Client-Side Javascript Vulnerabilities - Demystified

...

Agenda:

DOM XSS

PostMessage Misconfiguration

Parameter Pollution

XSLeak

Same Origin Policy:

Same Origin Policy is an important concept in the web application security model. This policy determines the access to data in between web applications.

http://store.company.com/somreandom.html

URL	SOP	Explanation
http://store.company.com/dir2/other.html		The same protocol, host and port.
http://store.company.com/dir/inner/another.html		The same protocol, host and port.
https://store.company.com/secure.html		Other protocol.
http://store.company.com:81/dir/etc.html		Other port.
http://news.company.com/dir/other.html		Other host.

Why does Client-side security matters?

Client Side vulnerability takes advantage of an authenticated session of a legitimate user.

Having arbitrary Javascript execution lets attacker to do (almost) anything on behalf of an authenticated user.

Example:

Modify Email/Password, steal your personal information and anything you can do from your browser.

DOM XSS

What is DOM XSS?

DOM XSS vulnerabilities arises when Javascript takes data from attacker controllable source and pass it to sink.

What are sinks ?

Sinks, on the other hand are the points in the flow of data at which the untrusted input gets outputted on the page or executed by JavaScript within the page.

Example:

URL: https://example.com/welcome.html

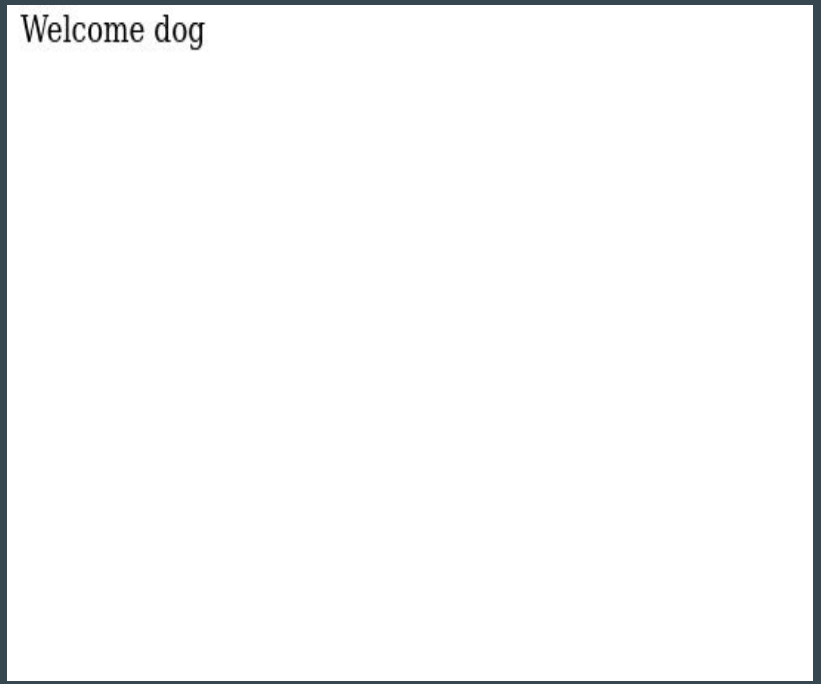
```
<html>
        <body>
        <script>
                var urlParams = new URLSearchParams(window.location.search);
                //Creates an object containing URL parameters

                var name;
                if (urlParams.has("name"))
                var name=urlParams.get("name");
                //Assigns the value of name parameter to variable name

                document.body.innerHTML="Welcome "+name
                // Display Greeting without name
        </script>
        </body>
</html>
```

URL: https://example.com/welcome.html?name=cat

https://example.com/welcome.html?name=dog



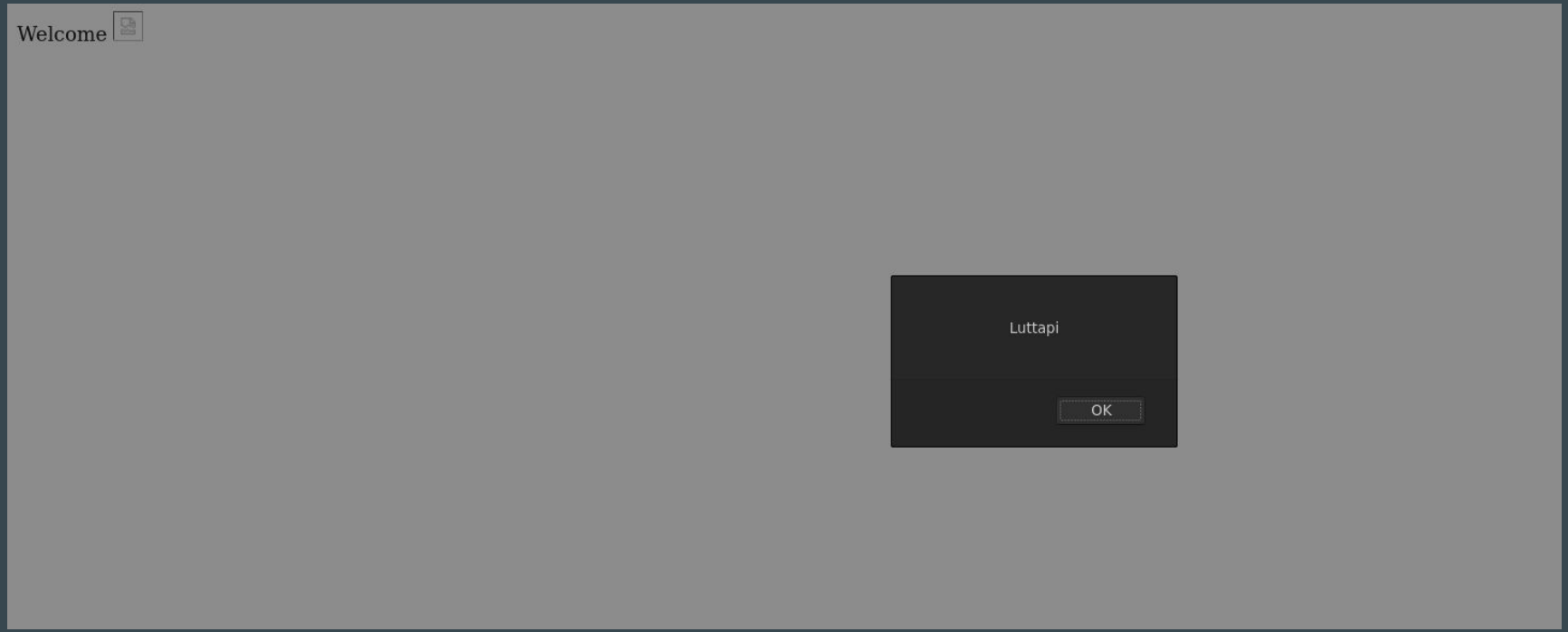
What if we input valid HTML code as name?

https://example.com/welcome.html?name=

Welcome

What about JS ?

https://example.com/welcome.html?name=



Similar Sinks:

Sink Name	Property susceptible to DOM-based XSS

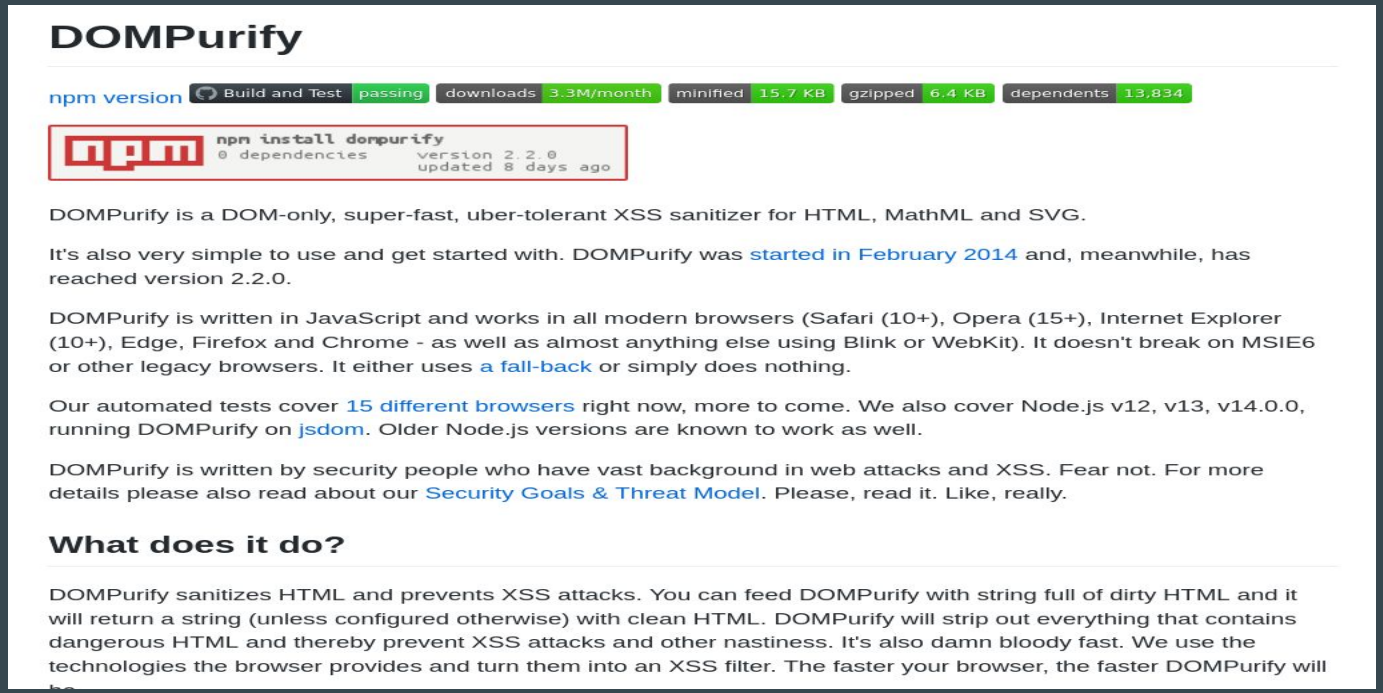
Execution Sink	`eval`
	`setTimeout`
	`setInterval`

HTML Element Sink	`document.write`
	`document.writeIn`
	`innerHTML`
	`outerHTML`

Set Location Sink	`location` `location.href`

Why did it happen and how can you fix?

SANITIZE the data before you use it !



DOMPurify

npm version | Build and Test passing | downloads 3.3M/month | minified 15.7 KB | gzipped 6.4 KB | dependents 13,834

npm install dompurify
0 dependencies version 2.2.0 updated 8 days ago

DOMPurify is a DOM-only, super-fast, uber-tolerant XSS sanitizer for HTML, MathML and SVG.

It's also very simple to use and get started with. DOMPurify was started in February 2014 and, meanwhile, has reached version 2.2.0.

DOMPurify is written in JavaScript and works in all modern browsers (Safari (10+), Opera (15+), Internet Explorer (10+), Edge, Firefox and Chrome - as well as almost anything else using Blink or WebKit). It doesn't break on MSIE6 or other legacy browsers. It either uses a fall-back or simply does nothing.

Our automated tests cover 15 different browsers right now, more to come. We also cover Node.js v12, v13, v14.0.0, running DOMPurify on jsdom. Older Node.js versions are known to work as well.

DOMPurify is written by security people who have vast background in web attacks and XSS. Fear not. For more details please also read about our Security Goals & Threat Model. Please, read it. Like, really.

What does it do?

DOMPurify sanitizes HTML and prevents XSS attacks. You can feed DOMPurify with string full of dirty HTML and it will return a string (unless configured otherwise) with clean HTML. DOMPurify will strip out everything that contains dangerous HTML and thereby prevent XSS attacks and other nastiness. It's also damn bloody fast. We use the technologies the browser provides and turn them into an XSS filter. The faster your browser, the faster DOMPurify will

PostMessage Misconfiguration

What are PostMessage and why does it exist?

The **`window.postMessage()`** method safely enables cross-origin communication between `Window` objects; *e.g.*, between a page and a pop-up that it spawned, or between a page and an iframe embedded within it.

```
targetWindow.postMessage(message, targetOrigin, [transfer]);
```

SOP restriction:

What could go wrong?

Case 1:

If you're rendering the input data from postMessage without origin check, it could lead to XSS

Case 2:

If you didn't specify the target domain to send message, there are chances for it to be stolen. Which could leak potentially sensitive information.

Getting XSS with postMessage

https://www.example.com/recieve.html

```
<html>

        <body bgcolor="white">
                This is https://othersite.com
                <hr>
                <div id="messageDOM"></div>
        <script>
        window.addEventListener("message",respond);
        function respond(message){
                var area=document.getElementById("messageDOM");
                area.innerHTML="Recieved Message: "+message.data;
        }

        </script>
        </body>
</html>
~
~
~
~
~
```

Harmless Usage

https://www.harmlesswebsite.com/sendmessage.html

```
<html>
        <body>
        <script>
                var childWin=window.open("https://example.com/recieve.html");
                setTimeout(()=>childWin.postMessage("Junk Message","*"),3000)
        </script>
        </body>
</html>
```

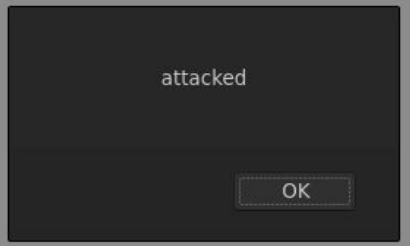
Fire XSS again!

https://attacker.com/exploit.html

```
<html>
        <body>
        <script>
                var childWin=window.open("https://example.com/recieve.html");
setTimeout(()=>{childWin.postMessage('<img src=x onerror=alert(`attacked`)>',"*")},3000)
        </script>
        </body>
</html>
~
```

This is https://example.com

Recieved Message: 



How to fix?

`Check for origin before use`

```
<html>

        <body bgcolor="white">
                This is https://example.com
                <hr>
                <div id="messageDOM"></div>
        <script>
        window.addEventListener("message",respond);
        function respond(message){
                if(message.origin !=="https://example.com")
                        return;
                var area=document.getElementById("messageDOM");
                area.innerHTML="Recieved Message: "+message.data;
        }

        </script>
        </body>
</html>
```

Hijacking Data from postMessage:

Developers should not only check the data origin before using but also specify trusted origin before sending the data.

If sensitive data is sent to arbitrarily controllable window without origin check then those data could be stolen.

Vulnerable Code

https://example.com/sendlogin.html

```
<html>
  <body>
    <script>
      window.opener.postMessage(`{'username':'secretadmin','accesstoken':'dG90YWxseXNlY3JldHBhc3N3b3JkCg=='}`,"*");
    </script>
  </body>
</html>
```

Why on earth would someone code like this?

Exploit?

https://attacker.com/exploit.html


```
<html>
  <body>
    <script>
      window.open("https://example.com/sendlogin.html");
      window.addEventListener("message",logit);
      function logit(message){
        console.log(message.data)
      }
    </script>
  </body>
</html>
```

How to uncover one myself?

Listen carefully! Yes listen for postmessages

Tools that might help you:

https://github.com/opnsec/postMessage-logger - Browser extenstion by opnsec

How to secure myself?

Send data only to trusted domain.

Avoid using * as target location

Double check before using regex with postmessage

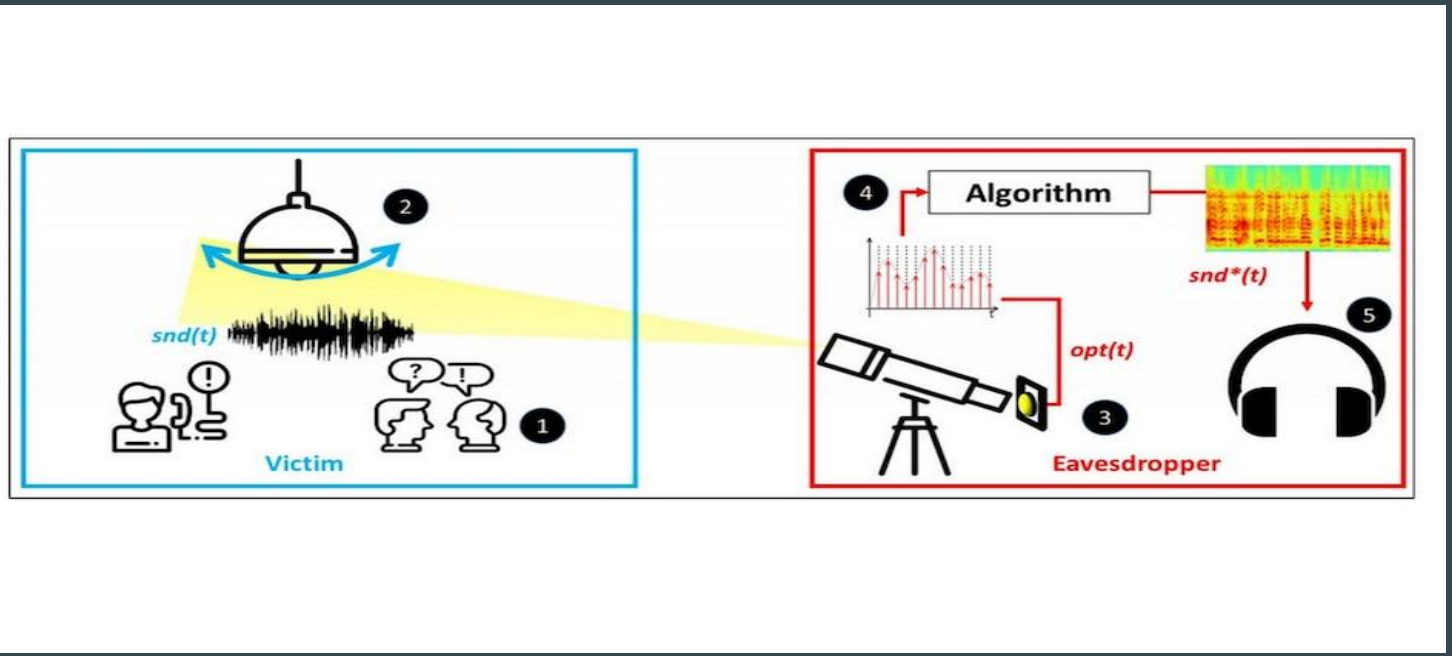
XSLeak

What are side channel attacks?

In computer security, a side-channel attack is any attack based on information gained from the implementation of a computer system, rather than weaknesses in the implemented algorithm itself.

SCA - Eavesdropping your conversation:

`Lamphone attack` by Israeli's Ben-Gurion University of the Negev and the Weizmann Institute of Science.



Back to XSLeak

Just like lamphone attack, xsleak simply exploit the fundamental browser and webpage implementation just using Javascript.

How do we do it?

- **Frame Count**
- **Cache and Error Events**
- **CSP Violation Events**
- **Media Size**
- **Redirects**
- **Request timing**

And Lot…….

https://github.com/xsleaks/xsleaks

Real world Example:

Leaking Facebook user's identity:

Description: Combination of browser and server behaviour for different users in redirection can be escalated to leak their Identity.

Observations:

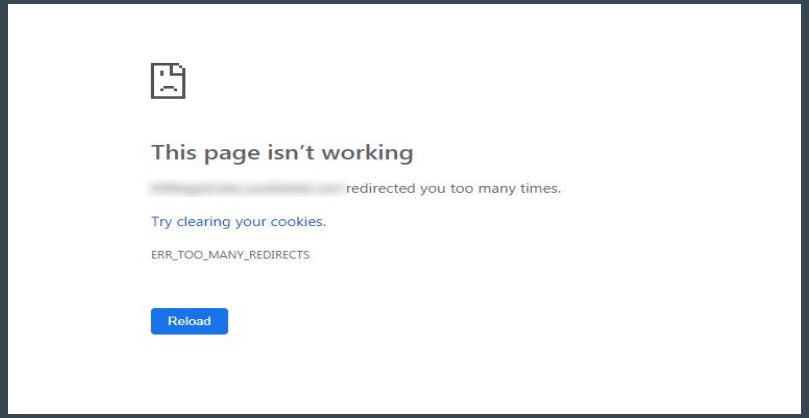
Server behaviour:

1,If a user visits https://facebook.com/<**own_username**>/archive?_fb_noscript=1 there will be no redirection.

2, ,If the same user visits https://facebook.com/<random_username>/archive?_fb_noscript=1 there will be a redirection changing URL to https://facebook.com/<random_username>

Browser behaviour:

If a page continuously redirects over 20 times browsers will throw out error stating "Too Many redirects".



These both can be chained to find the no of redirects by the server and therefore deanonymize the user cross-origin.

The Idea:

We know more than consecutive 20 redirects results in error. So lets make a webpage that redirects itself 18 times and to the Facebook URL in the 19th time. If the page loads without any error then it is not redirected else redirected. By knowing this we can find if the user is authenticated as that specific username.

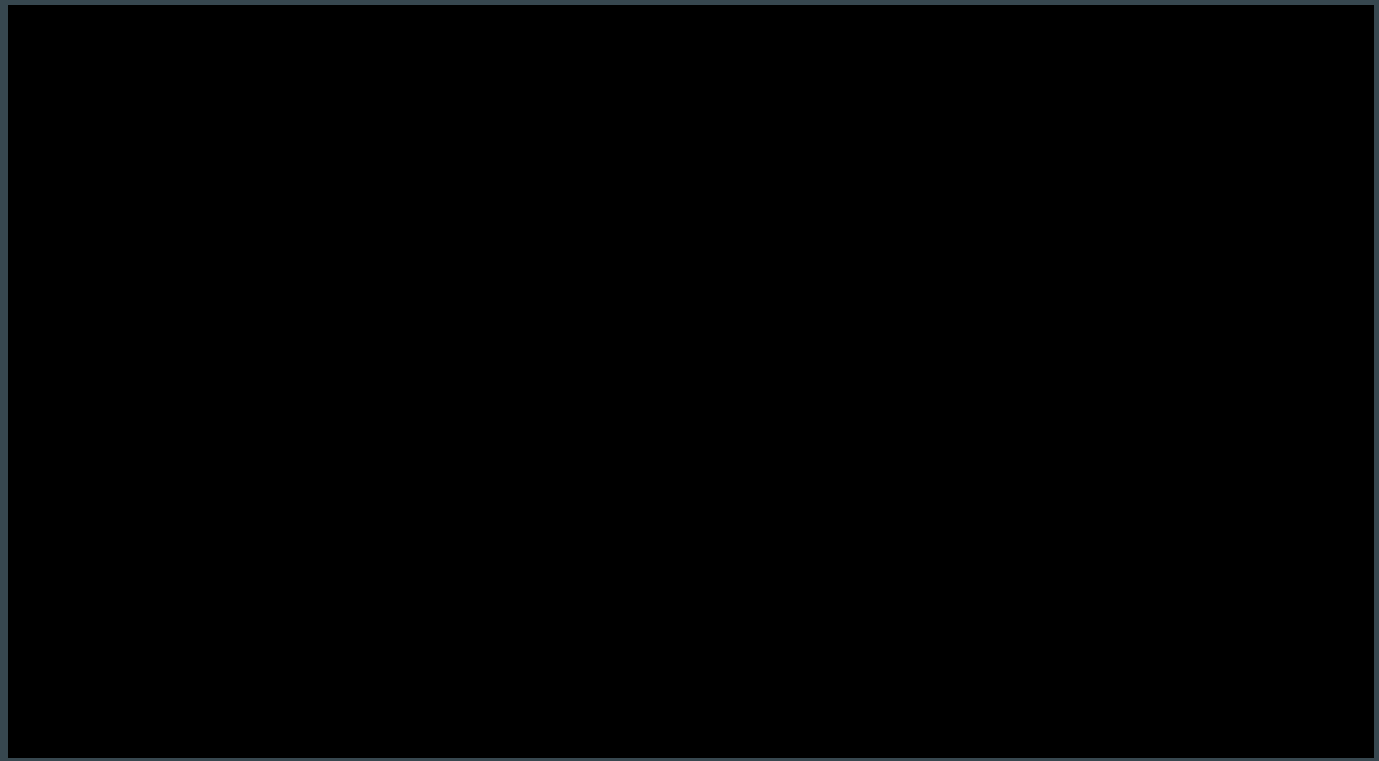
Check! How will you find if the page is loaded?

CORS with no-cors ! ***Yes.***

You can use fetch to request any website even though your that website isn't configured for CORS.

> `no-cors` — Prevents the method from being anything other than `HEAD`, `GET` or `POST`, and the headers from being anything other than simple headers. If any ServiceWorkers intercept these requests, they may not add or override any headers except for those that are simple headers. In addition, JavaScript may not access any properties of the resulting `Response`. This ensures that ServiceWorkers do not affect the semantics of the Web and prevents security and privacy issues arising from leaking data across domains.

Putting it all together!



Questions:

http://dc0471.org/discord