Encoding Differentials: Why Charset Matters

Encoding Differentials: Why Charset Matters - Stefan Schiller, Sonar.

- Published: 2024-07-15
- Original: <https://www.sonarsource.com/blog/encoding-differentials-why-charset-matters/>
- Preserved from: https://www.sonarsource.com/blog/encoding-differentials-why-charset-matters/ (live) on 2026-08-10
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

[[image: sonar logo]](https://www.sonarsource.com/)

TL;DR overview

- Encoding differentials occur when different layers of a web application interpret the same byte sequence under different character sets, creating security gaps that bypass input validation.
- Charset mismatches between the browser, server, and database can allow attackers to smuggle malicious payloads—such as XSS or SQL injection—past filters that rely on consistent encoding.
- Developers should explicitly declare character sets at every layer of the stack and avoid relying solely on content-type headers, which can be overridden or ignored by certain browsers.
- Sonar's static analysis can detect charset-related misconfigurations in source code before they become exploitable vulnerabilities in production.

Do you notice something in the following HTTP response?

Copy to clipboard

```
HTTP/1.1 200 OK
Server: Some Server
Content-Type: text/html
Content-Length: 1337

<!DOCTYPE html>
<html>
<head><title>Some Page</title></head>
<body>
...
```

Based on this small portion of the HTTP response, you can assume that this web application is **likely prone to an XSS vulnerability**.

How is this possible? Did you notice something?

If you have doubts about the `Content-Type` header, you are right. There is only a minor imperfection here: the header is **missing** a `charset` attribute. This does not sound like a big deal, however, this blog post will explain how attackers can exploit this to inject arbitrary JavaScript code into a website by **consciously changing the character set** that the browser assumes.

This blog post's content was also presented at the TROOPERS24 conference. A recording of the talk can be found here: From ASCII to UTF-16: Leveraging Encodings to Break Software.

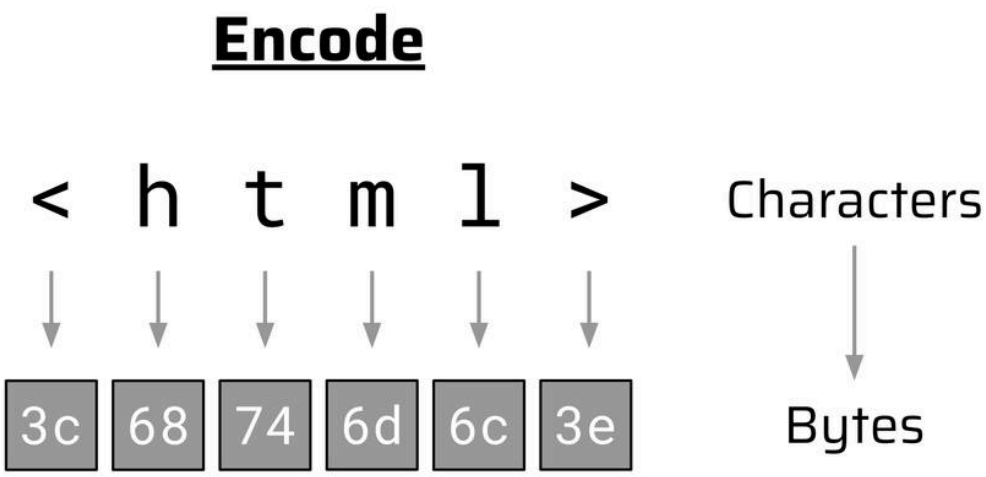
Character Encodings

A common `Content-Type` header in an HTTP response looks like this:

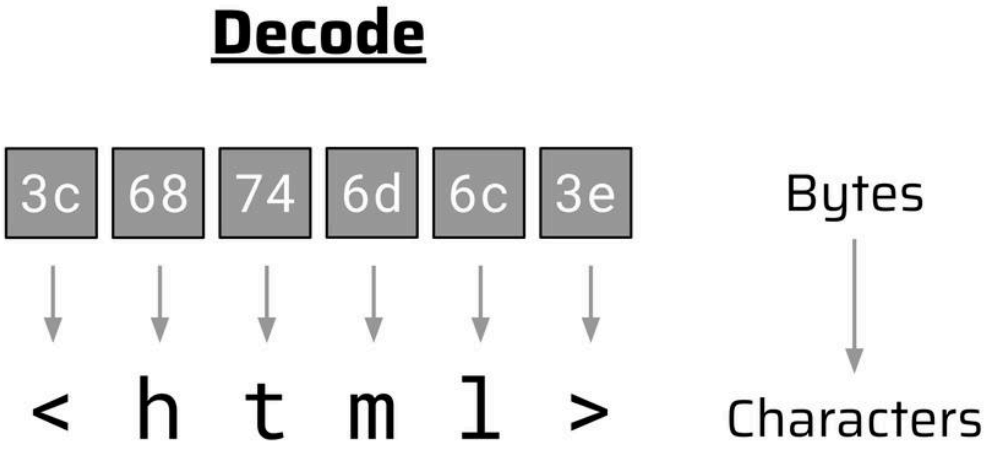
Copy to clipboard

```
HTTP/1.1 200 OK
Server: Some Server
Content-Type: text/html; {% mark yellow %}charset=utf-8{% mark %}
...
```

The `charset` attribute tells the browser that UTF-8 was used to encode the HTTP response body. A character encoding like UTF-8 defines a **mapping between characters and bytes**. When a web server serves an HTML document, it maps the characters of the document to the corresponding bytes and transmits these in the HTTP response body. This process turns characters into bytes (*encode*):



When the browser receives these bytes in the HTTP response body, it can translate them back to the characters of the HTML document. This process turns bytes into characters (*decode*):



UTF-8 is only one of **many character encodings** that a modern browser must support according to the HTML spec. There are plenty of others like `UTF-16`, `ISO-8859-xx`, `windows-125x`, `GBK`, `Big5`, etc. It is essential that the browser knows which of those encodings the server used or it **cannot properly decode** the bytes in the HTTP response body.

But what if there is no `charset` attribute in the `Content-Type` header or it is invalid?

In that case, the browser looks for a `<meta>` tag in the HTML document itself. This tag can also have a `charset` attribute that indicates the character encoding (e.g., `<meta charset="UTF-8">`). This is already an act of balance for the browser: In order to read the HTML document, it needs to decode the HTTP response body. Thus, it needs to assume *some* encoding, decode the HTTP body, look for a `<meta>` tag, and possibly re-decode the body with the indicated character encoding.

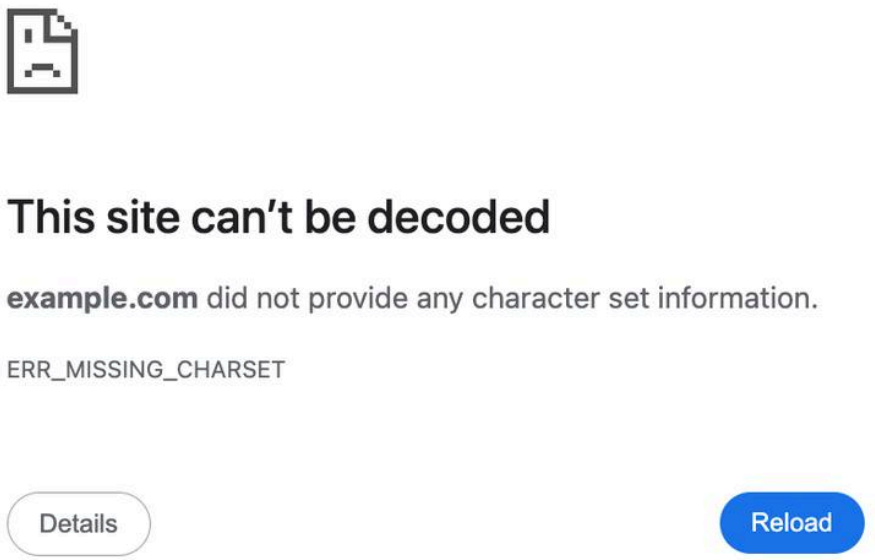
Another, less common way to indicate the character encoding is the Byte-Order Mark. This is a specific Unicode character (`U+FEFF`) that can be placed in front of a string to indicate the byte endianness and character encoding. It is mainly used in files, but since these might be served via a web server, modern browsers support it. A Byte-Order Mark at the beginning of an HTML document even takes precedence over a `charset` attribute in the `Content-Type` header and the `<meta>` tag.

In summary, there are three common ways that a browser uses to **determine the character encoding** of an HTML document, ordered by priority:

- Byte-Order Mark at the beginning of the HTML document
- `charset` attribute in the `Content-Type` header
- `<meta>` tag in the HTML document

Missing Charset Information

The Byte-Order Mark is generally very uncommon and the `charset` attribute is not always present in a `Content-Type` header or might be invalid. Also - especially for partial HTML responses - there is usually no `<meta>` tag that indicates a character encoding. In these cases, the browser does not have any information about what character set to use:



Have you ever seen this error message? Probably not, because **it does not exist**.

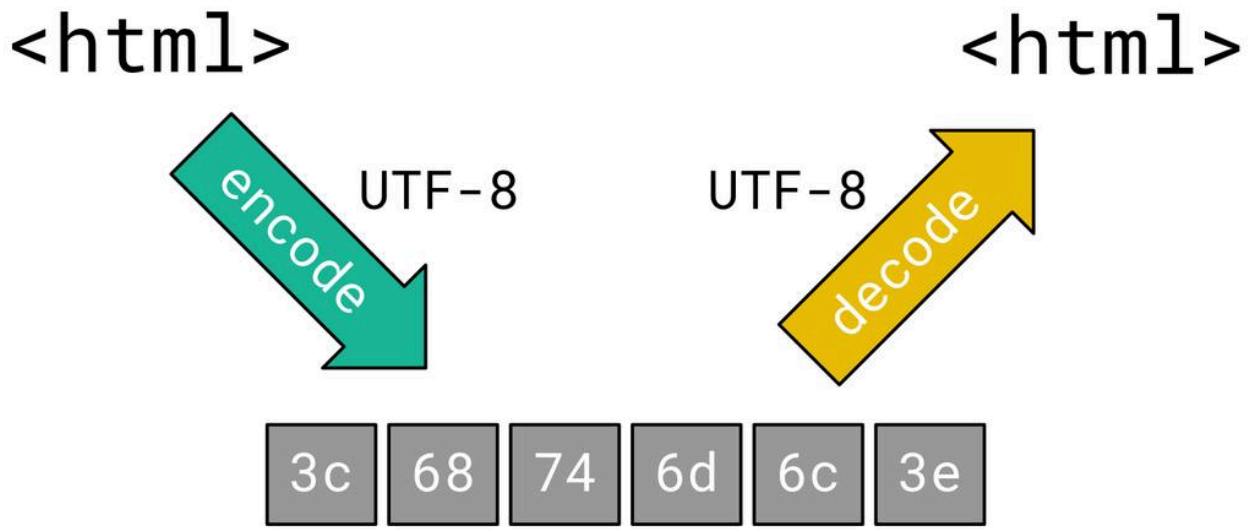
Similar to faulty HTML syntax, browsers try to recover from missing character set information when parsing the content served from a web server and **make the best of it**. This non-strict behavior contributes to a good user experience, but it may also **open doors for exploitation techniques** like mXSS.

For missing character information, browsers try to make an educated guess based on the content, which is called auto-detection. This is similar to MIME-type sniffing but operates on a character encoding level. Chromium's rendering engine Blink, for example, uses the Compact Encoding Detection (CED) library to automatically detect the character encoding. From an attacker's point of view, the **auto-detection feature is very powerful** as we will see.

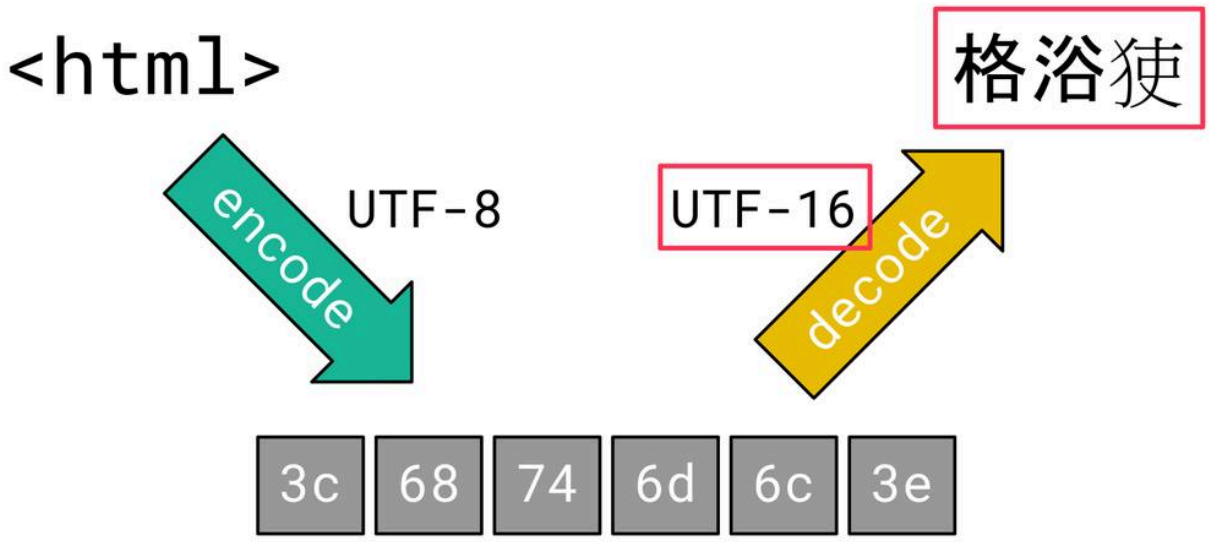
At this point, we are familiar with the different mechanisms a browser may use to determine the character encoding of an HTML document. But how could attackers exploit this?

Encoding Differentials

The purpose of character encoding is to translate characters into a computer-processable byte sequence. These bytes can be transmitted over a network and decoded back to characters by the receiver. This way, the **exact same characters** that the sender intended to transmit are restored:



This only works fine, when the sender and receiver agree upon the character encoding they use. If there is a **mismatch** between the character encoding used for encoding and decoding, the receiver may *see* different characters:



Such a mismatch between the character encoding used for encoding and decoding is what we refer to as *Encoding Differential* here.

For a web application, this becomes vital when user-controlled data is sanitized to prevent Cross-Site Scripting (XSS) vulnerabilities. If the character encoding that the browser assumes is different from what the web server intended, this could theoretically break the sanitization and lead to XSS vulnerabilities.

This itself is no big news and even Google was prone to an issue like this back in 2005. Google's 404 page did not provide charset information, which could be exploited by inserting a UTF-7 XSS

payload. In UTF-7, HTML special characters like angle brackets are encoded differently from ASCII which can be leveraged to bypass sanitization:

Copy to clipboard

```
+ADw-script+AD4-alert(1)+ADw-+AC8-script+AD4-
```

This greatly demonstrated the dangers of this encoding, which was deprecated in the following years to prevent security issues like this. Nowadays, the HTML spec even explicitly forbids the usage of UTF-7 to prevent XSS vulnerabilities.

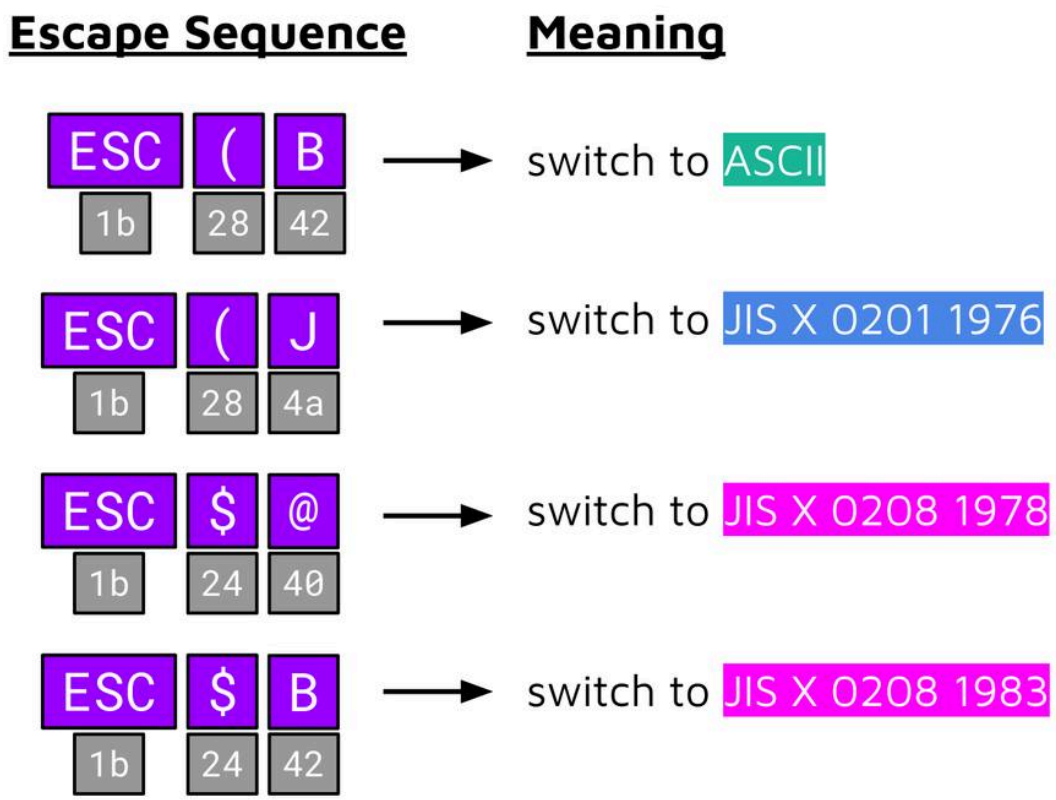
Although there are still a lot of other supported character encodings, most of these are not really useful from an attacker's point of view. All **HTML special characters** like angle brackets and quotes are **ASCII only** and since most character encodings are ASCII-compatible, there is **no difference** for these characters. Even for UTF-16, which is not ASCII-compatible due to its fixed amount of two bytes per character, it is usually not possible to smuggle ASCII characters, because their corresponding byte representation is the same, just with a trailing (little-endian) or leading (big-endian) zero byte.

However, there is a particularly interesting encoding: **ISO-2022-JP**.

ISO-2022-JP

ISO-2022-JP is a Japanese character encoding defined in RFC 1468. It is one of the official character encodings that user agents must support, as defined by the HTML standard. Particularly interesting about this encoding is that it supports certain **escape sequences** to **switch between different character sets**.

For example, if a byte sequence contains the bytes `0x1b`, `0x28`, `0x42`, these bytes are not decoded to a character but instead indicate that all following bytes should be decoded using ASCII. In total, there are four different escape sequences that can be used to switch between the character sets ASCII, JIS X 0201 1976, JIS X 0208 1978 and JIS X 0208 1983:

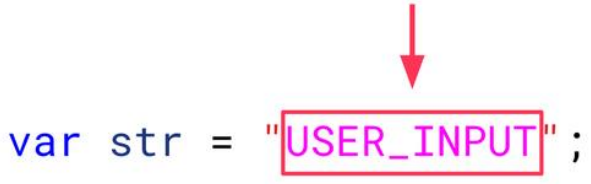


This feature of ISO-2022-JP not only provides great flexibility but can also break fundamental assumptions. And there is another catch: at the time of writing, **Chrome (Blink) and Firefox (Gecko) auto-detect this encoding. **A single occurrence of one of these escape sequences is usually enough to convince the auto-detection algorithm that the HTTP response body is encoded with ISO-2022-JP.

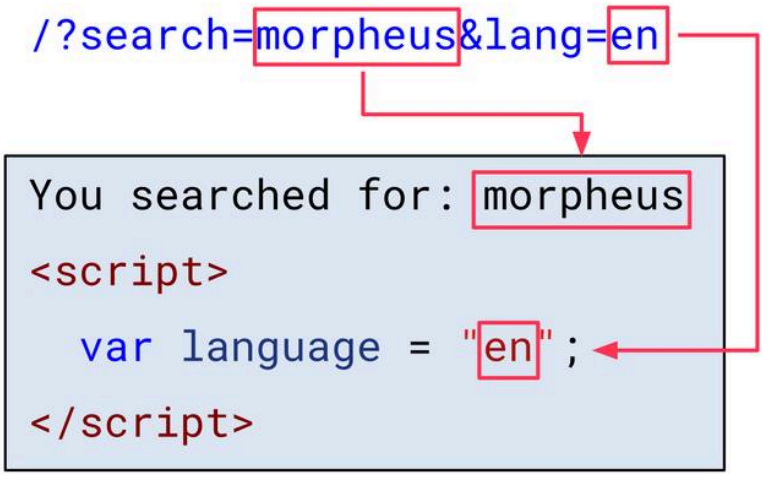
The following sections explain two different exploitation techniques that attackers may use when they can make the browser assume an ISO-2022-JP charset. Depending on the capabilities of the attacker, this can for example be achieved by directly controlling the `charset` attribute in the `Content-Type` header or by inserting a `<meta>` tag via an HTML injection vulnerability. If a web server provides an invalid `charset` attribute or none at all, there are usually no other prerequisites since attackers can easily switch the charset to ISO-2022-JP via auto-detection.

Technique 1: Negating Backslash Escaping

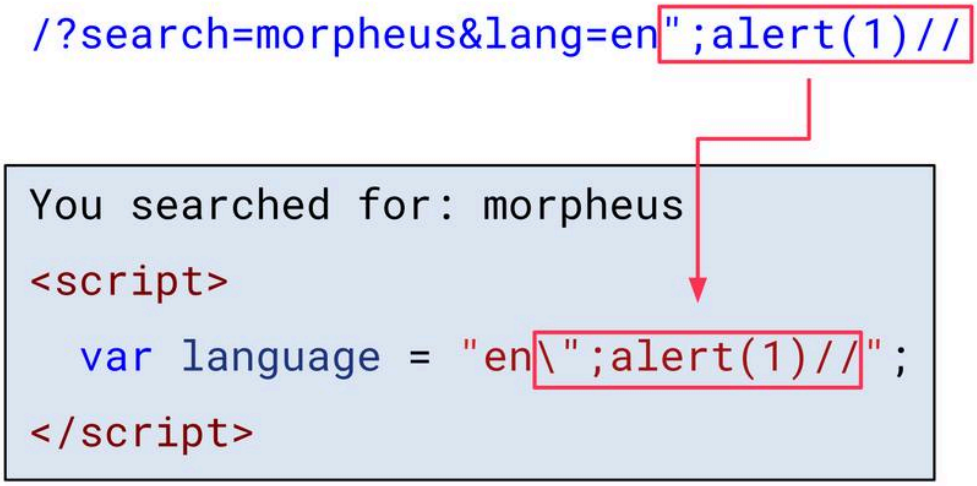
The scenario for this technique is that **user-controlled data** is placed **in a JavaScript string**:



Let's imagine a website that accepts two query parameters called `search` and `lang`. The first parameter is reflected in a plaintext context and the second parameter (`lang`) is inserted into a JavaScript string:

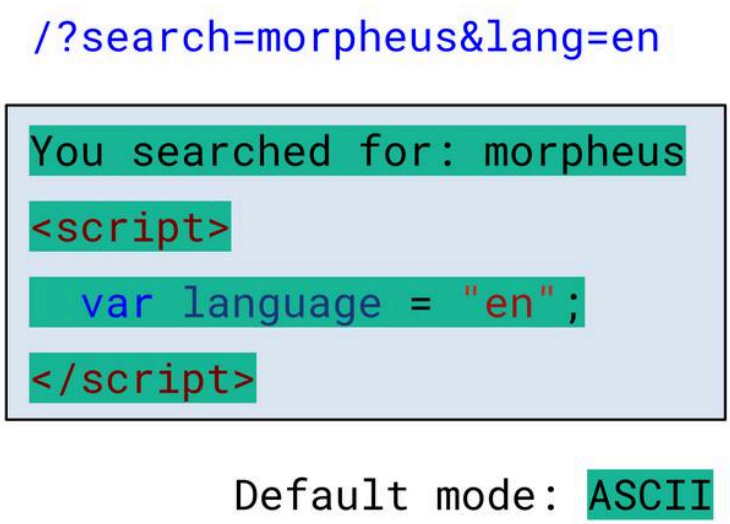


HTML special characters in the `search` parameter are HTML-encoded, and the `lang` parameter is properly sanitized by escaping double quotes (`"`) and backslashes (`\`). Thus, it is not possible to break out of the string context and inject JavaScript code:

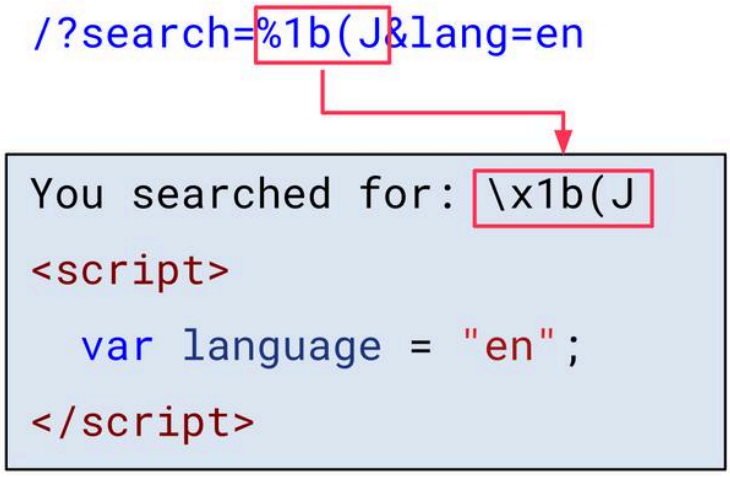


Proper sanitization: no XSS

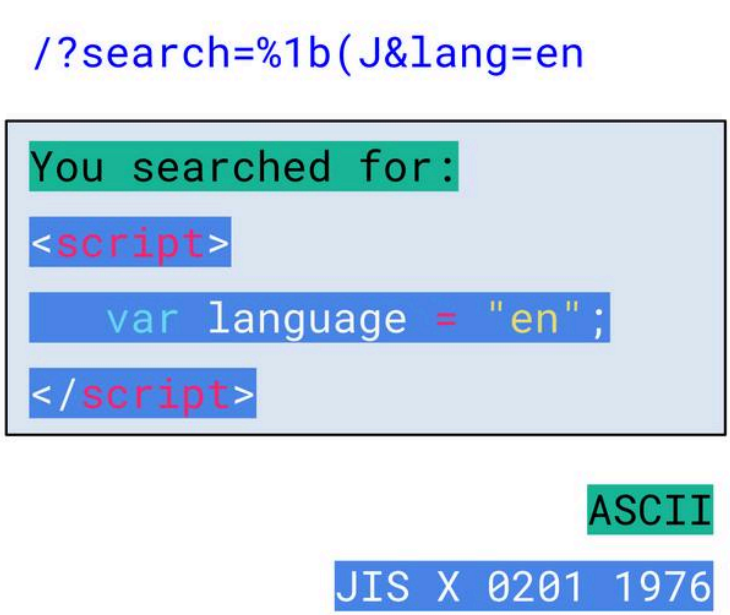
The default mode for ISO-2022-JP is ASCII. This means that all bytes of the received HTTP response body are decoded with ASCII and the resulting HTML document looks like what we would expect:



Now, let's assume an attacker inserts the escape sequence to switch to the JIS X 0201 1976 charset in the `search` parameter (`0x1b` , `0x28` , `0x4a`):



The browser now decodes all bytes following this escape sequence with JIS X 0201 1976:



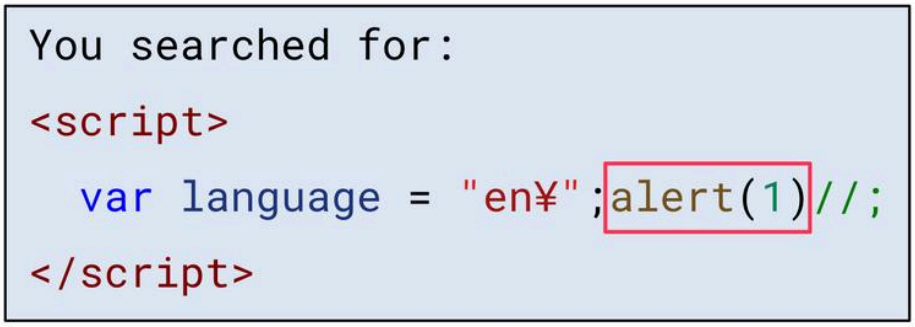
As we can see, this still results in the same characters as before, since JIS X 0201 1976 is *mainly* ASCII-compatible. However, if we closely inspect its code table, we can notice that there are two exceptions (highlighted in yellow):

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x	C0 codes[a]															
1x																
2x	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[	¥	]	^	_
6x	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{	l	}	¯	DEL
8x	C1 codes or Empty Block[a]															
9x																
Ax		。	「	」	、	・	ヲ	ァ	ィ	ゥ	ェ	ォ	ャ	ュ	ョ	ッ
Bx	ー	ア	イ	ウ	エ	オ	カ	キ	ク	ケ	コ	サ	シ	ス	セ	ソ
Cx	タ	チ	ツ	テ	ト	ナ	ニ	ヌ	ネ	ノ	ハ	ヒ	フ	ヘ	ホ	マ
Dx	ミ	ム	メ	モ	ヤ	ユ	ヨ	ラ	リ	ル	レ	ロ	ワ	ン	゙	゚
Ex																
Fx																

The byte `0x5c` is mapped to the yen character (`¥`) and the byte `0x7e` to the overline character (‾). This is different from ASCII, where `0x5c` is mapped to the backslash character (`\`) and `0x7e` to the tilde character (`~`).

This means that when the web server tries to escape a double quote in the `lang` parameter with a backslash, the browser does not *see* a backslash anymore, but instead a yen sign:

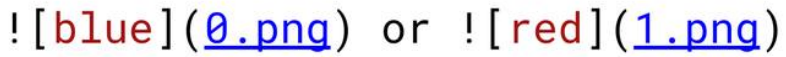
Accordingly, the inserted double quote actually designates the end of the string and allows an attacker to inject arbitrary JavaScript code:



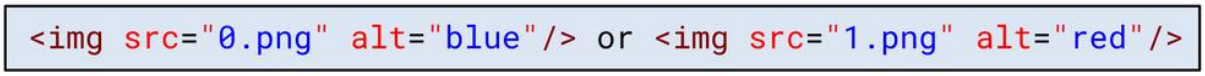
Although this technique is quite powerful, it is limited to bypassing sanitization in a JavaScript context since a backslash character does not have special meaning in HTML. The next section explains a more advanced technique that can be applied in a pure HTML context.

Technique 2: Breaking HTML Context

The scenario for this second technique is that an attacker can control values in **two different HTML contexts**. A common use case would be a website that supports markdown. For example, let's consider the following markdown text:



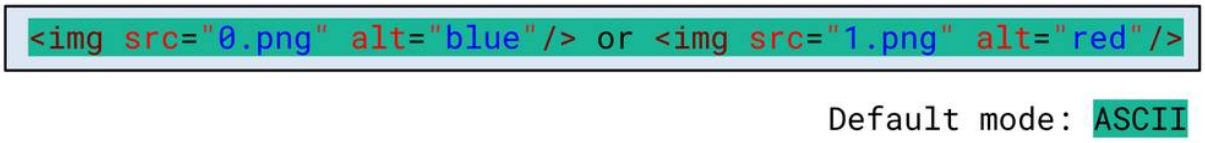
The resulting HTML code looks like this:



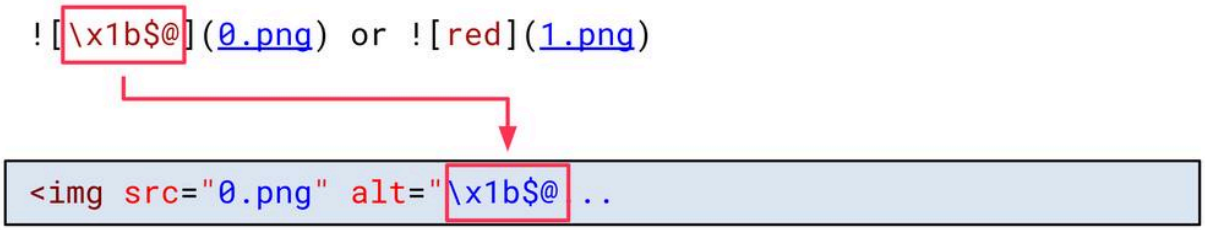
Essential for this technique is that an attacker can control values in two different HTML contexts. In this case, these are:

- Attribute context (image description/source)
- Plaintext context (text surrounding images)

By default, ISO-2022-JP is in ASCII mode and the browser *sees* the HTML document as expected:

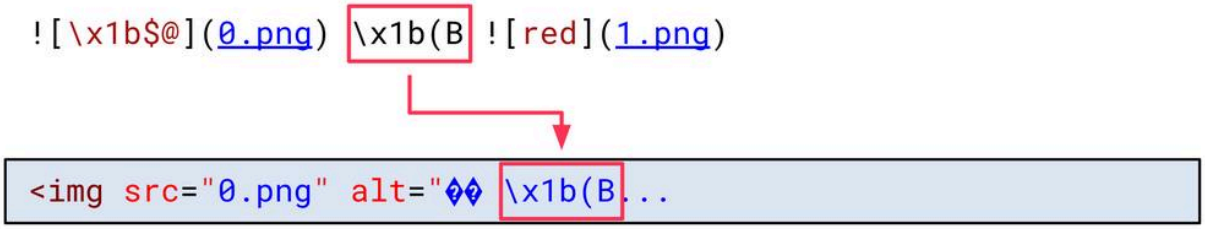


Now, let's assume an attacker inserts the escape sequence to switch the charset to JIS X 0208 1978 in the first image description:

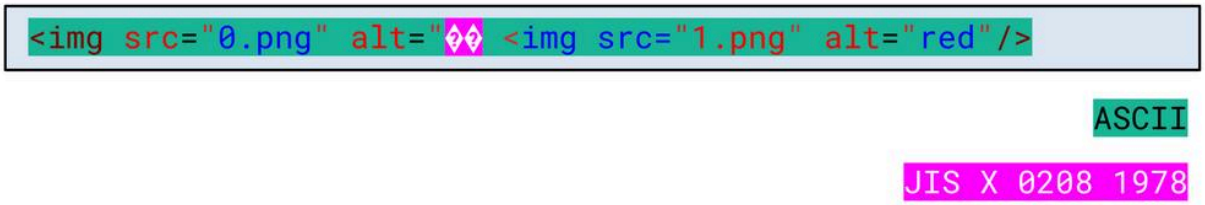


This makes the browser decode all bytes following with JIS X 0208 1978. This charset uses a fixed amount of 2 bytes per character and is not ASCII-compatible. This effectively breaks the HTML document:

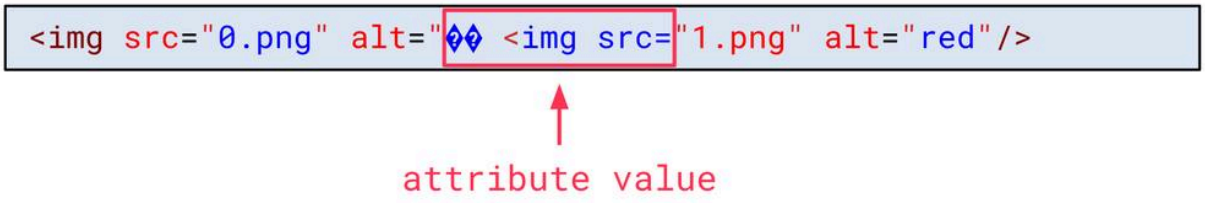
However, a second escape sequence can be inserted in the plaintext context between both images to switch the charset back to ASCII:



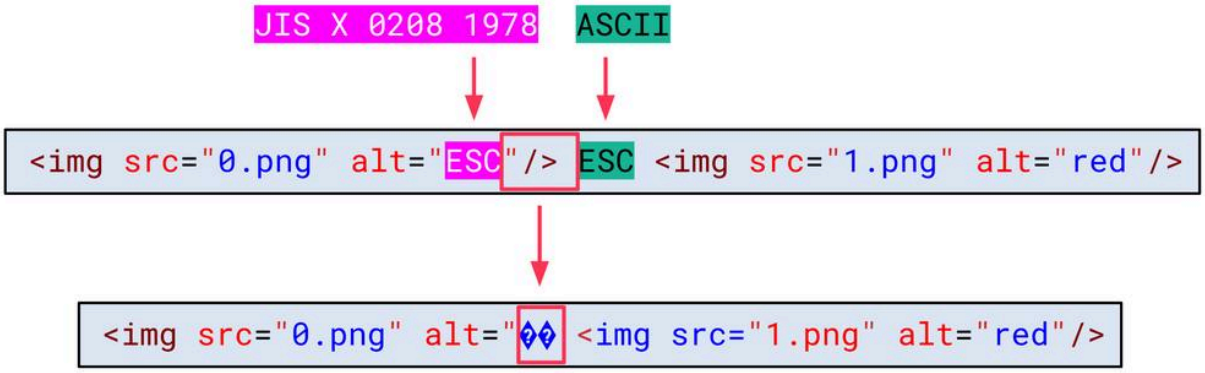
This way, all the following bytes are decoded using ASCII again:



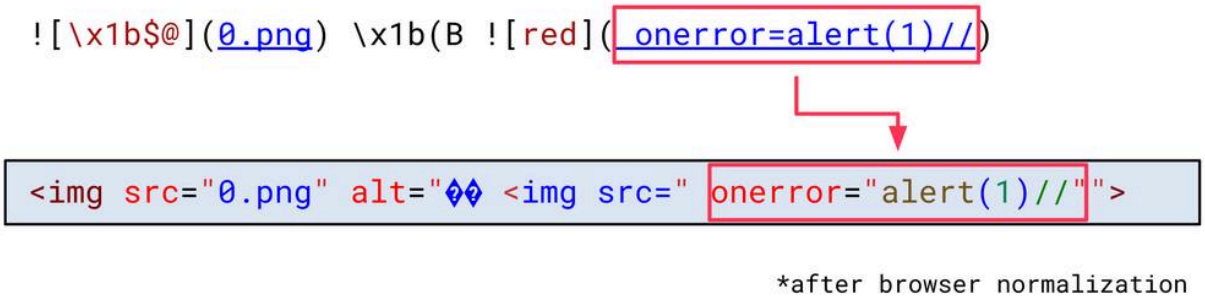
When inspecting the HTML syntax, though, we can notice that something changed. The beginning of the second `img` tag is now part of the `alt` attribute value:



The reason for this is that the 4 bytes in between both escape sequences were decoded using JIS X 0208 1978. This also **consumed the closing double-quote** of the attribute value:



At this point, the `src` attribute value of the second image is not an attribute value anymore. Thus, an attacker can replace this value with a JavaScript error handler:



This, again, allows an attacker to inject arbitrary JavaScript code.

Summary

In this blog post, we highlighted the importance of providing charset information when serving HTML documents. The absence of charset information can lead to severe XSS vulnerabilities when attackers are able to change the character set that the browser assumes.

We detailed how a browser determines the character set used to decode an HTTP response body and explained two different techniques that attackers may use to inject arbitrary JavaScript code into a website leveraging the ISO-2022-JP character encoding.

Although we consider a missing character set the actual vulnerability, a browser's auto-detection greatly increases its impact. Because of this, we hope that browsers will disable the auto-detection mechanism according to our suggestion - at least for the ISO-2022-JP character encoding.
