiss
Akamai Technologies, Inc.

March 5, 2018

Compression Dictionaries for HTTP/2
draft-vkrasnov-h2-compression-dictionaries-03

Abstract

This document specifies new HTTP/2 frame types and new HTTP/2 settings values that enable the use of previously transferred data as compression dictionaries, significantly improving overall compression ratio for a given connection.

In addition, this document proposes to define a set of industry standard, static, dictionaries to be used with any Lempel-Ziv based compression for the common textual MIME types prevalent on the web.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 6, 2018.

Copyright Notice

Table of Contents

1. Introduction

The HTTP/2 [RFC7540] protocol encourages the use of many small assets for CSS/JS/HTML, due to its multiplexed nature. Prior to HTTP/2, asset inlining was encouraged, resulting in fewer, larger assets per website.

The HTTP/2 protocol also allows for transmitted data to be compressed with a lossless compression format. The format used is specified in the "Content-Encoding" (see [RFC2616], section 14.11) header field. For example, "Content-Encoding: br" means the data was compressed using the Brotli format.

The nature of the compression algorithms, such as DEFLATE [RFC1951] and Brotli [RFC7932], used with HTTP in practice, require a certain "window" of data to perform backward matching. Therefore, larger files have much better compression ratio. To improve compression for

smaller files, these algorithms allow to use a chunk of arbitrary data as a "Custom Dictionary" and function as the initial sliding window.

Note: While that is not longer true for the latest stable version of Brotli, there's work underway to re-enable use of arbitrary compression dictionaries.

Compression is a compute-heavy operation, where investing additional compute power results in diminishing returns (in terms of compression ratio/CPU cycles). The "Custom Dictionary" technique is known to improve compression ratio significantly, with little additional computational cost. It is also supported by most Lempel-Ziv based compression formats.

This document introduces a mechanism for using previously transmitted data over HTTP/2 as a dictionary to be used with an underlying compression algorithm.

1.1. Conventions and Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

2. Preliminaries

2.1. Security Considerations

The use of compression over an encrypted connection could be used by malicious actors to potentially leak sensitive information. We will collaborate with industry experts to identify any additional attack vectors introduced by this draft, and include a set of best practices to both servers and clients that would implement it.

A list of attack vectors and potential mitigations is described later in this document.

2.2. Content Coding

A server that wishes to apply protocol level compression on a stream or use a stream as a dictionary SHOULD not apply non-identity content-coding (see [RFC7231], section 3.1.2.1) to that stream.

2.3. Compression Contexts

In the scope of this document, a compression context is a set of non-overlaping streams, that SHALL only be used as compression dictionaries for streams within the same compression context. While it is the responsibility of the server to implement best-practice techniques to mitigate cross-compression side channel attacks, compression contexts let the client mitigate some of the risks of cross-compression side channel attacks, by explicitly stating which requests can be cross-compressed with which requests.

For example a client may choose to disable compression for cross-site requests by assigning them to different compression contexts.

2.4. Server Push Interaction

Pushed streams may be cross-stream compressed or used as dictionaries, same as a regular stream. In some scenarios it may benefit the server to push a dummy resource to prime a dictionary.

2.5. HTTP/QUIC

Due to the nature of this draft, it is expected that a strict order is maintained between the definition and consumption of dictionaries. The nature of QUIC is such that frames and streams might not delivered in the order they are sent, therefore, a head-of-line blocking may occur when implementing compression dictionaries in HTTP/QUIC. This is similar to the tradeoff present in the HPACK/QUIC mapping.

3. HTTP/2 Extension

3.1. Extension Settings

The extension introduces a new SETTINGS value.

SETTINGS_COMPRESSION(0xTBA): For greater compression, and to prevent setting identifier depletion, the 32-bit value for this setting is defined as follows:

NDict: Indicates the number of dictionaries the client is willing to maintain. The default value is 0, the maximal value is 255.

DSize: Log2 of the maximal size of each dictionary. The default value is 0, the maximal value is 255. For example value of 17 indicates each dictionary MUST be smaller or equal to 2^17 (131,072 octets).

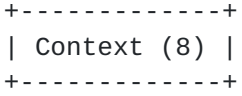
Fmt: Compression format to use, as a bitmask. 1st bit indicates brotli, 2nd bit indicates zlib. Other bits are reserved for future compression methods. A value of 0 indicates no support for cross-stream compression.

SDVersion: If greater than 0, indicates the version of static dictionaries to use. Maximal value is 255, the default value is 0, which indicates no static dictionaries are used.

3.2. Extension Frames

3.2.1. The SET_COMPRESSION_CONTEXT frame

The SET_COMPRESSION_CONTEXT frame (type=0xTBA).



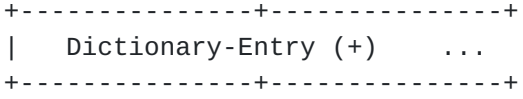
The SET_COMPRESSION_CONTEXT frame can be sent by the client on any stream in the idle state. The frame indicates the compression context ID for the given stream. Frames with an assigned context SHALL NOT be compressed using dictionaries from a different context. Frames with an assigned context SHALL NOT be used as a dictionary for streams with from a different context.

The SET_COMPRESSION_CONTEXT frame contains the following fields:

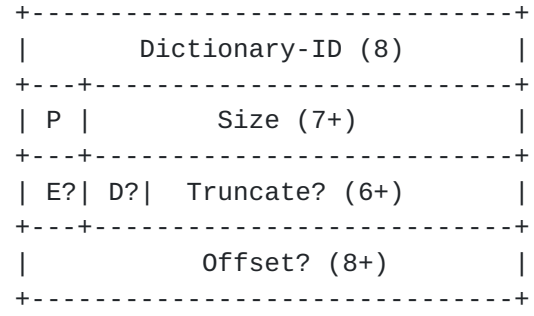
Context: an 8-bit context ID that indicates the compression context for the stream. If the frame is ommited, then the context value is assumed to be 0. The allowed context values are 0 through 255. A special context ID of 255 indicates the stream can only be compressed using the static dictionaries.

3.2.2. The SET_DICTIONARY Frame

The SET_DICTIONARY frame (type=0xTBA) contains one to many Dictionary-Entry.



A Dictionary-Entry field is encoded as follows:



The SET_DICTIONARY frame can be sent from the server to the client, on any client initiated stream in the open or half-closed (remote) states, or on any server initiated stream in the reserved (local) state. The SET_DICTIONARY frame MUST precede any DATA frames on that stream. The SET_DICTIONARY frame SHOULD be followed by sufficient DATA frames to build the dictionaries. If a RST frame was received for the stream before sufficient DATA was sent, the dictionaries are reset.

The Dictionary-Entry contains the following fields:

Dictionary-ID: an 8-bit ID, indicates the dictionary. MUST be lower than the value agreed by the SETTINGS_COMPRESSION setting.

Size: Indicates how many octets of the stream will be used for the dictionary. Size is represented as an integer with 7-bit prefix (see [RFC7541], Section 5.1). If P is set, the actual number of octets to use is 2 to the power of Size. If the computed value is greater than the length of the decompressed DATA, use all the available DATA.

Truncate: An optional field, represented as an integer with 6-bit prefix. Present when the APPEND flag is set. Truncate indicates the number of octets to keep of the existing dictionary, before appending the new data to it. If E is set, then Truncate is ignored, and new data is appended at the end. If Truncate is zero, then the dictionary is replaced, as if APPEND was unset. If the optional field D is set, then the first Truncate octets of the previous dictionary are used, otherwise the last Truncate octets are used.

Offset: An optional field, represented as an integer with 8-bit prefix. Present when the OFFSET flag is set. Offset indicates that the first Offset octets of the stream are ignored when building the dictionary.

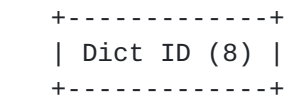
The flags defined for the SET_DICTIONARY frame apply to each Dictionary-Entry in the frame. The SET_DICTIONARY frame defines the following flags:

APPEND (0x1): Indicates that the data is to be appended to the existing dictionary with the given ID, as opposed to replacing it with the new data. Also indicates that fields E, D and Truncate are present.

OFFSET (0x2): Indicates the presence of the Offset field.

3.2.3. The USE_DICTIONARY Frame

The USE_DICTIONARY frame (type=0xTBA).



The USE_DICTIONARY frame indicates that the current stream is compressed with the indicated dictionary. The USE_DICTIONARY frame MUST be sent prior to any DATA frame on a given stream. SET_DICTIONARY and USE_DICTIONARY frames MAY be sent on the same stream. Only one USE_DICTIONARY frame MAY be sent for a stream.

The USE_DICTIONARY frame contains the following fields:

Dict ID: an 8-bit ID that indicates which dictionary to use. The dictionary MUST be previously defined by a SET_DICTIONARY frame, or by a static dictionary.

3.3. Static Dictionaries

This document proposes to generate a set of up to 8 standard dictionaries to be optionally bundled with supporting implementations. Each dictionary should be 32,768 or 65,536 octets long.

Each static dictionary will be identified by an integer ID in the range {0..7}.

If either endpoint supports the use of static dictionaries, it will indicate this by setting the SDVersion value of SETTINGS_COMPRESSION to greater than 0. The number will indicate the highest version of the dictionaries known.

The actual version used will be the lowest of the two values set by the endpoints.

If the client and the server agree on the use of static dictionaries, then both will initialize the first 8 dictionaries (IDs 0 through 7), with the contents of the static dictionaries. The static dictionaries belong to context 0.

If the value of the field NDict is lower than 8, then up to NDict dictionaries will be initialized.

4. Dictionary State

Both the server and the client MUST process the SET_DICTIONARY and USE_DICTIONARY frames in the order they are sent/received, with the exception when both are sent over the same stream. In that case USE_DICTIONARY is processed prior to the SET_DICTIONARY frames.

Doing otherwise will result in an illegal state of the dictionaries. This is similar to the way HEADER frames are processed in order to maintain legal HPACK state on the server and the client.

A possible dictionary implementation can be describes as follows:

```
struct {
    u8  id;
    u8  ctx;
    u64 size;
    u8  dict[size];
} D;
```

The collection of dictionaries could then be described as:

```
D dictionaries[NDict];
```

Initially all the dictionaries are unitialized:

```
for (i = 0; i < NDict; i++) {
    dictionaries[i] = {id = i, ctx = 0, size = 0, dict = {}};
}
```

Client side USE_DICTIONARY frame behaviour pseudo code:

```
dictionary = dictionaries[frame.Dictionary-ID]

if (dictionary.ctx != 0 && dictionary.ctx != stream.ctx)
    return PROTOCOL_ERROR

stream.decompressed_data = decompress(stream.dict, stream.data)
```

Client side SET_DICTIONARY frame behaviour pseudo code:

```
   foreach entry = frame.Dictionary-Entry {
       dictionary = dictionaries[entry.DICT_ID]

       if (entry.size == 0) {
           dictionary.size = 0
           dictionary.ctx = 0
           dictionary.dict = {}
           continue
       }

       if (dictionary.ctx != 0 && dictionary.ctx != stream.ctx) {
           return PROTOCOL_ERROR
       }

       dictionary.ctx = stream.ctx

       if (entry.P == 1) {
           size = 1 << entry.Size
       } else {
           size = entry.Size
       }

       if (frame.APPEND) {
           if (entry.E == 1) {
               truncate = dictionary.size
           } else {
               truncate = entry.Truncate
           }
       } else {
           truncate = 0
       }

       if (frame.OFFSET) {
           offset = entry.Offset
       } else {
           offset = 0
       }

       new_dict_data = stream.decompressed_data[offset:offset + size]
       if (entry.D == 1) {
           old_dict_data = head(dictionary.dict, truncate)
       } else {
           old_dict_data = tail(dictionary.dict, truncate)
       }

       dict_data = append(old_dict_data, new_dict_data)

       dictionary.dict = tail(dict_data, 1 << settings.DSize)
```

```
        dictionary.size = len(dictionary.dict)
    }
```

The server behaviour mirrors the client behaviour, but it is up to the server to choose the best dictionary.

4.1. Attack scenarios and mitigations

A single HTTP/2 connection is likely to be shared among multiple origins (over which it is authoritative) and among different navigation contexts to the same origin. When such sharing happens, and if compression contexts are shared between those instances, an attacker can use a BREACH-style attack in order to exfiltrate secrets from the context. Such secrets may include:

- Cookies set using Javascript (and in-particular "httponly" cookies set from anonymous functions in external JS, which is not accessible to scripts otherwise)
- CSRF tokens
- CSP nonces
- Application level secrets (e.g. financial information, stored credit cards numbers, codes, etc.)

The mechanism for such data theft can happen if the attacker can: * Download multiple similar payloads to the target page modulo the actual secret, while trying out multiple permutations of the secret. * Observe the on-the-wire transfer size using Resource Timing's "transferSize" property.

The rest of this section will describe different scenarios where those conditions are met as well as potential mitigations for them.

4.1.1. Cross-origin secret leak

An HTTP/2 session can be used to deliver resources from multiple origins over which the session has proved to be authoritative, through connection reuse (see [RFC7540] section 9.1.1 for more details). As a result, sharing compression contexts between such origins can be theoretically used to leak secrets from one of these origins to the next.

4.1.1.1. Mitigation

Limiting compression contexts to be used within the confines of a single origin.

4.1.2. Same-origin secret leak

Malicious pages on the origin as well as an XSS attacker can normally use "fetch()" or "XMLHttpRequest()" in order to inspect in-content secrets. This could be limited with CSP by only permitting the download of specific files, using nonces or using "connect-src 'none'" in order to limit arbitrary scripts from downloading files that contain secrets. However, using shared-dictionaries between secret resources and malicious ones can enable an attacker to guess said secrets and exfiltrate them (e.g. using other deficiencies in the defined CSP, if there are any).

Furthermore, said malicious page or XSS attack can also use as a dictionary resources fetched from the same origin in a different browsing context, enabling it to also inspect resources which cannot be fetched at all on its base page.

4.1.2.1. Mitigation

There's no obvious mitigation for this kind of attack, but a few options are:

- Limiting compression contexts to be used only within a single navigation context can limit the opportunity for the separate navigation context to inspect secrets from resources it is not allowed to fetch. At the same time this can be complex to implement, as the network layer is not aware of the navigation context and is supposed for example to dedupe outgoing requests from different compression contexts.

- "transferSize" padding/bucketing in such cases (e.g. pages with above mentioned CSP limitations) may be enough to render this attack not-practical.

- Limit dictionary sharing (or "transferSize" accuracy for resources that use shared dictionaries) only to non-credentialed resource fetches.

5. References

5.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, DOI 10.17487/RFC2616, June 1999, <https://www.rfc-editor.org/info/rfc2616>.

[RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <https://www.rfc-editor.org/info/rfc7231>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <https://www.rfc-editor.org/info/rfc7540>.

[RFC7541] Peon, R. and H. Ruellan, "HPACK: Header Compression for HTTP/2", RFC 7541, DOI 10.17487/RFC7541, May 2015, <https://www.rfc-editor.org/info/rfc7541>.

5.2. Informative References

[BREACH] Prado, A., Harris, N., and Y. Gluck, "BREACH: SSL, Gone in 30 Seconds", 2013, <http://breachattack.com/>.

[RFC1951] Deutsch, P., "DEFLATE Compressed Data Format Specification version 1.3", RFC 1951, DOI 10.17487/RFC1951, May 1996, <https://www.rfc-editor.org/info/rfc1951>.

[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>.

Authors' Addresses

Vlad Krasnov
Cloudflare, Inc.

Email: vlad@cloudflare.com

Yoav Weiss
Akamai Technologies, Inc.

Email: yoav@yoav.ws