MIME Sniffing

Living Standard — Last Updated 14 January 2021

Participate:

GitHub whatwg/mimesniff (new issue, open issues)
IRC: #whatwg on Freenode

Commits:

GitHub whatwg/mimesniff/commits
Snapshot as of this commit
@mimesniff

Tests:

web-platform-tests mimesniff/ (ongoing work)

Translations (non-normative):

日本語

Abstract

The MIME Sniffing standard defines sniffing resources.

Table of Contents

§ 1. Introduction

The HTTP `Content-Type` header field is intended to indicate the MIME type of an HTTP response. However, many HTTP servers supply a `Content-Type` header field value that does not match the actual contents of the response. Historically, web browsers have tolerated these servers by examining the content of HTTP responses in addition to the `Content-Type` header field in order to determine the effective MIME type of the response.

Without a clear specification for how to "sniff" the MIME type, each user agent has been forced to reverse-engineer the algorithms of other user agents in order to maintain interoperability. Inevitably, these efforts have not been entirely successful, resulting in divergent behaviors among user agents. In some cases, these divergent behaviors have had security implications, as a user agent could interpret an HTTP response as a different MIME type than the server intended.

These security issues are most severe when an "honest" server allows potentially malicious users to upload their own files and then serves the contents of those files with a low-privilege MIME type. For example, if a server believes that the client will treat a contributed file as an image (and thus treat it as benign), but a user agent believes the content to be HTML (and thus privileged to execute any scripts contained therein), an attacker might be able to steal the user's authentication credentials and mount other cross-site scripting attacks. (Malicious servers, of course, can specify an arbitrary MIME type in the `Content-Type` header field.)

This document describes a content sniffing algorithm that carefully balances the compatibility needs of user agent with the security constraints imposed by existing web content. The algorithm originated from research conducted by Adam Barth, Juan Caballero, and Dawn Song, based on content sniffing algorithms present in popular user agents, an extensive database of existing web content, and metrics collected from implementations deployed to a sizable number of users. [SECCONTSNIFF]

§ 2. Conformance requirements

The keywords "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. For readability, these keywords will generally not appear in all uppercase letters. [RFC2119]

Requirements phrased in the imperative as part of algorithms (such as "strip any leading space characters" or "return false and abort these steps") are to be interpreted with the meaning of the keyword used in introducing the algorithm.

Conformance requirements phrased as algorithms or specific steps can be implemented in any manner, so long as the end result is equivalent. In particular, note that the algorithms defined in this specification are intended to be easy to understand and are not intended to be performant.

§ 3. Terminology

This specification depends on the Infra Standard. [INFRA]

An **HTTP token code point** is U+0021 (!), U+0023 (#), U+0024 ($), U+0025 (%), U+0026 (&), U+0027 ('), U+002A (*), U+002B (+), U+002D (-), U+002E (.), U+005E (^), U+005F (_), U+0060 (`), U+007C (|), U+007E (~), or an ASCII alphanumeric.

Note

This matches the value space of the token token production. [HTTP]

An **HTTP quoted-string token code point** is U+0009 TAB, a code point in the range U+0020 SPACE to U+007E (~), inclusive, or a code point in the range U+0080 through U+00FF (ÿ), inclusive.

Note

This matches the effective value space of the quoted-string token production. By definition it is a superset of the HTTP token code points. [HTTP]

A **binary data byte** is a byte in the range 0x00 to 0x08 (NUL to BS), the byte 0x0B (VT), a byte in the range 0x0E to 0x1A (SO to SUB), or a byte in the range 0x1C to 0x1F (FS to US).

A **whitespace byte** (abbreviated 0xWS) is any one of the following bytes: 0x09 (HT), 0x0A (LF), 0x0C (FF), 0x0D (CR), 0x20 (SP).

A **tag-terminating byte** (abbreviated 0xTT) is any one of the following bytes: 0x20 (SP), 0x3E (">").

Equations are using the mathematical operators as defined in [ENCODING]. In addition, the bitwise NOT is represented by ~.

4. MIME types

4.1. MIME type representation

A **MIME type** represents an *internet media type* as defined by *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*. It can also be referred to as a MIME type record. [MIMETYPE]

Note

Standards are encouraged to consistently use the term MIME type to avoid confusion with the use of media type *as described in* Media Queries. *[MEDIAQUERIES]*

A MIME type's **type** is a non-empty ASCII string.

A MIME type's **subtype** is a non-empty ASCII string.

A MIME type's **parameters** is an ordered map whose keys and values are ASCII strings. It is initially empty.

4.2. MIME type miscellaneous

The **essence** of a MIME type *mimeType* is *mimeType*'s type, followed by U+002F (/), followed by *mimeType*'s subtype.

A MIME type is **supported by the user agent** if the user agent has the capability to interpret a resource of that MIME type and present it to the user.

This needs more work. See w3c/preload #113.

4.3. MIME type writing

A **valid MIME type string** is a string that matches the media-type token production. In particular, a valid MIME type string may include parameters. [RFC7231]

Note

A valid MIME type string is supposed to be used for conformance checkers only.

Example

`"text/html"` is a valid MIME type string.

`"text/html;"` is not a valid MIME type string, though parse a MIME type returns a MIME type record for it identical to if the input had been `"text/html"`.

A **valid MIME type string with no parameters** is a valid MIME type string that does not contain U+003B (;).

§ 4.4. Parsing a MIME type

To **parse a MIME type**, given a string *input*, run these steps:

1. Remove any leading and trailing HTTP whitespace from *input*.
2. Let *position* be a position variable for *input*, initially pointing at the start of *input*.
3. Let *type* be the result of collecting a sequence of code points that are not U+002F (/) from *input*, given *position*.
4. If *type* is the empty string or does not solely contain HTTP token code points, then return failure.
5. If *position* is past the end of *input*, then return failure.
6. Advance *position* by 1. (This skips past U+002F (/).)
7. Let *subtype* be the result of collecting a sequence of code points that are not U+003B (;) from *input*, given *position*.
8. Remove any trailing HTTP whitespace from *subtype*.
9. If *subtype* is the empty string or does not solely contain HTTP token code points, then return failure.
10. Let *mimeType* be a new MIME type record whose type is *type*, in ASCII lowercase, and subtype is *subtype*, in ASCII lowercase.
11. While *position* is not past the end of *input*:
    1. Advance *position* by 1. (This skips past U+003B (;).)
    2. Collect a sequence of code points that are HTTP whitespace from *input* given *position*.

       Note

       This is roughly equivalent to skip ASCII whitespace, except that HTTP whitespace is used rather than ASCII whitespace.

    3. Let *parameterName* be the result of collecting a sequence of code points that are not U+003B (;) or U+003D (=) from *input*, given *position*.
    4. Set *parameterName* to *parameterName*, in ASCII lowercase.
    5. If *position* is not past the end of *input*, then:
        1. If the code point at *position* within *input* is U+003B (;), then continue.
        2. Advance *position* by 1. (This skips past U+003D (=).)
    6. If *position* is past the end of *input*, then break.
    7. Let *parameterValue* be null.
    8. If the code point at *position* within *input* is U+0022 ("), then:
        1. Set *parameterValue* to the result of collecting an HTTP quoted string from *input*, given *position* and the *extract-value flag*.

2. Collect a sequence of code points that are not U+003B (;) from *input*, given *position*.

¶ Example

Given `text/html;charset="shift_jis"iso-2022-jp` you end up with `text/html;charset=shift_jis`.

9. Otherwise:

    1. Set *parameterValue* to the result of collecting a sequence of code points that are not U+003B (;) from *input*, given *position*.

    2. Remove any trailing HTTP whitespace from *parameterValue*.

    3. If *parameterValue* is the empty string, then continue.

10. If all of the following are true

    - *parameterName* is not the empty string
    - *parameterName* solely contains HTTP token code points
    - *parameterValue* solely contains HTTP quoted-string token code points
    - *mimeType*'s parameters[*parameterName*] does not exist

    then set *mimeType*'s parameters[*parameterName*] to *parameterValue*.

12. Return *mimeType*.

To **parse a MIME type from bytes**, given a byte sequence *input*, run these steps:

1. Let *string* be *input*, isomorphic decoded.

2. Return the result of parse a MIME type with *string*.

§ 4.5. Serializing a MIME type

To **serialize a MIME type**, given a MIME type *mimeType*, run these steps:

1. Let *serialization* be the concatenation of *mimeType*'s type, U+002F (/), and *mimeType*'s subtype.

2. For each *name* → *value* of *mimeType*'s parameters:

    1. Append U+003B (;) to *serialization*.

    2. Append *name* to *serialization*.

    3. Append U+003D (=) to *serialization*.

    4. If *value* does not solely contain HTTP token code points or *value* is the empty string, then:

        1. Precede each occurence of U+0022 (") or U+005C (\) in *value* with U+005C (\).

        2. Prepend U+0022 (") to *value*.

        3. Append U+0022 (") to *value*.

5. Append *value* to *serialization*.

3. Return *serialization*.

To **serialize a MIME type to bytes**, given a MIME type *mimeType*, run these steps:

1. Let *stringSerialization* be the result of serialize a MIME type with *mimeType*.

2. Return *stringSerialization*, isomorphic encoded.

4.6. MIME type groups

An **image MIME type** is a MIME type whose type is "`image`".

An **audio or video MIME type** is any MIME type whose type is "`audio`" or "`video`", or whose essence is "`application/ogg`".

A **font MIME type** is any MIME type whose type is "`font`", or whose essence is one of the following: [RFC8081]

- `application/font-cff`
- `application/font-off`
- `application/font-sfnt`
- `application/font-ttf`
- `application/font-woff`
- `application/vnd.ms-fontobject`
- `application/vnd.ms-opentype`

A **ZIP-based MIME type** is any MIME type whose subtype ends in "`+zip`" or whose essence is one of the following:

- `application/zip`

An **archive MIME type** is any MIME type whose essence is one of the following:

- `application/x-rar-compressed`
- `application/zip`
- `application/x-gzip`

An **XML MIME type** is any MIME type whose subtype ends in "`+xml`" or whose essence is "`text/xml`" or "`application/xml`". [RFC7303]

An **HTML MIME type** is any MIME type whose essence is "`text/html`".

A **scriptable MIME type** is an XML MIME type, HTML MIME type, or any MIME type whose essence is "`application/pdf`".

A **JavaScript MIME type** is any MIME type whose essence is one of the following:

- `application/ecmascript`
- `application/javascript`
- `application/x-ecmascript`
- `application/x-javascript`
- `text/ecmascript`
- `text/javascript`
- `text/javascript1.0`
- `text/javascript1.1`
- `text/javascript1.2`

- `text/javascript1.3`
- `text/javascript1.4`
- `text/javascript1.5`
- `text/jscript`
- `text/livescript`
- `text/x-ecmascript`
- `text/x-javascript`

A string is a **JavaScript MIME type essence match** if it is an ASCII case-insensitive match for one of the JavaScript MIME type essence strings.

Note

This hook is used by the `type` attribute of `script` elements.

A **JSON MIME type** is any MIME type whose subtype ends in "`+json`" or whose essence is "`application/json`" or "`text/json`".

§ 5. Handling a resource

A **resource** is ....

For each resource it handles, the user agent must keep track of the following associated metadata:

- A **supplied MIME type**, the MIME type determined by the supplied MIME type detection algorithm.
- A **check-for-apache-bug flag**, which defaults to unset.
- A **no-sniff flag**, which defaults to set if the user agent does not wish to perform sniffing on the resource and unset otherwise.

Note

The user agent can choose to use outside information, such as previous experience with a site, to determine whether to opt out of sniffing for a particular resource. The user agent can also choose to opt out of sniffing for all resources. However, opting out of sniffing does not exempt the user agent from using the MIME type sniffing algorithm.

- A **computed MIME type**, the MIME type determined by the MIME type sniffing algorithm.

§ 5.1. Interpreting the resource metadata

The supplied MIME type of a resource is provided to the user agent by an external source associated with that resource. The method of obtaining this information varies depending upon how the resource is retrieved.

To determine the supplied MIME type of a resource, user agents must use the following **supplied MIME type detection algorithm**:

1. Let *supplied-type* be null.
2. If the resource is retrieved via HTTP, execute the following steps:
    1. If one or more `Content-Type` headers are associated with the resource, execute the following steps:
        1. Set *supplied-type* to the value of the last `Content-Type` header associated with the resource.

            Note

            File extensions are not used to determine the supplied MIME type of a resource retrieved via HTTP because they are unreliable and easily spoofed.

        2. Set the check-for-apache-bug flag if *supplied-type* is **exactly** equal to one of the values in the following table:

Bytes in Hexadecimal	Bytes in ASCII
74 65 78 74 2F 70 6C 61 69 6E	`text/plain`

Bytes in Hexadecimal	Bytes in ASCII
74 65 78 74 2F 70 6C 61 69 6E 3B 20 63 68 61 72 73 65 74 3D 49 53 4F 2D 38 38 35 39 2D 31	`text/plain; charset=ISO-8859-1`
74 65 78 74 2F 70 6C 61 69 6E 3B 20 63 68 61 72 73 65 74 3D 69 73 6F 2D 38 38 35 39 2D 31	`text/plain; charset=iso-8859-1`
74 65 78 74 2F 70 6C 61 69 6E 3B 20 63 68 61 72 73 65 74 3D 55 54 46 2D 38	`text/plain; charset=UTF-8`

Note

The supplied MIME type detection algorithm detects these exact byte sequences because some older installations of Apache contain a bug that causes them to supply one of these Content-Type headers when serving files with unrecognized MIME types.

[HTTP]

3. If the resource is retrieved directly from the file system, set *supplied-type* to the MIME type provided by the file system.
4. If the resource is retrieved via another protocol (such as FTP), set *supplied-type* to the MIME type as determined by that protocol, if any.

   [FTP]
5. If *supplied-type* is not a MIME type, the supplied MIME type is undefined.

   Abort these steps.
6. The supplied MIME type is *supplied-type*.

§ 5.2. Reading the resource header

A **resource header** is the byte sequence at the beginning of a resource, as determined by reading the resource header.

To **read the resource header**, perform the following steps:

1. Let *buffer* be a byte sequence.
2. Read bytes of the resource into *buffer* until one of the following conditions is met:
   - the end of the resource is reached.
   - the number of bytes in *buffer* is greater than or equal to 1445.
   - a reasonable amount of time has elapsed, as determined by the user agent.

Note

If the number of bytes in buffer *is greater than or equal to 1445, the MIME type sniffing algorithm will be deterministic for the majority of cases.*

However, certain factors (such as a slow connection) may prevent the user agent from reading 1445 bytes in a reasonable amount of time.

3. The resource header is *buffer*.

Note

The resource header need only be determined once per resource.

§ 6. Matching a MIME type pattern

A **byte pattern** is a byte sequence used as a template to be matched against in the pattern matching algorithm.

A **pattern mask** is a byte sequence used to determine the significance of bytes being compared against a byte pattern in the pattern matching algorithm.

Note

In a pattern mask, 0xFF indicates the byte is strictly significant, 0xDF indicates that the byte is significant in an ASCII case-insensitive way, and 0x00 indicates that the byte is not significant.

To determine whether a byte sequence matches a particular byte pattern, use the following **pattern matching algorithm**. It is given a byte sequence *input*, a byte pattern *pattern*, a pattern mask *mask*, and a set of bytes to be ignored *ignored*, and returns true or false.

1. Assert: *pattern*'s length is equal to *mask*'s length.
2. If *input*'s length is less than *pattern*'s length, return false.
3. Let *s* be 0.
4. While *s* < *input*'s length:
    1. If *ignored* does not contain *input*[*s*], break.
    2. Set *s* to *s* + 1.
5. Let *p* be 0.
6. While *p* < *pattern*'s length:
    1. Let *maskedData* be the result of applying the bitwise AND operator to *input*[*s*] and *mask*[*p*].
    2. If *maskedData* is not equal to *pattern*[*p*], return false.
    3. Set *s* to *s* + 1.
    4. Set *p* to *p* + 1.
7. Return true.

§ 6.1. Matching an image type pattern

To determine which image MIME type byte pattern a byte sequence *input* matches, if any, use the following **image type pattern matching algorithm**:

1. Execute the following steps for each row *row* in the following table:
    1. Let *patternMatched* be the result of the pattern matching algorithm given *input*, the value in the first column of *row*, the value in the second column of *row*, and the value in the third column of *row*.
    2. If *patternMatched* is true, return the value in the fourth column of *row*.

Byte Pattern	Pattern Mask	Leading Bytes to Be Ignored	Image MIME Type	Note
00 00 01 00	FF FF FF FF	None.	`image/x-icon`	A Windows Icon signature.
00 00 02 00	FF FF FF FF	None.	`image/x-icon`	A Windows Cursor signature.
42 4D	FF FF	None.	`image/bmp`	The string "BM", a BMP signature.
47 49 46 38 37 61	FF FF FF FF FF FF	None.	`image/gif`	The string "GIF87a", a GIF signature.
47 49 46 38 39 61	FF FF FF FF FF FF	None.	`image/gif`	The string "GIF89a", a GIF signature.
52 49 46 46 00 00 00 00 57 45 42 50 56 50	FF FF FF FF 00 00 00 00 FF FF FF FF FF FF	None.	`image/webp`	The string "RIFF" followed by four bytes followed by the string "WEBPVP".
89 50 4E 47 0D 0A 1A 0A	FF FF FF FF FF FF FF FF	None.	`image/png`	An error-checking byte followed by the string "PNG" followed by CR LF SUB LF, the PNG signature.
FF D8 FF	FF FF FF	None.	`image/jpeg`	The JPEG Start of Image marker followed by the indicator byte of another marker.

2. Return undefined.

§ 6.2. Matching an audio or video type pattern

To determine which audio or video MIME type byte pattern a byte sequence *input* matches, if any, use the following **audio or video type pattern matching algorithm**:

1. Execute the following steps for each row *row* in the following table:
    1. Let *patternMatched* be the result of the pattern matching algorithm given *input*, the value in the first column of *row*, the value in the second column of *row*, and the value in the third column of *row*.
    2. If *patternMatched* is true, return the value in the fourth column of *row*.

Byte Pattern	Pattern Mask	Leading Bytes to Be Ignored	Audio or Video MIME Type	Note
2E 73 6E 64	FF FF FF FF	None.	`audio/basic`	The string ".snd", the basic audio signature.
46 4F 52 4D 00 00 00 00 41 49 46 46	FF FF FF FF 00 00 00 00 FF FF FF FF	None.	`audio/aiff`	The string "FORM" followed by four bytes followed by the string "AIFF", the AIFF signature.
49 44 33	FF FF FF	None.	`audio/mpeg`	The string "ID3", the ID3v2-tagged MP3 signature.
4F 67 67 53 00	FF FF FF FF FF	None.	`application/ogg`	The string "OggS" followed by NUL, the Ogg container signature.

Byte Pattern	Pattern Mask	Leading Bytes to Be Ignored	Audio or Video MIME Type	Note
4D 54 68 64 00 00 00 06	FF FF FF FF FF FF FF FF	None.	`audio/midi`	The string "MThd" followed by four bytes representing the number 6 in 32 bits (big-endian), the MIDI signature.
52 49 46 46 00 00 00 00 41 56 49 20	FF FF FF FF 00 00 00 00 FF FF FF FF	None.	`video/avi`	The string "RIFF" followed by four bytes followed by the string "AVI ", the AVI signature.
52 49 46 46 00 00 00 00 57 41 56 45	FF FF FF FF 00 00 00 00 FF FF FF FF	None.	`audio/wave`	The string "RIFF" followed by four bytes followed by the string "WAVE", the WAVE signature.

2. If *input* matches the signature for MP4, return "`video/mp4`".
3. If *input* matches the signature for WebM, return "`video/webm`".
4. If *input* matches the signature for MP3 without ID3, return "`audio/mpeg`".
5. Return undefined.

§ 6.2.1. Signature for MP4

To determine whether a byte sequence **matches the signature for MP4**, use the following steps:

1. Let *sequence* be the byte sequence to be matched, where *sequence*[*s*] is byte *s* in *sequence* and *sequence*[0] is the first byte in *sequence*.
2. Let *length* be the number of bytes in *sequence*.
3. If *length* is less than 12, return false.
4. Let *box-size* be the four bytes from *sequence*[0] to *sequence*[3], interpreted as a 32-bit unsigned big-endian integer.
5. If *length* is less than *box-size* or if *box-size* modulo 4 is not equal to 0, return false.
6. If the four bytes from *sequence*[4] to *sequence*[7] are not equal to 0x66 0x74 0x79 0x70 ("`ftyp`"), return false.
7. If the three bytes from *sequence*[8] to *sequence*[10] are equal to 0x6D 0x70 0x34 ("`mp4`"), return true.
8. Let *bytes-read* be 16.

   Note

   This ignores the four bytes that correspond to the version number of the "major brand".

9. While *bytes-read* is less than *box-size*, continuously loop through these steps:
   1. If the three bytes from *sequence*[*bytes-read*] to *sequence*[*bytes-read* + 2] are equal to 0x6D 0x70 0x34 ("`mp4`"), return true.
   2. Increment *bytes-read* by 4.

10. Return false.

6.2.2. Signature for WebM

To determine whether a byte sequence **matches the signature for WebM**, use the following steps:

1. Let *sequence* be the byte sequence to be matched, where *sequence*[*s*] is byte *s* in *sequence* and *sequence*[0] is the first byte in *sequence*.
2. Let *length* be the number of bytes in *sequence*.
3. If *length* is less than 4, return false.
4. If the four bytes from *sequence*[0] to *sequence*[3], are not equal to 0x1A 0x45 0xDF 0xA3, return false.
5. Let *iter* be 4.
6. While *iter* is less than *length* and *iter* is less than 38, continuously loop through these steps:
    1. If the two bytes from *sequence*[*iter*] to *sequence*[*iter* + 1] are equal to 0x42 0x82,
        1. Increment *iter* by 2.
        2. If *iter* is greater or equal than *length*, abort these steps.
        3. Let *number size* be the result of parsing a `vint` starting at *sequence*[*iter*].
        4. Increment *iter* by *number size*.
        5. If *iter* is less than *length* - 4, abort these steps.
        6. Let *matched* be the result of matching a padded sequence 0x77 0x65 0x62 0x6D ("webm") on *sequence* at offset *iter*.
        7. If *matched* is true, abort these steps and return true.
    2. Increment *iter* by 1.
7. Return false.

To **parse a `vint`** on a byte sequence *sequence* of size *length*, starting at index *iter* use the following steps:

1. Let *mask* be 128.
2. Let *max vint length* be 8.
3. Let *number size* be 1.
4. While *number size* is less than *max vint length*, and less than *length*, continuously loop through these steps:
    1. If the *sequence*[*index*] & *mask* is not zero, abort these steps.
    2. Let *mask* be the value of *mask* >> 1.
    3. Increment *number size* by one.
5. Let *index* be 0.
6. Let *parsed number* be *sequence*[*index*] & ~*mask*.
7. Increment *index* by one.
8. Let *bytes remaining* be the value of *number size*.

9. While *bytes remaining* is not zero, execute there steps:
    1. Let *parsed number* be *parsed number* << 8.
    2. Let *parsed number* be *parsed number* | *sequence*[*index*].
    3. Increment *index* by one.
    4. If *index* is greater or equal than *length*, abort these steps.
    5. Decrement *bytes remaining* by one.
10. Return *parsed number* and *number size*

Matching a padded sequence *pattern* on a sequence *sequence* at starting at byte *offset* and ending at by *end* means returning true if *sequence* has a length greater than *end*, and contains exactly, in the range [*offset*, *end*], the bytes in *pattern*, in the same order, eventually preceded by bytes with a value of 0x00, false otherwise.

§ 6.2.3. Signature for MP3 without ID3

To determine whether a byte sequence **matches the signature for MP3 without ID3**, use the following steps:

1. Let *sequence* be the byte sequence to be matched, where *sequence*[*s*] is byte *s* in *sequence* and *sequence*[0] is the first byte in *sequence*.
2. Let *length* be the number of bytes in *sequence*.
3. Initialize s to 0.
4. If the result of the operation match mp3 header is false, return false.
5. Parse an mp3 frame on *sequence* at offset *s*
6. Let *skipped-bytes* the return value of the execution of mp3 framesize computation
7. If *skipped-bytes* is less than 4, or *skipped-bytes* is greater than *s* - *length*, return false.
8. Increment s by *skipped-bytes*.
9. If the result of the operation match mp3 header operation is false, return false, else, return true.

To **match an mp3 header**, using a byte sequence *sequence* of length *length* at offset *s* execute these steps:

1. If *length* is less than 4, return false.
2. If *sequence*[*s*] is not equal to 0xff and *sequence*[*s* + 1] & 0xe0 is not equal to 0xe0, return false.
3. Let *layer* be the result of *sequence*[*s* + 1] & 0x06 >> 1.
4. If *layer* is 0, return false.
5. Let *bit-rate* be *sequence*[*s* + 2] & 0xf0 >> 4.
6. If *bit-rate* is 15, return false.
7. Let *sample-rate* be *sequence*[*s* + 2] & 0x0c >> 2.

8. If *sample-rate* is 3, return false.
9. Let *freq* be the value given by *sample-rate* in the table sample-rate.
10. Let *final-layer* be the result of 4 - (*sequence*[*s* + 1]).
11. If *final-layer* & 0x06 >> 1 is not 3, return false.
12. Return true.

To **compute an mp3 frame size**, execute these steps:

1. If *version* is 1, let *scale* be 72, else, let *scale* be 144.
2. Let *size* be *bitrate* * *scale* / *freq*.
3. If *pad* is not zero, increment *size* by 1.
4. Return *size*.

To **parse an mp3 frame**, execute these steps:

1. Let *version* be *sequence*[*s* + 1] & 0x18 >> 3.
2. Let *bitrate-index* be sequence[s + 2] & 0xf0 >> 4.
3. If the *version* & 0x01 is non-zero, let *bitrate* be the value given by *bitrate-index* in the table mp2.5-rates
4. If *version* & 0x01 is zero, let *bitrate* be the value given by *bitrate-index* in the table mp3-rates
5. Let *samplerate-index* be *sequence*[*s* + 2] & 0x0c >> 2.
6. Let *samplerate* be the value given by *samplerate-index* in the *sample-rate* table.
7. Let *pad* be *sequence*[*s* + 2] & 0x02 >> 1.

mp3-rates table

index	mp3-rates
0	0
1	32000
2	40000
3	48000
4	56000
5	64000
6	80000
7	96000
8	112000
9	128000
10	160000
11	192000
12	224000
13	256000
14	320000

mp2.5-rates

index	mp2.5-rates
0	0
1	8000
2	16000
3	24000
4	32000
5	40000
6	48000
7	56000
8	64000
9	80000
10	96000
11	112000
12	128000
13	144000
14	160000

sample-rate table

index	samplerate
0	44100
1	48000
2	32000

§ 6.3. Matching a font type pattern

To determine which font MIME type byte pattern a byte sequence *input* matches, if any, use the following **font type pattern matching algorithm**:

1. Execute the following steps for each row *row* in the following table:
    1. Let *patternMatched* be the result of the pattern matching algorithm given *input*, the value in the first column of *row*, the value in the second column of *row*, and the value in the third column of *row*.
    2. If *patternMatched* is true, return the value in the fourth column of *row*.

Byte Pattern	Pattern Mask	Leading Bytes to Be Ignored	Font MIME Type	Note
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 4C 50	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 FF FF	None.	`application/vnd.ms-fontobject`	34 bytes followed by the string "LP", the Embedded OpenType signature.
00 01 00 00	FF FF FF FF	None.	`font/ttf`	4 bytes representing the version number 1.0, a TrueType signature.

Byte Pattern	Pattern Mask	Leading Bytes to Be Ignored	Font MIME Type	Note
4F 54 54 4F	FF FF FF FF	None.	`font/otf`	The string "0TT0", the OpenType signature.
74 74 63 66	FF FF FF FF	None.	`font/collection`	The string "ttcf", the TrueType Collection signature.
77 4F 46 46	FF FF FF FF	None.	`font/woff`	The string "w0FF", the Web Open Font Format 1.0 signature.
77 4F 46 32	FF FF FF FF	None.	`font/woff2`	The string "w0F2", the Web Open Font Format 2.0 signature.

2. Return undefined.

§ 6.4. Matching an archive type pattern

To determine which archive MIME type byte pattern a byte sequence *input* matches, if any, use the following **archive type pattern matching algorithm**:

1. Execute the following steps for each row *row* in the following table:
    1. Let *patternMatched* be the result of the pattern matching algorithm given *input*, the value in the first column of *row*, the value in the second column of *row*, and the value in the third column of *row*.
    2. If *patternMatched* is true, return the value in the fourth column of *row*.

Byte Pattern	Pattern Mask	Leading Bytes to Be Ignored	Archive MIME Type	Note
1F 8B 08	FF FF FF	None.	`application/x-gzip`	The GZIP archive signature.
50 4B 03 04	FF FF FF FF	None.	`application/zip`	The string "PK" followed by ETX EOT, the ZIP archive signature.
52 61 72 20 1A 07 00	FF FF FF FF FF FF FF	None.	`application/x-rar-compressed`	The string "Rar " followed by SUB BEL NUL, the RAR archive signature.

2. Return undefined.

§ 7. Determining the computed MIME type of a resource

To determine the computed MIME type of a resource, user agents must use the following **MIME type sniffing algorithm**:

1. If the supplied MIME type is undefined or if the supplied MIME type's essence is "unknown/unknown", "application/unknown", or "*/*", execute the rules for identifying an unknown MIME type with the sniff-scriptable flag equal to the inverse of the no-sniff flag and abort these steps.
2. If the no-sniff flag is set, the computed MIME type is the supplied MIME type.

   Abort these steps.
3. If the check-for-apache-bug flag is set, execute the rules for distinguishing if a resource is text or binary and abort these steps.
4. If the supplied MIME type is an XML MIME type, the computed MIME type is the supplied MIME type.

   Abort these steps.
5. If the supplied MIME type's essence is "text/html", execute the rules for distinguishing if a resource is a feed or HTML and abort these steps.
6. If the supplied MIME type is an image MIME type supported by the user agent, let *matched-type* be the result of executing the image type pattern matching algorithm with the resource header as the byte sequence to be matched.
7. If *matched-type* is not undefined, the computed MIME type is *matched-type*.

   Abort these steps.
8. If the supplied MIME type is an audio or video MIME type supported by the user agent, let *matched-type* be the result of executing the audio or video type pattern matching algorithm with the resource header as the byte sequence to be matched.
9. If *matched-type* is not undefined, the computed MIME type is *matched-type*.

   Abort these steps.
10. The computed MIME type is the supplied MIME type.

§ 7.1. Identifying a resource with an unknown MIME type

The **sniff-scriptable flag** is used by the rules for identifying an unknown MIME type to determine whether to sniff for scriptable MIME types.

If the setting of the sniff-scriptable flag is not specified when calling the rules for identifying an unknown MIME type, the sniff-scriptable flag must default to unset.

To determine the computed MIME type of a resource *resource* with an unknown MIME type, execute the following **rules for identifying an unknown MIME type**:

1. If the sniff-scriptable flag is set, execute the following steps for each row *row* in the following table:

1. Let *patternMatched* be the result of the pattern matching algorithm given *resource*'s resource header, the value in the first column of *row*, the value in the second column of *row*, and the value in the third column of *row*.

2. If *patternMatched* is true, return the value in the fourth column of *row*.

Byte Pattern	Pattern Mask	Leading Bytes to Be Ignored	Computed MIME Type	Note
3C 21 44 4F 43 54 59 50 45 20 48 54 4D 4C TT	FF FF DF DF DF DF DF DF DF DF FF DF DF DF DF FF	Whitespace bytes.	`text/html`	The case-insensitive string "`<!DOCTYPE HTML`" followed by a tag-terminating byte.
3C 48 54 4D 4C TT	FF DF DF DF DF FF	Whitespace bytes.	`text/html`	The case-insensitive string "`<HTML`" followed by a tag-terminating byte.
3C 48 45 41 44 TT	FF DF DF DF DF FF	Whitespace bytes.	`text/html`	The case-insensitive string "`<HEAD`" followed by a tag-terminating byte.
3C 53 43 52 49 50 54 TT	FF DF DF DF DF DF DF DF	Whitespace bytes.	`text/html`	The case-insensitive string "`<SCRIPT`" followed by a tag-terminating byte.
3C 49 46 52 41 4D 45 TT	FF DF DF DF DF DF DF DF	Whitespace bytes.	`text/html`	The case-insensitive string "`<IFRAME`" followed by a tag-terminating byte.
3C 48 31 TT	FF DF FF FF	Whitespace bytes.	`text/html`	The case-insensitive string "`<H1`" followed by a tag-terminating byte.
3C 44 49 56 TT	FF DF DF DF FF	Whitespace bytes.	`text/html`	The case-insensitive string "`<DIV`" followed by a tag-terminating byte.
3C 46 4F 4E 54 TT	FF DF DF DF DF FF	Whitespace bytes.	`text/html`	The case-insensitive string "`<FONT`" followed by a tag-terminating byte.
3C 54 41 42 4C 45 TT	FF DF DF DF DF DF FF	Whitespace bytes.	`text/html`	The case-insensitive string "`<TABLE`" followed by a tag-terminating byte.
3C 41 TT	FF DF FF	Whitespace bytes.	`text/html`	The case-insensitive string "`<A`" followed by a tag-terminating byte.
3C 53 54 59 4C 45 TT	FF DF DF DF DF DF FF	Whitespace bytes.	`text/html`	The case-insensitive string "`<STYLE`" followed by a tag-terminating byte.
3C 54 49 54 4C 45 TT	FF DF DF DF DF DF FF	Whitespace bytes.	`text/html`	The case-insensitive string "`<TITLE`" followed by a tag-terminating byte.
3C 42 TT	FF DF FF	Whitespace bytes.	`text/html`	The case-insensitive string "`<B`" followed by a tag-terminating byte.

Byte Pattern	Pattern Mask	Leading Bytes to Be Ignored	Computed MIME Type	Note
3C 42 4F 44 59 TT	FF DF DF DF DF FF	Whitespace bytes.	`text/html`	The case-insensitive string "<BODY" followed by a tag-terminating byte.
3C 42 52 TT	FF DF DF FF	Whitespace bytes.	`text/html`	The case-insensitive string "<BR" followed by a tag-terminating byte.
3C 50 TT	FF DF FF	Whitespace bytes.	`text/html`	The case-insensitive string "<P" followed by a tag-terminating byte.
3C 21 2D 2D TT	FF FF FF FF FF	Whitespace bytes.	`text/html`	The string "<!--" followed by a tag-terminating byte.
3C 3F 78 6D 6C	FF FF FF FF FF	Whitespace bytes.	`text/xml`	The string "<?xml".
25 50 44 46 2D	FF FF FF FF FF	None.	`application/pdf`	The string "%PDF-", the PDF signature.

What about feeds?

2. Execute the following steps for each row *row* in the following table:

    1. Let *patternMatched* be the result of the pattern matching algorithm given *resource*'s resource header, the value in the first column of *row*, the value in the second column of *row*, and the value in the third column of *row*.

    2. If *patternMatched* is true, return the value in the fourth column of *row*.

Byte Pattern	Pattern Mask	Leading Bytes to Be Ignored	Computed MIME Type	Note
25 21 50 53 2D 41 64 6F 62 65 2D	FF FF FF FF FF FF FF FF FF FF FF	None.	`application/postscript`	The string "%!PS-Adobe-", the PostScript signature.
FE FF 00 00	FF FF 00 00	None.	`text/plain`	UTF-16BE BOM
FF FE 00 00	FF FF 00 00	None.	`text/plain`	UTF-16LE BOM
EF BB BF 00	FF FF FF 00	None.	`text/plain`	UTF-8 BOM

    User agents may implicitly extend this table to support additional MIME types.

    However, user agents should not implicitly extend this table to include additional byte patterns for any computed MIME type already present in this table, as doing so could introduce privilege escalation vulnerabilities.

    User agents must not introduce any privilege escalation vulnerabilities when extending this table.

3. Let *matchedType* be the result of executing the image type pattern matching algorithm given *resource*'s resource header.

4. If *matchedType* is not undefined, return *matchedType*.

5. Set *matchedType* to the result of executing the audio or video type pattern matching algorithm given *resource*'s resource header.

6. If *matchedType* is not undefined, return *matchedType*.
7. Set *matchedType* to the result of executing the archive type pattern matching algorithm given *resource*'s resource header.
8. If *matchedType* is not undefined, return *matchedType*.
9. If *resource*'s resource header contains no binary data bytes, return "`text/plain`".
10. Return "`application/octet-stream`".

§ 7.2. Sniffing a mislabeled binary resource

To determine whether a binary resource has been mislabeled as plain text, execute the following **rules for distinguishing if a resource is text or binary**:

1. Let *length* be the number of bytes in the resource header.
2. If *length* is greater than or equal to 2 and the first 2 bytes of the resource header are equal to 0xFE 0xFF (UTF-16BE BOM) or 0xFF 0xFE (UTF-16LE BOM), the computed MIME type is "`text/plain`".

   Abort these steps.
3. If *length* is greater than or equal to 3 and the first 3 bytes of the resource header are equal to 0xEF 0xBB 0xBF (UTF-8 BOM), the computed MIME type is "`text/plain`".

   Abort these steps.
4. If the resource header contains no binary data bytes, the computed MIME type is "`text/plain`".

   Abort these steps.
5. The computed MIME type is "`application/octet-stream`".

⚠Warning!

It is critical that the rules for distinguishing if a resource is text or binary never determine the computed MIME type to be a scriptable MIME type, as this could allow a privilege escalation attack.

§ 7.3. Sniffing a mislabeled feed

To determine whether a feed has been mislabeled as HTML, execute the following **rules for distinguishing if a resource is a feed or HTML**:

1. Let *sequence* be the resource header, where *sequence*[*s*] is byte *s* in *sequence* and *sequence*[0] is the first byte in *sequence*.
2. Let *length* be the number of bytes in *sequence*.

3. Initialize *s* to 0.

4. If *length* is greater than or equal to 3 and the three bytes from *sequence*[0] to *sequence*[2] are equal to 0xEF 0xBB 0xBF (UTF-8 BOM), increment *s* by 3.

5. While *s* is less than *length*, continuously loop through these steps:
    1. Enter loop *L*:
        1. If *sequence*[*s*] is undefined, the computed MIME type is the supplied MIME type.

           Abort these steps.

        2. If *sequence*[*s*] is equal to 0x3C ("<"), increment *s* by 1 and exit loop *L*.
        3. If *sequence*[*s*] is not a whitespace byte, the computed MIME type is the supplied MIME type.

           Abort these steps.

        4. Increment *s* by 1.
    2. Enter loop *L*:
        1. If *sequence*[*s*] is undefined, the computed MIME type is the supplied MIME type.

           Abort these steps.

        2. If *length* is greater than or equal to *s* + 3 and the three bytes from *sequence*[*s*] to *sequence*[*s* + 2] are equal to 0x21 0x2D 0x2D ("!--"), increment *s* by 3 and enter loop *M*:
            1. If *sequence*[*s*] is undefined, the computed MIME type is the supplied MIME type.

               Abort these steps.

            2. If *length* is greater than or equal to *s* + 3 and the three bytes from *sequence*[*s*] to *sequence*[*s* + 2] are equal to 0x2D 0x2D 0x3E ("-->"), increment *s* by 3 and exit loops *M* and *L*.
            3. Increment *s* by 1.
        3. If *length* is greater than or equal to *s* + 1 and *sequence*[*s*] is equal to 0x21 ("!"), increment *s* by 1 and enter loop *M*:
            1. If *sequence*[*s*] is undefined, the computed MIME type is the supplied MIME type.

               Abort these steps.

            2. If *length* is greater than or equal to *s* + 1 and *sequence*[*s*] is equal to 0x3E (">"), increment *s* by 1 and exit loops *M* and *L*.
            3. Increment *s* by 1.
        4. If *length* is greater than or equal to *s* + 1 and *sequence*[*s*] is equal to 0x3F ("?"), increment *s* by 1 and enter loop *M*:
            1. If *sequence*[*s*] is undefined, the computed MIME type is the supplied MIME type.

               Abort these steps.

            2. If *length* is greater than or equal to *s* + 2 and the two

bytes from *sequence*[*s*] to *sequence*[*s* + 1] are equal to 0x3F 0x3E ("?>"), increment *s* by 2 and exit loops *M* and *L*.

3. Increment *s* by 1.

5. If *length* is greater than or equal to *s* + 3 and the three bytes from *sequence*[*s*] to *sequence*[*s* + 2] are equal to 0x72 0x73 0x73 ("`rss`"), the computed MIME type is "`application/rss+xml`".

   Abort these steps.

6. If *length* is greater than or equal to *s* + 4 and the four bytes from *sequence*[*s*] to *sequence*[*s* + 3] are equal to 0x66 0x65 0x65 0x64 ("`feed`"), the computed MIME type is "`application/atom+xml`".

   Abort these steps.

7. If *length* is greater than or equal to *s* + 7 and the seven bytes from *sequence*[*s*] to *sequence*[*s* + 6] are equal to 0x72 0x64 0x66 0x3A 0x52 0x44 0x46 ("`rdf:RDF`"), increment *s* by 7 and enter loop *M*:

   1. If *sequence*[*s*] is undefined, the computed MIME type is the supplied MIME type.

      Abort these steps.

   2. If *length* is greater than or equal to *s* + 24 and the twenty-four bytes from *sequence*[*s*] to *sequence*[*s* + 23] are equal to 0x68 0x74 0x74 0x70 0x3A 0x2F 0x2F 0x70 0x75 0x72 0x6C 0x2E 0x6F 0x72 0x67 0x2F 0x72 0x73 0x73 0x2F 0x31 0x2E 0x30 0x2F ("`http://purl.org/rss/1.0/`"), increment *s* by 24 and enter loop *N*:

      1. If *sequence*[*s*] is undefined, the computed MIME type is the supplied MIME type.

         Abort these steps.

      2. If *length* is greater than or equal to *s* + 43 and the forty-three bytes from *sequence*[*s*] to *sequence*[*s* + 42] are equal to 0x68 0x74 0x74 0x70 0x3A 0x2F 0x2F 0x77 0x77 0x77 0x2E 0x77 0x33 0x2E 0x6F 0x72 0x67 0x2F 0x31 0x39 0x39 0x39 0x2F 0x30 0x32 0x2F 0x32 0x32 0x2D 0x72 0x64 0x66 0x2D 0x73 0x79 0x6E 0x74 0x61 0x78 0x2D 0x6E 0x73 0x23 ("`http://www.w3.org/1999/02/22-rdf-syntax-ns#`"), the computed MIME type is "`application/rss+xml`".

         Abort these steps.

      3. Increment *s* by 1.

   3. If *length* is greater than or equal to *s* + 24 and the twenty-four bytes from *sequence*[*s*] to *sequence*[*s* + 23] are equal to 0x68 0x74 0x74 0x70 0x3A 0x2F 0x2F 0x77 0x77 0x77 0x2E 0x77 0x33 0x2E 0x6F 0x72 0x67 0x2F

0x31 0x39 0x39 0x39 0x2F 0x30 0x32 0x2F 0x32 0x32 0x2D 0x72 0x64 0x66 0x2D 0x73 0x79 0x6E 0x74 0x61 0x78 0x2D 0x6E 0x73 0x23 ("`http://www.w3.org/1999/02/22-rdf-syntax-ns#`"), increment *s* by 24 and enter loop *N*:

1. If *sequence*[*s*] is undefined, the computed MIME type is the supplied MIME type.

   Abort these steps.

2. If *length* is greater than or equal to *s* + 43 and the forty-three bytes from *sequence*[*s*] to *sequence*[*s* + 42] are equal to 0x68 0x74 0x74 0x70 0x3A 0x2F 0x2F 0x70 0x75 0x72 0x6C 0x2E 0x6F 0x72 0x67 0x2F 0x72 0x73 0x73 0x2F 0x31 0x2E 0x30 0x2F ("`http://purl.org/rss/1.0/`"), the computed MIME type is "`application/rss+xml`".

   Abort these steps.

3. Increment *s* by 1.

4. Increment *s* by 1.

8. The computed MIME type is the supplied MIME type.

   Abort these steps.

6. The computed MIME type is the supplied MIME type.

Note

It might be more efficient for the user agent to implement the rules for distinguishing if a resource is a feed or HTML in parallel with its algorithm for detecting the character encoding of an HTML document.

§ 8. Context-specific sniffing

A **context** is ....

In certain contexts, it is only useful to identify resources that belong to a certain subset of MIME types.

In such contexts, it is appropriate to use a context-specific sniffing algorithm in place of the MIME type sniffing algorithm in order to determine the computed MIME type of a resource.

A **context-specific sniffing algorithm** determines the computed MIME type of a resource only if the resource is a MIME type relevant to a particular context.

§ 8.1. Sniffing in a browsing context

Use the MIME type sniffing algorithm.

§ 8.2. Sniffing in an image context

To determine the computed MIME type of a resource with an image MIME type, execute the following **rules for sniffing images specifically**:

1. If the supplied MIME type is an XML MIME type, the computed MIME type is the supplied MIME type.

   Abort these steps.

2. Let *image-type-matched* be the result of executing the image type pattern matching algorithm with the resource header as the byte sequence to be matched.

3. If *image-type-matched* is not undefined, the computed MIME type is *image-type-matched*.

   Abort these steps.

4. The computed MIME type is the supplied MIME type.

§ 8.3. Sniffing in an audio or video context

To determine the computed MIME type of a resource with an audio or video MIME type, execute the following **rules for sniffing audio and video specifically**:

1. If the supplied MIME type is an XML MIME type, the computed MIME type is the supplied MIME type.

Abort these steps.

2. Let *audio-or-video-type-matched* be the result of executing the audio or video type pattern matching algorithm with the resource header as the byte sequence to be matched.

3. If *audio-or-video-type-matched* is not undefined, the computed MIME type is *audio-or-video-type-matched*.

   Abort these steps.

4. The computed MIME type is the supplied MIME type.

§ 8.4. Sniffing in a plugin context

To determine the computed MIME type of a resource fetched in a plugin context, execute the following **rules for sniffing in a plugin context**:

1. If the supplied MIME type is undefined, the computed MIME type is "`application/octet-stream`".

2. The computed MIME type is the supplied MIME type.

§ 8.5. Sniffing in a style context

To determine the computed MIME type of a resource fetched in a style context, execute the following **rules for sniffing in a style context**:

1. If the supplied MIME type is undefined, ...

2. The computed MIME type is the supplied MIME type.

§ 8.6. Sniffing in a script context

To determine the computed MIME type of a resource fetched in a script context, execute the following **rules for sniffing in a script context**:

1. If the supplied MIME type is undefined, ...

2. The computed MIME type is the supplied MIME type.

§ 8.7. Sniffing in a font context

To determine the computed MIME type of a resource with a font MIME type, execute the following **rules for sniffing fonts specifically**:

1. If the supplied MIME type is an XML MIME type, the computed MIME type is the supplied MIME type.

   Abort these steps.

2. Let *font-type-matched* be the result of executing the font type pattern matching algorithm with the resource header as the byte sequence to be matched.

3. If *font-type-matched* is not undefined, the computed MIME type is *font-type-matched*.

   Abort these steps.

4. The computed MIME type is the supplied MIME type.

§ 8.8. Sniffing in a text track context

The computed MIME type is "`text/vtt`".

§ 8.9. Sniffing in a cache manifest context

The computed MIME type is "`text/cache-manifest`".

§ Acknowledgments

Special thanks to Adam Barth and Ian Hickson for maintaining previous incarnations of this document.

Thanks also to Alfred Hönes, Anne van Kesteren, Boris Zbarsky, Darien Maillet Valentine, David Singer, Domenic Denicola, Henri Sivonen, Jonathan Neal, Joshua Cranmer, Larry Masinter, 罗泽轩, Mariko Kosaka, Mark Pilgrim, Paul Adenot, Peter Occil, Rob Buis, Russ Cox, Simon Pieters, and triple-underscore for their invaluable contributions.

This standard is written by Gordon P. Hemsley (me@gphemsley.org).

§ Intellectual property rights

This is the Living Standard. Those interested in the patent-review version should view the Living Standard Review Draft.

§ Index

§ Terms defined by this specification

- serialize a MIME type to bytes, in §4.5
- sniff-scriptable flag, in §7.1
- subtype, in §4.1
- supplied MIME type, in §5
- supplied MIME type detection algorithm, in §5.1
- supported by the user agent, in §4.2
- tag-terminating byte, in §3
- type, in §4.1
- valid MIME type string, in §4.3
- valid MIME type string with no parameters, in §4.3
- whitespace byte, in §3
- XML MIME type, in §4.6
- ZIP-based MIME type, in §4.6

§ Terms defined by reference

- [FETCH] defines the following terms:
  - collecting an http quoted string
  - fetch
  - http whitespace
- [HTML] defines the following terms:
  - script
  - type
- [HTTP] defines the following terms:
  - media-type
  - quoted-string
  - token
- [INFRA] defines the following terms:
  - ascii alphanumeric
  - ascii case-insensitive
  - ascii lowercase
  - ascii string
  - ascii whitespace
  - break
  - byte
  - byte sequence
  - code point
  - collect a sequence of code points
  - collecting a sequence of code points
  - contain
  - continue
  - exist
  - for each
  - isomorphic decode
  - isomorphic encode
  - key
  - length
  - ordered map
  - position variable
  - set (for map)
  - skip ascii whitespace
  - string
  - value

§ References

§ Normative References

[ENCODING]

Anne van Kesteren. Encoding Standard. Living Standard. URL: https://encoding.spec.whatwg.org/

[FETCH]

Anne van Kesteren. Fetch Standard. Living Standard. URL: https://fetch.spec.whatwg.org/

[FTP]

J. Postel; J. Reynolds. File Transfer Protocol. October 1985. Internet Standard. URL: https://tools.ietf.org/html/rfc959

[HTTP]

R. Fielding, Ed.; J. Reschke, Ed.. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. June 2014. Proposed Standard. URL: https://httpwg.org/specs/rfc7230.html

[INFRA]

Anne van Kesteren; Domenic Denicola. Infra Standard. Living Standard. URL: https://infra.spec.whatwg.org/

[MIMETYPE]

N. Freed; N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types. November 1996. Draft Standard. URL: https://tools.ietf.org/html/rfc2046

[RFC2119]

S. Bradner. Key words for use in RFCs to Indicate Requirement Levels. March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119

[RFC7231]

R. Fielding, Ed.; J. Reschke, Ed.. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. June 2014. Proposed Standard. URL: https://httpwg.org/specs/rfc7231.html

[RFC7303]

H. Thompson; C. Lilley. XML Media Types. July 2014. Proposed Standard. URL: https://tools.ietf.org/html/rfc7303

[RFC8081]

C. Lilley. The "font" Top-Level Media Type. February 2017. Proposed Standard. URL: https://tools.ietf.org/html/rfc8081

[SECCONTSNIFF]

Adam Barth; Juan Caballero; Dawn Song. Secure Content Sniffing for Web Browsers, or How to Stop Papers from Reviewing Themselves. URL: https://www.adambarth.com/papers/2009/barth-caballero-song.pdf

§ Informative References

[HTML]

Anne van Kesteren; et al. HTML Standard. Living Standard. URL:

https://html.spec.whatwg.org/multipage/

[MEDIAQUERIES]

Florian Rivoal; Tab Atkins Jr.. Media Queries Level 4. URL: https://drafts.csswg.org/mediaqueries-4/