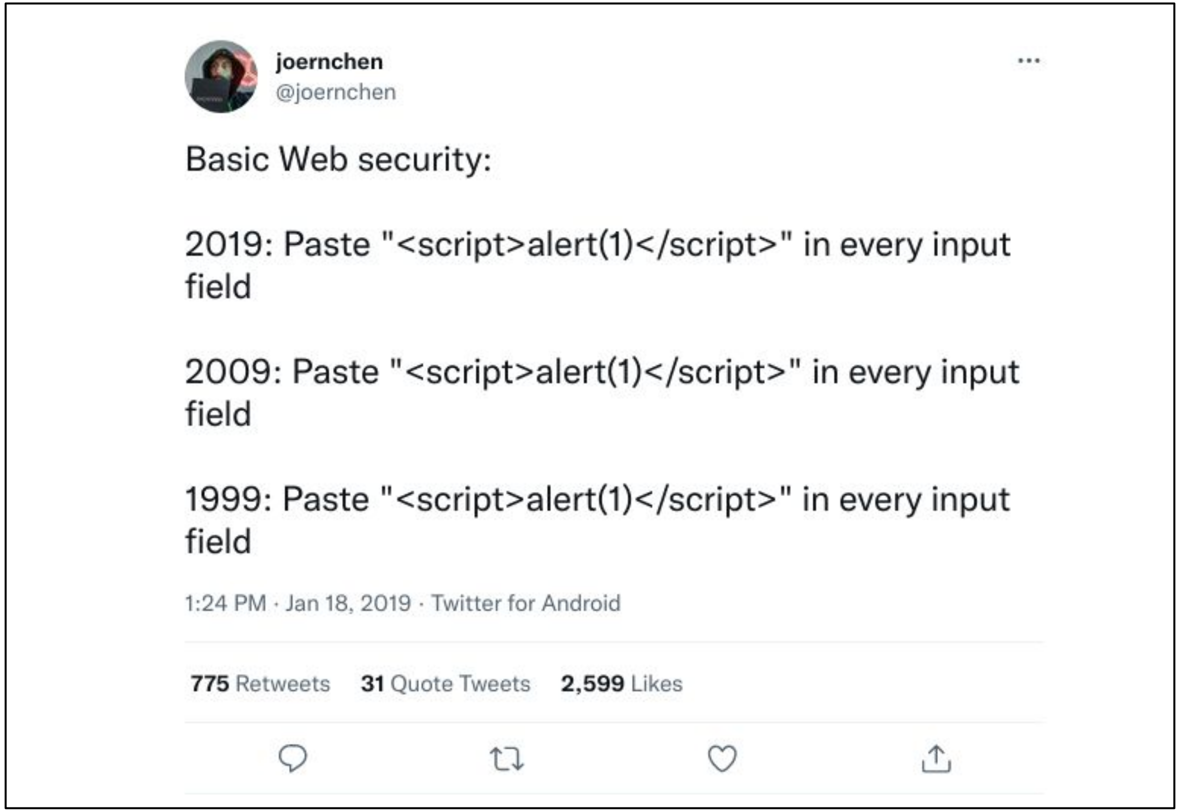
Making of:
The Sanitizer API

Frederik Braun
Staff Security Engineer at Mozilla
@freddyb

Questions? Comments?

✉️ **freddyb@mozilla.com**


joernchen
@joernchen

Basic Web security:

2019: Paste "<script>alert(1)</script>" in every input field

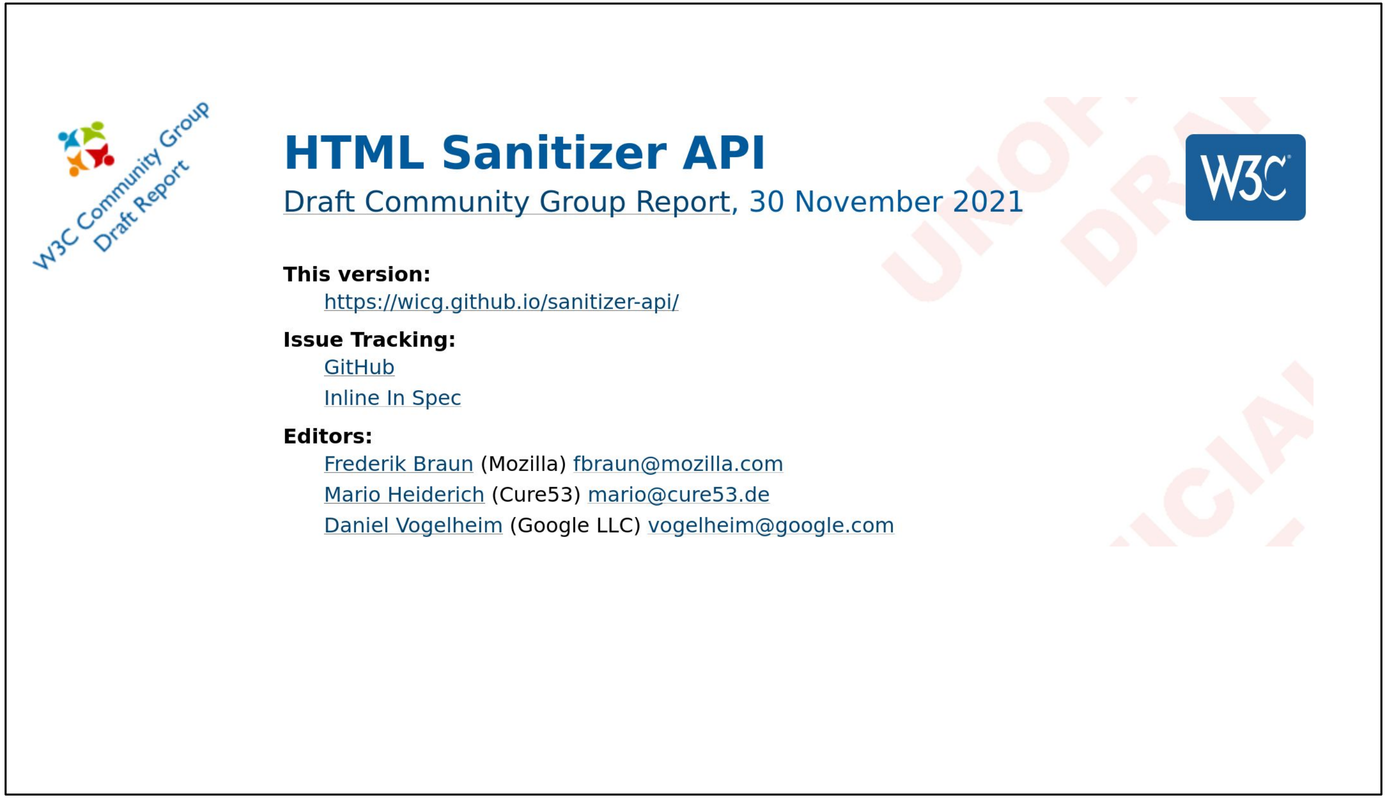
2009: Paste "<script>alert(1)</script>" in every input field

1999: Paste "<script>alert(1)</script>" in every input field

1:24 PM · Jan 18, 2019 · Twitter for Android

775 Retweets 31 Quote Tweets 2,599 Likes

[Source: https://twitter.com/joernchen/status/1086237923652046849]



HTML Sanitizer API

Draft Community Group Report, 30 November 2021

This version:
https://wicg.github.io/sanitizer-api/

Issue Tracking:
GitHub
Inline In Spec

Editors:
Frederik Braun (Mozilla) fbraun@mozilla.com
Mario Heiderich (Cure53) mario@cure53.de
Daniel Vogelheim (Google LLC) vogelheim@google.com

[Source: https://wicg.github.io/sanitizer-api/]

Frederik Braun
@freddyb

Staff Security Engineer
moz://a

[Subresource Integrity]()

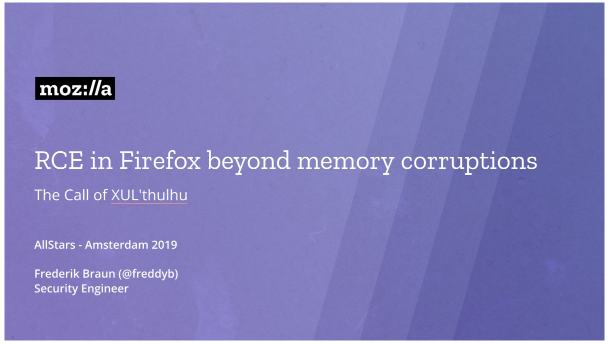
[X-Frame-Options: All about Clickjacking?]()

[eslint-plugin-no-unsanitized]()

```
foo.innerHTML = evil
```

DOM-based XSS

As an Aside



moz://a

RCE in Firefox beyond memory corruptions

The Call of XUL'thulhu

AllStars - Amsterdam 2019

Frederik Braun (@freddyb)
Security Engineer

moz://a

Finding & Fixing DOM-based XSS

Using Static Analysis

JSCamp 2021

Frederik Braun (@freddyb)
Staff Security Engineer at Mozilla

How people fix DOM-based XSS

Use `textContent`

Vulnerable code line doesn't need HTML?

Use **textContent** instead.

Done!

Encode and escape

Replace dangerous stuff with e.g., **HTML entities.**

Sanitize

If you want to allow some safe subset of HTML, use a **Sanitizer**

Fixing DOM-based XSS

Use `textContent`

Vulnerable code line doesn't need HTML?

Use **textContent** instead.

Done!

Encode and escape

Replace dangerous stuff with e.g., **HTML entities.**

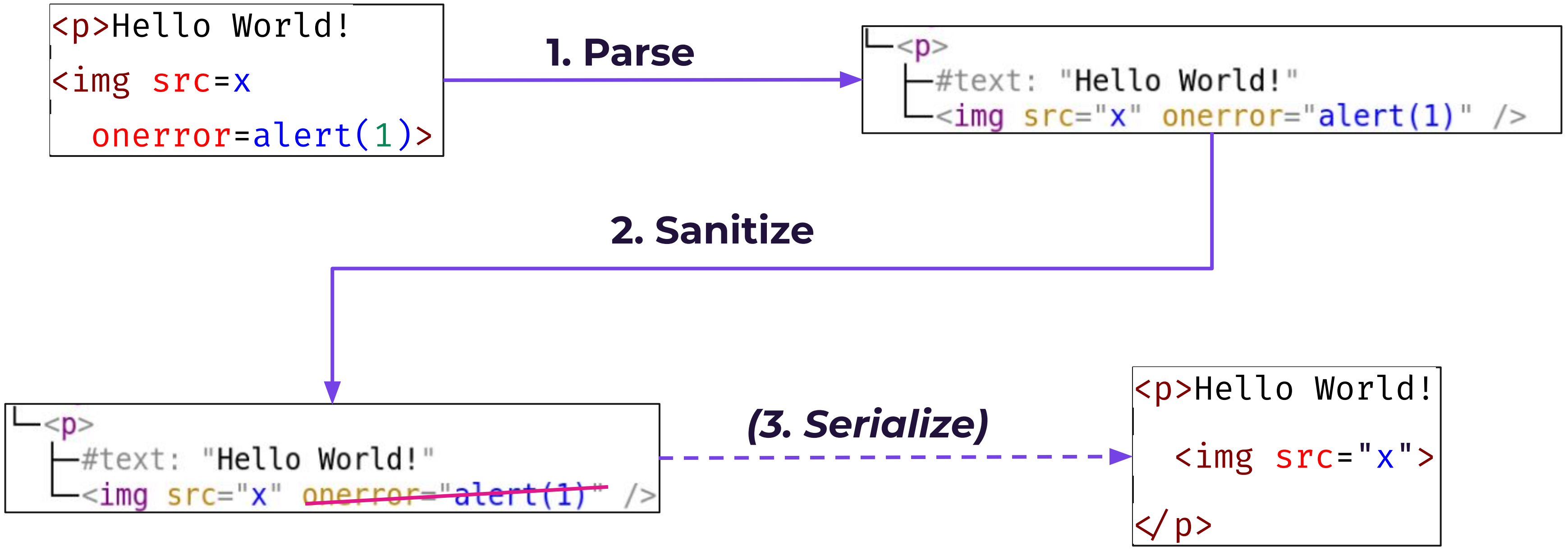
Sanitize

You are here 📌

If you want to allow some safe subset of HTML, use a **Sanitizer**

What is a Sanitizer?

What's in a Sanitizer?



```
<p>Hello World!
<img src=x
  onerror=alert(1)>
```

```
└<p>
  ├#text: "Hello World!"
  └<img src="x" onerror="alert(1)" />
```

```
└<p>
  ├#text: "Hello World!"
  └<img src="x" ~~onerror="alert(1)"~~ />
```

```
<p>Hello World!
  <img src="x">
</p>
```

Your Sanitizer is mostly an HTML Parser!

A Sanitizer API

Goals

1. Defining “Safe HTML”
2. Safe by Default
3. No Parsing Mistakes
4. Configurable
5. Responsibility shift to the browser

```
foo.innerHTML = evil
```

DOM-based XSS

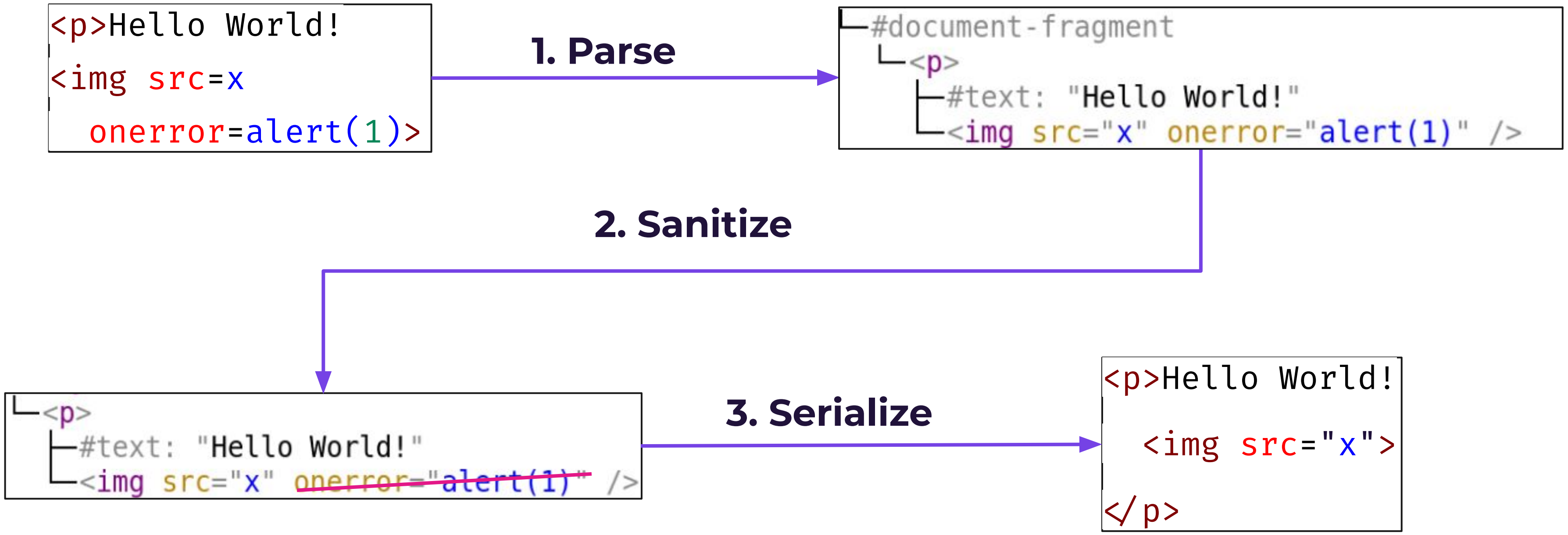
First Idea

```
mySanitizer = new Sanitizer(options)

mySanitizer.sanitize() // String
```

```
foo.innerHTML =
  mySanitizer.sanitize(evil)
```

What's in a Sanitizer?



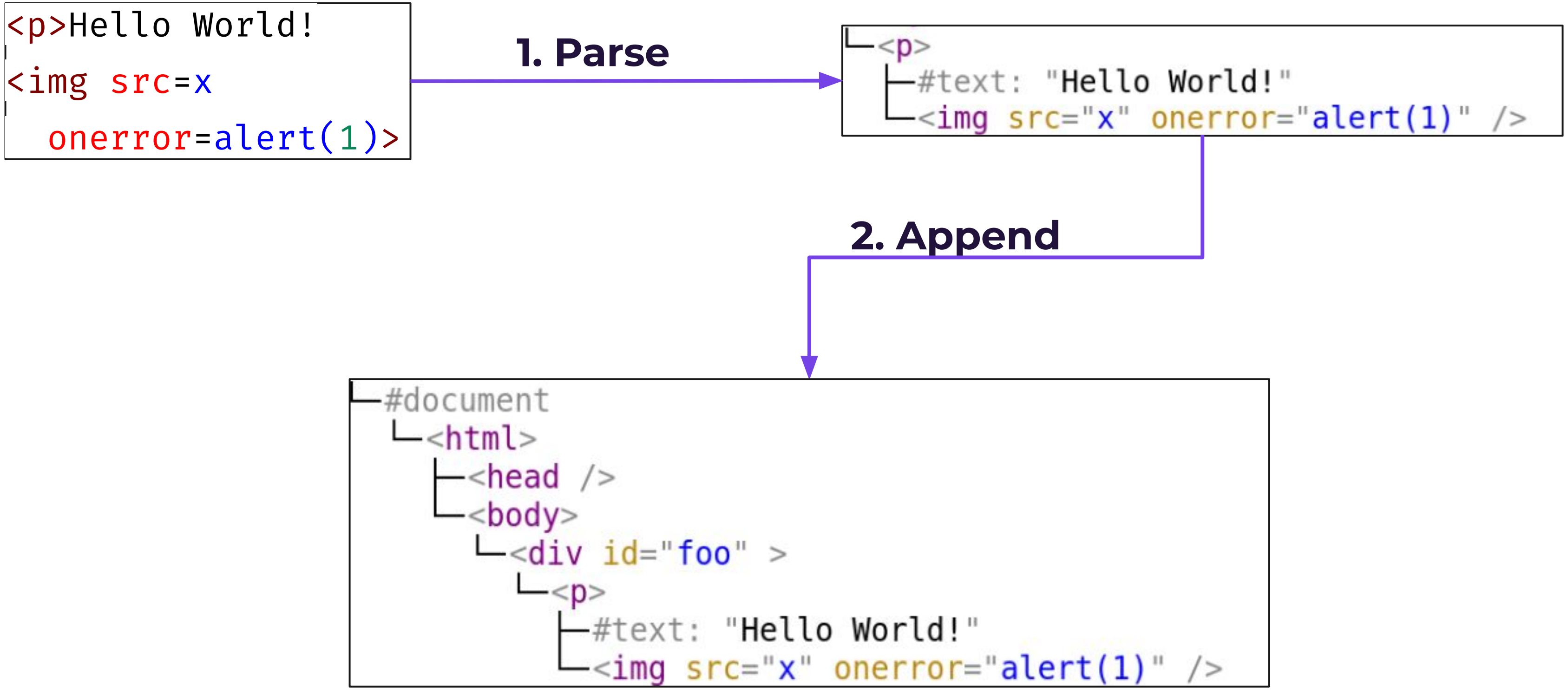
```
<p>Hello World!
<img src=x
  onerror=alert(1)>
```

```
└─#document-fragment
  └─<p>
    ├─#text: "Hello World!"
    └─<img src="x" onerror="alert(1)" />
```

```
└─<p>
  ├─#text: "Hello World!"
  └─<img src="x" ~~onerror="alert(1)"~~ />
```

```
<p>Hello World!
  <img src="x">
</p>
```

What's in "foo.innerHTML=" ?



```
<p>Hello World!
<img src=x
  onerror=alert(1)>
```

```
└─<p>
  ├─#text: "Hello World!"
  └─<img src="x" onerror="alert(1)" />
```

```
└─#document
  └─<html>
    ├─<head />
    └─<body>
      └─<div id="foo" >
        └─<p>
          ├─#text: "Hello World!"
          └─<img src="x" onerror="alert(1)" />
```

Now we’re using TWO HTML Parsers?!

API: Revision 1

```
mySanitizer = new Sanitizer(options)

mySanitizer.sanitize() // DocFragment

mySanitizer.sanitizeToString() // String
```

```
foo.append(mySanitizer.sanitize(evil))
```

Nothing *good* is designed in a vacuum

Looking for Bugs here. Anyone got some bugs?

Sanitizer is less expressive than `innerHTML`

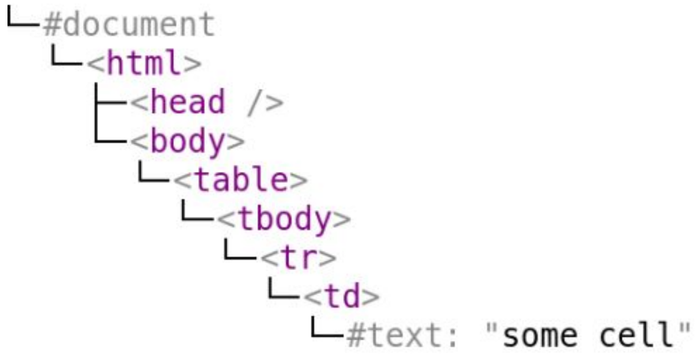
01

https://github.com/WICG/sanitizer-api/issues/42
Reported by Anne van Kesteren (@annevk)

innerHTML

Without the Sanitizer

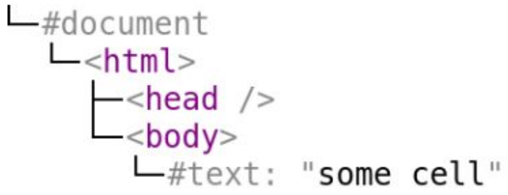
```
tableElement.innerHTML =
  "<tr><td>some cell</td></tr>"
```



```
└─#document
  └─<html>
    ├─<head />
    └─<body>
      └─<table>
        └─<tbody>
          └─<tr>
            └─<td>
              └─#text: "some cell"
```

With the Sanitizer

```
tableElement.append(
  mySanitizer.sanitize(sameInput))
```



```
└─#document
  └─<html>
    ├─<head />
    └─<body>
      └─#text: "some cell"
```

Parsing HTML fragments

§ 13.4 Parsing HTML fragments

The following steps form the **HTML fragment parsing algorithm.** The algorithm takes as input an `Element` node, referred to as the ***context*** element, which gives the context for the parser, as well as ***input***, a string to parse, and returns a list of zero or more nodes.

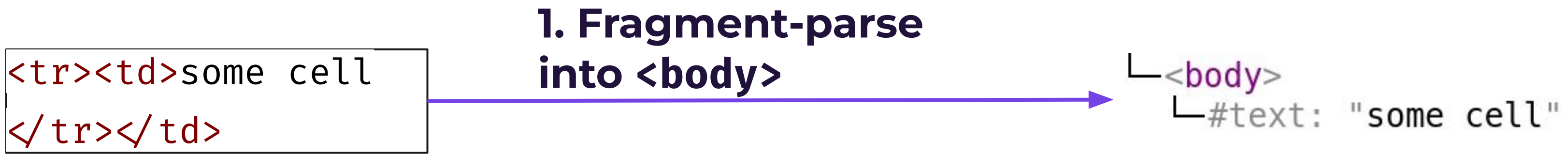
(...)

4. Set the state of the HTML parser's tokenization stage as follows, switching on the ***context*** element:

(long list of various html elements that cause different parsing behaviors)

[Source: https://html.spec.whatwg.org/multipage/parsing.html#parsing-html-fragments]

Fragment parsing without context



```
<tr><td>some cell
</tr></td>
```

1. Fragment-parse into <body>

```
└<body>
  └#text: "some cell"
```

Sanitizer Bypass with `iframe` `srcdoc`

02

https://bugzilla.mozilla.org/show_bug.cgi?id=1669945
Reported by Michał Bentkowski (@SecurityMB)

Michał’s Bypass: The code

```
<iframe id=ifr></iframe>
<script>
   const bypass =
      `<svg><font color><title><u rel="</title><img src onerror=alert(document.domain)>">`;
   ifr.srcdoc = new Sanitizer().sanitizeToString(bypass);

</script>
```

Parsing within sanitizeToString

```
<iframe id=ifr></iframe>
<script>
   const bypass =
      `<svg><font color><title><u rel="</title><img src onerror=alert(document.domain)>">`;
   ifr.srcdoc = new Sanitizer().sanitizeToString(bypass);

</script>
```

```
  └ <svg:svg>
    └ <svg:font color="">
      └ <svg:title>
        └ <html:u rel="</title><img src onerror=alert(1)>">
```

String returned from sanitizeToString

```
└ <svg:svg>
  └ <svg:font color="">
    └ <svg:title>
      └ <html:u rel="</title><img src onerror=alert(1)>">
```

↓

```
<svg>
<font color="">
<title>
<u rel="</title>
<img src onerror=alert(document.domain)>"></u></title></font></svg>
```

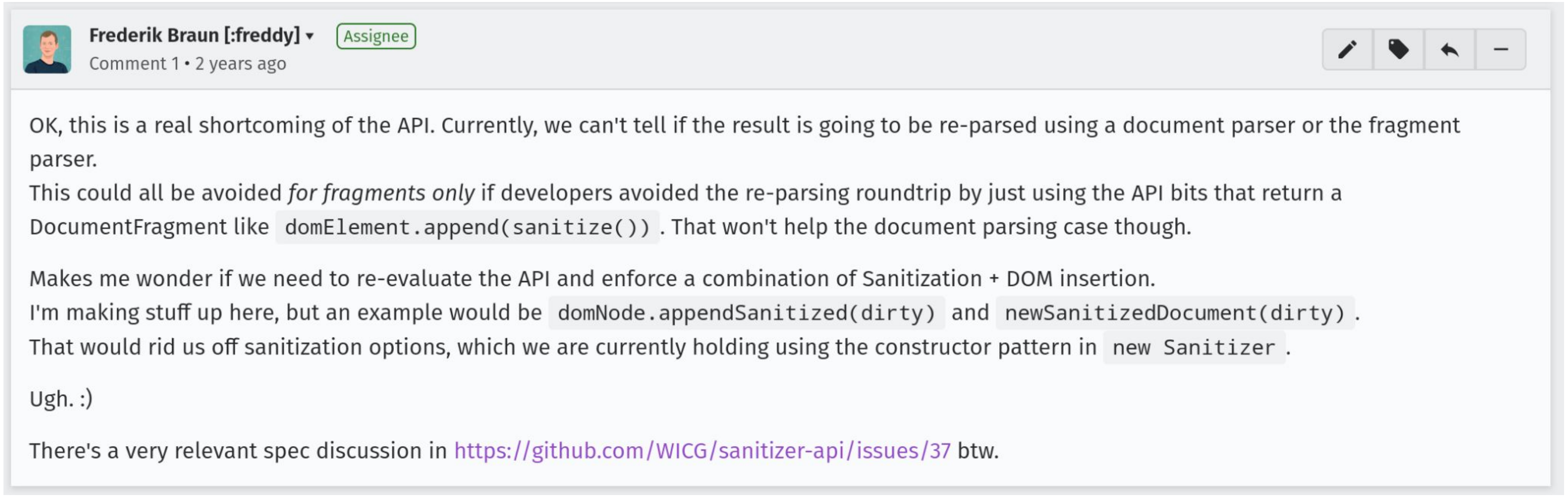
Parsing into the iframe srcdoc

```
<svg>
<font color="">
<title>
<u rel="</title>
<img src onerror=alert(document.domain)>"></u></title></font></svg>
```

```
┣ <svg:svg>
┗ <html:font color="">
  ┣ <html title>
  ┃ ┗ #text: <u rel="
  ┣ <html:img src="" onerror="alert(document.domain)"/>
  ┗ #text: ">
```

Burn all Parsers!

Parsers!!!1



Frederik Braun [:freddy] ▾ Assignee
Comment 1 • 2 years ago

OK, this is a real shortcoming of the API. Currently, we can't tell if the result is going to be re-parsed using a document parser or the fragment parser.
This could all be avoided *for fragments only* if developers avoided the re-parsing roundtrip by just using the API bits that return a DocumentFragment like `domElement.append(sanitize())`. That won't help the document parsing case though.

Makes me wonder if we need to re-evaluate the API and enforce a combination of Sanitization + DOM insertion.
I'm making stuff up here, but an example would be `domNode.appendSanitized(dirty)` and `newSanitizedDocument(dirty)`.
That would rid us off sanitization options, which we are currently holding using the constructor pattern in `new Sanitizer`.

Ugh. :)

There's a very relevant spec discussion in https://github.com/WICG/sanitizer-api/issues/37 btw.

```
foo.setHTML(evil, { sanitizer: mySanitizer })
foo.setHTML(evil)
```

Security Considerations

Server-Side Reflected and Stored XSS

DOM clobbering

XSS with Script gadgets

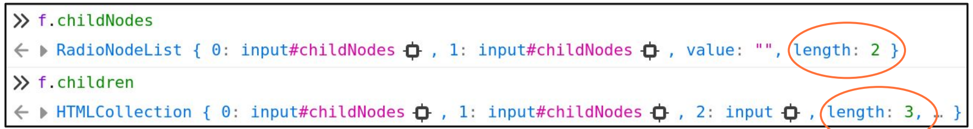
Mutated XSS

Server-Side XSS

The Sanitizer API is just for DOM-based XSS.

DOM clobbering

```html
<form id=f>
  <input id=childNodes>foo
  <input id=childNodes>bar
  <input type=text
         value="hidden-from-js">
```



```
>> f.childNodes
← ▸ RadioNodeList { 0: input#childNodes , 1: input#childNodes , value: "", length: 2 }
>> f.children
← ▸ HTMLCollection { 0: input#childNodes , 1: input#childNodes , 2: input , length: 3, … }
```

[More: https://portswigger.net/web-security/dom-based/dom-clobbering]

DOM clobbering

Sanitizer API is looking through clobbered properties

Preventing it in your app is currently out of scope.

You could configure the sanitizer to disallow e.g., `name` & `id` attributes.

XSS with Script gadgets

```
<button data-html="injection here"
  data-html-enabled="true"></button>
```

[Credit: https://research.google/pubs/pub46450/]

XSS with Script gadgets

The Sanitizer can not prevent these attacks.

But you can disallow e.g., `data-` or `role` attributes if you customize it according to your framework(s)

mXSS

```
<img src=" test.jpg" alt="``onload=xss()" />
```

```
<IMG alt=`` onload=xss() src="test.jpg">
```

[Credit: https://dl.acm.org/doi/10.1145/2508859.2516723]

mXSS

The Sanitizer offers help against mXSS.

Parse at your own peril.

Nothing *good* is developed without feedback.

We’re still not done here. Gimme moar bugs.

Bounties

1. Enable the Sanitizer

   Go to *about:config*. Toggle *dom.security.sanitizer.enabled*

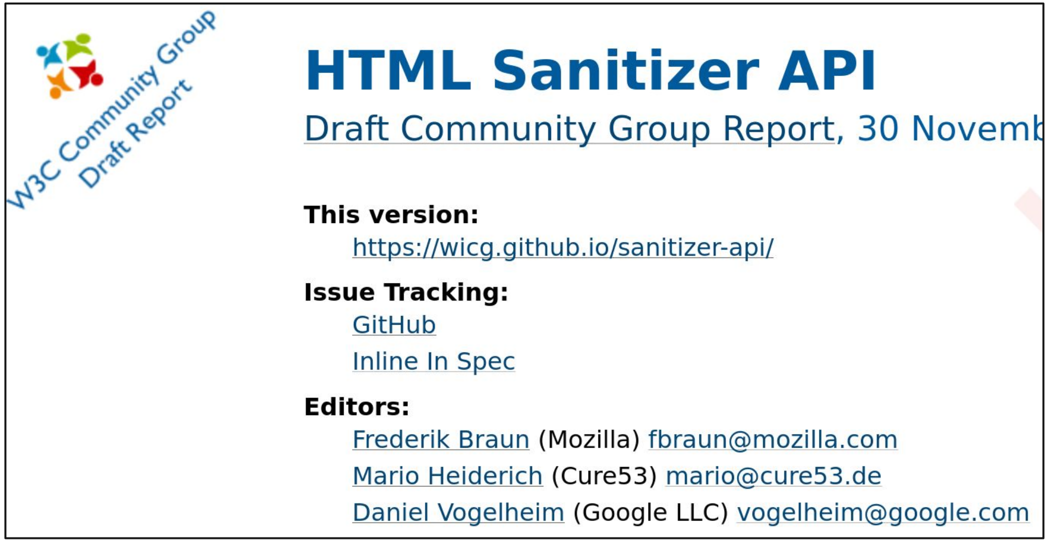
   about://flags#sanitizer-api or “*Experimental Web Platform Features*”

2. Go to empty web page and open Developer Tools

3. `document.body.setHTML(evil)`

4. Profit

Discussion


W3C Community Group Draft Report

HTML Sanitizer API

Draft Community Group Report, 30 Novemb

This version:
https://wicg.github.io/sanitizer-api/

Issue Tracking:
GitHub
Inline In Spec

Editors:
Frederik Braun (Mozilla) fbraun@mozilla.com
Mario Heiderich (Cure53) mario@cure53.de
Daniel Vogelheim (Google LLC) vogelheim@google.com

https://wicg.github.io/sanitizer-api/

Coding

1. Add more tests to web-platform-tests

2. Polyfill at https://github.com/mozilla/sanitizer-polyfill/

Burn all Parsers!

If you need an HTML Parser, make sure you pick the right one.

Thank you!

Frederik Braun (@freddyb)
Staff Security Engineer at Mozilla

Questions? Comments?

✉️ freddyb@mozilla.com