Exploiting CSP in Webkit to Break Authentication & Authorization

This blog post will discuss our findings that we presented in our Blackhat Europe talk titled "Exploiting CSP in Webkit to break Authentication/Authorization", a vulnerability that enabled us to takeover user accounts on most of the web applications out thereby exploiting a bug in CSP in WebKit.

By Guest

14 мин. на чтение

Посмотреть оригинал

When it comes to the web, browsers are the first line of defence. While built-in security features that come compiled with browsers are responsible for preventing a wide array of attacks, any seemingly trivial mistake in browsers' implementation can have devastating effects. In this paper, we talk about a vulnerability in WebKit, a browser engine used by Safari, and our journey throughout.

Safari by far is the second most used browser with ~19% of the world's browser market share. While many other browsers like Chrome, Edge, Opera share a common browser engine, Safari has its own, called- WebKit. And, unbeknownst to many of us, all browsers on iOS devices like iPhone or iPad must use WebKit as their core engine.

This blog post will discuss our findings that we presented in our Blackhat Europe talk titled "Exploiting CSP in Webkit to break Authentication/Authorization", a vulnerability that enabled us to take over user accounts on most of the web applications out thereby exploiting a bug in CSP in WebKit.

TLDR;

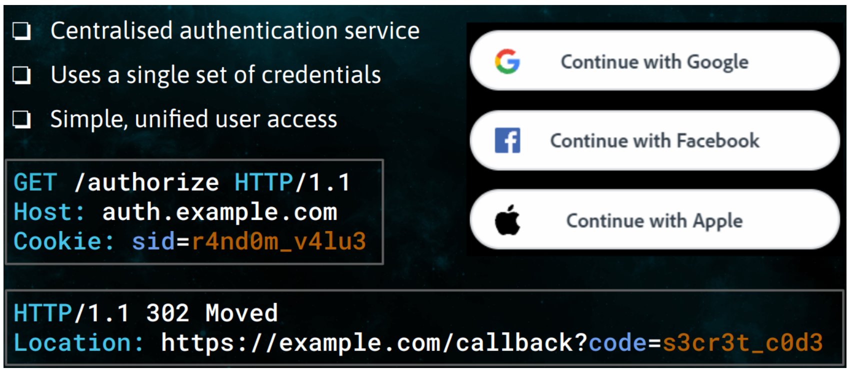
Long story short, there was a vulnerability that we reported to Safari that Apple didn't consider severe enough to fix quickly which then after waiting for a significant amount of time, we decided to exploit and earn some bounties by reporting them to bug bounty programs.

The bug was in the CSP implementation of WebKit, a browser engine used by Safari web browser, as well as all iOS web browsers including Firefox and Chrome. The failure of WebKit to adhere to W3C specification when a CSP violation event is triggered leaked cross-site destination URI while redirecting from one origin to a different one. Combined with the common practice of implementing SSO's cross-site redirects to authenticate and authorize, we were able to steal codes/acces_tokens or any other secrets that were part of the leaked URI which we could then further use arbitrarily. This allowed us to carry out attacks including but not limited to account takeovers, CSRF and sensitive information disclosure.

But first, let's discuss a few technologies and concepts before moving on to the vulnerability itself.

[Jump to POC]

Single Sign-On (SSO)



- ❏ Centralised authentication service
- ❏ Uses a single set of credentials
- ❏ Simple, unified user access

```
GET /authorize HTTP/1.1
Host: auth.example.com
Cookie: sid=r4nd0m_v4lu3
```

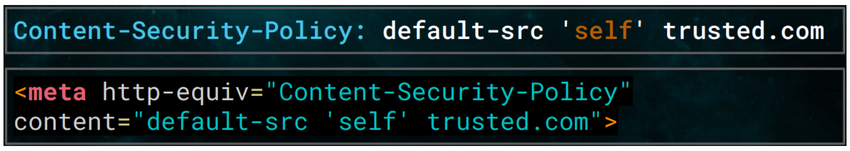
```
HTTP/1.1 302 Moved
Location: https://example.com/callback?code=s3cr3t_c0d3
```

Single Sign-On also commonly known as SSO is a technology that provides a centralised authentication service. Generally, users are asked to provide their single set of credentials, and authentication to all other applications are verified through a centralised authentication service. Its ease of use and effectiveness has made SSO really popular among modern applications. In addition to being simple and easy to use, SSO is also considered more secure.

SSO commonly works by generating an authentication token which is then passed to other applications and services. The authentication token enables services to verify and authorize an identity which is why securing and making sure that authentication tokens are kept safely is of utmost importance. Read more…

Content Security Policy (CSP)

Content-Security-Policy or CSP is a set of policy/rules defining how browsers should load resources like JavaScript, CSS or images and whether to allow or deny execution of such. It's defined using an HTTP response header called `Content-Security-Policy` or using `<meta>` tag with some exclusions like `report-uri` and `frame-ancestors`



```
Content-Security-Policy: default-src 'self' trusted.com
```

```
<meta http-equiv="Content-Security-Policy"
content="default-src 'self' trusted.com">
```

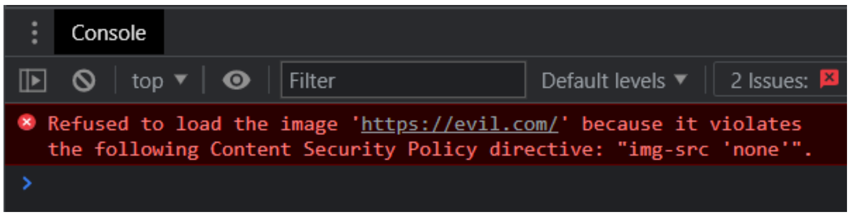
CSP has been around since 2010 and is now supported by all modern browsers. We can also see it being implemented widely in Web2.0 apps and a lot of improvements happening over the years. Though, setting the CSP header still seems to be a hard task because it depends on resources used by each particular site/page. That's probably why there is not something as a *default policy*. Read more…

CSP Violation Reports

It’s not that uncommon to make mistakes while implementing a Content-Security-Policy and for that reason, we also have something called Content-Security-Policy-Report-Only header. And what it does is- instead of actually executing the policy in action, it only reports violations which makes it much easier to debug if a policy is working as intended or not without breaking anything. So, when it comes to CSP, we always have what we call “violation reports” because what if a user tries to load a restricted resource, right? In that case, browsers block the request and dispatch a violation report. There are currently 4 ways to catch a dispatched report-

1. CSP’s report-uri directive
2. report-to
3. Reporting Observer
4. SecurityPolicyViolationEvent

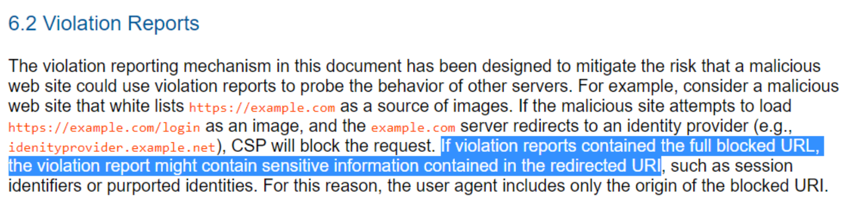
Throughout the post, we will be using SecurityPolicyViolationEvent for easier understanding. The report, in general, contains information on what caused the violation like violated-directive, original-policy, blockedURI or the URI the user requested and so on.



```
Refused to load the image 'https://evil.com/' because it violates
the following Content Security Policy directive: "img-src 'none'".
```

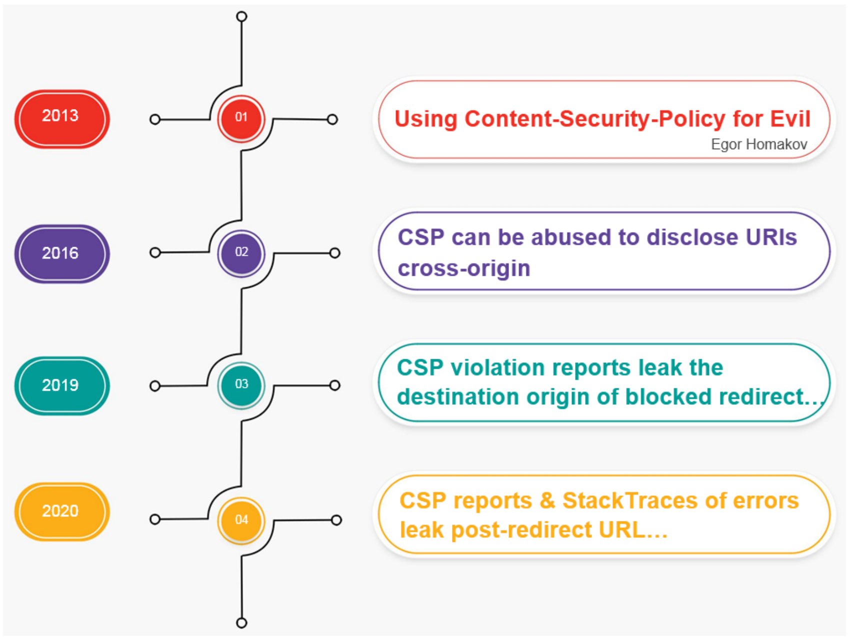
Here, for example, it discloses the URL the user tried to load i.e. evil.com. Though it’s not something sensitive here, it could be if it is a result of a redirect. In the case of OAuth, for example. So, to avoid leaking sensitive information, browsers only include the **origin of the original request**.

CSP1.0 Candidate Recommendation had it wrong and recommended to include the *origin of the blocked URI* which is still an issue and we have a long history of reports on that.



6.2 Violation Reports

The violation reporting mechanism in this document has been designed to mitigate the risk that a malicious web site could use violation reports to probe the behavior of other servers. For example, consider a malicious web site that white lists `https://example.com` as a source of images. If the malicious site attempts to load `https://example.com/login` as an image, and the `example.com` server redirects to an identity provider (e.g., `identityprovider.example.net`), CSP will block the request. If violation reports contained the full blocked URL, the violation report might contain sensitive information contained in the redirected URI, such as session identifiers or purported identities. For this reason, the user agent includes only the origin of the blocked URI.

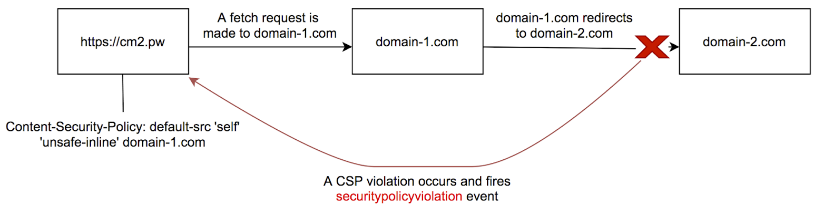

2013
01
Using Content-Security-Policy for Evil
Egor Homakov
2016
02
CSP can be abused to disclose URIs cross-origin
2019
03
CSP violation reports leak the destination origin of blocked redirect...
2020
04
CSP reports & StackTraces of errors leak post-redirect URL...

We found a few disclosed reports on Chromium's Issues leaking origin of `blocked-uri` in one way or another using CSP. All these reports share the same problem- the leakage of cross-origin redirect URL. The first one was published in 2013 when CSP was still in its infancy by Egor Homakov where he demonstrated what we could achieve by abusing CSP's `blocked-uri` directive. He said

> We can use CSP trick to detect if a user has authorized certain apps/clients

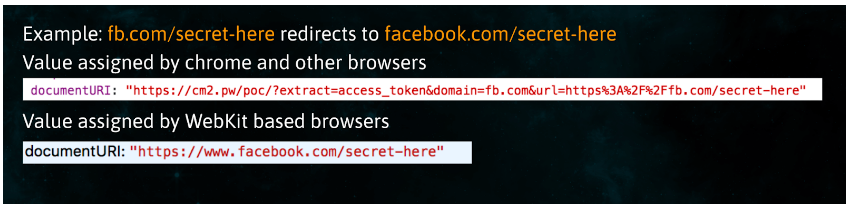
because all authorized apps will redirect to their homepage instead of login or authorization page. For example, if you've already authorized Instagram on your Facebook, sending a request to Facebook OAuth endpoint for Instagram would redirect to Instagram instead of Facebook. Therefore, further specifications of CSP was changed to recommend including the URL of the original request instead of redirected URL.

Root Cause of the Vulnerability


https://cm2.pw
A fetch request is made to domain-1.com
domain-1.com
domain-1.com redirects to domain-2.com
domain-2.com
Content-Security-Policy: default-src 'self' 'unsafe-inline' domain-1.com
A CSP violation occurs and fires securitypolicyviolation event

We can define a CSP policy in such a way that whenever a redirect happens to a domain with a different origin (also known as cross-origin) a CSP violation occurs, and a SecurityPolicyViolation event is fired. As you can see in this figure, our domain `cm2.pw` with described CSP policy makes a fetch request to `domain-1.com`, the response header from `domain-1.com` contains a `Location` header that now redirects to a domain with a different origin `domain-2.com`, but due to our CSP policy, the request to `domain-2.com` is blocked and a CSP violation event is fired.

Now when a SecurityPolicyViolation event is fired it uses SecurityPolicyViolationEvent interface at target with different attributes. While setting values for these attributes, a failure by WebKit to adhere to W3C specification resulted in leaking 30x redirection location in `documentURI` attribute.



Example: fb.com/secret-here redirects to facebook.com/secret-here

Value assigned by chrome and other browsers

```
documentURI: "https://cm2.pw/poc/?extract=access_token&domain=fb.com&url=https%3A%2F%2Ffb.com/secret-here"
```

Value assigned by WebKit based browsers

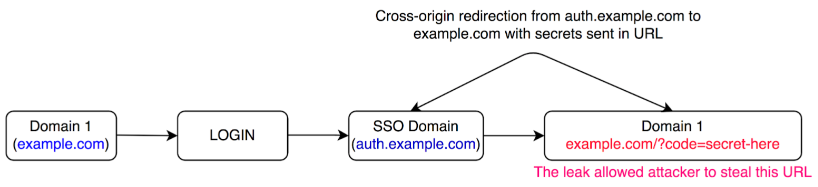
```
documentURI: "https://www.facebook.com/secret-here"
```

As you can see in the figure above,

`fb.com/secret-here` redirects to `facebook.com/secret-here`

Then you can see differences in values for attribute documentURI set by Safari and other browsers. While other browsers who have followed the specification mentioned by W3C seem to set the value of a URI with values from where the original fetch request was made, Safari who failed to follow the W3C specification set those values with the final redirection URI.

How can this be exploited in SSO?

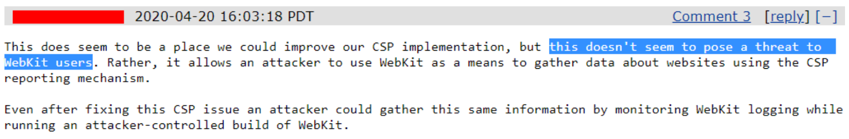

Cross-origin redirection from auth.example.com to example.com with secrets sent in URL
Domain 1 (example.com)
LOGIN
SSO Domain (auth.example.com)
Domain 1 example.com/?code=secret-here
The leak allowed attacker to steal this URL

As you can see in this figure here, when a user loads `example.com` it first checks if the user is authenticated or not and if the user is not authenticated, then as shown in the figure it will send the user to `auth.example.com` which is an SSO Domain that provides centralized authentication. Now after verifying the user, the user is then redirected back to `example.com` with an authentication token. Now the general common practice while implementing SSO is to use a different sub-domain or an entirely different domain, which then doesn't meet the criteria to be defined as same-origin, so the final redirection that happens from SSO domain to the final domain, or in the case as shown in the figure from `auth.example.com` to `example.com` will be cross-origin thus allowing an attacker to steal the final redirection URI.

In today's modern applications where authentication and authorization are heavily dependent on technologies such as OAuth, SSO and JWT where secrets are passed through URLs the leak of the final redirection URI makes applications using these technologies almost always vulnerable.

Responsible Disclosure to Safari

The responsible disclosure didn't go as we hoped it would. We first reported the vulnerability to Safari in February 2020 and had to wait for a few weeks before we even received an automatic acknowledgement. After 2 months of waiting, we finally received a reply on April 20 and to our surprise, it said "*this doesn't seem to pose a threat to WebKit users*". They further went on to explain that even after fixing the issue, an attacker could gather the same information by monitoring WebKit logs while running an attacker-controlled build of WebKit.



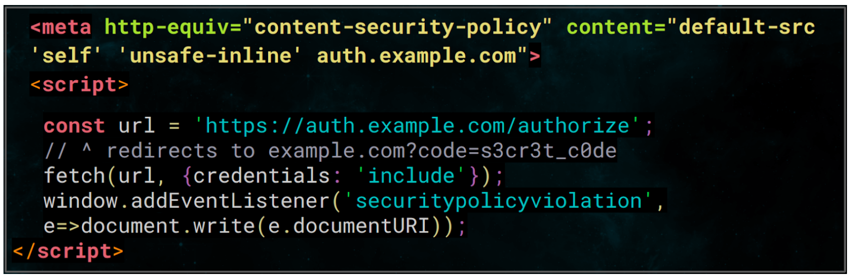
[redacted] 2020-04-20 16:03:18 PDT Comment 3 [reply] [−]

```
This does seem to be a place we could improve our CSP implementation, but this doesn't seem to pose a threat to
WebKit users. Rather, it allows an attacker to use WebKit as a means to gather data about websites using the CSP
reporting mechanism.

Even after fixing this CSP issue an attacker could gather this same information by monitoring WebKit logging while
running an attacker-controlled build of WebKit.
```

To us, it seemed as if they failed to understand the vulnerability and the risk it posed. Trying to convince them, time went by but we saw no progress at all. So, in January of 2021, after waiting for almost a year, we decided to explore the potential of vulnerability by targeting bug bounty programs.

Setting up POC



```
  <meta http-equiv="content-security-policy" content="default-src
  'self' 'unsafe-inline' auth.example.com">
  <script>

   const url = 'https://auth.example.com/authorize';
   // ^ redirects to example.com?code=s3cr3t_c0de
   fetch(url, {credentials: 'include'});
   window.addEventListener('securitypolicyviolation',
   e=>document.write(e.documentURI));
</script>
```

The exploit was simple- we set up a page with CSP as shown above which whitelists the domain we are sending requests to, add an event listener for SecurityPolicyViolation and send the request. Sending a request to an authorization endpoint issues a redirect to another domain with code or token appended to the URL, which is typically the case for OAuth or SSOs whereas the policy only allows loading of resources from the page itself and the domain specified- `auth.example.com`. An attempt to load resources from any other domains would violate the policy and trigger the SecurityPolicyViolation event. The `documentURI` then would have the value of the actual blocked URL, not just the origin.

Playground

We have also prepared a playground for everyone, to make it easier to experiment with the vulnerability. The usage is simple and pretty straightforward, we pass in a few parameters as shown in the usage. The page then sets the appropriate CSP policy using the domain from the query string and displays the TODO. There, we have added an Exploit button as a workaround for ITP and after a little wait, it issues a fetch request to the URL specified in the query string which redirects to another domain causing a CSP violation. The event listener would then print the secret token on the screen.

https://cm2.pw/csp-playground

https://github.com/threatnix/csp-playground

Impact

It took us a while to understand the full impact the vulnerability had. The vulnerability did not just affect Safari browser but because of

Apple's App Store policies that states that Apps that browse the web must use the iOS WebKit framework and WebKit Javascript. This means that web browsers can't implement their own rendering engines; they must embed a version of Safari's rendering engine. Which made browsers like Firefox, chrome, opera in iOS devices vulnerable.

Due to the nature of the design of OAuth, almost every implementation of OAuth that exists was vulnerable. Using the bug we could takeover Facebook, Instagram and WhatsApp accounts. The vulnerability also affected google's SSO which means google products using google account's central authentication were also vulnerable. Crypto-currency platforms, Social Networking platforms, e-commerce platforms and more. In today's modern applications it's very rare to find an application not using OAuth, so let's just say the numbers were very high.

Since the vulnerability affected SSO, it's obvious that different SSO providers were also affected by this.

Many applications make use of Login with `<third-party>` features such as "Login with Facebook" or Google to provide their user with an easy and seamless authentication experience. And since most of these features make use of OAuth, these applications implementing “Login with” feature were also affected thus allowing an attacker to takeover accounts on platforms implementing these features. For example: If you have an e-commerce website where you have a feature that allows users to login with a Facebook account, then an attacker could basically takeover accounts of your users in your e-commerce website by stealing facebook’s OAuth tokens.

Roadblocks

Let’s discuss some roadblocks and common misunderstandings that we saw among security teams of different bug bounty programs.

ITP or Intelligent Tracking Prevention is a privacy feature developed and implemented by WebKit and has been in use since October 2017. ITP uses machine learning model to detect Cross-Site Tracking Capabilities and aims to reduce them by taking different actions such as Full Third-Party Cookie Blocking, reducing the accessibility and longevity of first-party cookies, LocalStorage Capping and deletion, downgrading Third-Party Referrers and more.

Due to the nature of the attack where it relied on making third-party requests, ITP rendered the attack unusable on latest versions of WebKit. According to documentation from WebKit to circumvent it, there needs to be a user interaction (it could either be a tap, a click, or use of the keyboard) with your website as first party. So in a real attack scenario, we open up a tab using `window.open`, and then we keep polling the vulnerable endpoint until the user interacts with it.

When we started submitting this vulnerability to several bug bounty programs we noticed that a few of them had similar misunderstandings around them.

A common misunderstanding among security teams of several programs was around authorization code, unaware about its capabilities and how sensitive it is. The misunderstanding was that without the ability to exchange authorization code for an access token, the authorization code was useless since exchanging them required `client_id` and `client_secret`. While the statement above is true, in a case where OAuth was used for authentication, simply having access to authorization code was enough to get access to victim’s account. Making a request to an authorization endpoint with stolen authorization code with attacker generated state or nonce value would allow an attacker to authenticate into victim’s account.

Another common misunderstanding was with regards to state or nonce value and its capabilities to prevent attacks in OAuth. While using state in OAuth is an excellent way to prevent CSRF attacks and to redirect users to where they were before the authentication process started. Apart from that the state really doesn't provide any additional security features. Even if the state value is bound to a token, an attacker can generate a fresh state value, which then can be attached to an OAuth endpoint URL to send to the victim. The resulting token of the request then can be used by an attacker without any issue. The same thing was also demonstrated in our demo above where we saw that nonce value or state value did nothing to prevent the attack.

Stats

These are some of the stats based on bug bounties programs and reports we made.

In total, we were able to harvest more than $100k in bounties. The responses from the bug bounty programs were kinda mixed. While some programs took the issue very seriously and applied a fix in under 21 hrs which was really impressive, some took months to just understand the vulnerability, and some of the programs also relied on WebKit to come up with a fix. Some programs like DropBox and Poloniex took it one step further by using their contacts at Apple and asking them to come up with a fix.

The highest single reward was given by Coinbase which also was the fastest program to fix the issue. Apple took more than a year, a total of 457 days, to come up with a fix and in the end decided that the bug does not qualify to be awarded a bounty because it does not meet the published bounty categories.

And ironically while this bug in WebKit made so many applications on the internet vulnerable, Apple's implementation of SSO and OAuth were not vulnerable.

Fixes

We covered how we can exploit the vulnerability and how severe it could be. Now, let's discuss some fixes as well.

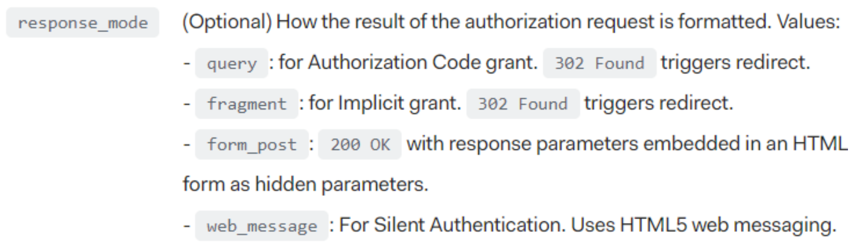
As a developer or a site owner, we can use any or a combination of the following methods to prevent such vulnerabilities altogether.

- Use POST instead of GET

This also aligns with the practice of never sending any sensitive information over URL. For example, SAML (Security Assertion Markup Language) already uses POST instead of GET for transporting tokens.

- Use postMessage

This approach uses HTML5 postMessage for cross-domain communication.



`response_mode`	(Optional) How the result of the authorization request is formatted. Values: - `query` : for Authorization Code grant. `302 Found` triggers redirect. - `fragment` : for Implicit grant. `302 Found` triggers redirect. - `form_post` : `200 OK` with response parameters embedded in an HTML form as hidden parameters. - `web_message` : For Silent Authentication. Uses HTML5 web messaging.

These two fixes are also supported by OAuth or OpenID Connect as can be seen in the image above. They could be specified via `response_mode` as long as the authorization server supports. However, it's still necessary to disable or remove support for query and fragment modes.

- Use SameSite Cookies

SameSite cookies not only protects against this particular vulnerability but makes almost all client-side vulnerabilities infeasible. SameSite cookies protected quite a few programs in our campaign as well.

- For OAuth, use PKCE for defence in depth

PKCE for OAuth makes the exploit kinda useless because we also need `code_verifier` to make use of the leaked code.

- Ask for manual authorization each time a user logs in

This approach prompts for authorization every single time even if already authorized and is also supported by OpenID using the prompt parameter- Apple does the exact same and many others too.

- Use client-side redirects (or JavaScript) instead of 30x

One can also use client-side redirects instead of 301/302 in combination with X-Frame-Options or CSP’s frame-ancestors- Twitter, for example, uses this method of redirect and many programs, we reported the vulnerability to, also fixed the issue using this approach.

- Validate referrer/origin against client (client_id, redirect_uri) - Not recommended

The last one is to validate referrer/origin against the client. Though, we do not recommend using this approach as it suffers from the same issues as with CSRF protection with referrer/origin.

Browsers' Mitigation Strategies

Browsers also have some mitigations in place to avoid such leaks besides Same Origin Policy (SOP).

However, the strategies listed here are specific to CSP- we have something we will shortly talk about yet another exploit that these strategies do not cover.

1. Includes only the URL of the original request, not the redirect target

The violation reports now include the URL of the original request, meaning we only get what we already know.

1. Ignores the path component of a source expression if the resource being loaded is the result of a redirect

Furthermore, browsers ignore the path component of a source expression if the resource being loaded is the result of a redirect. This is to prevent leaking of path information which was demonstrated in Homakov's post. The spec also gives an example for easier understanding but we'll leave it at that.

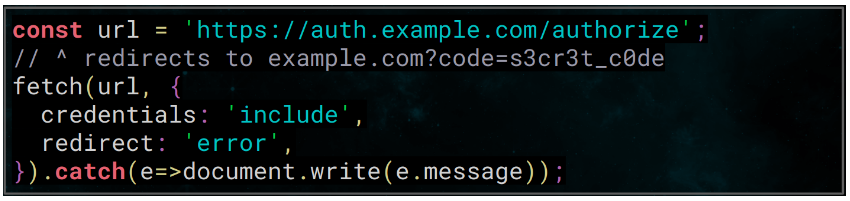
Bypasses & a new 0day

Unfortunately, Apple fixed it a few weeks back and it no longer works in the latest version of Safari but let's discuss it anyway along with bypasses.

After our reports, many programs, including Facebook & Coinbase, came up with their own fixes- their fixes revolved around checking if the User-Agent is of Safari and only if it is, they would use a different approach. For all other user-agents, their workflow remained the same. So, it seemed a simple change of user-agent would bypass the fix and an obvious place to try it was on iPhones/iPads. Since all browsers on iOS devices must use WebKit as their engine, they are all actually the same. Just using Chrome didn't work but changing to "Desktop Site" did. However, the change of User-agent wasn't inherited in subsequent fetch requests. So, we tried all requests from a project called

HTTPLeaks by Cure53 and found that `<a ping>` (anchor with ping attribute) did the work.

We had yet another bypass but during our hunt for bypasses, we noticed something peculiar. After some digging, we realized that we have already found another 0day. This time, it wasn't CSP but Cross-Origin-Resource-Sharing or CORS. The finding was as easy as shown here, we just needed to add a catch handler to our fetch request and it would disclose the destination URL.



```
const url = 'https://auth.example.com/authorize';
// ^ redirects to example.com?code=s3cr3t_c0de
fetch(url, {
  credentials: 'include',
  redirect: 'error',
}).catch(e=>document.write(e.message));
```

We shared our plan to disclose the vulnerability with Apple and surprisingly it got fixed right before our talk, without us ever being notified. It's a little disappointing that they never get in touch with us but I guess it's fine as long as the bug is fixed.

DEMO

https://youtu.be/UFwMCKbkd9k

Key Takeaways

1. SameSite Cookies can be super helpful
2. Use defence in depth to mitigate browser flaws
3. Browsers do not always follow the spec completely

Timeline

Feb 21, 2020: Initial report sent to Safari

Mar 12, 2020: Sent a ping

Mar 16, 2020: Automatic acknowledgement received

Apr 20, 2020: Received a reply from Apple that said the vulnerability doesn't pose a threat

Jan 12, 2021: We started targeting other bug bounty programs

May 24, 2021: Apple patched the vulnerability & shipped the new versions of WebKit

P.S. The vulnerabilities were assigned CVE-2021-30682 and CVE-2021-30888 respectively.

Update: You can see the talk as was presented on GreHack 2021 here: https://www.youtube.com/watch?v=2dS34u3T-80&t=19302s