Exploring the DOMPurify library: Bypasses and Fixes (1/2)

Exploring the DOMPurify library: Bypasses and Fixes (1/2) - kevin_mizu, mizu.re.

- Published: date not stated
- Original: <https://mizu.re/post/exploring-the-dompurify-library-bypasses-and-fixes>
- Preserved from: https://mizu.re/post/exploring-the-dompurify-library-bypasses-and-fixes (stored) on 2026-08-16
- Licence: unknown

Rights remain with the original author and publisher. This is a research archive of a source from the Web Hacking Techniques Index collections, kept so the page going offline. To read the original, follow the link above.

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.

Exploring the DOMPurify library: Bypasses and Fixes (1/2) | mizu.re

keyboard_arrow_up

title: Exploring the DOMPurify library: Bypasses and Fixes (1/2) date: Nov 17, 2024 tags: Article Web mXSS

Exploring the DOMPurify library: Bypasses and Fixes (1/2)

Introduction

This article will be part of a two-article series focusin Introductionel free to skip to "DOMPurify 3.1.0 bypass (found by @IceFont)".

How does client-side HTML sanitizer works?

Before diving into the technical details, I believe it's important to quickly explain how a client-side HTML sanitizer works.

Essentially, what you need to keep in mind is that using a client-side sanitizer leverages the browser's HTML parser, limiting the potential for parsing differentials to occur. For instance, using a client-side HTML sanitizer, by design, issues involving incorrect comment parsing won't have any impact, as the same HTML parser is used twice anyway.

```
package main

import (
    "fmt"

    "github.com/microcosm-cc/bluemonday"
)

func main() {
    unsafeHTML := `<!--><img src=x onerror=alert()>>`
    p := bluemonday.NewPolicy()
    p.AllowComments()
    safeHTML := p.Sanitize(unsafeHTML)
    fmt.Println("Sanitized HTML:", safeHTML) // <!--><img src=x onerror=alert()>-->
}
```

Fig. 1: Golang bluemonday HTML sanitizer bypass due to inconsistent HTML comment parsing in x/net/html found by @gregxsunday (ref).

If you want to easily reproduce this issue on your side, you can use pybluemonday version <= 0.0.9, which contains all the vulnerable versions for this issue.

As an example of how internally a client-side HTML sanitizer works, here is a simplified version of the DOMPurify's workflow, as it is the subject of this article :)

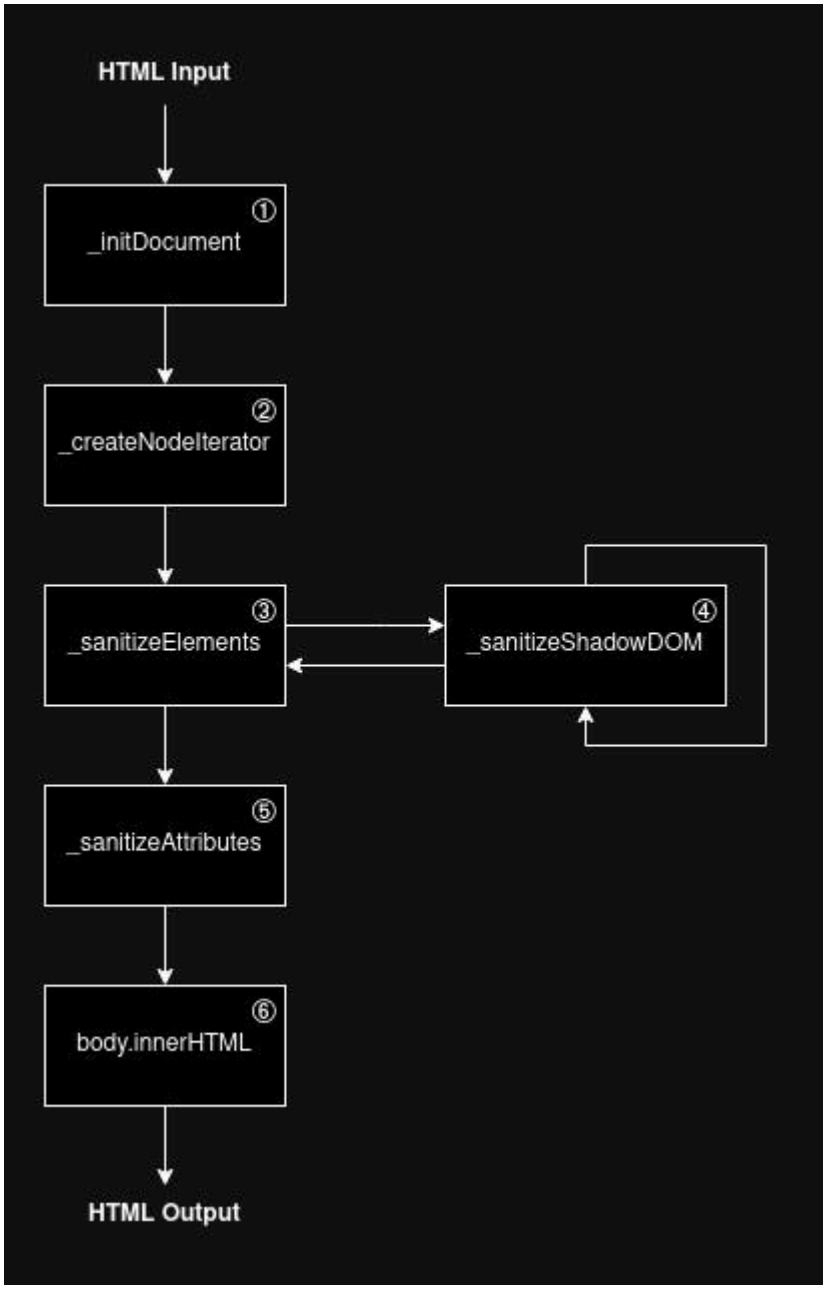


Fig. 2: Simplified DOMPurify execution flow.

- **_initDocument**: Uses the DOMParser API to parse the HTML as the browser would.
- **_createNodeIterator**: Uses NodeIterator to iterate over the DOM tree.
- **_sanitizeElements**: Checks for DOM Clobbering, mXSS, etc., and ensures the current tag is allowed.
- **_sanitizeShadowDOM**: The NodeIterator API doesn't iterate over the <template> tag by default. Recursively sanitizes when it reaches a DocumentFragment.

- **_sanitizeAttributes**: Sanitizes HTML attributes using DOM APIs.
- **body.innerHTML**: Serializes the clean HTML output and returns it.

This is a highly simplified version of DOMPurify's logic. I recommend reading the code directly if you want to understand all its security measures.

Why are mutation XSS (mXSS) possible?

Based on the previous section, you might be thinking:

>

How could a client-side sanitizer be bypassed if it has the same parser as the browser?

That's a good point, and it is mostly due to the way HTML works. The first reason, as well explained in the specification, is that parsing an HTML string twice can lead to different outputs each time.

[image: image]

Fig. 3: HTML Specification - Serialising html fragments (ref).

A "well-known" example used by @SecurityMB to bypass DOMPurify < 2.0.17 is related to the <form> child restriction, which blocks it from having another nested <form>:

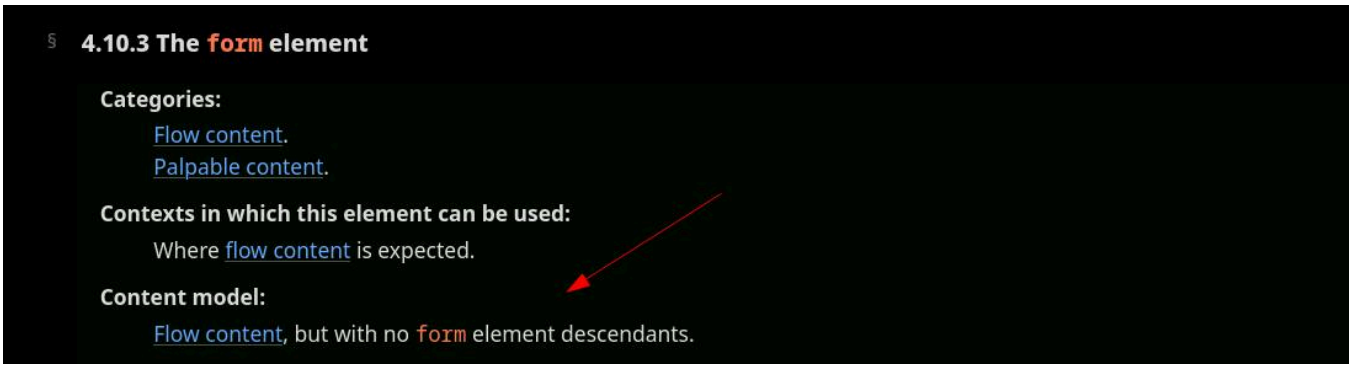


Fig. 4: HTML Specification - The form element (ref).

Fig. 5: Double parsing mutation using <form> element's parsing properties.

You can manually edit the string at the top, and the result will appear at the bottom. It uses 'pipelines', meaning that here, we see the result of double HTML parsing using the DOMParser method. Thanks to @BitK_ for this excellent interactive DOM Tree rendering tool (link).

Additionally, when parsing an HTML DOM tree from a string, there are several rules that describe how each tag has to be interpreted. Among these rules described in the HTML specification, some are related to the concept of namespace.

- <html> | HTML namespace
- <svg> | SVG namespace
- <math> | MathML namespace

Each of these namespaces has its own parsing rules, meaning that a tag, depending on its context, can be interpreted in completely different ways. This is one of the key reasons that makes HTML sanitization complicated, even on the client side.

For example, the <style> element is treated as text in the HTML namespace, while within the MathML or SVG namespace, it would be treated as HTML.

Fig. 6: Parsing of the <style> element in the HTML namespace.

Fig. 7: Parsing of the <style> element in the SVG namespace.

The above two behaviors are also used with HTML integration points and MathML text integration points to switch from the SVG and MathML namespace to the HTML one.

List of MathML text integration points:

- <mi>
- <mo>
- <mn>
- <ms>
- <mtext>

List of HTML integration points:

- <annotation-xml>
- <foreignObject>
- <desc>
- <title>

Fig. 8: Example of HTML integration points usage.

Many more advanced mutation techniques have already been discovered and documented by great researchers. Since explaining every existing mutation and their potential dangers would take too long, I recommend checking out these resources if you haven't already (I'm probably missing a lot of great ones):

| Author | Ressource | | | @cure53berlin | mXSS Attacks: Attacking well-secured Web-Applications by using innerHTML Mutations. | | | @Checkmarx | Mutation Cross-Site Scripting (mXSS) Vulnerabilities Discovered in Mozilla-Bleach. | | | @garethheyes | Bypassing DOMPurify again with mutation XSS. | | | @klikkioy | Yahoo Mail stored XSS. | | | @LiveOverflow | Generic HTML Sanitizer Bypass Investigation. | | | @LiveOverflow | XSS on Google Search - Sanitizing HTML in The Client? | | | @SecurityMB | Write-up of DOMPurify 2.0.0 bypass using mutation XSS. | | | @SecurityMB | Mutation XSS via namespace confusion - DOMPurify < 2.0.17 bypass. | | | @SecurityMB | invalid parsing of HTML by tree_builder_simulator leading to mutation XSS (Chromium). | | | @SecurityMB | Sanitizer bypass if the sanitized markup is assigned to srcdoc (Firefox). | | | @ryotkak | Bypassing DOMPurify with good old XML. | | | @gregxsunday | $3,133.70 XSS in golang's net/html library - My first Google bug bounty. | | | @Sonar_Research | mXSS cheatsheet. | | | @S1r1u5_ | MXSS Explained: Server Side HTML Sanitizers are Doomed to Fail with this XSS!. | | | @S1r1u5_ | MXSS Evolution and Timeline. | | | @wir3less2 | XSS in Gmail's Amp4Email | | | Me | Playing with DOMPurify custom elements handling. | |

Now that we have all the necessary information to understand the upcoming sections, let's start discussing the bypasses.

DOMPurify 3.1.0 bypass (found by @IceFont)

A bit of context on recent DOMPurify researches

The story begins on April 26, 2024, when @cure53berlin posted about a full DOMPurify bypass in versions <= 3.1.0, discovered by @IcesFont.

Fig. 9: Tweet announcing the DOMPurify <= 3.1.0 bypass (ref).

This bypass involved a lot of new mutation concepts, making it really hard to replicate. Thankfully, @IcesFont graciously gave me more details about how his bypass worked, which greatly helped my understanding

I really want to highlight that it's thanks to @IcesFont's work that I was able to find bypasses in versions 3.1.1 and 3.1.2.

With that said, we can dive into how @IcesFont bypassed DOMPurify <= 3.1.0 in default configurations :D

Node flattening

When parsing an HTML tree, there are many factors to consider. One aspect that might not immediately come to mind is **how deep a DOM tree can be**? Interestingly, the HTML specification does not provide explicit guidelines on how this should be handled.

[image: image]

Fig. 10: HTML Specification - Tree construction (ref).

Because of this, each HTML parsing implementation can define its own limit and act differently when reaching it, which significantly increases the risk of parsing discrepancies.

| Language | Library | Nested node limit | Handling | | | Chromium | DOMParser | 512 | Flattening | | | Firefox | DOMParser | 512 | Flattening | | | Safari | DOMParser | 512 | Flattening | | | Ruby | nokogiri (updated version of libxml2) | 256 | Removing | | | C | libxml2 | 255 | Removing | | | PHP | php-xml (libxml2) | 255 | Removing | | | Python | lxml (libxml2) | 255 | Removing | | | Python | html.parser | No limit? | - | | | javascript | parse5 | No limit? | - | | | javascript | htmlparser2 | No limit? | - | | | Golang | x/net/html | No limit? | - | | | Rust | html5ever | No limit? | - | | | Java | Jsoup | No limit? | - | | | Perl | HTML::TreeBuilder | No limit? | - | |

Fig. 11: Handling of nested node limits depending on HTML parsers.

For instance, this is how your browser is currently handling it: (If I'm not mistaken and this hasn't changed, the output should be different. If not, please DM me on Twitter)

Fig. 12: Nested nodes with a depth of 511.

Fig. 13: Nested nodes with a depth of 512.

Something that makes this behavior even more interesting for mXSS is related to the timing of when this mutation occurs. As we can see from the live examples above, even the <style> tag is flattened out of the <svg> tag, it remains part of the SVG namespace.

This strongly indicates that the flattening occurs after the node has been parsed. As a result, it's possible to create an "invalid" HTML DOM tree, which would lead to another mutation if it is serialized and parsed again.

For example, if an <a> tag is a child of another <a> tag within the HTML namespace, it gets popped out. However, if we flatten an <a> tag from the SVG namespace into the HTML namespace, it won't get popped out!

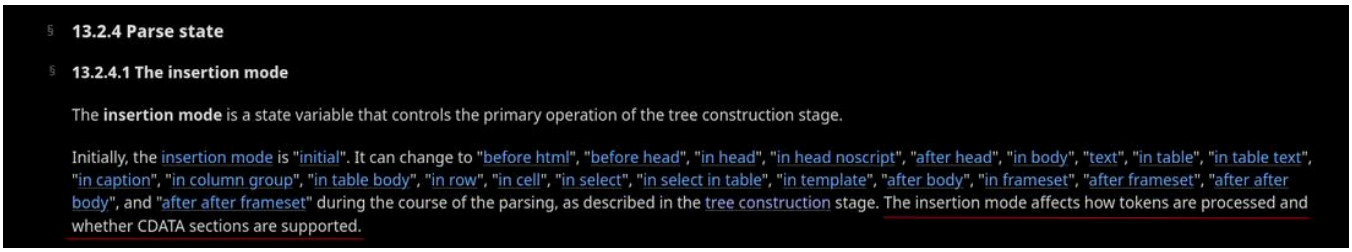
Fig. 14: Nested <a> without flattening.

Fig. 15: Nested <a> with flattening.

Being able to return "invalid" HTML out of a sanitizer is a **strong** mutation gadget, as most of the time it will result in a mutation when reparsing it.

HTML Parsing states

The last piece requires a deep understanding of how HTML parse states are handled. For this bypass, we are going to focus on two concepts: HTML insertion modes and the stack of open elements. As explained in the HTML specification, HTML insertion modes aim to define how tokens are processed while parsing an HTML string.



§ **13.2.4 Parse state**

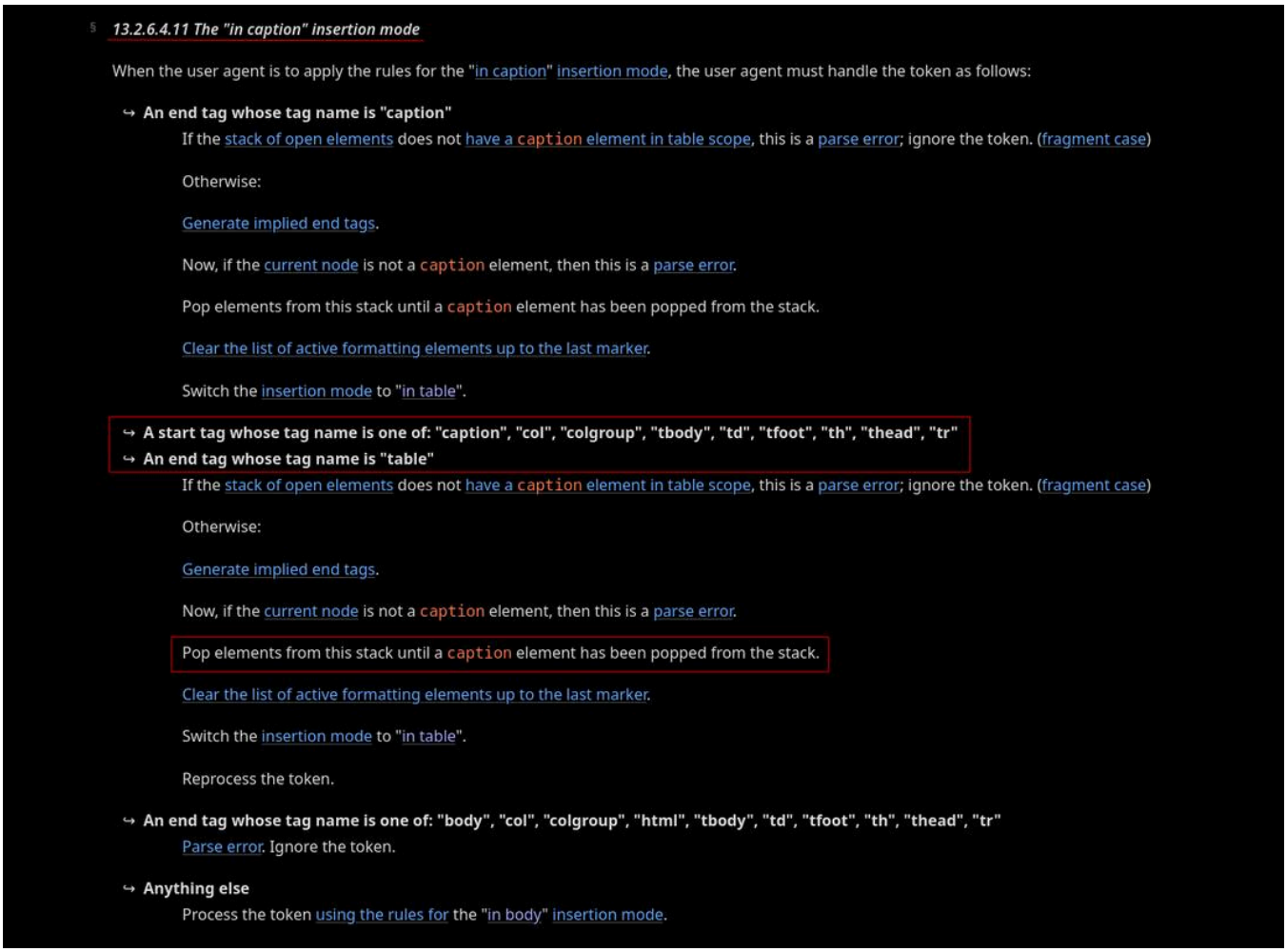
§ **13.2.4.1 The insertion mode**

The **insertion mode** is a state variable that controls the primary operation of the tree construction stage.

Initially, the insertion mode is "initial". It can change to "before html", "before head", "in head", "in head noscript", "after head", "in body", "text", "in table", "in table text", "in caption", "in column group", "in table body", "in row", "in cell", "in select", "in select in table", "in template", "after body", "in frameset", "after frameset", "after after body", and "after after frameset" during the course of the parsing, as described in the tree construction stage. The insertion mode affects how tokens are processed and whether CDATA sections are supported.

Fig. 16: HTML Specification - The insertion mode (ref)

For instance, based on the in caption insertion mode definition, if the parser finds a <caption> start tag, it needs to **pop elements** from the **stack of open elements** until a <caption> element has been popped out.



§ *13.2.6.4.11 The "in caption" insertion mode*

When the user agent is to apply the rules for the "in caption" insertion mode, the user agent must handle the token as follows:

↪ **An end tag whose tag name is "caption"**

If the stack of open elements does not have a caption element in table scope, this is a parse error; ignore the token. (fragment case)

Otherwise:

Generate implied end tags.

Now, if the current node is not a caption element, then this is a parse error.

Pop elements from this stack until a caption element has been popped from the stack.

Clear the list of active formatting elements up to the last marker.

Switch the insertion mode to "in table".

↪ **A start tag whose tag name is one of: "caption", "col", "colgroup", "tbody", "td", "tfoot", "th", "thead", "tr"**

↪ **An end tag whose tag name is "table"**

If the stack of open elements does not have a caption element in table scope, this is a parse error; ignore the token. (fragment case)

Otherwise:

Generate implied end tags.

Now, if the current node is not a caption element, then this is a parse error.

Pop elements from this stack until a caption element has been popped from the stack.

Clear the list of active formatting elements up to the last marker.

Switch the insertion mode to "in table".

Reprocess the token.

↪ **An end tag whose tag name is one of: "body", "col", "colgroup", "html", "tbody", "td", "tfoot", "th", "thead", "tr"**

Parse error. Ignore the token.

↪ **Anything else**

Process the token using the rules for the "in body" insertion mode.

Fig. 17: HTML Specification - Parsing main incaption (ref).

>

What is the stack of open elements?

Essentially, it's a LIFO (Last In First Out) stack of HTML elements. This stack grows as the HTML parser processes the provided string.

[image: image]

Fig. 18: HTML Specification - The stack of open elements (ref).

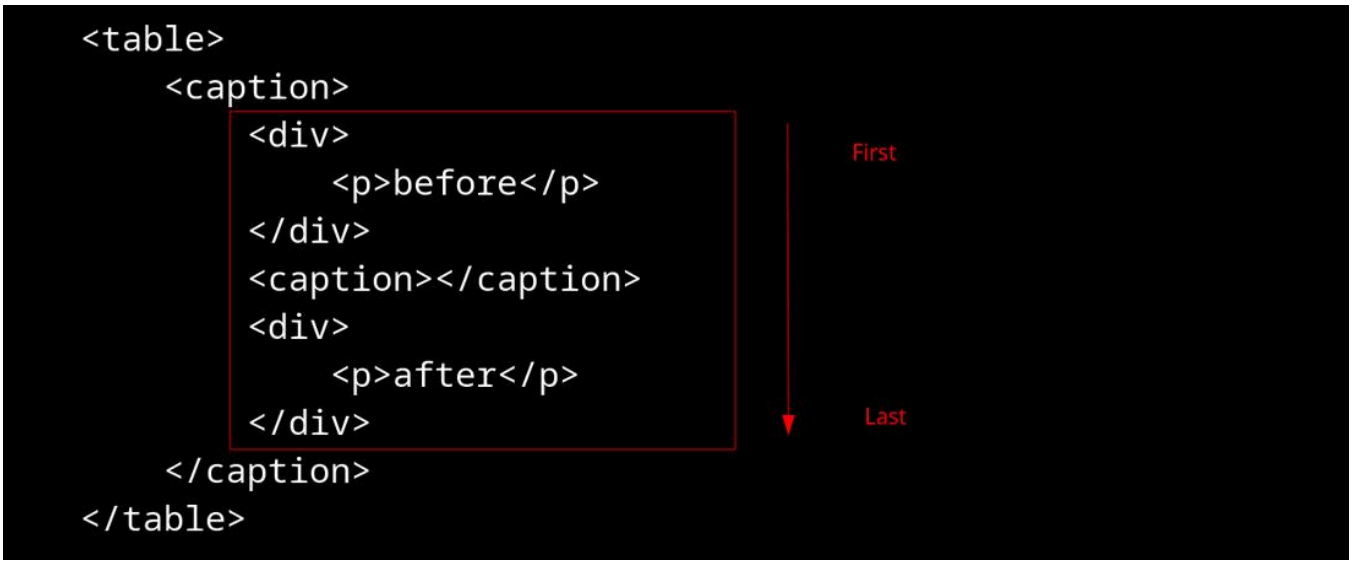


Fig. 19: Example of stack of open elements for a caption element.

If we revisit the in caption insertion mode: popping out elements from the stack of open elements until finding a <caption> element will result in popping out all elements below the nested <caption> element (even if they are valid in that context :D).

Fig. 20: Example of in caption handling in the case of nested <caption>.

What makes it even more interesting is that, even if this is HTML namespace specific, it doesn't take into account the namespace of the tag that gets popped out as they are part of the stack of o pen elements.

Fig. 21: Example of in caption handling in the case of nested <caption> with nested SVG namespace elements.

Finally, to generate this situation using node flattening, @IcesFont used the fact that the in caption insertion mode falls back to the in body insertion mode, which "resets" the parent in table insertion mode.

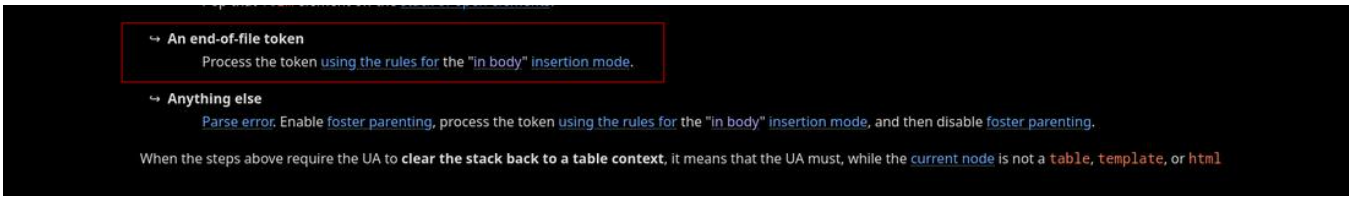


Fig. 22: HTML Specification - Parsing main incaption (ref).

Because of that, it is possible to get a valid context where <caption> can be nested, allowing for the creation of the above "invalid" situation using flattening :D

Fig. 23: Parsing of nested <caption> using the in table insertion mode without flattening.

Fig. 24: Parsing of nested <caption> using the in table insertion mode with flattening.

Proof Of Concept

If we bring everything that has been explained in this section together, it is possible to craft the following HTML payload, which bypasses DOMPurify version <= 3.1.0

Unfortunately, Firefox does not mutate when a <table> is present at the same level as the second <caption> tag, making Firefox not vulnerable to this mutation. However, @kinugawamasato discovered another mutation using deep nesting, which works on Firefox, Chromium, and Safari (we won't cover that one here).

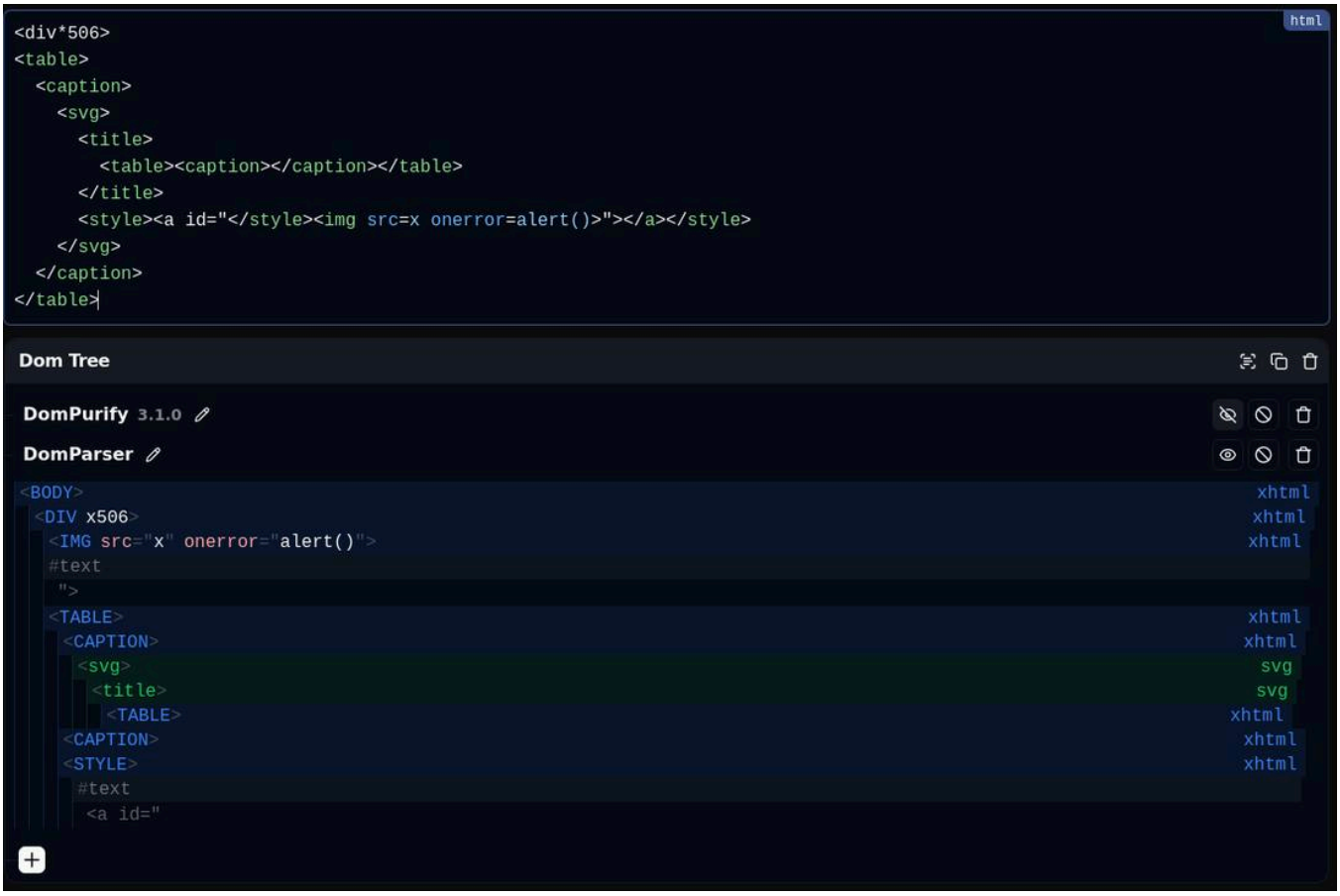
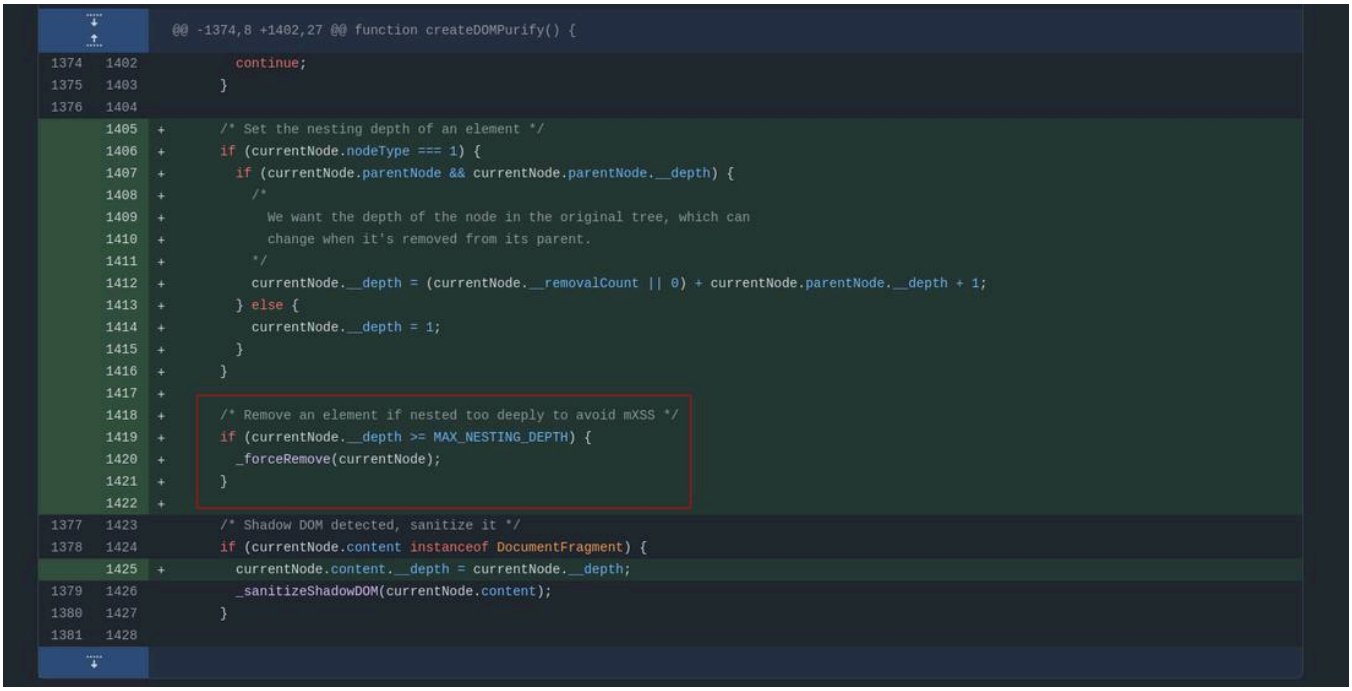


Fig. 25: DOMPurify <= 3.1.0 bypass found by @IcesFont.

DOMPurify 3.1.1 bypass

DOMPurify 3.1.0 fix

This issue has been fixed by @cure53berlin with the help of @IcesFont using a custom depth counter to limit the maximum nested depth to 255. Why not use a browser API to get the current depth of a node? Because there is no such API :(



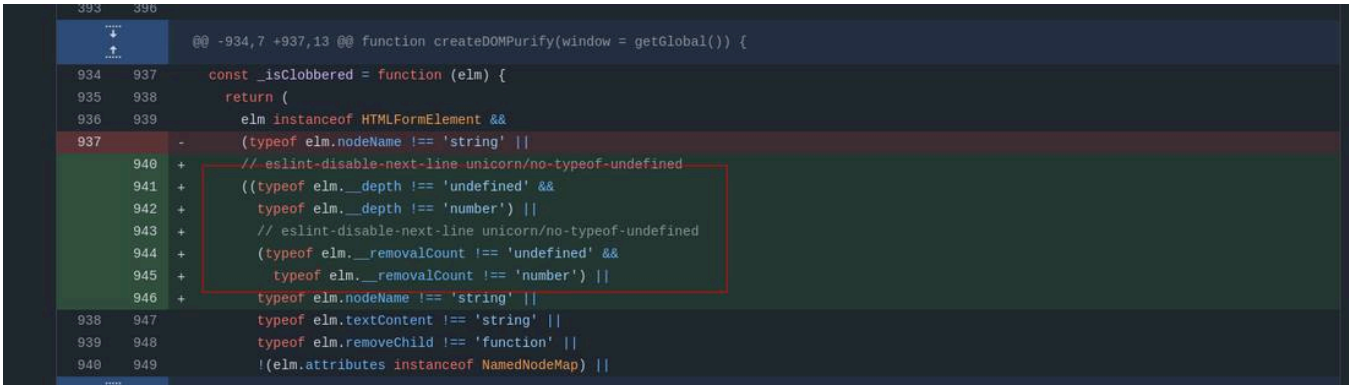
```
@@ -1374,8 +1402,27 @@ function createDOMPurify() {
        continue;
      }

+     /* Set the nesting depth of an element */
+     if (currentNode.nodeType === 1) {
+       if (currentNode.parentNode && currentNode.parentNode.__depth) {
+         /*
+           We want the depth of the node in the original tree, which can
+           change when it's removed from its parent.
+         */
+         currentNode.__depth = (currentNode.__removalCount || 0) + currentNode.parentNode.__depth + 1;
+       } else {
+         currentNode.__depth = 1;
+       }
+     }
+
+     /* Remove an element if nested too deeply to avoid mXSS */
+     if (currentNode.__depth >= MAX_NESTING_DEPTH) {
+       _forceRemove(currentNode);
+     }
+
      /* Shadow DOM detected, sanitize it */
      if (currentNode.content instanceof DocumentFragment) {
+       currentNode.content.__depth = currentNode.__depth;
        _sanitizeShadowDOM(currentNode.content);
      }
```

Fig. 26: GitHub diff between DOMPurify versions 3.1.1 and 3.1.0 (ref).

"Basically", the fix will use a custom node attribute (__depth) taken from the parentNode and adds 1 (the __removalCount is used when removing nodes to properly track the depth update). Then, if the attribute's value gets bigger than 255, it removes the node and all its children.

Additionally, to make sure that the **depth attribute doesn't get clobbered using <form><input id="**depth">, the _isClobbered function has been updated to enforce it to be an integer.



```
@@ -934,7 +937,13 @@ function createDOMPurify(window = getGlobal()) {
  const _isClobbered = function (elm) {
    return (
      elm instanceof HTMLFormElement &&
-     (typeof elm.nodeName !== 'string' ||
+     // eslint-disable-next-line unicorn/no-typeof-undefined
+     ((typeof elm.__depth !== 'undefined' &&
+       typeof elm.__depth !== 'number') ||
+       // eslint-disable-next-line unicorn/no-typeof-undefined
+       (typeof elm.__removalCount !== 'undefined' &&
+         typeof elm.__removalCount !== 'number') ||
+       typeof elm.nodeName !== 'string' ||
        typeof elm.textContent !== 'string' ||
        typeof elm.removeChild !== 'function' ||
        !(elm.attributes instanceof NamedNodeMap) ||
```

Fig. 27: DOMPurify's 3.1.1 _isClobbered function.

DOM Clobbering issue

Even if the fix might look great at first glance, a small mistake has been made regarding how the .parentNode property is accessed. In the fix, currentNode.parentNode.__depth is being used. Why is this a problem? Essentially, it allows clobbering the parentNode property with a node that doesn't have the __depth property yet, allowing the count to reset!

```
<div id="parent">
  <form id="f">
    <input name="parentNode">
  </form>
</div>
<script>
  parent.__depth = 250;
  f.parentNode.__depth; // undefined
</script>
```

Fig. 28: Example of __depth clobbering through the .parentNode property.

Using this bug twice in a row is required for the fix, as 255 * 2 = 510, which doesn't reach the flattening limit. This can be done by using the nested <form> mutation described in the "Why are mutation XSS (mXSS) possible?" section.

Proof Of Concept

For the same reason as the previous bypass, this one isn't working in Firefox.

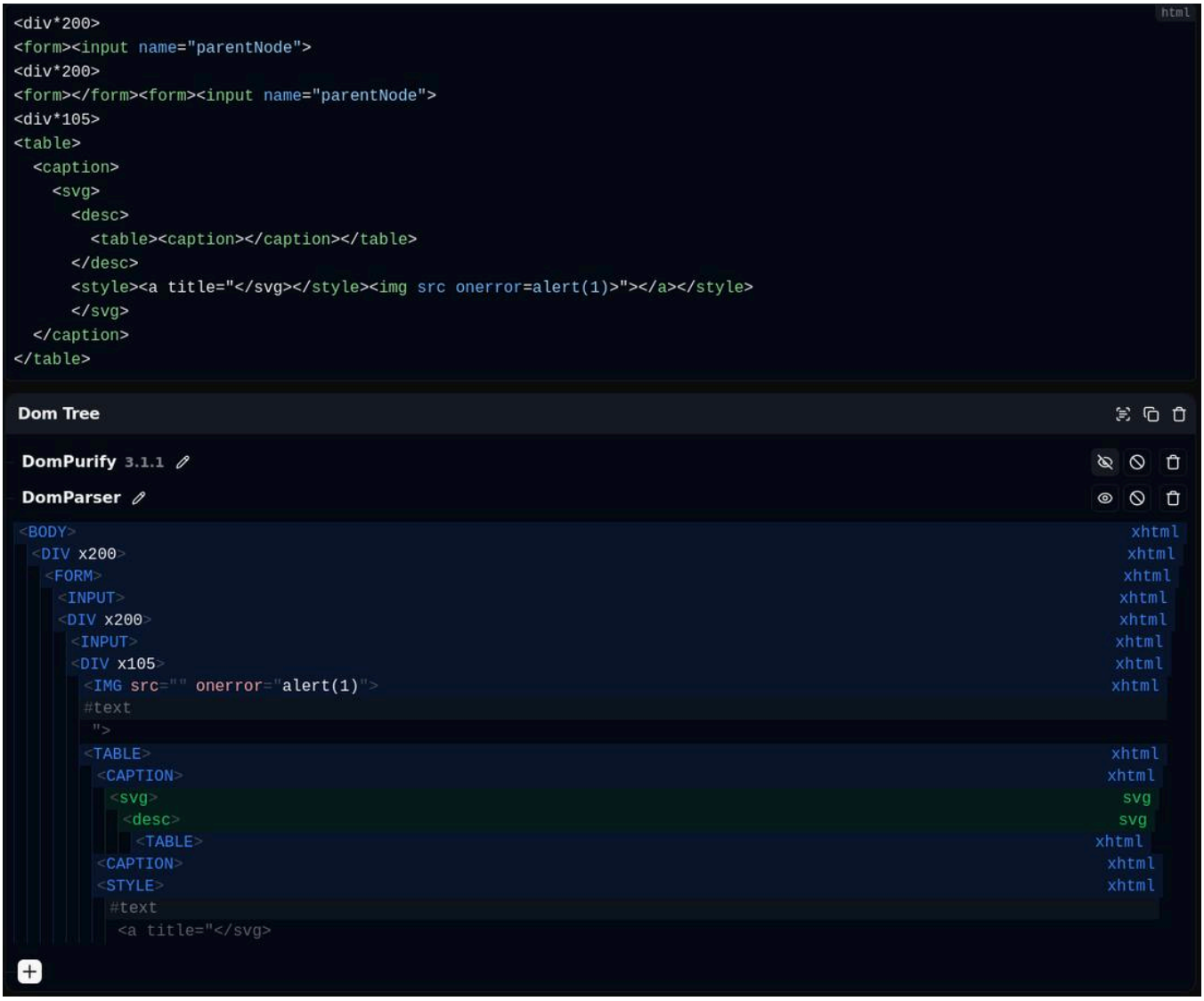


Fig. 29: DOMPurify <= 3.1.1 bypass.

DOMPurify 3.1.2 bypass

DOMPurify 3.1.1 fix

The fix in this version was much stricter than the previous one, not only because of my report but also because @hash_kitten found another full bypass involving only HTML insertion modes and the stack of open elements. We aren't going to cover this bypass in this article, but it motivated @cure53berlin to block every HTML integration point, preventing any switch from the SVG to HTML namespace.

Fig. 30: GitHub diff between DOMPurify versions 3.1.2 and 3.1.1 (ref).

Additionally, to make sure no DOM Clobbering issues were left, the getParentNode method was used, which resolves the value using the property getter itself.

```
@@ -1401,15 +1402,16 @@ function createDOMPurify() {
1401 1402         if (_sanitizeElements(currentNode)) {
1402 1403           continue;
1403 1404         }
     1405 +       const parentNode = getParentNode(currentNode);
1404 1406
1405 1407         /* Set the nesting depth of an element */
1406 1408         if (currentNode.nodeType === 1) {
1407      -         if (currentNode.parentNode && currentNode.parentNode.__depth) {
     1409 +         if (parentNode && parentNode.__depth) {
1408 1410             /*
1409 1411               We want the depth of the node in the original tree, which can
1410 1412               change when it's removed from its parent.
1411 1413             */
1412      -           currentNode.__depth = (currentNode.__removalCount || 0) + currentNode.parentNode.__depth + 1;
     1414 +           currentNode.__depth = (currentNode.__removalCount || 0) + parentNode.__depth + 1;
1413 1415           } else {
1414 1416             currentNode.__depth = 1;
1415 1417           }
```

Fig. 31: GitHub diff between DOMPurify versions 3.1.2 and 3.1.1 (ref).

Second-order DOM Clobbering

This time, the DOM Clobbering issue was inherent to the sanitization order used by DOMPurify. If we refer back to the previous sanitization flow graph, DOM Clobbering checks occur within the _sanitizeElement function.

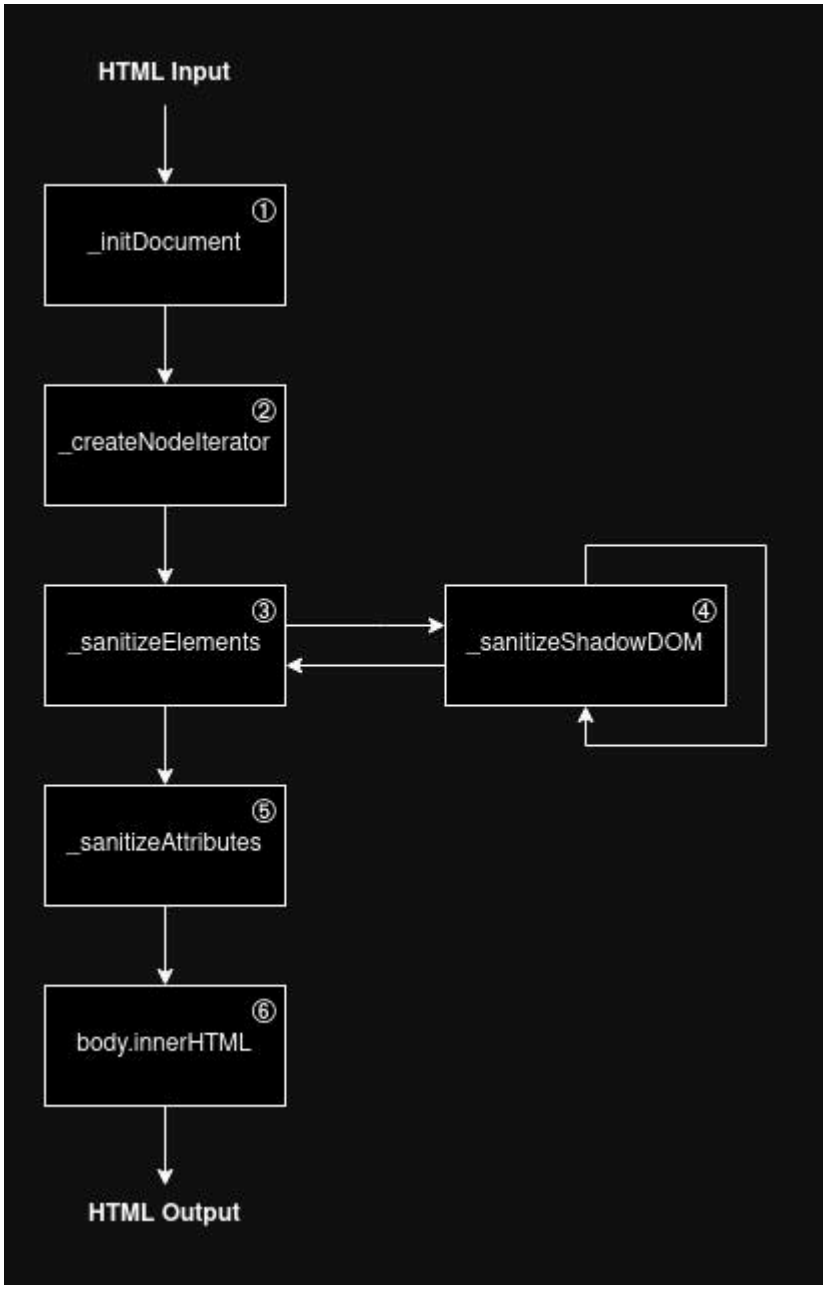


Fig. 32: Simplified DOMPurify execution flow.

This means that any attribute modification or normalization occurring after the _sanitizeElement function might create a "second-order" DOM Clobbering.

```
<form id="x"></form>
<input form="y" name="z">
<script>
  console.log(x.z); // undefined
  x.id = "y";
  console.log(y.z); // <input form="y" name="z">
</script>
```

Fig. 33: Second order DOM Clobbering example 1.

```
<form id="x">
  <input id="i" form="y" name="z">
</form>
<script>
  console.log(x.z); // undefined
  i.removeAttribute("form");
  console.log(x.z); // <input form="y" name="z">
</script>
```

Fig. 34: Second order DOM Clobbering example 2 (this one was found by @ryotkak).

If we take a look at the _sanitizeAttributes function, we can see that it:

- Takes the current attribute value.
- Trims it (removes spaces).
- Sanitizes it.
- Sets the clean value in place of the previous one.

```
const _sanitizeAttributes = function (currentNode) {
  // ...
  const { attributes } = currentNode;
  // ...
  while (l--) {
    const attr = attributes[l];
    // ...
    stringTrim(attrValue);
    // Sanitize
    try {
      // ...
      currentNode.setAttribute(name, value);
      // ...
    } catch (_) {}
  }
};
```

Fig. 35: Simplified DOMPurify _sanitizeAttributes function.

As an example:

Fig. 36: Example of DOMPurify attribute normalization.

Because of that we can now clobber the __depth attribute itself, allowing us to break the depth count at the beginning!

```
<form id="x "></form>
<input form="x" name="__depth">
<script>
f = document.getElementById("x ");
f.setAttribute("id", f.id.trim());
depth = f.__depth + 1; // [object HTMLInputElement]1

if (depth >= 255) {
   // This never gets reached.
}
</script>
```

Fig. 37: Example of __depth count breaking using "second-order" DOM Clobbering.

"Elevator" HTML mutation

At this point, even though we have demonstrated how to break the flattening limitation, it is still not enough as we can't use any HTML integration points, which were mandatory for the @IcesFont mutation.

This time, to find a valid mutation that doesn't require any HTML integration points, I decided to take another approach... fuzzing!

```
<div id="elem"></div>

<script>
  // init
  const tags = ["a", "abbr", "acronym", "address", "area", "article", "aside", "audio", "b", "base", "basefont", "bgsound", '

  // check for mutations
  var found = []
  var parse = (str) => (new DOMParser).parseFromString(str, "text/html").documentElement.innerHTML;
  var check = (output, payload) => {
    if (/* INSERT SOME CONDITION HERE */) {
      console.log(output);
    }
  }

  // fuzzing context
  var fuzz = () => {
    for (i in tags) {
      for (j in tags) {
        for (k in tags) {
          payload = `<${tags[i]}><${tags[j]}><${tags[k]}></${tags[k]}></${tags[j]}><style></style></${tags[i]}>`;
          check(parse(payload), payload);
  }}}}

  // start fuzzing
  fuzz();
</script>
```

Fig. 38: Example of a script to fuzz HTML mutation.

I'll be honest, my fuzzing approach wasn't optimized at all. I was only looking for any HTML parsing that resulted in popping out a <style> element using a custom JS script. At least, thanks to this, I found an interesting mutation:

Fig. 39: "Elevator" HTML mutation example 1.

Fig. 40: "Elevator" HTML mutation example 2.

Take care about the <style> tag :D The <button> tags can be replaced by <dd>, <dt>, <li> or <table>.

Essentially, the tags between two <button> elements determine where the stack of open elements gets popped down. What makes this behavior even more interesting is that it can even traverse namespaces as long as one tag from each traversed namespace is present between the two <button> elements.

Fig. 41: Example of "elevator" mutation without a tag from each namespace to traverse.

Fig. 42: Example of "elevator" mutation with a tag from each namespace to traverse.

I tried to figure out where in the specification this behavior was described, and it seems to be related to this:

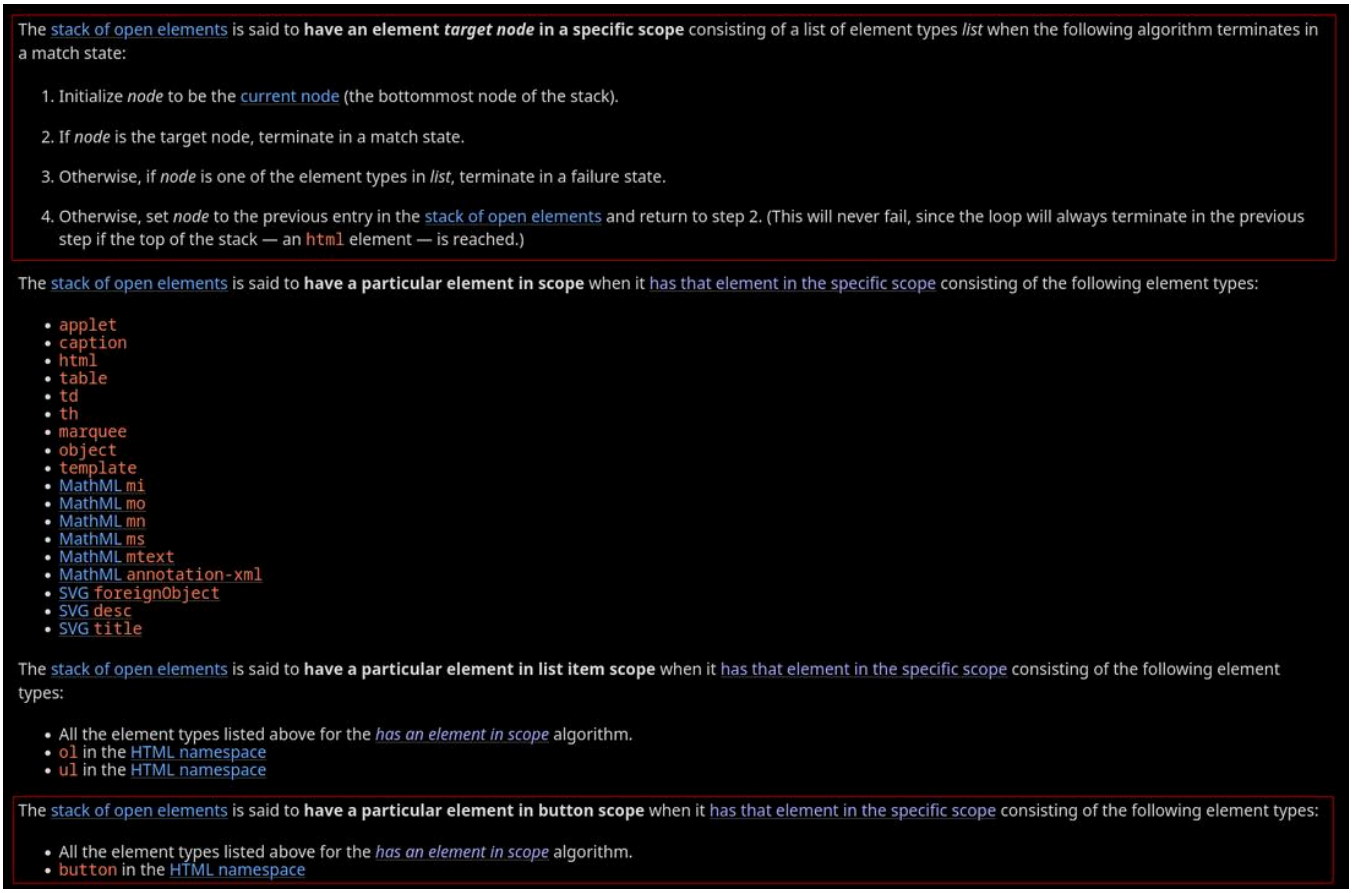


Fig. 43: HTML Specification - Has an element in the specific scope (ref)

Even if this mutation is quite powerful, it wasn't enough to bypass DOMPurify <= 3.1.2 for two reasons:

- It is required to use the same tag in SVG and HTML namespaces.
- It is not possible to use HTML integration points.

```
const COMMON_SVG_AND_HTML_ELEMENTS = addToSet({}, [
  'title',
  'style',
  'font',
  'a',
  'script',
]);
```

Fig. 44: List of tags allowed in both the HTML and SVG namespace by DOMPurify (ref).

Therefore, after playing a bit with this mutation, I've found another case where it can occur:

Fig. 45: "Elevator" HTML mutation using the <image> tag conversion to <img>.

You can try to update the <image> tag with an <img> tag in the HTML namespace, you should see that it doesn't work anymore.

As we can see, the <image> tag conversion to <img> in the HTML namespace leads to the same behavior if there is another <image> tag in the SVG namespace subtree. Additionally, thanks to the <a> tag, which is allowed in both SVG and HTML namespaces by DOMPurify, it is possible to trigger the bug properly!

>

What makes this mutation more interesting than the <button> one for a DOMPurify bypass?

Basically, DOMPurify blocks the usage of HTML integration points only if it is used to switch from the SVG to HTML namespace. For instance, the following is fully valid and won't be removed.

Fig. 46: Example of HTML integration points usage without switching to HTML with DOMPurify.

Based on this, we can use node flattening to flatten the <image> tag out of the <svg>, which will create the right combination for DOMPurify sanitizing!

Proof Of Concept

If we bring everything that has been explained in this section together, it is possible to craft the following HTML bypass which bypasses DOMPurify version <= 3.1.2

Unfortunately, for an unknown reason, second-order DOM clobbering isn't working on Firefox in the context of DOMPurify sanitization, making Firefox not vulnerable again...

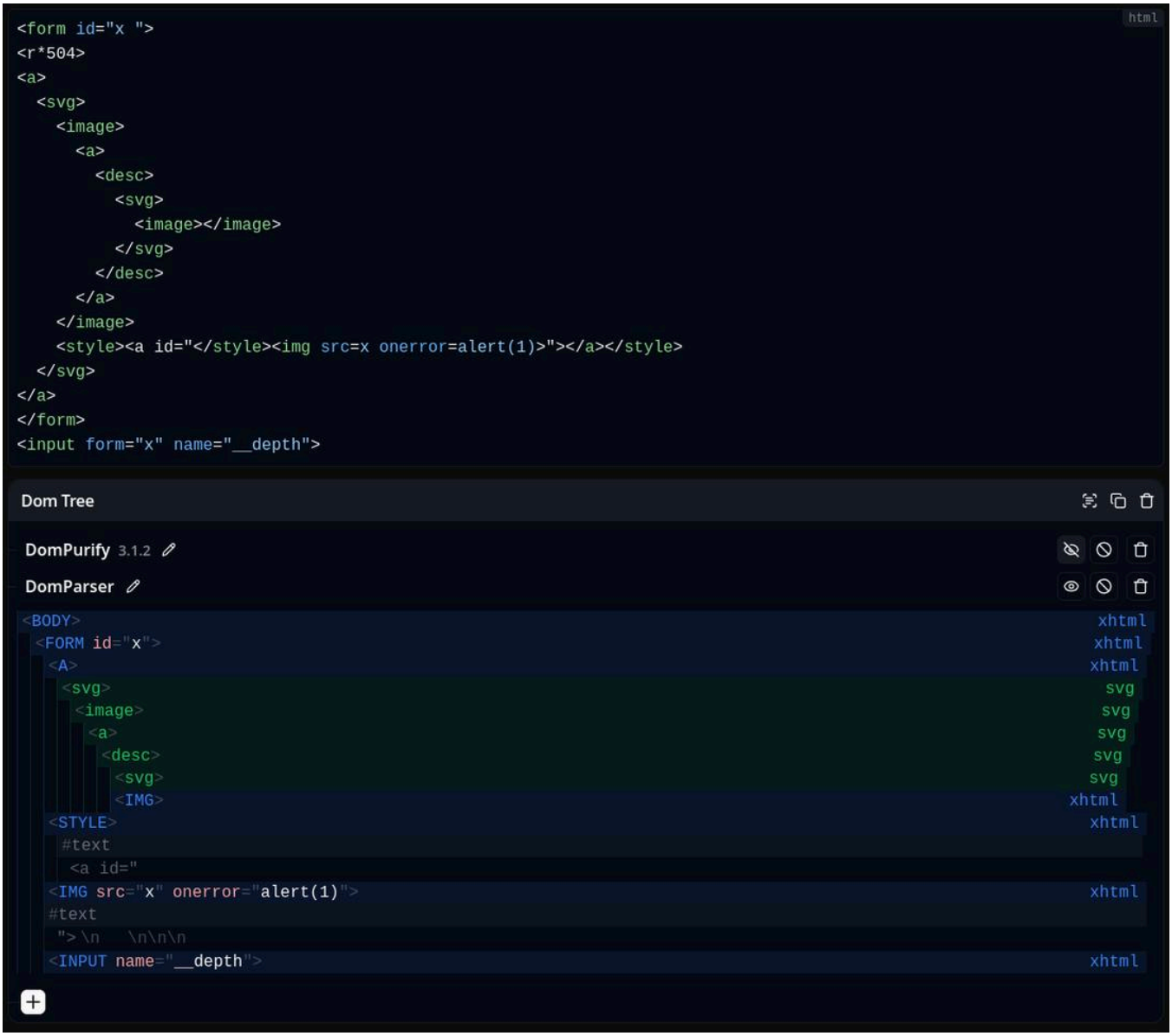


Fig. 47: DOMPurify <= 3.1.2 bypass.

DOMPurify Triple HTML Parsing bypass (found with @hash_kitten and @ryotkak)

Form reordering and node flattening

Now that we've covered full bypasses for versions <= 3.1.2, we are going to focus on something a bit different that we have found with @ryotkak and @hash_kitten.

One of the main problems of most DOMPurify bypasses is that if the HTML gets parsed (server-side or client-side) at least once before reaching the DOMPurify sink, the payload will be broken as the mutation occurs only in a two-parsing window. I've even seen some applications using DOMPurify twice in a row "just in case". An example that I've faced recently is in the Mermaid library:

```
export const sanitizeText = (text: string, config: MermaidConfig): string => {
  if (!text) {
    return text;
  }
  if (config.dompurifyConfig) {
    text = DOMPurify.sanitize(sanitizeMore(text, config), config.dompurifyConfig).toString(); // sanitizeMore uses DOMPurif
  } else {
    text = DOMPurify.sanitize(sanitizeMore(text, config), { // sanitizeMore uses DOMPurify.sanitize internaly.
      FORBID_TAGS: ['style'],
    }).toString();
  }
  return text;
};
```

Fig. 48: Mermaid.js's sanitizeText function.

One way to overcome that, which we found, is by mixing <form> / <table> reordering and node flattening again. How? For this, we need to "chain" several mutations.

The first one is related to how nested form parsing reacts if a <table>, <marquee>, <applet>, or <object> is present between them. Under those conditions, tags at the same level as the first <form> tag get bumped into it.

Fig. 49: <form> / <table> reordering

On Firefox, chaining it with the nested <form> mutation, this is enough to trigger a triple parsing mutation bug that bumps up an element. However, this is not the case on Chromium and Safari. I thought this might be related to HTML quirks mode, but I was wrong, and I have no idea where this parsing difference comes from. ¯\( )/¯

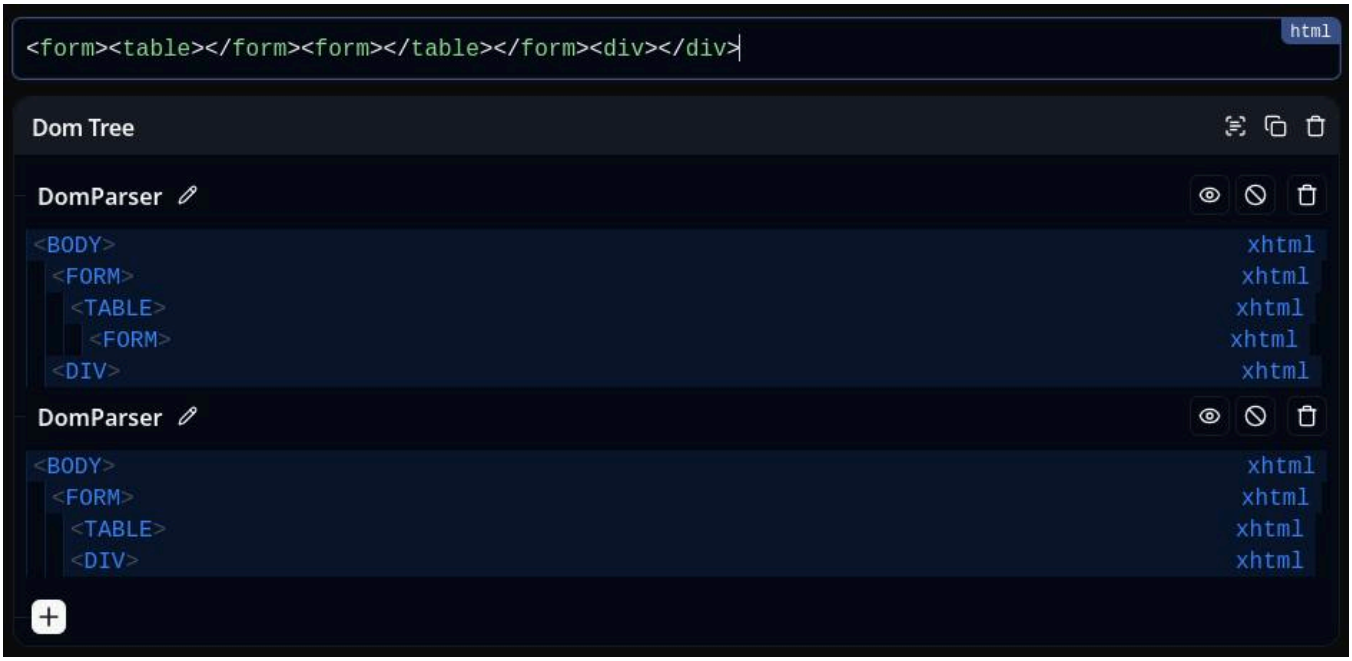


Fig. 50: Firefox <form> / <table> reordering (triple HTML parsing).

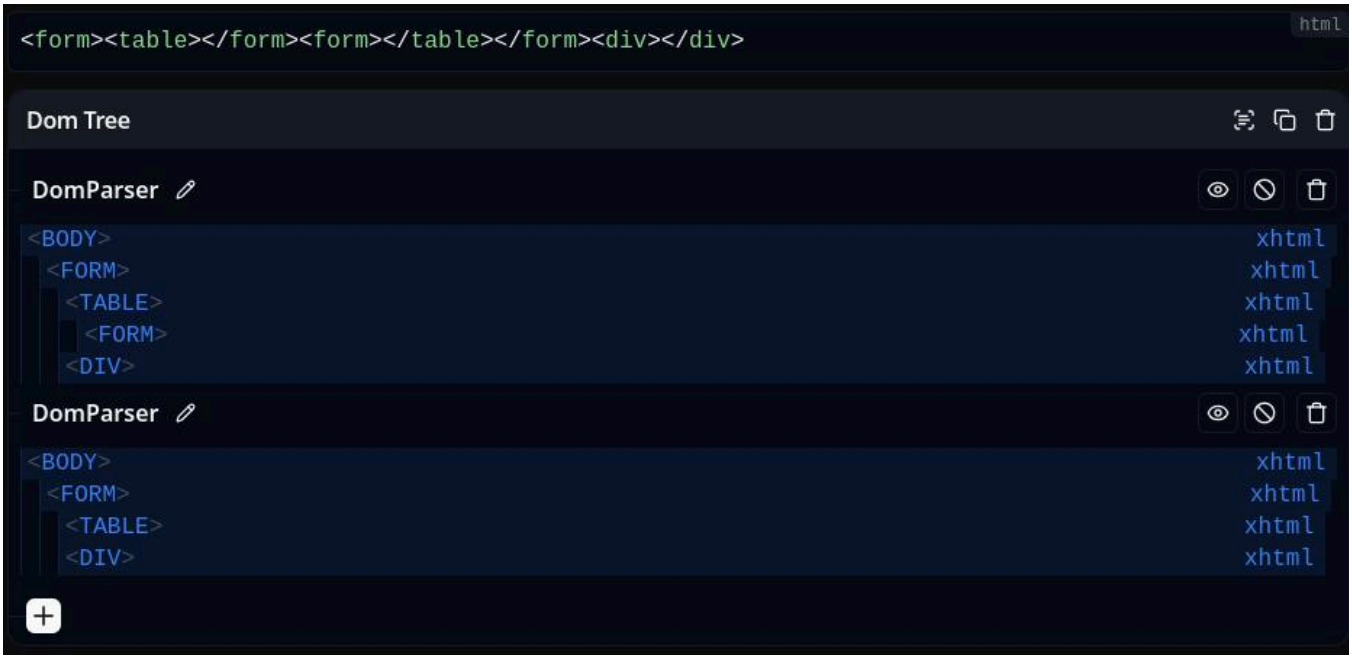


Fig. 51: Chromium <form> / <table> reordering (triple HTML parsing).

Therefore, after some fuzzing, @ryotkak and @hash_kitten found that mixing the mutation and adding any tag before the <table> one allows the behavior to work on both Firefox, Chromium and Safari.

Fig. 52: <form> / <table> reordering (triple HTML parsing) working on Firefox, Chromium and Safari.

Using this, it is possible to control how much an element gets bumped up by simply repeating the payload several times in a row :D

Fig. 53: Example of two-level bumped tag using <form> / <table> reordering.

Don't forgot that the <table> tag can be replaced with <marquee>, <applet> or <object>.

The last thing to do is to craft a payload that reaches the node flattening only on the second parsing, forcing the XSS mutation to occur on the third one!

Proof Of Concept

If we bring everything that has been explained in this section together, it is possible to craft the following HTML payload, which bypasses DOMPurify version <= 3.1.2 in the case of triple HTML parsing

This time it works on Firefox!

Live Proof of Concept

```
var n = 510;
var payload = `
${"<form><h1></form><table><form></form></table></form></table></h1></form>".repeat(n)}
<math>
  <mi>
    <style><!--</style>
    <style id="--></style></mi></math><img src='x' onerror='alert(1)'>"></style>
  </mi>
</math>
`;
document.body.innerHTML = DOMPurify.sanitize(payload)
document.body.innerHTML = document.body.innerHTML;
```

Fig. 54: DOMPurify <= 3.1.2 triple HTML parsing bypass example 1.

The previous payload shows the case where the third HTML parsing occurs after the DOMPurify sanitization. Therefore, as we discussed earlier, this can be used in the case of pre-HTML parsing (client-side or server-side) before the DOMPurify sanitization. If we mix this payload with the DOMPurify <= 3.1.2 bypass, it is possible to have a working payload in most cases!

Live Proof of Concept

```
var n = 503;
var dirty = `
${"<form><h1></form><table><form></form></table></form></table></h1></form>\n".repeat(n)}
<a>
  <svg>
    <desc>
      <svg>
        <image>
          <a>
            <desc>
              <svg>
                <image></image>
              </svg>
            </desc>
          </a>
        </image>
        <title><a id="</title><img src=x onerror=alert(1)>"></a></title>
      </svg>
    </desc>
  </svg>
</a>
`;
var step1 = DOMPurify.sanitize(dirty);
document.body.innerHTML = DOMPurify.sanitize(step1);
```

Fig. 55: DOMPurify <= 3.1.0 triple HTML parsing bypass example 2.

A double DOMPurify.sanitize has been used for the showcase, I believe it shows how strong this payload is! :D

Oh, and this works perfectly on outdated mermaid.js versions, but I leave it as an exercise :p

What's next?

DOMPurify 3.1.2 fix

Because of the triple HTML parsing bypass, @cure53berlin decided to fix the problem at its root cause: HTML attributes. Since all the recent DOMPurify bypasses involve namespace confusion attacks using HTML attributes to smuggle an HTML comment, they decided to remove any attribute containing this pattern. Thanks to this mitigation, even an n-time HTML parsing mutation will be detected and sanitized from the first parsing by DOMPurify.

```
@@ -1187,6 +1206,12 @@ function createDOMPurify() {
1187 1206          continue;
1188 1207        }
1189 1208
     1209 +      /* Work around a security issue with comments inside attributes */
     1210 +      if (SAFE_FOR_XML && regExpTest(/((--!?|])>)|<\/(style|title)/i, value)) {
     1211 +        _removeAttribute(name, currentNode);
     1212 +        continue;
     1213 +      }
     1214 +
1190 1215        /* Sanitize attribute content to be template-safe */
1191 1216        if (SAFE_FOR_TEMPLATES) {
1192 1217          arrayForEach([MUSTACHE_EXPR, ERB_EXPR, TMPLIT_EXPR], expr => {
```

Fig. 58: Fig. 30: GitHub diff between DOMPurify versions 3.1.3 and 3.1.2 (ref).

Conclusion

To conclude, this article has covered four DOMPurify bypasses: three related to the default configurations for versions <= 3.1.0, 3.1.1, and 3.1.2, and one based on triple HTML parsing payloads. We've seen that HTML can be highly unpredictable, with many specific behaviors such as node flattening, insertion modes, and the stack of open elements capable of generating various mutations that lead to unexpected results.

Moreover, while the latest DOMPurify fix is robust, it also means that the library's security now relies heavily on a single regular expression. In the second article, we will explore how and why this reliance can become problematic in certain configurations and use cases :D

Finally, I would like to thank @IcesFont, @hash_kitten, and @ryotkak for allowing me to write about all the findings. Additionally, I want to extend my gratitude to @cure53berlin for their incredible responsiveness to each report

DOMPurify is an amazing library; keep using it!

> Click here to continue with Part 2.

Bibliography

- cure53. DOMPurify. https://github.com/cure53/DOMPurify
- @gregxsunday. $3,133.70 XSS in golang's net/html library - My first Google bug bounty. https://www.youtube.com/watch?v=H1TVk3HhL9E
- WhatWG. HTML specification - Serialising html fragments. https://html.spec.whatwg.org/#serialising-html-fragments
- @SecurityMB. Mutation XSS via namespace confusion - DOMPurify < 2.0.17 bypass. https://research.securitum.com/mutation-xss-via-mathml-mutation-dompurify-2-0-17-bypass/
- @BitK_. DOM Explorer. https://yeswehack.github.io/Dom-Explorer/
- WhatWG. HTML specification. https://html.spec.whatwg.org/
- WhatWG. HTML namespace. https://html.spec.whatwg.org
- W3.org. SVG namespace. https://www.w3.org/TR/SVG2/
- W3.org. MathML namespace. https://www.w3.org/TR/MathML/chapter2.xml
- WhatWG. HTML specification - HTML integration points. https://html.spec.whatwg.org/#html-integration-point

- WhatWG. HTML specification - MathML text integration points. https://html.spec.whatwg.org/#html-integration-point
- WhatWG. HTML Specification - Tree construction. https://html.spec.whatwg.org/#tree-construction
- WhatWG. HTML Specification - HTML insertion modes. https://html.spec.whatwg.org/#the-insertion-mode
- WhatWG. HTML Specification - stack of open elements. https://html.spec.whatwg.org/#the-stack-of-open-elements
- WhatWG. HTML Specification - in caption insertion mode. https://html.spec.whatwg.org/#parsing-main-incaption
- WhatWG. HTML Specification - Has an element in the specific scope. https://html.spec.whatwg.org/#has-an-element-in-the-specific-scope
- Mermaid. Mermaid.js. https://github.com/mermaid-js/mermaid

Archived from https://mizu.re/post/exploring-the-dompurify-library-bypasses-and-fixes. Rendered offline from the local Markdown copy.