Working Group	A. Barth	
Internet-Draft	U.C. Berkeley	
Expires: April 3, 2010	I. Hickson	
	Google, Inc.	
	September 30, 2009	

TOC

Content-Type Processing Model
draft-abarth-mime-sniff-03

Status of this Memo

Copyright Notice

Abstract

Many web servers supply incorrect Content-Type headers 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 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.

Table of Contents

1. Introduction

TOC

The HTTP Content-Type header 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 been tolerated these servers by examining the content of HTTP responses in addition to the Content-Type header 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 developed 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.) In the absense of mime 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 as 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 avoid employing 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 exactly 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.

---

2. Metadata

TOC

The explicit Content-Type metadata associated with the resource (the resource's type information) depends on the protocol that was used to fetch the resource.

For HTTP resources, only the last Content-Type HTTP header, if any, contributes any type information; the official type of the resource is then the value of that header, interpreted as described by the HTTP specifications. If the Content-Type HTTP header is present but the value of the last such header cannot be interpreted as described by the HTTP specifications (e.g. because its value doesn't contain a U+002F SOLIDUS ('/') character), then the resource has no type information (even if there are multiple Content-Type HTTP headers and one of the other ones is syntactically correct).

For resources fetched from the file system, user agents should use platform-specific conventions, e.g. operating system file extension/type mappings.

> Note: It is essential that file extensions are not used for determining the media type for resources fetched over HTTP because file extensions can often by supplied by malicious parties.

For resources fetched over most other protocols, e.g. FTP, there is no type information.

The algorithm for extracting an encoding from a Content-Type, given a string s, is as follows. It either returns an encoding or nothing.

1. Find the first seven characters in s that are an ASCII case-insensitive match for the word "charset". If no such match is found, return nothing.

2. Skip any U+0009, U+000A, U+000C, U+000D, or U+0020 characters that immediately follow the word 'charset' (there might not be any).

3. If the next character is not a U+003D EQUALS SIGN ('='), return nothing.

4. Skip any U+0009, U+000A, U+000C, U+000D, or U+0020 characters that immediately follow the equals sign (there might not be any).

5. Process the next character as follows:

   *If it is a U+0022 QUOTATION MARK ('"') and there is a later U+0022 QUOTATION MARK ('"') in s, or

   *If it is a U+0027 APOSTROPHE ("'") and there is a later U+0027 APOSTROPHE ("'") in s

   > Return the string between this character and the next earliest occurrence of this character.

   *If it is an unmatched U+0022 QUOTATION MARK ('"'),

   *If it is an unmatched U+0027 APOSTROPHE ("'"), or

   *If there is no next character

   > Return nothing.

   *Otherwise

   > Return the string from this character to the first U+0009, U+000A, U+000C, U+000D, U+0020, or U+003B character or the end of s, whichever comes first.

Note: The above algorithm is a willful violation of the HTTP specification. [RFC2616]

TOC

3. Web Pages

The /sniffed type/ of a resource is found as follows:

1. Let /official type/ be the type given by the Content-Type metadata for the resource, ignoring parameters. Comparisons with this type, as defined by MIME specifications, are done in an ASCII case-insensitive manner. [RFC2046]

2. If the user agent is configured to strictly obey Content-Type headers for this resource, then jump to the last step in this set of steps.

3. If the resource was fetched over an HTTP protocol and there is an HTTP Content-Type header and the value of the last such header has bytes that exactly match 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.

4. If there is no /official type/, jump to the unknown type step below.

5. If /official type/ is "unknown/unknown", "application/unknown", or "*/*", jump to the unknown type step below.

6. If /official type/ ends in "+xml", or if it is either "text/xml" or "application/xml", then the /sniffed type/ of the resource is /official type/; return that and abort these steps.

7. If /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, passing it the /official type/.

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

9. The /sniffed type/ of the resource is /official type/.

---

4. Text or Binary

TOC

1. The user agent MAY wait for 512 or more bytes of the resource to be available.

2. Let n be the smaller of either 512 or the number of bytes already available.

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

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

...then the /sniffed type/ of the resource is "text/plain". Abort these steps.

4. If none of the first n bytes of the resource are binary data bytes then the /sniffed type/ of the resource is "text/plain". Abort these steps.

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

5. If the first bytes of the resource match one of the byte 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 the /sniffed type/ of the resource is the type given in the corresponding cell in the "sniffed type" column on that row; abort these steps.

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

6. Otherwise, the /sniffed type/ of the resource is "application/octet-stream".

---

5. Unknown Type

TOC

1. The user agent MAY wait for 512 or more bytes of the resource to be available.

2. Let /stream length/ be the smaller of either 512 or the number of bytes already available.

3. For each row in the table below:

   *If the row has no "WS" bytes:

      1. Let /pattern length/ be the length of the pattern (number of bytes described by the cell in the second column of the row).

      2. If /stream length/ is smaller than /pattern length/ then skip this row.

      3. Apply the "and" operator to the first /pattern length/ bytes of the resource and the given mask (the bytes in the cell of first column of that row), and let the result be the data.

      4. If the bytes of the data matches the given pattern bytes exactly, then the /sniffed type/ of the resource is the type given in the cell of the third column in that row; abort these steps.

*If the row has a "WS" byte:

1. Let /index pattern/ be an index into the mask and pattern byte strings of the row.

2. Let /index stream/ be an index into the byte stream being examined.

3. Loop: If /index stream/ points beyond the end of the byte stream, then this row doesn't match, skip this row.

4. Examine the /index stream/th byte of the byte stream as follows:

   -If the /index pattern/th byte of the pattern is a normal hexadecimal byte and not a "WS" byte:

   > If the "and" operator, applied to the /index stream/th byte of the stream and the /index pattern/th byte of the mask, yield a value different that the /index pattern/th byte of the pattern, then skip this row.
   >
   > Otherwise, increment /index pattern/ to the next byte in the mask and pattern and /index stream/ to the next byte in the byte stream.

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

   > "WS" means "whitespace", and allows insignificant whitespace to be skipped when sniffing for a type signature.
   >
   > If the /index stream/th byte 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 byte in the byte stream.
   >
   > Otherwise, increment only the /index pattern/ to the next byte in the mask and pattern.

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

6. Otherwise, the /sniffed type/ of the resource is the type given in the cell of the third column in that row; abort these steps.

4. If none of the first n bytes of the resource are binary data bytes then the sniffed type of the resource is "text/plain". Abort these steps.

5. Otherwise, the sniffed type of the resource is "application/octet-stream".

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	WS 3C 21 44 4F 43 54 59 50 45 20 48 54 4D 4C	text/html	Scriptable
Comment: "<!DOCTYPE HTML", case-insensitive, with leading spaces.			
FF FF DF DF DF DF	WS 3C 48 54 4D 4C	text/html	Scriptable
Comment: "<HTML", case-insensitive, with leading spaces.			
FF FF DF DF DF DF	WS 3C 48 45 41 44	text/html	Scriptable
Comment: "<HEAD", case-insensitive, with leading spaces.			
FF FF DF DF DF DF DF DF	WS 3C 53 43 52 49 50 54	text/html	Scriptable
Comment: "<SCRIPT", case-insensitive, with leading spaces.			
FF FF DF DF DF DF DF DF	WS 3C 49 46 52 41 4d 45	text/html	Scriptable
Comment: "<IFRAME", case-insensitive, with leading spaces.			
FF FF DF FF	WS 3C 48 31	text/html	Scriptable
Comment: "<H1", case-insensitive, with leading spaces.			
FF FF DF DF DF	WS 3C 44 49 56	text/html	Scriptable
Comment: "<DIV", case-insensitive, with leading spaces.			
FF FF DF DF DF DF	WS 3C 46 4f 4e 54	text/html	Scriptable
Comment: "<FONT", case-insensitive, with leading spaces.			
FF FF DF DF DF DF DF	WS 3C 54 41 42 4c 45	text/html	Scriptable
Comment: "<TABLE", case-insensitive, with leading spaces.			
FF FF DF	WS 3C 41	text/html	Scriptable
Comment: "<A", case-insensitive, with leading spaces.			
FF FF DF DF DF DF DF	WS 3C 53 54 59 4c 45	text/html	Scriptable
Comment: "<STYLE", case-insensitive, with leading spaces.			
FF FF DF DF DF DF DF	WS 3C 54 49 54 4c 45	text/html	Scriptable
Comment: "<TITLE", case-insensitive, with leading spaces.			
FF FF DF	WS 3C 42	text/html	Scriptable
Comment: "<B", case-insensitive, with leading spaces.			

FF FF DF DF DF DF	WS 3C 42 4f 44 59	text/html	Scriptable
Comment: "<BODY", case-insensitive, with leading spaces.			
FF FF DF DF	WS 3C 42 52	text/html	Scriptable
Comment: "<BR", case-insensitive, with leading spaces.			
FF FF DF	WS 3C 50	text/html	Scriptable
Comment: "<P", case-insensitive, with leading spaces.			
FF FF FF FF FF	WS 3C 21 2d 2d	text/html	Scriptable
Comment: The string "<!--", an HTML comment, with leading spaces.			
FF FF FF FF FF FF	WS 3C 3f 78 6d 6c	text/xml	Scriptable
Comment: The string "<?xml", case-sensitive, with leading spaces.			
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 byte 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.	Safe

			microsoft.icon	
Comment: A Windows Icon signature.				
FF FF FF FF FF FF	52 61 72 20 1A 07	application/	Safe	
FF	00	x-rar-compressed		
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 desired, by implicitly adding to the above table. However, user agents should 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 scriping attacks, but then a user agent detects it as HTML anyway and allows script to execute).
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

If the resource's /official type/ is "image/svg+xml", then the /sniffed type/ of the resource is its /official type/ (an XML type).
Otherwise, if the first bytes of the resource match one of the byte sequences in the first column of the following table, then the /sniffed type/ of the resource is the type given in the corresponding cell in the second column on the same row:

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, the /sniffed type/ of the resource is the same as its /official type/.

---

7. Feed or HTML

TOC

1. The user agent MAY wait for 512 or more bytes of the resource to be available.

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

3. If at any point this algorithm requires the user agent to determine the value of a byte in s which is not yet available, or which is past the first 512 bytes of the resource, or which is beyond the end of the resource, the algorithm stops and the /sniffed type/ of the resource is "text/html".

   > Note: User agents are allowed, by the first step of this algorithm, to wait until the first 512 bytes of the resource are available.

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 start: 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

     > The sniffed type of the resource is "text/html". Abort these steps.

7. If the bytes 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 bytes 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 start 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 start in the overall algorithm in this section.

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

10. Otherwise, if the bytes in s starting at pos match any of the sequences of bytes 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	The /sniffed type/ of the resource is "application/rss+xml"; abort these steps.	rss
66 65 65 64	The /sniffed type/ of the resource is "application/atom+xml"; 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 byte sequences above match the bytes in s starting at pos, then the /sniffed type/ of the resource is "text/html". Abort these steps.

11. Initialize /RDF flag/ to 0.

12. Initialize /RSS flag/ to 0.

13. If the bytes 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 bytes 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 the /sniffed type/ of the resource is "application/rss+xml". Abort these steps.

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

18. Jump back to step 13 of this algorithm.

19. The /sniffed type/ of the resource is "text/html".

For efficiency reasons, implementations may 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.

TODO: * Transcribe the tables into C and auto generate the tables. * Investigate charset parsing.

Authors' Addresses

TOC

	Adam Barth
	University of California, Berkeley
Email:	abarth@eecs.berkeley.edu
URI:	http://www.adambarth.com/
	Ian Hickson
	Google, Inc.
Email:	ian@hixie.ch
URI:	http://ln.hixie.ch/