None	A. Barth	
Internet-Draft	I. Hickson	
Expires: July 3, 2011	Google, Inc.	
	December 30, 2010	

TOC

Media Type Sniffing
draft-ietf-websec-mime-sniff-00

Abstract

Many web servers supply incorrect Content-Type header fields with their HTTP responses. In order to be compatible with these servers, user agents consider the content of HTTP responses as well as the Content-Type header fields when determining the effective media type of the response. This document describes an algorithm for determining the effective media type of HTTP responses that balances security and compatibility considerations.
Please send feedback on this draft to apps-discuss@ietf.org.

Status of this Memo

Copyright Notice

Table of Contents

1. Introduction

TOC

The HTTP Content-Type header field indicates the media type of an HTTP response. However, many HTTP servers supply a Content-Type 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 to determine the effective media type of the response.

Without a clear specification of how to "sniff" the media type, each user agent implementor was forced to reverse engineer the behavior of the other user agents and to develop their own algorithm. These divergent algorithms have lead to a lack of interoperability between user agents and to security issues when the server intends an HTTP response to be interpreted as one media type but some user agents interpret the responses as another media type.

These security issues are most severe when an "honest" server lets potentially malicious users upload files and then serves the contents of those files with a low-privilege media type (such as text/plain or image/jpeg). (Malicious servers, of course, can specify an arbitrary media type in the Content-Type header field.) In the absence of media type sniffing, this user-generated content would not be interpreted as a high-privilege media type, such as text/html. However, if a user agent does interpret a low-privilege media type, such as image/gif, as a high-privilege media type, such as text/html, the user agent has created a privilege escalation vulnerability in the server. For example, a malicious user might be able to leverage content sniffing to mount a cross-site script attack by including JavaScript code in the uploaded file that a user agent treats as text/html.

This document describes a content sniffing algorithm that carefully balances the compatibility needs of user agent implementors with the security constraints. The algorithm has been constructed with reference to 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 [BarthCaballeroSong2009] (Barth, A., Caballero, J., and D. Song, "Secure Content Sniffing for Web Browsers, or How to Stop Papers from Reviewing Themselves," 2009.).

WARNING! Whenever possible, user agents SHOULD NOT employ a content sniffing algorithm. However, if a user agent does employ a content sniffing algorithm, the user agent SHOULD use the algorithm in this document because using a different content sniffing algorithm than servers expect causes security problems. 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.

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

---

2. Metadata

TOC

The explicit media type metadata information associated with sequence of octets depends on the protocol that was used to fetch the octets. For octets received via HTTP, the Content-Type HTTP header field, if present, indicates the media type. Let the official-type be the media type indicted by the HTTP Content-Type header field, if present. If the Content-Type header field is absent or if its value cannot be interpreted as a media type (e.g. because its value doesn't contain a U+002F SOLIDUS ('/') character), then there is no official-type.

> Note: If an HTTP response contains multiple Content-Type header fields, the user agent MUST use the textually last Content-Type header field to the official-type. For example, if the last Content-Type header field contains the value "foo", then there is no official media type because "foo" cannot be interpreted as a media type (even if the HTTP response contains another Content-Type header field that could be interpreted as a media type).

For octets fetched from the file system, user agents should use platform-specific conventions (e.g., operating system file extension/ type mappings) to determine the official-type.

> Note: It is essential that file extensions are not used for determining the media type for octets fetched over HTTP because, in some cases, file extensions can be supplied by malicious parties. For example, most PHP installations let the attacker append

> arbitrary path information to URLs (e.g., http://example.com/
> foo.php/bar.html) and thereby determine the file extension.

For octets fetched over some other protocols, e.g. FTP, there is no type information.
Note: Comparisons between media types, as defined by MIME specifications, are done in an ASCII case-insensitive manner. [RFC2046]

---

3. Web Pages

TOC

The user agent MUST use the following algorithm to determine the sniffed-type of a sequence of octets:

1. If the user agent is configured to strictly obey the official-type, then let the sniffed-type be the official-type and abort these steps.

2. If the octets were fetched via HTTP and there is an HTTP Content-Type header field and the value of the last such header field has octets that *exactly* match the octets contained in one of the following lines:

Bytes in Hexadecimal	Textual Representation
74 65 78 74 2f 70 6c 61 69 6e	text/plain
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

   ...then jump to the "text or binary" section below.

3. If there is no official-type, jump to the "unknown type" section below.

4. If the official-type is "unknown/unknown", "application/unknown", or "*/*", jump to the "unknown" type section below.

5. If the official-type ends in "+xml", or if it is either "text/xml" or "application/xml", then let the sniffed-type be the official-type and abort these steps.

6. If the official-type is an image type supported by the user agent (e.g., "image/png", "image/gif", "image/jpeg", etc), then jump to the "images" section below.

7. If the official-type is "text/html", then jump to the "feed or HTML" section below.

8. Let the sniffed-type be the official type.

---

4. Text or Binary

TOC

This section defines the *rules for distingushing if a resource is text or binary*.

1. The user agent MAY wait for 512 or more octets be to arrive.

   > Note: Waiting for 512 octets octets to arrive causes the text-or-binary algorithm to be deterministic for a given sequence of octets. However, in some cases, the user agent might need to wait an arbitrary length of time for these octets to arrive. User agents SHOULD wait for 512 octets to arrive, when feasible.

2. Let n be the smaller of either 512 or the number of octets that have already arrived.

3. If n is greater than or equal to 3, and the first 2 or 3 octets match one of the following octet sequences:

Bytes in Hexadecimal	Description
FE FF	UTF-16BE BOM
FF FE	UTF-16LE BOM
EF BB BF	UTF-8 BOM

...then let the sniffed-type be "text/plain" and abort these steps.

4. If none of the first n octets are binary data octets then let the sniffed-type be "text/plain" and abort these steps.

Binary Data Byte Ranges
0x00 -- 0x08
0x0B
0x0E -- 0x1A
0x1C -- 0x1F

5. If the first octets match one of the octet sequences in the "pattern" column of the table in the "unknown type" section below, ignoring any rows whose cell in the "security" column says "scriptable" (or "n/a"), then let the sniffed-type be the type given in the corresponding cell in the "sniffed type" column on that row and abort these steps.

> WARNING! It is critical that this step not ever return a scriptable type (e.g., text/html), because otherwise that would allow a privilege escalation attack.

6. Otherwise, let the sniffed-type be "application/octet-stream" and abort these steps.

---

5. Unknown Type

TOC

1. The user agent MAY wait for 512 or more octets to arrive for the same reason as in the "text or binary" section above.

2. Let n be the smaller of either 512 or the number of octets that have already arrived.

3. For each row in the table below:

   *If the row has no "WS" octets:

      1. Let pattern-length be the length of the pattern.

      2. If n is smaller than pattern-length then skip this row.

3. Apply the bit-wise "and" operator to the first pattern-length octets and the given mask, and let the result be the masked-data.

4. If the octets of the masked-data matches the given pattern octets exactly, then let the sniffed-type be the type given in the cell of the third column in that row and abort these steps.

*If the row has a "WS" octet or a "_>" octet:

1. Let index-pattern be an index into the mask and pattern octet strings of the row.

2. Let index-stream be an index into the octet stream being examined.

3. LOOP: If index-stream points beyond the end of the octet stream, then this row doesn't match and skip this row.

4. Examine the index-stream-th octet of the octet stream as follows:

-If the index-pattern-th octet of the pattern is a normal hexadecimal octet and not a "WS" octet or a "SB" octet:

> If the bit-wise "and" operator, applied to the index-stream-th octet of the stream and the index-pattern-th octet of the mask, yield a value different than the index-pattern-th octet of the pattern, then skip this row.
>
> Otherwise, increment index-pattern to the next octet in the mask and pattern and index-stream to the next octet in the octet stream.

-Otherwise, if the index-pattern-th octet of the pattern is a "WS" octet:

> "WS" means "whitespace", and allows insignificant whitespace to be skipped when sniffing for a type signature.
>
> If the index-stream-th octet of the stream is one of 0x09 (ASCII TAB), 0x0A (ASCII LF), 0x0C (ASCII FF), 0x0D (ASCII CR), or 0x20 (ASCII space), then increment only the index-stream to the next octet in the octet stream.

> Otherwise, increment only the index-pattern to the next octet in the mask and pattern.

- Otherwise, if the index-pattern-th octet of the pattern is a "_>" octet:

  > "_>" means "space-or-bracket", and allows HTML tag names to terminate with either a space or a greater than sign.
  >
  > If index-stream-th octet of the stream different than 0x20 (ASCII space) or 0x3E (ASCII ">"), then skip this row.
  >
  > Otherwise, increment index-pattern to the next octet in the mask and pattern and index-stream to the next octet in the octet stream.

5. If index-pattern does not point beyond the end of the mask and pattern octet strings, then jump back to the LOOP step in this algorithm.

6. Otherwise, let the sniffed-type be the type given in the cell of the third column in that row and abort these steps.

4. If none of the first n octets are binary data (as defined in the "text or binary" section), then let the sniffed-type be "text/plain" and abort these steps.

5. Otherwise, let the sniffed-type be "application/octet-stream" and abort these steps.

The table used by the above algorithm is:

Mask in Hex	Pattern in Hex	Sniffed Type	Security
FF FF FF DF DF DF DF DF DF DF FF DF DF DF DF FF	WS 3C 21 44 4F 43 54 59 50 45 20 48 54 4D 4C _>	text/html	Scriptable
Comment: <!DOCTYPE HTML			
FF FF DF DF DF DF FF	WS 3C 48 54 4D 4C _>	text/html	Scriptable
Comment: <HTML			
FF FF DF DF DF DF FF	WS 3C 48 45 41 44 _>	text/html	Scriptable
Comment: <HEAD			
FF FF DF DF DF DF DF DF FF	WS 3C 53 43 52 49 50 54 _>	text/html	Scriptable
Comment: <SCRIPT			
FF FF DF DF DF DF DF DF FF	WS 3C 49 46 52 41 4d 45 _>	text/html	Scriptable
Comment: <IFRAME			
FF FF DF FF FF	WS 3C 48 31 _>	text/html	Scriptable
Comment: <H1			
FF FF DF DF DF FF	WS 3C 44 49 56 _>	text/html	Scriptable
Comment: <DIV			
FF FF DF DF DF DF FF	WS 3C 46 4f 4e 54 _>	text/html	Scriptable
Comment: <FONT			
FF FF DF DF DF DF DF FF	WS 3C 54 41 42 4c 45 _>	text/html	Scriptable
Comment: <TABLE			
FF FF DF FF	WS 3C 41 _>	text/html	Scriptable
Comment: <A			
FF FF DF DF DF DF DF FF	WS 3C 53 54 59 4c 45 _>	text/html	Scriptable
Comment: <STYLE			
FF FF DF DF DF DF DF FF	WS 3C 54 49 54 4c 45 _>	text/html	Scriptable
Comment: <TITLE			

FF FF DF FF	WS 3C 42 _>	text/html	Scriptable
Comment: <B			
FF FF DF DF DF DF	WS 3C 42 4f 44 59	text/html	Scriptable
FF	_>		
Comment: <BODY			
FF FF DF DF FF	WS 3C 42 52 _>	text/html	Scriptable
Comment: <BR			
FF FF DF FF	WS 3C 50 _>	text/html	Scriptable
Comment: <P			
FF FF FF FF FF FF	WS 3C 21 2d 2d _>	text/html	Scriptable
Comment: <!--			
FF FF FF FF FF FF	WS 3C 3f 78 6d 6c	text/xml	Scriptable
Comment: <?xml (Note the case sensitivity and lack of trailing _>)			
FF FF FF FF FF	25 50 44 46 2D	application/pdf	Scriptable
Comment: The string "%PDF-", the PDF signature.			
FF FF FF FF FF FF	25 21 50 53 2D 41	application/	Safe
FF FF FF FF FF	64 6F 62 65 2D	postscript	
Comment: The string "%!PS-Adobe-", the PostScript signature.			
FF FF 00 00	FE FF 00 00	text/plain	n/a
Comment: UTF-16BE BOM			
FF FF 00 00	FF FE 00 00	text/plain	n/a
Comment: UTF-16LE BOM			
FF FF FF 00	EF BB BF 00	text/plain	n/a
Comment: UTF-8 BOM			
FF FF FF FF FF FF	47 49 46 38 37 61	image/gif	Safe
Comment: The string "GIF87a", a GIF signature.			
FF FF FF FF FF FF	47 49 46 38 39 61	image/gif	Safe
Comment: The string "GIF89a", a GIF signature.			
FF FF FF FF FF FF	89 50 4E 47 0D 0A	image/png	Safe
FF FF	1A 0A		
Comment: The PNG signature.			
FF FF FF	FF D8 FF	image/jpeg	Safe
Comment: A JPEG SOI marker followed by a octet of another marker.			

FF FF	42 4D	image/bmp	Safe
Comment: The string "BM", a BMP signature.			
FF FF FF FF	00 00 01 00	image/vnd. microsoft.icon	Safe
Comment: A Windows Icon signature.			
FF FF FF FF FF FF FF	52 61 72 20 1A 07 00	application/ x-rar-compressed	Safe
Comment: A RAR archive.			
FF FF FF FF	50 4B 03 04	application/zip	Safe
Comment: A ZIP archive.			
FF FF FF	1F 8B 08	application/ x-gzip	Safe
Comment: A GZIP archive.			

User agents MAY support additional types if necessary, by implicitly adding to the above table. However, user agents SHOULD NOT not use any other patterns for types already mentioned in the table above because this could then be used for privilege escalation (where, e.g., a server uses the above table to determine that content is not HTML and thus safe from cross-site scripting attacks, but then a user agent detects it as HTML anyway and allows script to execute). In extending this table, user agents SHOULD NOT introduce any privilege escalation vulnerabilities.
Note: The column marked "security" is used by the algorithm in the "text or binary" section, to avoid sniffing text/plain content as a type that can be used for a privilege escalation attack.

---

6. Image

TOC

This section defines the *rules for sniffing images specifically*.
If the official-type is "image/svg+xml", then let the sniffed-type be the official-type (an XML type) and abort these steps.
If the first octets match one of the octet sequences in the first column of the following table, then let the sniffed-type be the type given in the corresponding cell in the second column on the same row and abort these steps:

Bytes in Hexadecimal	Sniffed Type	Comment
47 49 46 38 37 61	image/gif	"GIF87a"
47 49 46 38 39 61	image/gif	"GIF89a"
89 50 4E 47 0D 0A 1A 0A	image/png	
FF D8 FF	image/jpeg	
42 4D	image/bmp	"BM"
00 00 01 00	image/vnd.microsoft.icon	

Otherwise, let the sniffed-type be the official-type and abort these steps.

---

7. Feed or HTML

TOC

1. The user agent MAY wait for 512 or more octets to arrive for the same reason as in the "text or binary" section above.

2. Let s be the stream of octets, and let s[i] represent the octet in s with position i, treating s as zero-indexed (so the first octet is at i=0).

3. If at any point this algorithm requires the user agent to determine the value of a octet in s which has not yet arrived, or which is past the first 512 octets, or which is beyond the end of the octet stream, the algorithm stops and the sniffed-type is "text/html".

   > Note: User agents are allowed, by the first step of this algorithm, to wait until the first 512 octets have arrived.

4. Initialize pos to 0.

5. If s[0] equals 0xEF, s[1] equals 0xBB, and s[2] equals 0xBF, then set pos to 3. (This skips over a leading UTF-8 BOM, if any.)

6. LOOP: Examine s[pos].

   *If it equals 0x09 (ASCII tab), 0x20 (ASCII space), 0x0A (ASCII LF), or 0x0D (ASCII CR)

   > Increase pos by 1 and repeat this step.

   *If it equals 0x3C (ASCII "<")

> Increase pos by 1 and go to the next step.

*If it is anything else

> Let the sniffed-type be "text/html" and abort these steps.

7. If the octets with positions pos to pos+2 in s are exactly equal to 0x21, 0x2D, 0x2D respectively (ASCII for "!--"), then:

    1. Increase pos by 3.

    2. If the octets with positions pos to pos+2 in s are exactly equal to 0x2D, 0x2D, 0x3E respectively (ASCII for "-->"), then increase pos by 3 and jump back to the previous step (the step labeled loop start) in the overall algorithm in this section.

    3. Otherwise, increase pos by 1.

    4. Return to step 2 in these substeps.

8. If s[pos] equals 0x21 (ASCII "!"):

    1. Increase pos by 1.

    2. If s[pos] equals 0x3E, then increase pos by 1 and jump back to the step labeled LOOP in the overall algorithm in this section.

    3. Otherwise, return to step 1 in these substeps.

9. If s[pos] equals 0x3F (ASCII "?"):

    1. Increase pos by 1.

    2. If s[pos] and s[pos+1] equal 0x3F and 0x3E respectively, then increase pos by 1 and jump back to the step labeled LOOP in the overall algorithm in this section.

    3. Otherwise, return to step 1 in these substeps.

10. Otherwise, if the octets in s starting at pos match any of the sequences of octets in the first column of the following table, then the user agent MUST follow the steps given in the corresponding cell in the second column of the same row.

Bytes in Hexadecimal	Requirement	Comment
72 73 73	Let the sniffed-type be "application/rss+xml" and abort these steps.	rss
66 65 65 64	Let the sniffed-type be "application/atom+xml" and abort these steps.	feed
72 64 66 3A 52 44 46	Continue to the next step in this algorithm.	rdf:RDF

If none of the octet sequences above match the octets in s starting at pos, then let the sniffed-type be "text/html" and abort these steps.

11. Initialize RDF-flag to 0.

12. Initialize RSS-flag to 0.

13. If the octets with positions pos to pos+23 in s are exactly 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 respectively (ASCII for "http://purl.org/rss/1.0/"), then:

    1. Increase pos by 23.

    2. Set RSS-flag to 1.

14. If the octets with positions pos to pos+42 in s are exactly 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 respectively (ASCII for "http://www.w3.org/1999/02/22-rdf-syntax-ns#"), then:

    1. Increase pos by 42.

    2. Set RDF-flag to 1.

15. Increase pos by 1.

16. If RDF-flag is 1 and RSS-flag is 1, then let the sniffed-type be "application/rss+xml" and abort these steps.

17. If pos points beyond the end of the octet stream s, then continue to step 19 of this algorithm.

18. Jump back to step 13 of this algorithm.

19. Let the sniffed-type be "text/html" and abort these steps.

For efficiency reasons, implementations might wish to implement this algorithm and the algorithm for detecting the character encoding of HTML documents in parallel.

---

8. References

TOC

[BarthCaballeroSong2009]	Barth, A., Caballero, J., and D. Song, “Secure Content Sniffing for Web Browsers, or How to Stop Papers from Reviewing Themselves,” 2009.

Authors' Addresses

TOC

	Adam Barth
	Google, Inc.
Email:	ietf@adambarth.com
URI:	http://www.adambarth.com/
	Ian Hickson
	Google, Inc.
Email:	ian@hixie.ch
URI:	http://ln.hixie.ch/