moz://a

Finding & Fixing DOM-based XSS

Using Static Analysis

enterJS 2022

Frederik Braun (@freddyb@security.plumbing)
Staff Security Engineer at Mozilla

Questions? Comments?

Feel free to write me at **fbraun@mozilla.com**
or **@freddyb@security.plumbing**

Agenda

1. DOM-based Cross-Site Scripting (XSS)
2. Linting & Static Analysis
3. Case Study: DOM-based XSS in Firefox UI code
4. The Sanitizer API

Intro

DOM-based XSS

```
let imageURL = location.hash.slice(1);
let html = `
   <div class="image-box">
     <img class="image"
          src="${imageURL}" />
   </div>`;
 // ( ... )
 main.innerHTML = html;
```


```
let imageURL = location.hash.slice(1);
let html = `
   <div class="image-box">
     <img class="image"
          src="${imageURL}" />
   </div>`;
 // ( ... )
 main.innerHTML = html;
```

Source

```
let imageURL = location.hash.slice(1);
let html = `
    <div class="image-box">
      <img class="image"
           src="${imageURL}" />
    </div>`;
  // ( ... )
  main.innerHTML = html;
```

Source

Sink

```
let imageURL = location.hash.slice(1);
let html = `
   <div class="image-box">
     <img class="image"
          src="${imageURL}" />
   </div>`;
 // ( ... )
 main.innerHTML = html;
```

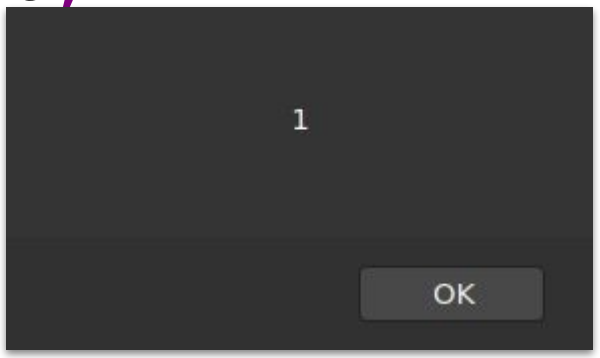
" onerror="alert(1)

" onerror="alert(1)

```
let imageURL = location.hash.slice(1);
let html = `
    <div class="image-box">
      <img class="image"
           src="${imageURL}" />
    </div>`;
  // ( ... )
  main.innerHTML = html;
```

" onerror="alert(1)

" onerror="alert(1)



Finding ALL the DOM-XSS

DOM XSS Sinks

and where to find them

What else, besides `innerHTML`?

What if we could just **disallow using all those *sinks***?

DOM XSS Sinks

and where to find them

What other *functions* or *properties* should we be concerned about?

DOM XSS Sinks

Some Examples

1. Assignments
   `outerHTML` or **`innerHTML`**
   with **`=`** or **`+=`**

2. Function Calls
   `insertAdjacentHTML(), document.write(), document.writeln()`

3. Let’s disallow using these *sinks* with a *linter*

Restricting

Can we look at just the sinks (and not the sources)?

- Tracing information through the code would require us to have a model of the browser (DOM, etc.)
- Sources are too numerous!
  URL parameters, forms, cookies, ...
- Are there some reasonable short cuts?

Linting & Static Analysis

Finding DOM-based XSS

Approaches

Static Analysis

You are here 📌

- also known as “Source Code Analysis”
- Scanning through the source code

Dynamic Analysis

- also known as “Runtime Analysis”
- Executing the code

Caveats & Alternative Approaches

Static Analysis

You are here 📌

- No visibility into types or data
- Prone to reporting **False Positives**
- Blocked by minification, bundling, or obfuscation
- Fast

Dynamic Analysis

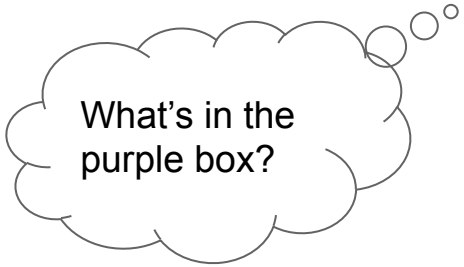
- Less prone to false positives
- Limited to the code that is being executed and instrumented
- Slow

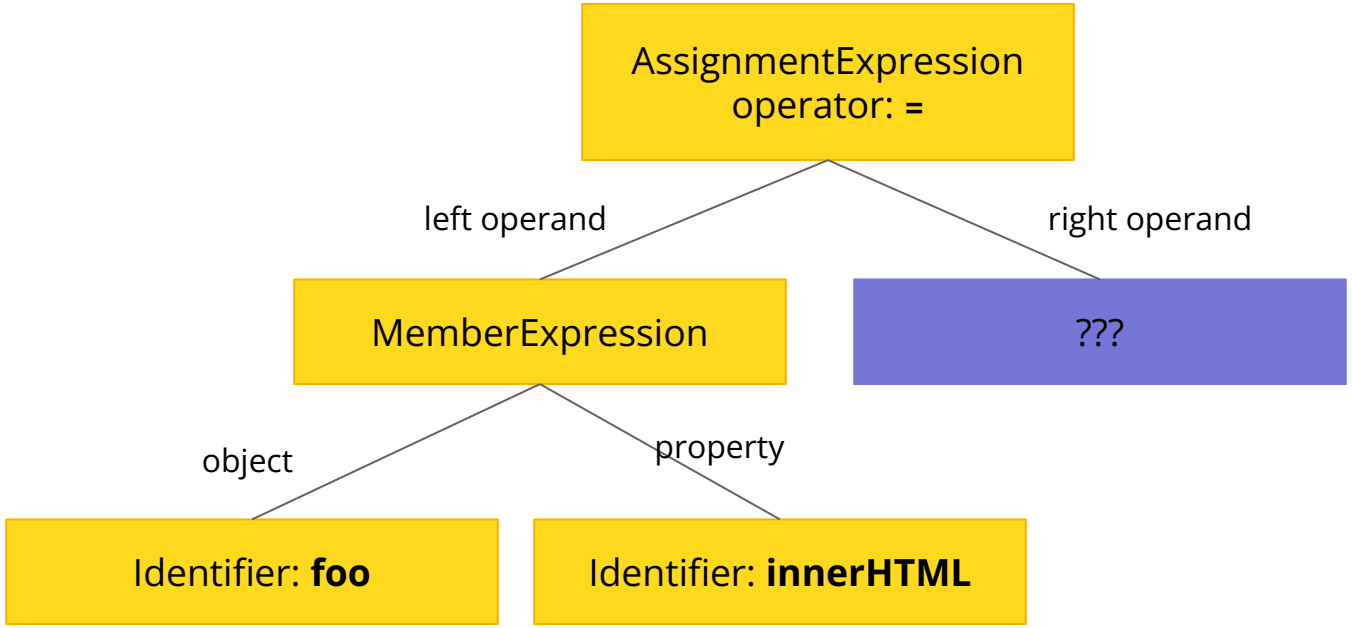

ESLint

```
foo.innerHTML = ""

foo.innerHTML = evil
```

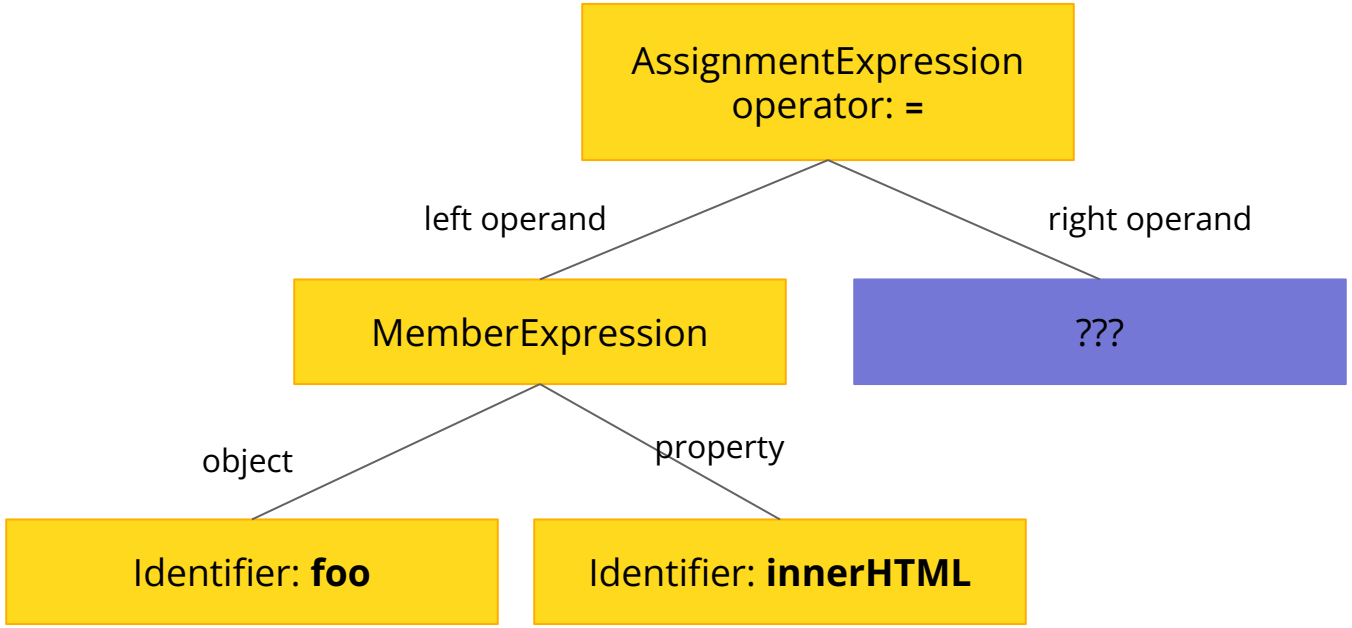
Abstract Syntax Tree (AST)





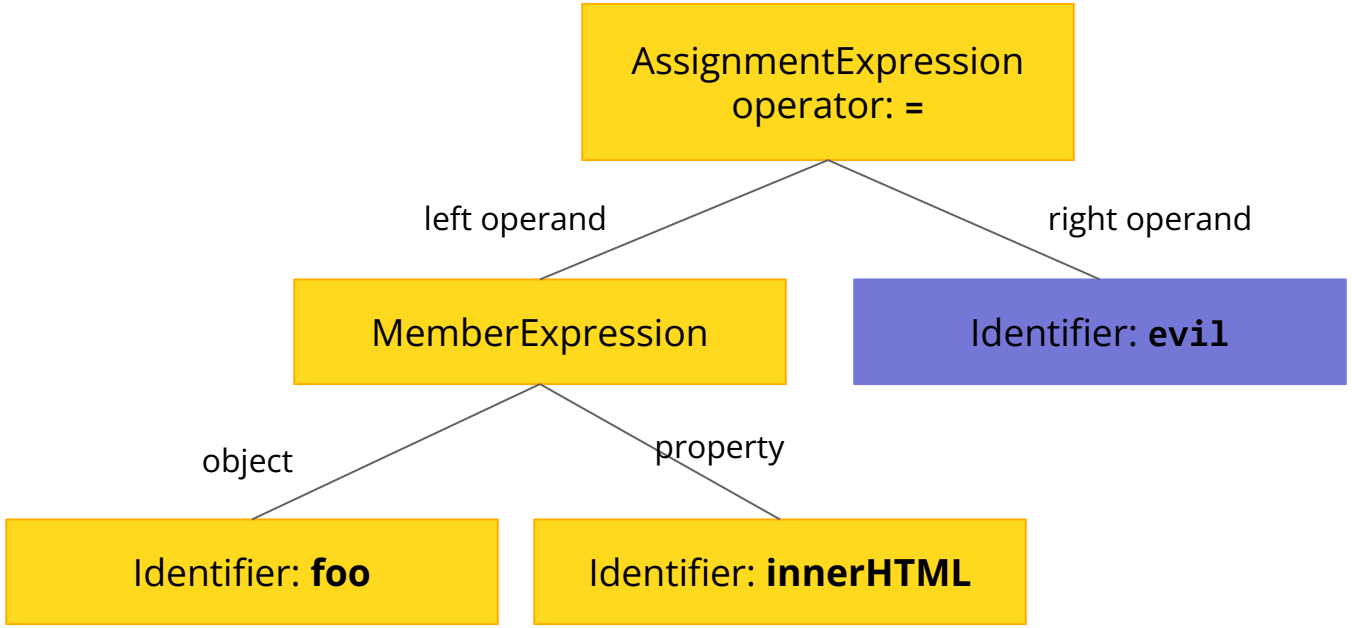

```
foo.innerHTML = evil
```

Abstract Syntax Tree (AST)



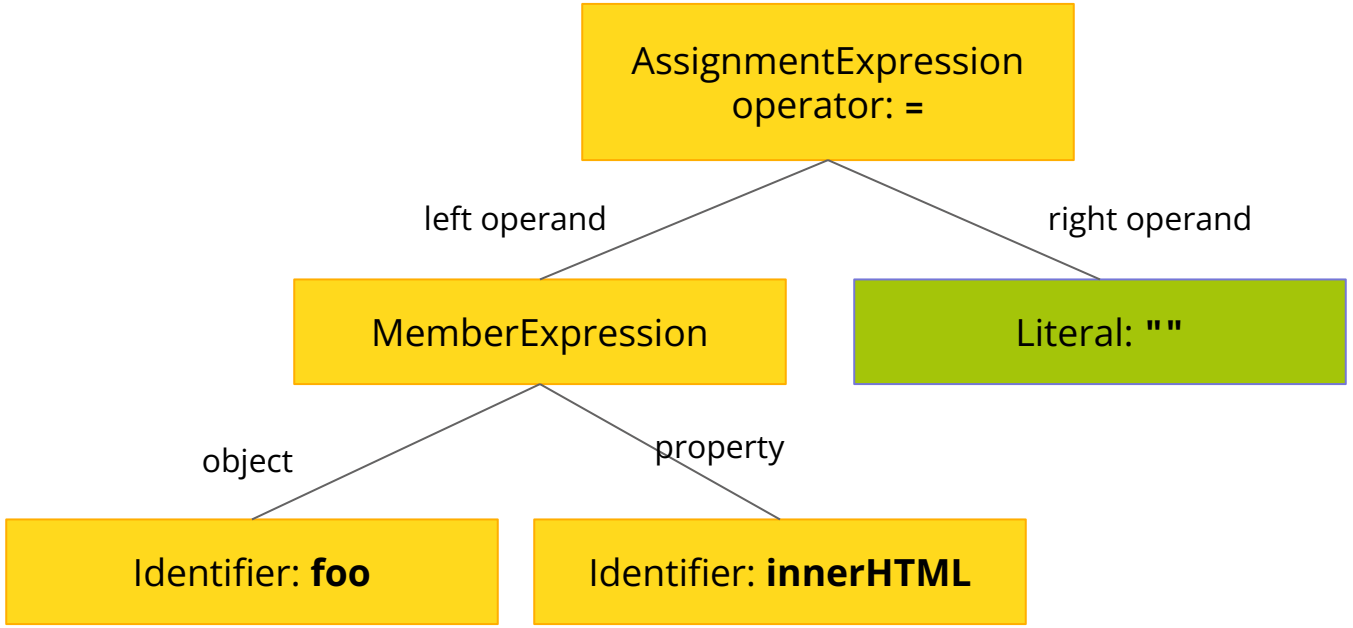

```
foo.innerHTML = evil
```

Abstract Syntax Tree (AST)




```
foo.innerHTML = evil
```

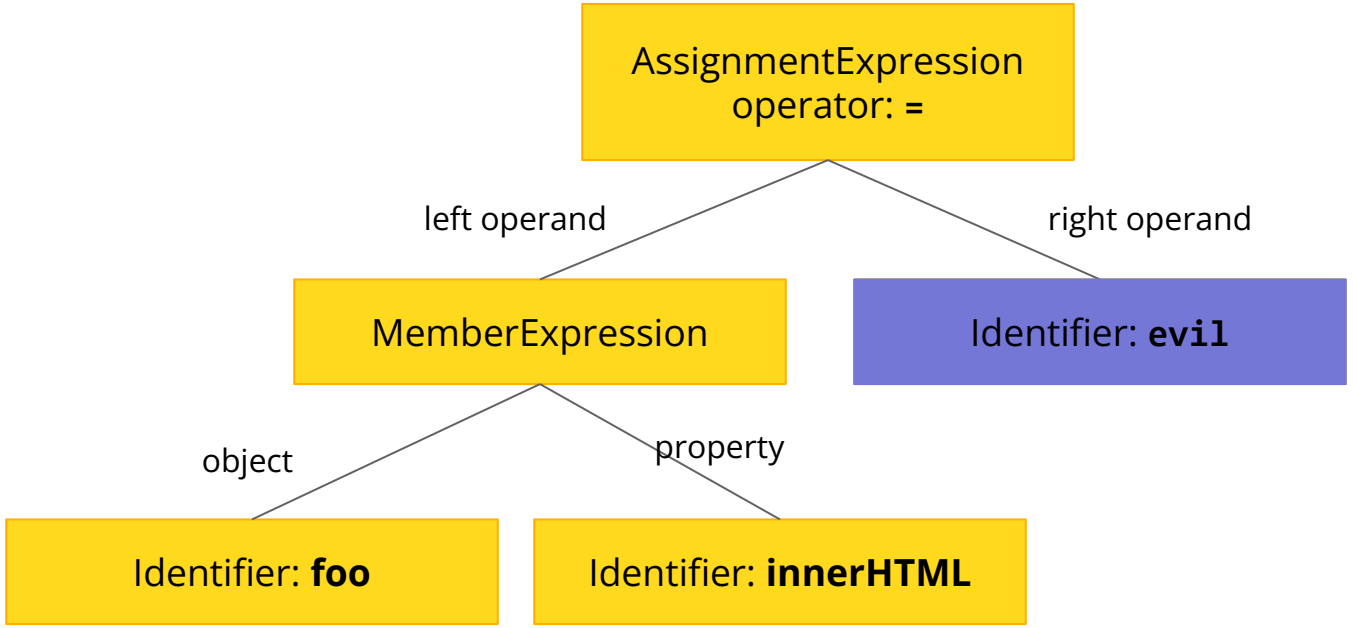
Abstract Syntax Tree (AST)




```
foo.innerHTML = ""
```

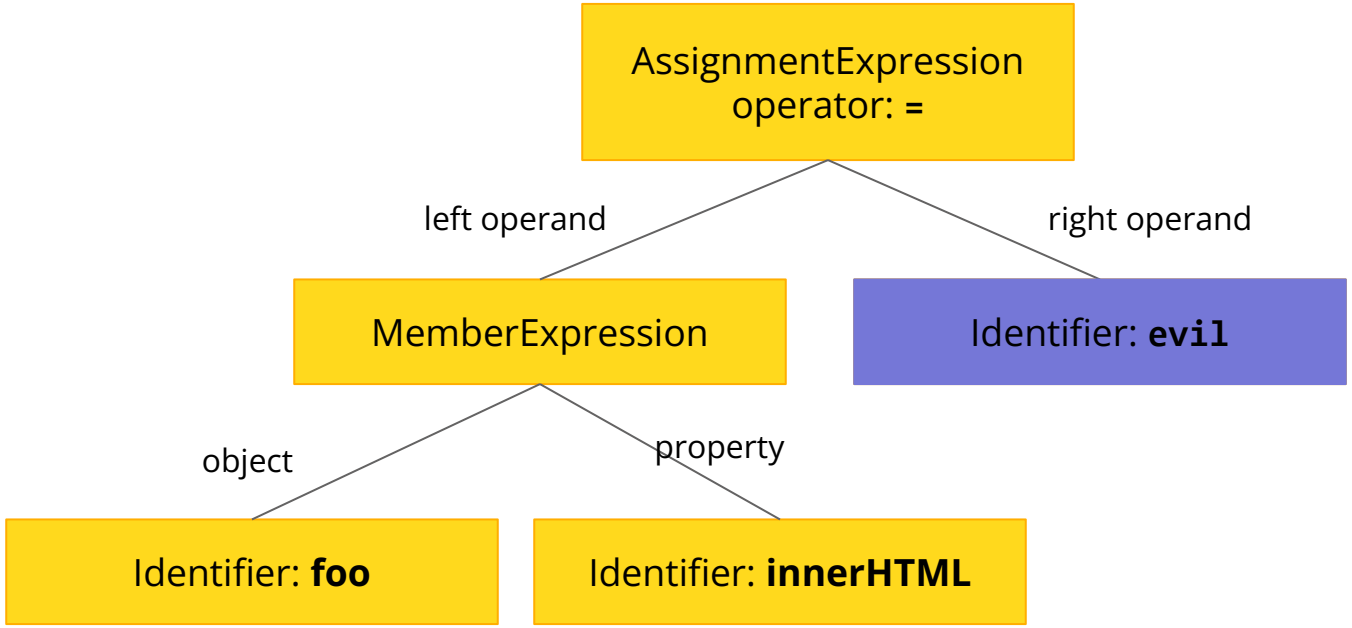
Abstract Syntax Tree (AST)

What do we actually know about `evil`?




```
foo.innerHTML = evil
```

Abstract Syntax Tree (AST)




```
foo.innerHTML = evil
```

Avoiding False Positives

1. Allow pure hardcoded strings.

2. Trace variables back to definition

3. Allow configuring a Sanitizer:
   Maybe the return value of `sanitize(evil)` isn't evil anymore?

4. Ignore code in `tests/`

mozilla/eslint-plugin-no-unsanitized:

Custom ESLint rule to disallows unsafe innerHTML, outerHTML, insertAdjacentHTML and alike

https://github.com/mozilla/eslint-plugin-no-unsanitized

mozilla/eslint-plugin-no-unsanitized:

Custom ESLint rule to disallows unsafe innerHTML, outerHTML, insertAdjacentHTML and alike

https://github.com/mozilla/eslint-plugin-no-unsanitized

Case Study

Firefox Browser Frontend

eslint-plugin-no-unsanitized versus Firefox

Numbers from Spring 2017

1000+

when searching with `grep`

34

linter violations

2

critical vulnerabilities

Integration

Preventing this from happening again

Fix obvious bugs first

- Focus on what's doable
- Remove XSS and self-XSS issues,

Exceptions

- Allow for exceptions.
- per-directory
  per file
  or per line
- Ensure they are temporary!

Add linter to CI

- Check all commits
- Violations won't be merged

Fixing DOM-based XSS

Introducing the Sanitizer API

HTML Sanitizer API

Draft Community Group Report, 14 June 2022

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

Sanitizer API - Upcoming Browser Specification

Easy to Use

1. `<img src=x onerror=alert(1)>`
2. ✨ Sanitizer API ✨
3. `<img src="x">`

Sanitizer API - Upcoming Browser Specification

Easy to Use

1. `<img src=x onerror=alert(1)>`
2. ✨ Sanitizer API ✨
3. `<img src="x">`

Easy to Include

- Safe By Default
- Customizable Sanitizer logic

Sanitizer API - Upcoming Browser Specification

Easy to Use

1. `<img src=x onerror=alert(1)>`
2. ✨ Sanitizer API ✨
3. `<img src="x">`

Easy to Include

- Safe By Default
- Customizable Sanitizer logic

Battle-tested

- No parsing mistakes!
- Shifts the responsibility to the browser (updates about every 4 weeks)

New `Element.setHTML()` function

```
let mySanitizer = new Sanitizer(/* config optional */);


someElement.setHTML('<img src="x" onerror="alert(1)">',
  { sanitizer: mySanitizer}
);


// someElement.innerHTML is now `<img src="x">`
```

Sanitizer Object is *also* optional

New `Element.setHTML()` function

```
someElement.setHTML(`<img src="x"
onerror="alert(1)">`);

// someElement.innerHTML is now `<img src="x">`
```

You can fix DOM XSS

Thank You!

Frederik Braun (@freddyb@security.plumbing)
Staff Security Engineer at Mozilla

Backup Slides

Sanitizers: Uncovered

What's in a Sanitizer?

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

1. Parse

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

2. Sanitize

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

3. Serialize

```
<p>Hello
World!

  <img
src="x">

</p>
```

What's in "foo.innerHTML=" ?

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

1. Parse with foo as *context*

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

2. Append

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

Now we're using TWO HTML Parsers?!

Sanitizer is less expressive than `innerHTML`

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"
```

Nothing *good* is developed without feedback.

Try it out

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

Discuss it out

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/

Code

1. Add more cross-browser test cases to wpt
   https://github.com/web-platform-tests/wpt/

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

That's it. Really.

Frederik Braun (@freddyb@security.plumbing)
Staff Security Engineer at Mozilla