Element: setHTMLUnsafe() method

- Syntax
  - Parameters
  - Return value
  - Exceptions
- Description
  - Security considerations
  - When should setHTMLUnsafe() be used?
- Examples
  - setHTMLUnsafe() with Trusted Types
  - Using setHTMLUnsafe() without Trusted Types
  - setHTMLUnsafe() live example
    - HTML
    - JavaScript
    - Results
- Specifications
- Browser compatibility
- See also

{{APIRef("DOM")}}

> [!WARNING] This method parses its input as HTML, writing the result into the DOM. APIs like this are known as injection sinks, and are potentially a vector for cross-site scripting (XSS) attacks, if the input originally came from an attacker.

> You can mitigate this risk by always passing `TrustedHTML` objects instead of strings and enforcing trusted types. See Security considerations for more information.

> [!NOTE] {{domxref("Element.setHTML()")}} should almost always be used instead of this method — on browsers where it is supported — as it always removes XSS-unsafe HTML entities.

The **`setHTMLUnsafe()`** method of the {{domxref("Element")}} interface is used to parse HTML input into a {{domxref("DocumentFragment")}}, optionally filtering out unwanted elements and attributes, and those that don't belong in the context, and then using it to replace the element's subtree in the DOM.

Syntax

```
setHTMLUnsafe(input)
setHTMLUnsafe(input, options)
```

Parameters

- `input`
  - : A {{domxref("TrustedHTML")}} instance or string defining HTML to be parsed.
- `options` {{optional_inline}}
  - : An options object with the following optional parameters:
    - `sanitizer` {{optional_inline}}
      - : A {{domxref("Sanitizer")}} or {{domxref("SanitizerConfig")}} object that defines what elements of the input will be allowed or removed. This can also be a string with the value `"default"`, which applies a `Sanitizer` with the (XSS-safe) default sanitizer configuration. If not specified, no sanitizer is used.
        Note that if you're using the same configuration multiple times, it's expected to be more efficient to use a `Sanitizer` and modify it when you need to.

Return value

None (`undefined`).

Exceptions

- {{jsxref("TypeError")}}
  - : This is thrown if:
    - `input` is passed a string when Trusted Types are enforced by a CSP and no default policy is defined.
    - `options.sanitizer` is passed a:
      - {{domxref("SanitizerConfig")}} that isn't valid. For example, a configuration that includes both "allowed" and "removed" configuration settings.
      - string that does not have the value `"default"`.
      - value that is not a {{domxref("Sanitizer")}}, {{domxref("SanitizerConfig")}}, or string.

Description

The **`setHTMLUnsafe()`** method is used to parse an HTML input into a {{domxref("DocumentFragment")}}, optionally sanitizing it of unwanted elements and attributes, and discarding elements that the HTML specification doesn't allow in the target element (such as {{htmlelement("li")}} inside a {{htmlelement("div")}}). The `DocumentFragment` is then used to replace the element's subtree in the DOM.

Unlike with {{domxref("Element.innerHTML")}}, declarative shadow roots in the input will be parsed into the DOM. If the string of HTML defines more than one declarative shadow root in a particular shadow host then only the first {{domxref("ShadowRoot")}} is created — subsequent declarations are parsed as `<template>` elements within that shadow root.

`setHTMLUnsafe()` doesn't perform any sanitization by default. If no sanitizer is passed as a parameter, all HTML entities in the input will be injected. It is therefore potentially even less safe that {{domxref("Element.innerHTML")}}, which disables {{htmlelement("script")}} execution when parsing.

Security considerations

The suffix "Unsafe" in the method name indicates that it does not enforce removal of all XSS-unsafe HTML entities (unlike {{domxref("Element.setHTML()")}}). While it can do so if used with an appropriate sanitizer, it doesn't have to use an effective sanitizer, or any sanitizer at all! The method is therefore a possible vector for cross-site scripting (XSS) attacks, where potentially

unsafe strings provided by a user are injected into the DOM without first being sanitized.

You should mitigate this risk by always passing {{domxref("TrustedHTML")}} objects instead of strings, and enforcing trusted types using the `require-trusted-types-for` CSP directive. This ensures that the input is passed through a transformation function, which has the chance to sanitize the input to remove potentially dangerous markup (such as {{htmlelement("script")}} elements and event handler attributes), before it is injected.

Using `TrustedHTML` makes it possible to audit and check that sanitization code is effective in just a few places, rather than scattered across all your injection sinks. You should not have to pass a sanitizer to the method when using `TrustedHTML`.

If for any reason you can't use `TrustedHTML` (or even better, `setHTML()`) then the next safest option is to use `setHTMLUnsafe()` with the default sanitizer configuration.

When should `setHTMLUnsafe()` be used?

`setHTMLUnsafe()` should almost never be used if {{domxref("Element.setHTML()")}} is available, because there are very few (if any) cases where user-provided HTML input should need to include XSS-unsafe elements. Not only is `setHTML()` safe, but it avoids having to consider trusted types.

Using `setHTMLUnsafe()` might be appropriate if:

- You can't use `setHTML()` or trusted types (for whatever reason) and you want to have the safest possible filtering. In this case you might use `setHTMLUnsafe()` with the default {{domxref("Sanitizer")}} to filter all XSS-unsafe elements.
- You can't use `setHTML()` and the input might contain declarative shadow roots, so you can't use {{domxref("Element.innerHTML")}}.
- You have an edge case where you have to allow HTML input that includes a known set of unsafe HTML entities.
  You can't use `setHTML()` in this case, because it strips all unsafe entities. You could use `setHTMLUnsafe()` without a sanitizer or `innerHTML`, but that would allow all unsafe entities.
  A better option here is to call `setHTMLUnsafe()` with a sanitizer that allows just those dangerous elements and attributes we actually need. While this is still unsafe, it is safer than allowing all of them.

For the last point, consider a situation where your code relies on being able to use unsafe `onclick` handlers. The following code shows the effect of the different methods and sanitizers for this case.

```
const target = document.querySelector("#target");


const input = "<img src=x onclick=alert('onclick') onerror=alert('onerror')>";
```

```
// Safe - removes all XSS-unsafe entities.
target.setHTML(input);

// Removes no event handler attributes
target.setHTMLUnsafe(input);
target.innerHTML = input;

// Safe - removes all XSS-unsafe entities.
const configSafe = new Sanitizer();
target.setHTMLUnsafe(input, { sanitizer: configSafe });

// Removes all XSS-unsafe entities except `onclick`
const configLessSafe = new Sanitizer();
config.allowAttribute("onclick");
target.setHTMLUnsafe(input, { sanitizer: configLessSafe });
```

Examples

setHTMLUnsafe() with Trusted Types

To mitigate the risk of XSS, we'll first create a `TrustedHTML` object from the string containing the HTML, and then pass that object to `setHTMLUnsafe()`. Since trusted types are not yet supported on all browsers, we define the trusted types tinyfill. This acts as a transparent replacement for the Trusted Types JavaScript API:

```
if (typeof trustedTypes === "undefined")
  trustedTypes = { createPolicy: (n, rules) => rules };
```

Next we create a {{domxref("TrustedTypePolicy")}} that defines a {{domxref("TrustedTypePolicy/createHTML", "createHTML()")}} for transforming an input string into {{domxref("TrustedHTML")}} instances. Commonly implementations of `createHTML()` use a library such as DOMPurify to sanitize the input as shown below:

```
const policy = trustedTypes.createPolicy("my-policy", {
  createHTML: (input) => DOMPurify.sanitize(input),
});
```

Then we use this `policy` object to create a `TrustedHTML` object from the potentially unsafe input string:

```
// The potentially malicious string
const untrustedString = "abc <script>alert(1)<" + "/script> def";
// Create a TrustedHTML instance using the policy
const trustedHTML = policy.createHTML(untrustedString);
```

Now that we have `trustedHTML`, the code below shows how you can use it with `setHTMLUnsafe()`. The input has been through the transformation function, so we don't pass a sanitizer to the method.

```
// Get the target Element with id "target"
const target = document.getElementById("target");

// setHTMLUnsafe() with no sanitizer
target.setHTMLUnsafe(trustedHTML);
```

Using setHTMLUnsafe() without Trusted Types

This example demonstrates the case where we aren't using trusted types, so we'll be passing sanitizer arguments.

The code creates an untrusted string and shows a number of ways a sanitizer can be passed to the method.

```
// The potentially malicious string
const untrustedString = "abc <script>alert(1)<" + "/script> def";

// Get the target Element with id "target"
const target = document.getElementById("target");

// Define custom Sanitizer and use in setHTMLUnsafe()
// This allows only elements: div, p, button, script
const sanitizer1 = new Sanitizer({
  elements: ["div", "p", "button", "script"],
});
target.setHTMLUnsafe(untrustedString, { sanitizer: sanitizer1 });
```

```
// Define custom SanitizerConfig within setHTMLUnsafe()
// Removes the <script> element but allows other potentially unsafe entities.
target.setHTMLUnsafe(untrustedString, {
  sanitizer: { removeElements: ["script"] },
});
```

`setHTMLUnsafe()` live example

This example provides a "live" demonstration of the method when called with different sanitizers. The code defines buttons that you can click to inject a string of HTML. One button injects the HTML without sanitizing it at all, and the second uses a custom sanitizer that allows `<script>` elements but not other unsafe items. The original string and injected HTML are logged so you can inspect the results in each case.

> [!NOTE] Because we want to show how the sanitizer argument is used, the following code injects a string rather than a trusted type. You should not do this in production code.

HTML

The HTML defines two {{htmlelement("button")}} elements for calling the method with different sanitizers, another button to reset the example, and a {{htmlelement("div")}} element to inject the string into.

```
<button id="buttonNoSanitizer" type="button">None</button>
<button id="buttonAllowScript" type="button">allowScript</button>

<button id="reload" type="button">Reload</button>
<div id="target">Original content of target element</div>
```

```
<pre id="log"></pre>
```

```
#log {
  height: 320px;
  overflow: scroll;
  padding: 0.5rem;
  border: 1px solid black;
  margin: 5px;
```

```
}
```

JavaScript

```
const logElement = document.querySelector("#log");
function log(text) {
  logElement.textContent += text;
}
```

```
if ("Sanitizer" in window) {
```

First we define the string to sanitize, which will be the same for all cases. This contains the {{htmlelement("script")}} element and the `onclick` handler, both of which are considered XSS-unsafe. We also define the handler for the reload button.

```
// Define unsafe string of HTML
const unsanitizedString = `
  <div>
    <p>Paragraph to inject into shadow DOM.
      <button onclick="alert('You clicked the button!')">Click me</button>
    </p>
    <script src="path/to/a/module.js" type="module"><\/script>
    <p data-id="123">Para with <code>data-</code> attribute</p>
  </div>
`;

const reload = document.querySelector("#reload");
reload.addEventListener("click", () => document.location.reload());
```

Next we define the click handler for the button that sets the HTML with no sanitizer. Generally we would expect the method to drop elements in the string that aren't allowed in the context (such as table-specific elements in a `<div>` element), but otherwise match the input string. In this case the strings should match.

```
const buttonNoSanitizer = document.querySelector("#buttonNoSanitizer");
buttonNoSanitizer.addEventListener("click", () => {
  // Set unsafe HTML without specifying a sanitizer
  target.setHTMLUnsafe(unsanitizedString);

  // Log HTML before sanitization and after being injected
  logElement.textContent =
```

```
    "No sanitizer: string should be injected without filtering\n\n";
  log(`\nunsanitized: ${unsanitizedString}`);
  log(`\n\nsanitized: ${target.innerHTML}`);
});
```

The next click handler sets the target HTML using a custom sanitizer that allows only {{htmlelement("div")}}, {{htmlelement("p")}}, and {{htmlelement("script")}} elements. Note that because we're using the `setHTMLUnsafe()` method, `<script>` are not removed!

```
const allowScriptButton = document.querySelector("#buttonAllowScript");
allowScriptButton.addEventListener("click", () => {
  // Set the content of the element using a custom sanitizer
  const sanitizer1 = new Sanitizer({
    elements: ["div", "p", "script"],
  });
  target.setHTMLUnsafe(unsanitizedString, { sanitizer: sanitizer1 });

  // Log HTML before sanitization and after being injected
  logElement.textContent = "Sanitizer: {elements: ['div', 'p', 'script']}\n";
  log(`\nunsanitized: ${unsanitizedString}`);
  log(`\n\nsanitized: ${target.innerHTML}`);
});
```

```
} else {
  log("The HTML Sanitizer API is NOT supported in this browser.");
  // Provide fallback or alternative behavior
}
```

Results

Click the "None" and "allowScript" buttons to see the effects of no sanitizer and a custom sanitizer, respectively.

When you click the "None" button, you should see that the input and output match, as no sanitizer is applied. When you click the "allowScript" button the `<script>` element is still present, but the `<button>` element is removed. With this approach you can create safe HTML, but you aren't forced to.

{{EmbedLiveSample("setHTMLUnsafe() live example","100","450px")}}

Specifications

{{Specifications}}

Browser compatibility

{{Compat}}

See also

- {{domxref("ShadowRoot.setHTMLUnsafe()")}}
- {{domxref("Element.innerHTML")}}
- {{domxref("Document.parseHTML_static", "Document.parseHTML()")}} and {{domxref("Document.parseHTMLUnsafe_static", "Document.parseHTMLUnsafe()")}}

□□□ #2
□ □□□ □□□ 2026-10-31 03:27:53 UTC
□ □□□ □□□ 2026-10-31 03:27:53 UTC