Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: 28 January 2021

J. Yasskin
Google

27 July 2020

Web Bundles
draft-yasskin-wpack-bundled-exchanges-03

Abstract

Web bundles provide a way to bundle up groups of HTTP responses, with the URLs and request URLs that produced them, to transmit or store them together. They can include multiple top-level resources with one identified as the default by a primaryUrl metadata, provide random access to their component exchanges, and efficiently store 8-bit resources.

Note to Readers

Discussion of this draft takes place on the wpack mailing list (wpack@ietf.org), which is archived at https://www.ietf.org/mailman/listinfo/wpack (https://www.ietf.org/mailman/listinfo/wpack).

The source code and issues list for this draft can be found in https://github.com/WICG/webpackage (https://github.com/WICG/webpackage).

Status of This Memo

Table of Contents

1. Introduction 3
  1.1. Terminology 3
  1.2. Mode of specification 3
2. Semantics 3
  2.1. Stream attributes and operations 4
  2.2. Load a bundle's metadata 4
    2.2.1. Load a bundle's metadata from the end 5
  2.3. Load a response from a bundle 5
3. Format 6
  3.1. Top-level structure 6
  3.2. Serving constraints 7
  3.3. Load a bundle's metadata 7
    3.3.1. Parsing the index section 10
    3.3.2. Parsing the manifest section 13
    3.3.3. Parsing the signatures section 13
    3.3.4. Parsing the critical section 15
    3.3.5. The responses section 15
    3.3.6. Starting from the end 15
  3.4. Load a response from a bundle 16
  3.5. Parsing CBOR items 18
    3.5.1. Parse a known-length item 18
    3.5.2. Parsing variable-length data from a bytestring . . . 19
    3.5.3. Parsing the type and argument of a CBOR item 19
  3.6. Interpreting CBOR HTTP headers 20
4. Guidelines for bundle authors 21
5. Security Considerations 21
  5.1. Version skew 21
  5.2. Content sniffing 21
6. IANA considerations 23
  6.1. Internet Media Type Registration 23
  6.2. Web Bundle Section Name Registry 24
7. References 25

7.1. Normative References 25
7.2. Informative References 26
Appendix A. Change Log 27
Appendix B. Acknowledgements 27
Author's Address 28

1. Introduction

To satisfy the use cases in [I-D.yasskin-wpack-use-cases], this document proposes a new bundling format to group HTTP resources. Several of the use cases require the resources to be signed: that's provided by bundling signed exchanges ([I-D.yasskin-http-origin-signed-responses]) rather than natively in this format.

1.1. Terminology

Exchange (noun) An HTTP request+response pair. This can either be a request from a client and the matching response from a server or the request in a PUSH_PROMISE and its matching response stream. Defined by Section 8 of [RFC7540].

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 BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

1.2. Mode of specification

This specification defines how conformant bundle parsers work. It does not constrain how encoders produce a bundle: although there are some guidelines in Section 4, encoders MAY produce any sequence of bytes that a conformant parser would parse into the intended semantics.

This specification uses the conventions and terminology defined in the Infra Standard ([INFRA]).

2. Semantics

A bundle is logically a set of HTTP exchanges, with a URL identifying the primary resource of the bundle.

While the order of the exchanges is not semantically meaningful, it can significantly affect performance when the bundle is loaded from a network stream.

A bundle is parsed from a stream of bytes, which is assumed to have the attributes and operations described in Section 2.1.

Bundle parsers support two operations, Load a bundle’s metadata (Section 2.2) and Load a response from a bundle (Section 2.3) each of which can return an error instead of their normal result.

A client is expected to load the metadata for a bundle as soon as it starts downloading it or otherwise discovers it. Then, when fetching ([FETCH]) a request, the client is expected to match it against the requests in the metadata, and if one matches, load that request’s response.

2.1. Stream attributes and operations

* A sequence of *available bytes*. As the stream delivers bytes, these are appended to the available bytes.

* An *EOF* flag that’s true if the available bytes include the entire stream.

* A *current offset* within the available bytes.

* A *seek to offset N* operation to set the current offset to N bytes past the beginning of the available bytes. A seek past the end of the available bytes blocks until N bytes are available. If the stream ends before enough bytes are received, either due to a network error or because the stream has a finite length, the seek fails.

* A *read N bytes* operation, which blocks until N bytes are available past the current offset, and then returns them and seeks forward by N bytes. If the stream ends before enough bytes are received, either due to a network error or because the stream has a finite length, the read operation returns an error instead.

2.2. Load a bundle’s metadata

This takes the bundle’s stream and returns either an error (where an error is a "format error" or a "version error"), an error with a fallback URL (which is also the primaryUrl when the bundle parses successfully), or a map ([INFRA]) of metadata containing at least keys named:

primaryUrl The URL of the main resource in the bundle. If the client can’t process the bundle for any reason, this is also the fallback URL, a reasonable URL to try to load instead.

requests A map ([INFRA]) whose keys are URLs and whose values consist of either:

* A single "ResponseMetadata" value for a non-content-negotiated resource or

* A set of content-negotiated resources represented by

  - A "Variants" header field value ([I-D.ietf-httpbis-variants]) and

  - A map ([INFRA]) from each of the possible combinations of one available-value for each variant-axis to a "ResponseMetadata" structure. Load a response from a bundle can use the "ResponseMetadata" structures to find the matching response.

The map may include other items added by sections defined in the Web Bundle Section Name Registry.

This operation only waits for a prefix of the stream that, if the bundle is encoded with the "responses" section last, ends before the first response.

This operation's implementation is in Section 3.3.

2.2.1. Load a bundle's metadata from the end

If a bundle's bytes are embedded in a longer sequence rather than being streamed, a parser can also load them starting from a pointer to the last byte of the bundle. This returns the same data as Section 2.2.

This operation's implementation is in Section 3.3.6.

2.3. Load a response from a bundle

This takes the stream of bytes representing the bundle, a request ([FETCH]), and the "ResponseMetadata" returned from Section 2.2 for the appropriate content-negotiated resource within the request's URL, and returns the response ([FETCH]) matching that request.

This operation can be completed without inspecting bytes other than those that make up the loaded response, although higher-level operations like proving that an exchange is correctly signed ([I-D.yasskin-http-origin-signed-responses]) may need to load other responses.

A client will generally want to load the response for a request that the client generated. For a URL with multiple variants, the client SHOULD use the algorithm in Section 4 of [I-D.ietf-httpbis-variants] to select the best variant.

This operation's implementation is in Section 3.4.

3. Format

3.1. Top-level structure

This section is non-normative.

A bundle holds a series of named sections. The beginning of the bundle maps section names to the range of bytes holding that section. The most important section is the "index" (Section 3.3.1), which similarly maps serialized HTTP requests to the range of bytes holding that request's serialized response. Byte ranges are represented using an offset from some point in the bundle _after_ the encoding of the range itself, to reduce the amount of work needed to use the shortest possible encoding of the range.

Future specifications can define new sections with extra data, and if necessary, these sections can be marked "critical" (Section 3.3.4) to prevent older parsers from using the rest of the bundle incorrectly.

The bundle is a CBOR item ([CBORbis]) with the following CDDL ([CDDL]) schema:

```
webbundle = [
  ;  in UTF-8.
  magic: h'F0 9F 8C 90 F0 9F 93 A6',
  version: bytes .size 4,
  primary-url: whatwg-url,
  section-lengths: bytes
    .cbor [* (section-name: tstr, length: uint) ],
  sections: [* any ],
  length: bytes .size 8,  ; Big-endian number of bytes in the bundle.
]

$section-name /=
  "index" / "manifest" / "signatures" / "critical" / "responses"

$section /= index / manifest / signatures / critical / responses

responses = [*response]

whatwg-url = tstr
```

3.2. Serving constraints

When served over HTTP, a response containing an "application/webbundle" payload MUST include at least the following response header fields, to reduce content sniffing vulnerabilities (Section 5.2):

* Content-Type: application/webbundle

* X-Content-Type-Options: nosniff

3.3. Load a bundle’s metadata

A bundle holds a series of sections, which can be accessed randomly using the information in the "section-lengths" CBOR item, which holds a list of alternating section names and section lengths:

```
section-lengths = [* (section-name: tstr, length: uint) ],
```

To implement Section 2.2, the parser MUST run the following steps, taking the "stream" as input.

1. Seek to offset 0 in "stream". Assert: this operation doesn’t fail.

2. If reading 10 bytes from "stream" returns an error or doesn’t return the bytes with hex encoding "86 48 F0 9F 8C 90 F0 9F 93 A6" (the CBOR encoding of the 6-item array initial byte and 8-byte bytestring initial byte, followed by 🌐📦 in UTF-8), return a "format error".

3. Let "version" be the result of reading 5 bytes from "stream". If this is an error, return a "format error".

4. Let "urlType" and "urlLength" be the result of reading the type and argument of a CBOR item from "stream" (Section 3.5.3). If this is an error or "urlType" is not 3 (a CBOR text string), return a "format error".

5. Let "fallbackUrlBytes" be the result of reading "urlLength" bytes from "stream". If this is an error, return a "format error".

6. Let "fallbackUrl" be the result of parsing ([URL]) the UTF-8 decoding of "fallbackUrlBytes" with no base URL. If either the UTF-8 decoding or parsing fails, return a "format error".

Note: From this point forward, errors also include the fallback URL to help clients recover.

7. If "version" does not have the hex encoding "44 31 00 00 00" (the CBOR encoding of a 4-byte byte string holding an ASCII "1" followed by three 0 bytes), return a "version error" with "fallbackUrl".

   Note: RFC EDITOR PLEASE DELETE THIS NOTE; Implementations of drafts of this specification MUST NOT use the version "1" in this byte string, and MUST instead define an implementation-specific string to identify which draft is implemented. This string SHOULD match the version used in the draft's MIME type (Section 6.1).

8. Let "sectionLengthsLength" be the result of getting the length of the CBOR bytestring header from "stream" (Section 3.5.2). If this is an error, return a "format error" with "fallbackUrl".

9. If "sectionLengthsLength" is 8192 (8*1024) or greater, return a "format error" with "fallbackUrl".

10. Let "sectionLengthsBytes" be the result of reading "sectionLengthsLength" bytes from "stream". If "sectionLengthsBytes" is an error, return a "format error" with "fallbackUrl".

11. Let "sectionLengths" be the result of parsing one CBOR item (Section 3.5) from "sectionLengthsBytes", matching the section-lengths rule in the CDDL ([CDDL]) above. If "sectionLengths" is an error, return a "format error" with "fallbackUrl".

12. Let ("sectionsType", "numSections") be the result of parsing the type and argument of a CBOR item from "stream" (Section 3.5.3).

13. If "sectionsType" is not "4" (a CBOR array) or "numSections" is not half of the length of "sectionLengths", return a "format error" with "fallbackUrl".

14. Let "sectionsStart" be the current offset within "stream".

   For example, if "sectionLengthsLength" were 52 and "sectionLengths" contained 4 items (2 sections), "sectionsStart" would be 65 (10 initial bytes + a 2-byte bytestring header to describe a 52-byte bytestring + 52 bytes of section lengths + a 1-byte array header for the 2 sections).

15. Let "knownSections" be the subset of the Section 6.2 that this client has implemented.

16. Let "ignoredSections" be an empty set.

17. Let "sectionOffsets" be an empty map ([INFRA]) from section names to (offset, length) pairs. These offsets are relative to the start of "stream".

18. Let "currentOffset" be "sectionsStart".

19. For each (""name"", "length") pair of adjacent elements in "sectionLengths":

    1. If ""name""'s specification in "knownSections" says not to process other sections, add those sections' names to "ignoredSections".

       Note: The "ignoredSections" enables sections that supercede other sections to be introduced in the future. Implementations that don't implement any such sections are free to omit the relevant steps.

    2. If "sectionOffsets["name"]" exists, return a "format error" with "fallbackUrl". That is, duplicate sections are forbidden.

    3. Set "sectionOffsets["name"]" to ("currentOffset", "length").

    4. Set "currentOffset" to "currentOffset + length".

20. If the "responses" section is not last in "sectionLengths", return a "format error" with "fallbackUrl". This allows a streaming parser to assume that it'll know the requests by the time their responses arrive.

21. Let "metadata" be a map ([INFRA]) initially containing the single key/value pair ""primaryUrl""/"fallbackUrl".

22. For each ""name"" --> ("offset", "length") triple in "sectionOffsets":

    1. If ""name"" isn't in "knownSections", continue to the next triple.

    2. If ""name""'s Metadata field (Section 6.2) is "No", continue to the next triple.

3. If ""name"" is in "ignoredSections", continue to the next triple.

4. Seek to offset "offset" in "stream". If this fails, return a "format error" with "fallbackUrl".

5. Let "sectionContents" be the result of reading "length" bytes from "stream". If "sectionContents" is an error, return a "format error" with "fallbackUrl".

6. Follow ""name""'s specification from "knownSections" to process the section, passing "sectionContents", "stream", "sectionOffsets", and "metadata". If this returns an error, return a "format error" with "fallbackUrl".

23. Assert: "metadata" has an entry with the key "primaryUrl".

24. If "metadata" doesn't have an entry with key "requests", return a "format error" with "fallbackUrl".

25. Return "metadata".

3.3.1. Parsing the index section

The "index" section defines the set of HTTP requests in the bundle and identifies their locations in the "responses" section. It consists of a map from URL strings to arrays consisting of a "Variants" header field value ([I-D.ietf-httpbis-variants]) followed by one "location-in-responses" pair for each of the possible combinations of available-values within the "Variants" value in lexicographic (row-major) order.

For example, given a "variants-value" of "Accept-Encoding;gzip;br, Accept-Language;en;fr;ja", the list of "location-in-responses" pairs will correspond to the "VariantKey"s:

* gzip;en

* gzip;fr

* gzip;ja

* br;en

* br;fr

* br;ja

The order of variant-axes is important. If the "variants-value" were "Accept-Language;en;fr;ja, Accept-Encoding;gzip;br" instead, the "location-in-responses" pairs would instead correspond to:

* en;gzip

* en;br

* fr;gzip

* fr;br

* ja;gzip

* ja;br

As a special case, an empty "variants-value" indicates that there is only one resource at the specified URL and that no content negotiation is performed.

```
index = {* whatwg-url => [ variants-value, +location-in-responses ] }
variants-value = bstr
location-in-responses = (offset: uint, length: uint)
```

A "ResponseMetadata" struct identifies a byte range within the bundle stream, defined by an integer offset from the start of the stream and the integer number of bytes in the range.

To parse the index section, given its "sectionContents", the "sectionOffsets" map, and the "metadata" map to fill in, the parser MUST do the following:

1. Let "index" be the result of parsing "sectionContents" as a CBOR item matching the "index" rule in the above CDDL (Section 3.5). If "index" is an error, return an error.

2. Let "requests" be an initially-empty map ([INFRA]) from URLs to response descriptions, each of which is either a single "location-in-stream" value or a pair of a "Variants" header field value ([I-D.ietf-httpbis-variants]) and a map from that value's possible "Variant-Key"s to "location-in-stream" values, as described in Section 2.2.

3. Let "MakeRelativeToStream" be a function that takes a "location-in-responses" value ("offset", "length") and returns a "ResponseMetadata" struct or error by running the following sub-steps:

1. If "offset" + "length" is larger than "sectionOffsets["responses"].length", return an error.

2. Otherwise, return a "ResponseMetadata" struct whose offset is "sectionOffsets["responses"].offset" + "offset" and whose length is "length".

4. For each ("url", "responses") entry in the "index" map:

   1. Let "parsedUrl" be the result of parsing ([URL]) "url" with no base URL.

   2. If "parsedUrl" is a failure, its fragment is not null, or it includes credentials, return an error.

   3. If the first element of "responses" is the empty string:

      1. If the length of "responses" is not 3 (i.e. there is more than one "location-in-responses" in responses), return an error.

      2. Otherwise, assert that "requests"["parsedUrl"] does not exist, and set "requests"["parsedUrl"] to "MakeRelativeToStream(location-in-responses)", where "location-in-responses" is the second and third elements of "responses". If that returns an error, return an error.

   4. Otherwise:

      1. Let "variants" be the result of parsing the first element of "responses" as the value of the "Variants" HTTP header field (Section 2 of [I-D.ietf-httpbis-variants]). If this fails, return an error.

      2. Let "variantKeys" be the Cartesian product of the lists of available-values for each variant-axis in lexicographic (row-major) order. See the examples above.

      3. If the length of "responses" is not "2 * len(variantKeys) + 1", return an error.

      4. Set "requests"["parsedUrl"] to a map from "variantKeys"["i"] to the result of calling "MakeRelativeToStream" on the "location-in-responses" at "responses"["2*i+1"] and "responses"["2*i+2"], for "i" in ["0", "len(variantKeys)"). If any "MakeRelativeToStream" call returns an error, return an error.

5. Set "metadata["requests"]" to "requests".

3.3.2. Parsing the manifest section

The "manifest" section records a single URL identifying the manifest of the bundle. The URL MUST refer to the one or more response(s) contained in the bundle itself.

The bundle can contain multiple resources at this URL, and the client is expected to content-negotiate for the best one. For example, a client might select the one with an "accept" header of "application/manifest+json" ([appmanifest]) and an "accept-language" header of "es-419".

```
manifest = whatwg-url
```

To parse the manifest section, given its "sectionContents" and the "metadata" map to fill in, the parser MUST do the following:

1. Let "urlString" be the result of parsing "sectionContents" as a CBOR item matching the above "manifest" rule (Section 3.5. If "urlString" is an error, return that error.

2. Let "url" be the result of parsing ([URL]) "urlString" with no base URL.

3. If "url" is a failure, its fragment is not null, or it includes credentials, return an error.

4. Set "metadata["manifest"]" to "url".

3.3.3. Parsing the signatures section

The "signatures" section vouches for the resources in the bundle.

The section can contain as many signatures as needed, each by some authority, and each covering an arbitrary subset of the resources in the bundle. Intermediates, including attackers, can remove signatures from the bundle without breaking the other signatures.

The bundle parser’s client is responsible to determine the validity and meaning of each authority’s signatures. In particular, the algorithm below does not check that signatures are valid. For example, a client might:

* Use the ecdsa_secp256r1_sha256 algorithm defined in Section 4.2.3 of [TLS1.3] to check the validity of any signature with an EC public key on the secp256r1 curve.

* Reject all signatures by an RSA public key.

* Treat an X.509 certificate with the CanSignHttpExchanges extension (Section 4.2 of [I-D.yasskin-http-origin-signed-responses]) and a valid chain to a trusted root as an authority that vouches for the authenticity of resources claimed to come from that certificate's domains.

* Treat an X.509 certificate with another extension or EKU as vouching that a particular analysis has run over the signed resources without finding malicious behavior.

A client might also choose different behavior for those kinds of authorities and keys.

```
signatures = [
  authorities: [*authority],
  vouched-subsets: [*{
    authority: index-in-authorities,
    sig: bstr,
    signed: bstr  ; Expected to hold a signed-subset item.
  }],
]
authority = augmented-certificate
index-in-authorities = uint

signed-subset = {
  validity-url: whatwg-url,
  auth-sha256: bstr,
  date: uint,
  expires: uint,
  subset-hashes: {+
    whatwg-url => [variants-value, +resource-integrity]
  },
  * tstr => any,
}
resource-integrity = (
  header-sha256: bstr,
  payload-integrity-header: tstr
)
```

The "augmented-certificate" CDDL rule comes from Section 3.3 of [I-D.yasskin-http-origin-signed-responses].

To parse the signatures section, given its "sectionContents", the "sectionOffsets" map, and the "metadata" map to fill in, the parser MUST do the following:

1. Let "signatures" be the result of parsing "sectionContents" as a CBOR item matching the "signatures" rule in the above CDDL (Section 3.5).

2. Set "metadata["authorities"]" to the list of authorities in the first element of the "signatures" array.

3. Set "metadata["vouched-subsets"]" to the second element of the "signatures" array.

3.3.4. Parsing the critical section

The "critical" section lists sections of the bundle that the client needs to understand in order to load the bundle correctly. Other sections are assumed to be optional.

```
critical = [*tstr]
```

To parse the critical section, given its "sectionContents" and the "metadata" map to fill in, the parser MUST do the following:

1. Let "critical" be the result of parsing "sectionContents" as a CBOR item matching the above "critical" rule (Section 3.5). If "critical" is an error, return that error.

2. For each value "sectionName" in the "critical" list, if the client has not implemented sections named "sectionName", return an error.

This section does not modify the returned metadata.

3.3.5. The responses section

The responses section does not add any items to the bundle metadata map. Instead, its offset and length are used in processing the index section (Section 3.3.1).

3.3.6. Starting from the end

The length of a bundle is encoded as a big-endian integer inside a CBOR byte string at the end of the bundle.

```
+------------+-----+----+----+----+----+----+----+----+----+----+
| first byte | ... | 48 | 00 | 00 | 00 | 00 | 00 | BC | 61 | 4E |
+------------+-----+----+----+----+----+----+----+----+----+----+
            /       \
      0xBC614E-10=12345668 omitted bytes
```

Figure 1: Example trailing bytes

Parsing from the end allows the bundle to be appended to another format such as a self-extracting executable.

To implement Section 2.2.1, taking a sequence of bytes "bytes", the client MUST:

1. Let "byteStringHeader" be "bytes[bytes.length - 9]". If "byteStringHeader" is not "0x48" (the CBOR ([CBORbis]) initial byte for an 8-byte byte string), return an error.

2. Let "bundleLength" be "[bytes[bytes.length - 8], bytes[bytes.length])" (the last 8 bytes) interpreted as a big-endian integer.

3. If "bundleLength > bytes.length", return an error.

4. Let "stream" be a new stream whose:

   * Available bytes are "[bytes[bytes.length - bundleLength], bytes[bytes.length])".

   * EOF flag is set.

   * Current offset is initially 0.

   * The seek to offset N and read N bytes operations succeed immediately if "currentOffset + N <= bundleLength" and fail otherwise.

5. Return the result of running Section 3.3 with "stream" as input.

3.4. Load a response from a bundle

The result of Load a bundle’s metadata maps each URL and Variant-Key ([I-D.ietf-httpbis-variants]) to a response, which consists of headers and a payload. The headers can be loaded from the bundle’s stream before waiting for the payload, and similarly the payload can be streamed to downstream consumers.

```
response = [headers: bstr .cbor headers, payload: bstr]
```

To implement Section 2.3, the parser MUST run the following steps, taking the bundle’s "stream", a "request" ([FETCH]), and a "responseMetadata" returned by Section 2.2 .

1. Seek to offset "responseMetadata.offset" in "stream". If this fails, return an error.

2. Read 1 byte from "stream". If this is an error or isn't "0x82", return an error.

3. Let "headerLength" be the result of getting the length of a CBOR bytestring header from "stream" (Section 3.5.2). If "headerLength" is an error, return that error.

4. If "headerLength" is 524288 (512*1024) or greater, return an error.

5. Let "headerCbor" be the result of reading "headerLength" bytes from "stream" and parsing a CBOR item from them matching the "headers" CDDL rule. If either the read or parse returns an error, return that error.

6. Let ("headers", "pseudos") be the result of converting "headerCbor" to a header list and pseudoheaders using the algorithm in Section 3.6. If this returns an error, return that error.

7. If "pseudos" does not have a key named ':status' or its size isn't 1, return an error.

8. If "pseudos[':status']" isn't exactly 3 ASCII decimal digits, return an error.

9. Let "payloadLength" be the result of getting the length of a CBOR bytestring header from "stream" (Section 3.5.2). If "payloadLength" is an error, return that error.

10. If "payloadLength" is greater than 0 and "headers" does not contain a "Content-Type" header, return an error.

    The client MUST interpret the following payload as this specified media type instead of trying to sniff a media type from the bytes of the payload, for example by appending an artificial "X-Content-Type-Options: nosniff" header field ([FETCH]) to "headers".

11. If "stream.currentOffset + payloadLength != responseMetadata.offset + responseMetadata.length", return an error.

12. Let "body" be a new body ([FETCH]) whose stream is a tee'd copy of "stream" starting at the current offset and ending after "payloadLength" bytes.

    TODO: Add the rest of the details of creating a "ReadableStream" object.

13. Let "response" be a new response ([FETCH]) whose:

    * Url list is "request"'s url list,

    * status is "pseudos[':status']",

    * header list is "headers", and

    * body is "body".

14. Return "response".

3.5. Parsing CBOR items

Parsing a bundle involves parsing many CBOR items. All of these items need to be deterministically encoded.

3.5.1. Parse a known-length item

To parse a CBOR item ([CBORbis]), optionally matching a CDDL rule ([CDDL]), from a sequence of bytes, "bytes", the parser MUST do the following:

1. If "bytes" are not a well-formed CBOR item, return an error.

2. If "bytes" does not satisfy the core deterministic encoding requirements from Section 4.2.1 of [CBORbis], return an error. This format does not use floating point values or tags, so this specification does not add any deterministic encoding rules for them.

3. If "bytes" includes extra bytes after the encoding of a CBOR item, return an error.

4. Let "item" be the result of decoding "bytes" as a CBOR item.

5. If a CDDL rule was specified, but "item" does not match it, return an error.

6. Return "item".

3.5.2. Parsing variable-length data from a bytestring

Bundles encode variable-length data in CBOR bytestrings, which are prefixed with their length. This algorithm returns the number of bytes in the variable-length item and sets the stream's current offset to the first byte of the contents.

To get the length of a CBOR bytestring header from a bundle's stream, the parser MUST do the following:

1. Let ("type", "argument") be the result of parsing the type and argument of a CBOR item from the stream (Section 3.5.3). If this returns an error, return that error.

2. If "type" is not "2", the item is not a bytestring. Return an error.

3. Return "argument".

3.5.3. Parsing the type and argument of a CBOR item

To parse the type and argument of a CBOR item from a bundle's stream, the parser MUST do the following. This algorithm returns a pair of the CBOR major type 0-7 inclusive, and a 64-bit integral argument for the CBOR item:

1. Let "firstByte" be the result of reading 1 byte from the stream. If "firstByte" is an error, return that error.

2. Let "type" be "(firstByte & 0xE0) / 0x20".

3. If "firstByte & 0x1F" is:

   0..23, inclusive Return ("type", "firstByte & 0x1F").

   24 Let "content" be the result of reading 1 byte from the stream. If "content" is an error or is less than 24, return an error.

   25 Let "content" be the result of reading 2 bytes from the stream. If "content" is an error or its first byte is 0, return an error.

   26 Let "content" be the result of reading 4 bytes from the stream. If "content" is an error or its first two bytes are 0, return an error.

   27 Let "content" be the result of reading 8 bytes from the

stream. If "content" is an error or its first four bytes are 0, return an error.

28..31, inclusive Return an error. Note: This intentionally does not support indefinite-length items.

4. Let "argument" be the big-endian integer encoded in "content".

5. Return ("type", "argument").

3.6. Interpreting CBOR HTTP headers

Bundles represent HTTP requests and responses as a list of headers, matching the following CDDL ([CDDL]):

```
headers = {* bstr => bstr}
```

Pseudo-headers starting with a ":" provide the non-header information needed to create a request or response as appropriate

To convert a CBOR item "item" into a [FETCH] header list and pseudoheaders, parsers MUST do the following:

1. If "item" doesn't match the "headers" rule in the above CDDL, return an error.

2. Let "headers" be a new header list ([FETCH]).

3. Let "pseudos" be an empty map ([INFRA]).

4. For each pair ("name", "value") in "item":

    1. If "name" contains any upper-case or non-ASCII characters, return an error. This matches the requirement in Section 8.1.2 of [RFC7540].

    2. If "name" starts with a ':':

        1. Assert: "pseudos[name]" does not exist, because CBOR maps cannot contain duplicate keys.

        2. Set "pseudos[name]" to "value".

        3. Continue.

    3. If "name" or "value" doesn't satisfy the requirements for a header in [FETCH], return an error.

4. Assert: "headers" does not contain ([FETCH]) "name", because CBOR maps cannot contain duplicate keys and an earlier step rejected upper-case bytes.

   Note: This means that a response cannot set more than one cookie, because the "Set-Cookie" header ([RFC6265]) has to appear multiple times to set multiple cookies.

5. Append ("name", "value") to "headers".

5. Return ("headers", "pseudos").

4. Guidelines for bundle authors

Bundles SHOULD consist of a single CBOR item satisfying the core deterministic encoding requirements (Section 3.5) and matching the "webbundle" CDDL rule in Section 3.1.

5. Security Considerations

5.1. Version skew

Bundles currently have no mechanism for ensuring that the signed exchanges they contain constitute a consistent version of those resources. Even if a website never has a security vulnerability when resources are fetched at a single time, an attacker might be able to combine a set of resources pulled from different versions of the website to build a vulnerable site. While the vulnerable site could have occurred by chance on a client's machine due to normal HTTP caching, bundling allows an attacker to guarantee that it happens. Future work in this specification might allow a bundle to constrain its resources to come from a consistent version.

5.2. Content sniffing

While modern browsers tend to trust the "Content-Type" header sent with a resource, especially when accompanied by "X-Content-Type-Options: nosniff", plugins will sometimes search for executable content buried inside a resource and execute it in the context of the origin that served the resource, leading to XSS vulnerabilities. For example, some PDF reader plugins look for "%PDF" anywhere in the first 1kB and execute the code that follows it.

The "application/webbundle" format defined above includes URLs and request headers early in the format, which an attacker could use to cause these plugins to sniff a bad content type.

To avoid vulnerabilities, in addition to the response header requirements in Section 3.2, servers are advised to only serve an "application/webbundle" resource from a domain if it would also be safe for that domain to serve the bundle's content directly, and to follow at least one of the following strategies:

1. Only serve bundles from dedicated domains that don't have access to sensitive cookies or user storage.

2. Generate bundles "offline", that is, in response to a trusted author submitting content or existing signatures reaching a certain age, rather than in response to untrusted-reader queries.

3. Do all of:

    1. If the bundle's contained URLs (e.g. in the manifest and index) are derived from the request for the bundle, percent-encode (https://url.spec.whatwg.org/#percent-encode) ([URL]) any bytes that are greater than 0x7E or are not URL code points (https://url.spec.whatwg.org/#url-code-points) ([URL]) in these URLs. It is particularly important to make sure no unescaped nulls (0x00) or angle brackets (0x3C and 0x3E) appear.

    2. Similarly, if the request headers for any contained resource are based on the headers sent while requesting the bundle, only include request header field names *and values* that appear in a static allowlist. Keep the set of allowed request header fields smaller than 24 elements to prevent attackers from controlling a whole CBOR length byte.

    3. Restrict the number of items a request can direct the server to include in a bundle to less than 12, again to prevent attackers from controlling a whole CBOR length byte.

    4. Do not reflect request header fields into the set of response headers.

If the server serves responses that are written by a potential attacker but then escaped, the "application/webbundle" format allows the attacker to use the length of the response to control a few bytes before the start of the response. Any existing mechanisms that prevent polyglot documents probably keep working in the face of this new attack, but we don't have a guarantee of that.

To encourage servers to include the "X-Content-Type-Options: nosniff" header field, clients SHOULD reject bundles served without it.

6. IANA considerations

6.1. Internet Media Type Registration

IANA is requested to register the MIME media type ([IANA.media-types]) for web bundles, application/webbundle, as follows:

* Type name: application

* Subtype name: webbundle

* Required parameters:

  - v: A string denoting the version of the file format. ([RFC5234] ABNF: "version = 1*(DIGIT/%x61-7A)") The version defined in this specification is "1".

    Note: RFC EDITOR PLEASE DELETE THIS NOTE; Implementations of drafts of this specification MUST NOT use simple integers to describe their versions, and MUST instead define implementation-specific strings to identify which draft is implemented.

* Optional parameters: N/A

* Encoding considerations: binary

* Security considerations: See Section 5 of this document.

* Interoperability considerations: N/A

* Published specification: This document

* Applications that use this media type: None yet, but it is expected that web browsers will use this format.

* Fragment identifier considerations: N/A

* Additional information:

  - Deprecated alias names for this type: N/A

  - Magic number(s): 86 48 F0 9F 8C 90 F0 9F 93 A6

  - File extension(s): .wbn

  - Macintosh file type code(s): N/A

* Person & email address to contact for further information: See the Author's Address section of this specification.

* Intended usage: COMMON

* Restrictions on usage: N/A

* Author: See the Author's Address section of this specification.

* Change controller: The IESG iesg@ietf.org (mailto:iesg@ietf.org)

* Provisional registration? Yes.

6.2. Web Bundle Section Name Registry

IANA is directed to create a new registry with the following attributes:

Name: Web Bundle Section Names

Review Process: Specification Required

Initial Assignments:

Section Name	Specification	Metadata	Metadata Fields
"index"	Section 3.3.1	Yes	"requests"
"manifest"	Section 3.3.2	Yes	"manifest"
"signatures"	Section 3.3.3	Yes	"authorities", "vouched-subsets"
"critical"	Section 3.3.4	Yes	
"responses"	Section 3.3.5	No	

Table 1

Requirements on new assignments:

Section Names MUST be encoded in UTF-8.

Assignments must specify whether the section is parsed during Load a bundle's metadata (Metadata=Yes) or not (Metadata=No).

The section’s specification can use the bytes making up the section, the bundle’s stream (Section 2.1), and the "sectionOffsets" map (Section 3.3), as input, and MUST say if an error is returned, and otherwise what items, if any, are added to the map that Section 3.3 returns. A section’s specification MAY say that, if it is present, another section is not processed.

7. References

7.1. Normative References

[appmanifest]
Caceres, M., Christiansen, K., Lamouri, M., Kostiainen, A., Dolin, R., and M. Giuca, "Web App Manifest", World Wide Web Consortium WD WD-appmanifest-20180523, 23 May 2018, <https://www.w3.org/TR/2018/WD-appmanifest-20180523>.

[CBORbis] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", Work in Progress, Internet-Draft, draft-ietf-cbor-7049bis-14, 16 June 2020, <http://www.ietf.org/internet-drafts/draft-ietf-cbor-7049bis-14.txt>.

[CDDL] Birkholz, H., Vigano, C., and C. Bormann, "Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures", RFC 8610, DOI 10.17487/RFC8610, June 2019, <https://www.rfc-editor.org/info/rfc8610>.

[FETCH] WHATWG, "Fetch", July 2020, <https://fetch.spec.whatwg.org/>.

[I-D.ietf-httpbis-variants]
Nottingham, M., "HTTP Representation Variants", Work in Progress, Internet-Draft, draft-ietf-httpbis-variants-06, 3 November 2019, <http://www.ietf.org/internet-drafts/draft-ietf-httpbis-variants-06.txt>.

[I-D.yasskin-http-origin-signed-responses]
Yasskin, J., "Signed HTTP Exchanges", Work in Progress, Internet-Draft, draft-yasskin-http-origin-signed-responses-08, 4 November 2019, <http://www.ietf.org/internet-drafts/draft-yasskin-http-origin-signed-responses-08.txt>.

[IANA.media-types]
IANA, "Media Types", <http://www.iana.org/assignments/media-types>.

[INFRA] WHATWG, "Infra", July 2020, <https://infra.spec.whatwg.org/>.

[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>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/info/rfc5234>.

[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>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[URL] WHATWG, "URL", July 2020, <https://url.spec.whatwg.org/>.

7.2. Informative References

[I-D.yasskin-wpack-use-cases]
Yasskin, J., "Use Cases and Requirements for Web Packages", Work in Progress, Internet-Draft, draft-yasskin-wpack-use-cases-00, 30 October 2019, <http://www.ietf.org/internet-drafts/draft-yasskin-wpack-use-cases-00.txt>.

[RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, DOI 10.17487/RFC6265, April 2011, <https://www.rfc-editor.org/info/rfc6265>.

[TLS1.3] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

Appendix A. Change Log

RFC EDITOR PLEASE DELETE THIS SECTION.

draft-03

* Make the manifest optional.

* Update the reference to draft-yasskin-wpack-use-cases.

* Retitle to "web bundles".

draft-02

* Fix the initial bytes of the format.

* Allow empty responses to omit their content type.

* Provisionally register application/webbundle.

draft-01

* Include only section lengths in the section index, requiring sections to be listed in order.

* Have the "index" section map URLs to sets of responses negotiated using the Variants system ([I-D.ietf-httpbis-variants]).

* Require the "manifest" to be embedded into the bundle.

* Add a content sniffing security consideration.

* Add a version string to the format and its mime type.

* Add a fallback URL in a fixed location in the format, and use that fallback URL as the primary URL of the bundle.

* Add a "signatures" section to let authorities (like domain-trusted X.509 certificates) vouch for subsets of a bundle.

* Use the CBORbis "deterministic encoding" requirements instead of "canonicalization" requirements.

Appendix B. Acknowledgements

Thanks to the Chrome loading team, especially Kinuko Yasuda and Kouhei Ueno for making the format work well when streamed.

Author's Address

Jeffrey Yasskin
Google

Email: jyasskin@chromium.org

Network Working Group   J. Yasskin
Internet-Draft   Google
Intended status: Informational   27 July 2020
Expires: 28 January 2021

Use Cases and Requirements for Web Packages
draft-yasskin-wpack-use-cases-01

Abstract

This document lists use cases for signing and/or bundling collections of web pages, and extracts a set of requirements from them.

Note to Readers

Discussion of this draft takes place on the ART area mailing list (art@ietf.org), which is archived at https://mailarchive.ietf.org/arch/search/?email_list=art (https://mailarchive.ietf.org/arch/search/?email_list=art).

The source code and issues list for this draft can be found in https://github.com/WICG/webpackage (https://github.com/WICG/webpackage).

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction 3
2. Use cases 4
2.1. Essential 4
2.1.1. Offline installation 4
2.1.2. Offline browsing 6
2.1.3. Save and share a web page 6
2.1.4. Privacy-preserving prefetch 7
2.2. Nice-to-have 7
2.2.1. Packaged Web Publications 8
2.2.2. Avoiding Censorship 9
2.2.3. Third-party security review 9
2.2.4. Building packages from multiple libraries 10
2.2.5. Cross-CDN Serving 10
2.2.6. Pre-installed applications 11
2.2.7. Protecting Users from a Compromised Frontend 12
2.2.8. Installation from a self-extracting executable 13
2.2.9. Packages in version control 13
2.2.10. Subresource bundling 13
2.2.11. Archival 14
3. Requirements 15
3.1. Essential 15
3.1.1. Indexed by URL 15
3.1.2. Request headers 15
3.1.3. Response headers 15
3.1.4. Signing as an origin 15
3.1.5. Random access 16
3.1.6. Resources from multiple origins in a package 16
3.1.7. Cryptographic agility 16
3.1.8. Unsigned content 16
3.1.9. Certificate revocation 16
3.1.10. Downgrade prevention 16
3.1.11. Metadata 17
3.1.12. Implementations are hard to get wrong 17
3.2. Nice to have 17
3.2.1. Streamed loading 17
3.2.2. Signing without origin trust 17
3.2.3. Additional signatures 17

3.2.4. Binary 18
3.2.5. Deduplication of diamond dependencies 18
3.2.6. Old crypto can be removed 18
3.2.7. Compress transfers 18
3.2.8. Compress stored packages 18
3.2.9. Subsetting and reordering 18
3.2.10. Packaged validity information 18
3.2.11. Signing uses existing TLS certificates 18
3.2.12. External dependencies 19
3.2.13. Trailing length 19
3.2.14. Time-shifting execution 19
3.2.15. Service Worker integration 19
4. Non-goals 19
4.1. Store confidential data 19
4.2. Generate packages on the fly 20
4.3. Non-origin identity 20
4.4. DRM 20
4.5. Ergonomic replacement for HTTP/2 PUSH 20
5. Security Considerations 21
6. IANA Considerations 21
7. Informative References 21
Appendix A. Acknowledgements 23
Author's Address 23

1. Introduction

People would like to use content offline and in other situations where there isn't a direct connection to the server where the content originates. However, it's difficult to distribute and verify the authenticity of applications and content without a connection to the network. The W3C has addressed running applications offline with Service Workers ([ServiceWorkers]), but not the problem of distribution.

Previous attempts at packaging web resources (e.g. Resource Packages (https://www.mnot.net/blog/2010/02/18/resource_packages) and the W3C TAG's packaging proposal (https://w3ctag.github.io/packaging-on-the-web/)) were motivated by speeding up the download of resources from a single server, which is probably better achieved through other mechanisms like HTTP/2 PUSH, possibly augmented with a simple manifest of URLs a page plans to use (https://lists.w3.org/Archives/Public/public-web-perf/2015Jan/0038.html). This attempt is instead motivated by avoiding a connection to the origin server at all. It may still be useful for the earlier use cases, so they're still listed, but they're not primary.

2. Use cases

These use cases are in rough descending priority order. If use cases have conflicting requirements, the design should enable more important use cases.

2.1. Essential

2.1.1. Offline installation

Alex can download a file containing a website (a PWA (https://developers.google.com/web/progressive-web-apps/checklist)) including a Service Worker from origin "O", and transmit it to their peer Bailey, and then Bailey can install the Service Worker with a proof that it came from "O". This saves Bailey the bandwidth costs of transferring the website.

There are roughly two ways to accomplish this:

1. Package just the Service Worker Javascript and any other Javascript that it importScripts() (https://w3c.github.io/ServiceWorker/#importscripts), with their URLs and enough metadata to synthesize a navigator.serviceWorker.register(scriptURL, options) call (https://w3c.github.io/ServiceWorker/#navigator-service-worker-register), along with an uninterpreted but signature-checked blob of data that the Service Worker can interpret to fill in its caches.

2. Package the resources so that the Service Worker can fetch() them to populate its cache.

Associated requirements for just the Service Worker:

* Indexed by URL: The "register()" and "importScripts()" calls have semantics that depend on the URL.

* Signing as an origin: To prove that the file came from "O".

* Signing uses existing TLS certificates: So "O" doesn’t have to spend lots of money buying a specialized certificate.

* Cryptographic agility: Today’s algorithms will eventually be obsolete and will need to be replaced.

* Certificate revocation: "O"’s certificate might be compromised or mis-issued, and the attacker shouldn’t then get an infinite ability to mint packages.

* Downgrade prevention: "O"'s site might have an XSS vulnerability, and attackers with an old signed package shouldn't be able to take advantage of the XSS forever.

* Metadata: Just enough to generate the "register()" call, which is less than a full W3C Application Manifest.

Additional associated requirements for packaged resources:

* Indexed by URL: Resources on the web are addressed by URL.

* Request headers: If Bailey's running a different browser from Alex or has a different language configured, the "accept*" headers are important for selecting which resource to use at each URL.

* Response headers: The meaning of a resource is heavily influenced by its HTTP response headers.

* Resources from multiple origins in a package: So the site can be built from multiple components (Section 2.2.4).

* Metadata: The browser needs to know which resource within a package file to treat as its Service Worker and/or initial HTML page.

2.1.1.1. Online use

Bailey may have an internet connection through which they can, in real time, fetch updates to the package they received from Alex.

2.1.1.2. Fully offline use

Or Bailey may not have any internet connection a significant fraction of the time, either because they have no internet at all, because they turn off internet except when intentionally downloading content, or because they use up their plan partway through each month.

Associated requirements beyond Offline installation:

* Packaged validity information: Even without a direct internet connection, Bailey should be able to check that their package is still valid.

2.1.2. Offline browsing

Alex can download a file containing a large website (e.g. Wikipedia) from its origin, save it to transferrable storage (e.g. an SD card), and hand it to their peer Bailey. Then Bailey can browse the website with a proof that it came from "O". Bailey may not have the storage space to copy the website before browsing it.

This use case is harder for publishers to support if we specialize Section 2.1.1 for Service Workers since it requires the publisher to adopt Service Workers before they can sign their site.

Associated requirements beyond Offline installation:

* Random access: To avoid needing a long linear scan before using the content.

* Compress stored packages: So that more content can fit on the same storage device.

2.1.3. Save and share a web page

Casey is viewing a web page and wants to save it either for offline use or to show it to their friend Dakota. Since Casey isn't the web page's publisher, they don't have the private key needed to sign the page. Browsers currently allow their users to save pages, but each browser uses a different format (MHTML, Web Archive, or files in a directory), so Dakota and Casey would need to be using the same browser. Casey could also take a screenshot, at the cost of losing links and accessibility.

Associated requirements:

* Unsigned content: A client can't sign content as another origin.

* Resources from multiple origins in a package: General web pages include resources from multiple origins.

* Indexed by URL: Resources on the web are addressed by URL.

* Response headers: The meaning of a resource is heavily influenced by its HTTP response headers.

2.1.4. Privacy-preserving prefetch

Lots of websites link to other websites. Many of these source sites would like the targets of these links to load quickly. The source could use "<link rel="prefetch">" to prefetch the target of a link, but if the user doesn't actually click that link, that leaks the fact that the user saw a page that linked to the target. This can be true even if the prefetch is made without browser credentials because of mechanisms like TLS session IDs.

Because clients have limited data budgets to prefetch link targets, this use case is probably limited to sites that can accurately predict which link their users are most likely to click. For example, search engines can predict that their users will click one of the first couple results, and news aggreggation sites like Reddit or Slashdot can hope that users will read the article if they've navigated to its discussion.

Two search engines have built systems to do this with today's technology: Google's AMP (https://www.ampproject.org/) and Baidu's MIP (https://www.mipengine.org/) formats and caches allow them to prefetch search results while preserving privacy, at the cost of showing the wrong URLs for the results once the user has clicked. A good solution to this problem would show the right URLs but still avoid a request to the publishing origin until after the user clicks.

Associated requirements:

* Signing as an origin: To prove the content came from the original origin.

* Streamed loading: If the user clicks before the target page is fully transferred, the browser should be able to start loading early parts before the source site finishes sending the whole page.

* Compress transfers

* Subsetting and reordering: If a prefetched page includes subresources, its publisher might want to provide and sign both WebP and PNG versions of an image, but the source site should be able to transfer only best one for each client.

2.2. Nice-to-have

2.2.1. Packaged Web Publications

The W3C's Publishing Working Group (https://www.w3.org/publishing/groups/publ-wg/), merged from the International Digital Publishing Forum (IDPF) and in charge of EPUB maintenance, wants to be able to create publications on the web and then let them be copied to different servers or to other users via arbitrary protocols. See their Packaged Web Publications use cases (https://www.w3.org/TR/pwp-ucr/#pwp) for more details.

Associated requirements:

* Indexed by URL: Resources on the web are addressed by URL.

* Signing as an origin: So that readers can be sure their copy is authentic and so that copying the package preserves the URLs of the content inside it.

* Downgrade prevention: An early version of a publication might contain incorrect content, and a publisher should be able to update that without worrying that an attacker can still show the old content to users.

* Metadata: A publication can have copyright and licensing concerns; a title, author, and cover image; an ISBN or DOI name; etc.; which should be included when that publication is packaged.

Other requirements are similar to those from Offline installation:

* Random access: To avoid needing a long linear scan before using the content.

* Compress stored packages: So that more content can fit on the same storage device.

* Request headers: If different users' browsers have different capabilities or preferences, the "accept*" headers are important for selecting which resource to use at each URL.

* Response headers: The meaning of a resource is heavily influenced by its HTTP response headers.

* Signing uses existing TLS certificates: So a publisher doesn't have to spend lots of money buying a specialized certificate.

* Cryptographic agility: Today's algorithms will eventually be obsolete and will need to be replaced.

* Certificate revocation: The publisher's certificate might be compromised or mis-issued, and an attacker shouldn't then get an infinite ability to mint packages.

2.2.2. Avoiding Censorship

Some users want to retrieve resources that their governments or network providers don't want them to see. Right now, it's straightforward for someone in a privileged network position to block access to particular hosts, but TLS makes it difficult to block access to particular resources on those hosts.

Today it's straightforward to retrieve blocked content from a third party, but there's no guarantee that the third-party has sent the user an accurate representation of the content: the user has to trust the third party.

With signed web packages, the user can re-gain assurance that the content is authentic, while still bypassing the censorship. Packages don't do anything to help discover this content.

Systems that make censorship more difficult can also make legitimate content filtering more difficult. Because the client that processes a web package always knows the true URL, this forces content filtering to happen on the client instead of on the network.

Associated requirements:

* Indexed by URL: So the user can see that they're getting the content they expected.

* Signing as an origin: So that readers can be sure their copy is authentic and so that copying the package preserves the URLs of the content inside it.

2.2.3. Third-party security review

Some users may want to grant certain permissions only to applications that have been reviewed for security by a trusted third party. These third parties could provide guarantees similar to those provided by the iOS, Android, or Chrome OS app stores, which might allow browsers to offer more powerful capabilities than have been deemed safe for unaudited websites.

Binary transparency for websites is similar: like with Certificate Transparency [RFC6962], the transparency logs would sign the content of the package to provide assurance that experts had a chance to audit the exact package a client received.

Associated requirements:

* Additional signatures

2.2.4. Building packages from multiple libraries

Large programs are built from smaller components. In the case of the web, components can be included either as Javascript files or as "<iframe>"d subresources. In the first case, the packager could copy the JS files to their own origin; but in the second, it may be important for the "<iframe>"d resources to be able to make same-origin (https://html.spec.whatwg.org/multipage/origin.html#same-origin) requests back to their own origin, for example to implement federated sign-in.

Associated requirements:

* Resources from multiple origins in a package: Each component may come from its own origin.

* Deduplication of diamond dependencies: If we have dependencies A->B->D and A->C->D, it's important that a request for a D resource resolves to a single resource that both B and C can handle.

2.2.4.1. Shared libraries

In ecosystems like Electron (https://electron.atom.io/) and Node (https://nodejs.org/en/), many packages may share some common dependencies. The cost of downloading each package can be greatly reduced if the package can merely point at other dependencies to download instead of including them all inline.

Associated requirements:

* External dependencies

2.2.5. Cross-CDN Serving

When a web page has subresources from a different origin, retrieval of those subresources can be optimized if they're transferred over the same connection as the main resource. If both origins are distributed by the same CDN, in-progress mechanisms like [I-D.ietf-httpbis-http2-secondary-certs] allow the server to use a single connection to send both resources, but if the resource and subresource don't share a CDN or don't use a CDN at all, existing mechanisms don't help.

If the subresource is signed by its publisher, the main resource's server can forward it to the client.

There are some yet-to-be-solved privacy problems if the client and server want to avoid transferring subresources that are already in the client's cache: naively telling the server that a resource is already present is a privacy leak.

Associated requirements:

* Streamed loading: To get optimal performance, the browser should be able to start loading early parts of a resource before the distributor finishes sending the whole resource.

* Signing as an origin: To prove the content came from the original origin.

* Compress transfers

2.2.6. Pre-installed applications

Device manufacturers would like to ship their devices with some web applications pre-installed and usable even if the application is first used without an internet connection. Thereafter, the application should use the normal Service Worker update mechanism to stay up to date.

One way to accomplish this would be to pre-create a browser profile in the device's default browser and navigate it to each of the pre-installed apps before recording the device image. However, this means end-users miss the browser's initial setup flow and possibly that any "unique" cookies the sites set are now shared across everyone who bought the device. It also doesn't help users who change their default browser.

If multiple browsers supported an unsigned web package format, with an option to trust it as if it were signed if it's in a particular section of the filesystem that's as protected as the browser's executable, and if registering a Service Worker from a page inside a package passed the full package contents to the Service Worker's "install" event, the device manufacturer could provide web packages for each pre-installed application that would work in the user's chosen browser.

Associated requirements:

* Service Worker integration: To pass the package into the "install" event and from there get its contents into a "Cache".

2.2.7. Protecting Users from a Compromised Frontend

If an attacker gains control over a frontend server, any user who visits that server while they have control can have their web app upgraded to a hostile version. On the other hand, native applications either control their own update process or delegate it to an app store, which allows them to protect users by requiring that updates are signed by a trusted key. This protection isn't perfect--it's a Trust-On-First-Use mechanism that doesn't protect users who first install the application while the attacker controls the server they get it from, and attackers can bypass it by compromising the app's build system--but since both of those risks also apply to web apps, it does make the attack surface for native applications smaller than for web apps.

Not all application developers should choose to require signed updates, since doing so adds the risk of losing the signing key, but having this option gives security-sensitive applications like Dashlane (https://app.dashlane.com/) an incentive to build native apps instead of web apps.

It has been difficult to add a signature requirement for web app upgrades because we haven't had a way to sign web resources. Web Packaging is expected to provide that, so we'll be able to consider the best way to do it.

Both HTTP Strict Transport Security (HSTS, [RFC6797]) and HTTP Public Key Pinning (HPKP, [RFC7469]) have established ways to pin assertions about a site's security for a bounded time after a visit. We could do the same with a web app's signing key.

Note that HPKP has been turned off in Chromium (https://groups.google.com/a/chromium.org/d/topic/blink-dev/he9tr7p3rZ8/discussion) because it was difficult to use and made it too easy to "brick" a website. To reduce the chance of bricking the website, this key pinning design could require an active Service Worker before enforcing the pins. It could also avoid the need for users to take manual action to recover from a lost signing key by allowing a new key to be used if it's seen consistently for a site-chosen amount of time, instead of waiting for the whole pin to expire. However, these mitigations don't guarantee that browsers would find the tradeoffs more acceptable than they did for HPKP.

One can think of a CDN as a potentially-compromised frontend and use this mechanism to limit the damage it can cause. However, this doesn't make it safe to use a wholly-untrustworthy CDN because of the risk to first-time users.

Associated requirements:

* Signing without origin trust: To let a backend system vouch for the content. This would likely be augmented with origin trust by receiving the signed content over TLS.

* Streamed loading: To get optimal performance, the browser should be able to start loading early parts of a resource before the server finishes sending the whole resource.

2.2.8. Installation from a self-extracting executable

The Node and Electron communities would like to install packages using self-extracting executables. The traditional way to design a self-extracting executable is to concatenate the package to the end of the executable, have the executable look for a length at its own end, and seek backwards from there for the start of the package.

Associated requirements:

* Trailing length

2.2.9. Packages in version control

Once packages are generated, they should be stored in version control. Many popular VC systems auto-detect text files in order to "fix" their line endings. If the first bytes of a package look like text, while later bytes store binary data, VC may break the package.

Associated requirements:

* Binary

2.2.10. Subresource bundling

Text based subresources often benefit from improved compression ratios when bundled together.

At the same time, the current practice of JS and CSS bundling, by compiling everything into a single JS file, also has negative side-effects:

1. Dependent execution - in order to start executing _any_ of the bundled resources, it is required to download, parse and execute _all_ of them.

2. Loss of caching granularity - Modification of _any_ of the resources results in caching invalidation of _all_ of them.

3. Loss of module semantics - ES6 modules must be delivered as independent resources. Therefore, current bundling methods, which deliver them with other resources under a common URL, require transpilation to ES5 and result in loss of ES6 module semantics.

An on-the-fly readable packaging format, that will enable resources to maintain their own URLs while being physically delivered with other resources, can resolve the above downsides while keeping the upsides of improved compression ratios.

To improve cache granularity, the client needs to tell the server which versions of which resources are already cached, which it could do with a Service Worker or perhaps with [I-D.ietf-httpbis-cache-digest].

Associated requirements:

* Indexed by URL

* Streamed loading: To solve downside 1.

* Compress transfers: To keep the upside.

* Response headers: At least the Content-Type is needed to load JS and CSS.

* Unsigned content: Signing same-origin content wastes space.

2.2.11. Archival

Existing formats like WARC ([ISO28500]) do a good job of accurately representing the state of a web server at a particular time, but a browser can't currently use them to give a person the experience of that website at the time it was archived. It's not obvious to the author of this draft that a new packaging format is likely to improve on WARC, compared to, for example, implementing support for WARC in browsers, but folks who know about archiving seem interested, e.g.: https://twitter.com/anjacks0n/status/950861384266416134 (https://twitter.com/anjacks0n/status/950861384266416134).

Because of the time scales involved in archival, any signatures from the original host would likely not be trusted anymore by the time the archive is viewed, so implementations would need to sandbox the content instead of running it on the original origin.

Associated requirements:

* Indexed by URL

* Response headers: To accurately record the server's response.

* Unsigned content: To deal with expired signatures.

* Time-shifting execution

3. Requirements

3.1. Essential

3.1.1. Indexed by URL

Resources should be keyed by URLs, matching how browsers look resources up over HTTP.

3.1.2. Request headers

Resource keys should include request headers like "accept" and "accept-language", which allows content-negotiated resources to be represented.

This would require an extension to [MHTML], which uses the "content-location" response header to encode the requested URL, but has no way to encode other request headers. MHTML also has no instructions for handling multiple resources with the same "content-location".

This also requires an extension to [ZIP]: we'd need to encode the request headers into ZIP's filename fields.

3.1.3. Response headers

Resources should include their HTTP response headers, like "content-type", "content-encoding", "expires", "content-security-policy", etc.

This requires an extension to [ZIP]: we'd need something like [JAR]'s "META-INF" directory to hold extra metadata beyond the resource's body.

3.1.4. Signing as an origin

Resources within a package are provably from an entity with the ability to serve HTTPS requests for those resources' origin [RFC6454].

Note that previous attempts to sign HTTP messages ([I-D.thomson-http-content-signature], [I-D.burke-content-signature], and [I-D.cavage-http-signatures]) omit a description of how a client should use a signature to prove that a resource comes from a particular origin, and they're probably not usable for that purpose.

This would require an extension to the [ZIP] format, similar to [JAR]'s signatures.

In any cryptographic system, the specification is responsible to make correct implementations easier to deploy than incorrect implementations (Section 3.1.12).

3.1.5. Random access

When a package is stored on disk, the browser can access arbitrary resources without a linear scan.

[MHTML] would need to be extended with an index of the byte offsets of each contained resource.

3.1.6. Resources from multiple origins in a package

A package from origin "A" can contain resources from origin "B" authenticated at the same level as those from "A".

3.1.7. Cryptographic agility

Obsolete cryptographic algorithms can be replaced.

Planning to upgrade the cryptography also means we should include some way to know when it's safe to remove old cryptography (Section 3.2.6).

3.1.8. Unsigned content

Alex can create their own package without a CA-signed certificate, and Bailey can view the content of the package.

3.1.9. Certificate revocation

When a package is signed by a revoked certificate, online browsers can detect this reasonably quickly.

3.1.10. Downgrade prevention

Attackers can't cause a browser to trust an older, vulnerable version of a package after the browser has seen a newer version.

3.1.11. Metadata

Metadata like that found in the W3C's Application Manifest [W3C.WD-appmanifest-20170828] can help a client know how to load and display a package.

3.1.12. Implementations are hard to get wrong

The design should incorporate aspects that tend to cause incorrect implementations to get noticed quickly, and avoid aspects that are easy to implement incorrectly. For example:

* Explicitly specifying a cryptographic algorithm identifier in [RFC7515] made it easy for implementations to trust that algorithm, which caused vulnerabilities (https://paragonie.com/blog/2017/03/jwt-json-web-tokens-is-bad-standard-that-everyone-should-avoid).

* [ZIP]'s duplicate specification of filenames makes it easy for implementations to check the signature of one copy but use the other (https://nakedsecurity.sophos.com/2013/07/10/anatomy-of-a-security-hole-googles-android-master-key-debacle-explained/).

* Following Langley's Law (https://blog.gerv.net/2016/09/introducing-deliberate-protocol-errors-langleys-law/) when possible makes it hard to deploy incorrect implementations.

3.2. Nice to have

3.2.1. Streamed loading

The browser can load a package as it downloads.

This conflicts with ZIP, since ZIP's index is at the end.

3.2.2. Signing without origin trust

It's possible to sign a resource with a key that has some effect on trust other than asserting that the origin's owner vouches for it. These keys could be expressed as raw public keys or as certificates with other key usages.

3.2.3. Additional signatures

Third-parties can vouch for packages by signing them.

3.2.4. Binary

The format is identified as binary by tools that might try to "fix" line endings.

This conflicts with using an [MHTML]-based format.

3.2.5. Deduplication of diamond dependencies

Nested packages that have multiple dependency routes to the same sub-package, can be transmitted and stored with only one copy of that sub-package.

3.2.6. Old crypto can be removed

The ecosystem can identify when an obsolete cryptographic algorithm is no longer needed and can be removed.

3.2.7. Compress transfers

Transferring a package over the network takes as few bytes as possible. This is an easier problem than Compress stored packages since it doesn’t have to preserve Random access.

3.2.8. Compress stored packages

Storing a package on disk takes as few bytes as possible.

3.2.9. Subsetting and reordering

Resources can be removed from and reordered within a package, without breaking signatures (Section 3.1.4).

3.2.10. Packaged validity information

Certificate revocation and Downgrade prevention information can itself be packaged or included in other packages.

3.2.11. Signing uses existing TLS certificates

A "normal" TLS certificate can be used for signing packages. Avoiding extra requirements like "code signing" certificates makes packaging more accessible to all sites.

3.2.12. External dependencies

Sub-packages can be "external" to the main package, meaning the browser will need to either fetch them separately or already have them. (#35, App Installer Story (https://github.com/WICG/webpackage/issues/35))

3.2.13. Trailing length

The package’s length in bytes appears a fixed offset from the end of the package.

This conflicts with [MHTML].

3.2.14. Time-shifting execution

In some unsigned packages, Javascript time-telling functions should return the timestamp of the package, rather than the true current time.

We should explore if this has security implications.

3.2.15. Service Worker integration

When a web page inside a package registers a Service Worker, that Service Worker’s "install" event should receive a reference to the full package, with a way to copy the package’s contents into a "Cache" object. ([ServiceWorkers])

4. Non-goals

Some features often come along with packaging and signing, and it’s important to explicitly note that they don’t appear in the list of Requirements.

4.1. Store confidential data

Packages are designed to hold public information and to be shared to people with whom the original publisher never has an interactive connection. In that situation, there’s no way to keep the contents confidential: even if they were encrypted, to make the data public, anyone would have to be able to get the decryption key.

It’s possible to maintain something similar to confidentiality for non-public packaged data, but doing so complicates the format design and can give users a false sense of security.

We believe we’ll cause fewer privacy breaches if we omit any mechanism for encrypting data, than if we include something and try to teach people when it’s unsafe to use.

4.2. Generate packages on the fly

See discussion at WICG/webpackage#6 (https://github.com/WICG/webpackage/issues/6#issuecomment-275746125).

4.3. Non-origin identity

A package can be primarily identified as coming from something other than a Web Origin (https://html.spec.whatwg.org/multipage/browsers.html#concept-origin).

4.4. DRM

Special support for blocking access to downloaded content based on licensing. Note that DRM systems can be shipped inside the package even if the packaging format doesn’t specifically support them.

4.5. Ergonomic replacement for HTTP/2 PUSH

HTTP/2 PUSH ([RFC7540], section 8.2) is hard for developers to configure, and an explicit package format might be easier. However, experts in this area believe we should focus on improving PUSH directly instead of routing around it with a bundling format.

Trying to bundle resources in order to speed up page loads has a long history, including Resource Packages (https://www.mnot.net/blog/2010/02/18/resource_packages) from 2010 and the W3C TAG’s packaging proposal (https://w3ctag.github.io/packaging-on-the-web/) from 2015.

However, the HTTPWG is doing a lot of work to let servers optimize the PUSHed data, and packaging would either have to re-do that or accept lower performance. For example:

* [I-D.vkrasnov-h2-compression-dictionaries] should allow individual small resources to be compressed as well as they would be in a bundle.

* [I-D.ietf-httpbis-cache-digest] tells the server which resources it doesn’t need to PUSH.

Associated requirements:

* Streamed loading: If the whole package has to be downloaded before the browser can load a piece, this will definitely be slower than PUSH.

* Compress transfers: Keep up with [I-D.vkrasnov-h2-compression-dictionaries].

* Indexed by URL: Resources on the web are addressed by URL.

* Request headers: PUSH_PROMISE (http://httpwg.org/specs/rfc7540.html#PUSH_PROMISE) ([RFC7540], section 6.6) includes request headers.

* Response headers: PUSHed resources include their response headers.

5. Security Considerations

The security considerations will depend on the solution designed to satisfy the above requirements. See [I-D.yasskin-dispatch-web-packaging] for one possible set of security considerations.

6. IANA Considerations

This document has no actions for IANA.

7. Informative References

[I-D.burke-content-signature]
Burke, B., "HTTP Header for digital signatures", Work in Progress, Internet-Draft, draft-burke-content-signature-00, 7 March 2011, <http://www.ietf.org/internet-drafts/draft-burke-content-signature-00.txt>.

[I-D.cavage-http-signatures]
Cavage, M. and M. Sporny, "Signing HTTP Messages", Work in Progress, Internet-Draft, draft-cavage-http-signatures-12, 21 October 2019, <http://www.ietf.org/internet-drafts/draft-cavage-http-signatures-12.txt>.

[I-D.ietf-httpbis-cache-digest]
Oku, K. and Y. Weiss, "Cache Digests for HTTP/2", Work in Progress, Internet-Draft, draft-ietf-httpbis-cache-digest-05, 2 July 2018, <http://www.ietf.org/internet-drafts/draft-ietf-httpbis-cache-digest-05.txt>.

[I-D.ietf-httpbis-http2-secondary-certs]
Bishop, M., Sullivan, N., and M. Thomson, "Secondary Certificate Authentication in HTTP/2", Work in Progress, Internet-Draft, draft-ietf-httpbis-http2-secondary-certs-06, 14 May 2020, <http://www.ietf.org/internet-drafts/draft-ietf-httpbis-http2-secondary-certs-06.txt>.

[I-D.thomson-http-content-signature]
Thomson, M., "Content-Signature Header Field for HTTP", Work in Progress, Internet-Draft, draft-thomson-http-content-signature-00, 2 July 2015, <http://www.ietf.org/internet-drafts/draft-thomson-http-content-signature-00.txt>.

[I-D.vkrasnov-h2-compression-dictionaries]
Krasnov, V. and Y. Weiss, "Compression Dictionaries for HTTP/2", Work in Progress, Internet-Draft, draft-vkrasnov-h2-compression-dictionaries-03, 5 March 2018, <http://www.ietf.org/internet-drafts/draft-vkrasnov-h2-compression-dictionaries-03.txt>.

[I-D.yasskin-dispatch-web-packaging]
Yasskin, J., "Web Packaging", Work in Progress, Internet-Draft, draft-yasskin-dispatch-web-packaging-00, 30 June 2017, <http://www.ietf.org/internet-drafts/draft-yasskin-dispatch-web-packaging-00.txt>.

[ISO28500] "WARC file format", ISO 28500:2017, 2017, <https://www.iso.org/standard/68004.html>.

[JAR] "JAR File Specification", 2014, <https://docs.oracle.com/javase/7/docs/technotes/guides/jar/jar.html>.

[MHTML] Palme, J., Hopmann, A., and N. Shelness, "MIME Encapsulation of Aggregate Documents, such as HTML (MHTML)", RFC 2557, DOI 10.17487/RFC2557, March 1999, <https://www.rfc-editor.org/info/rfc2557>.

[RFC6454] Barth, A., "The Web Origin Concept", RFC 6454, DOI 10.17487/RFC6454, December 2011, <https://www.rfc-editor.org/info/rfc6454>.

[RFC6797] Hodges, J., Jackson, C., and A. Barth, "HTTP Strict Transport Security (HSTS)", RFC 6797, DOI 10.17487/RFC6797, November 2012, <https://www.rfc-editor.org/info/rfc6797>.

[RFC6962] Laurie, B., Langley, A., and E. Kasper, "Certificate Transparency", RFC 6962, DOI 10.17487/RFC6962, June 2013, <https://www.rfc-editor.org/info/rfc6962>.

[RFC7469] Evans, C., Palmer, C., and R. Sleevi, "Public Key Pinning Extension for HTTP", RFC 7469, DOI 10.17487/RFC7469, April 2015, <https://www.rfc-editor.org/info/rfc7469>.

[RFC7515] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, <https://www.rfc-editor.org/info/rfc7515>.

[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>.

[ServiceWorkers] "Service Workers Nightly", W3C ED, <https://w3c.github.io/ServiceWorker/>.

[W3C.WD-appmanifest-20170828] Caceres, M., Christiansen, K., Lamouri, M., Kostiainen, A., and R. Dolin, "Web App Manifest", World Wide Web Consortium WD WD-appmanifest-20170828, 28 August 2017, <https://www.w3.org/TR/2017/WD-appmanifest-20170828>.

[ZIP] "APPNOTE.TXT - .ZIP File Format Specification", 1 October 2014, <https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT>.

Appendix A. Acknowledgements

Thanks to Yoav Weiss for the Subresource bundling use case and discussions about content distributors.

Author's Address

Jeffrey Yasskin
Google

Email: jyasskin@chromium.org