Chapter 24
Cross Origin Resource Sharing (CORS)

Revision 1.0, October 2025.

Objectives

- Analyze the trust model involved in cross origin attacks
- Learn about when CORS is used
- Understand the role of HTTP request and response options and headers
- Learn what it means to "Preflight" web requests

As we learned in Chapter 22, cross origin attacks have been and continue to be a risk for web services. As an attempt to strengthen web defenses against cross origin attacks, work began in the early 2000's to try to somehow fix the HTTP protocol to make these attacks more difficult. In 2006, the World Wide Web Consortium release a working draft document of a new standard. In 2009, the document was revised and renamed Cross Origin Resourced Sharing – W3C Working Draft 17[1]. That document lays out the features that are supported today and is used in security for `fetch` (the replacement for `XMLHttpRequest`)[2].

Note that CORS is a pretty controversial topic, with some people thinking it is a great idea and others thinking that it is either unnecessary or just hate it because it often (legitimately) blocks requests to their web applications. If you hang out in web programming circles for a while, you will definitely hear lot of complaints about CORS.

However, CORS is an intrinsic part of the web infrastructure, so it is important that we understand what is does and how it works. We will start by introducing the trust model that underlies CORS, then learn about the CORS additions (basically new headers) to the web protocol. We will introduce an important part of CORS called "preflighting" of a web request.

24.1 The CORS Trust Model

You recall from Chapter 22 on CSRF that the goal was for an evil (untrusted) website to serve you a web page that contains a web request contained in a form or JavaScript. Our example started with a client at their web browser who had an active session open with their bank. The client then decides, probably in another browser window, to make a request to a questionable

[1] `https://www.w3.org/TR/2009/WD-cors-20090317/`
[2] `https://www.w3.org/TR/cors/`

site, `bad-site.com`. That site then returns a web page that contains some JavaScript that will generate a web request. When this JavaScript executes, it causes an unauthorized request from the client to their bank to transfer money to an account controlled by the attacker. Without sufficient checking and proper session management, as we discussed in Chapter 23, your bank happily transferred the funds to the attacker's account.

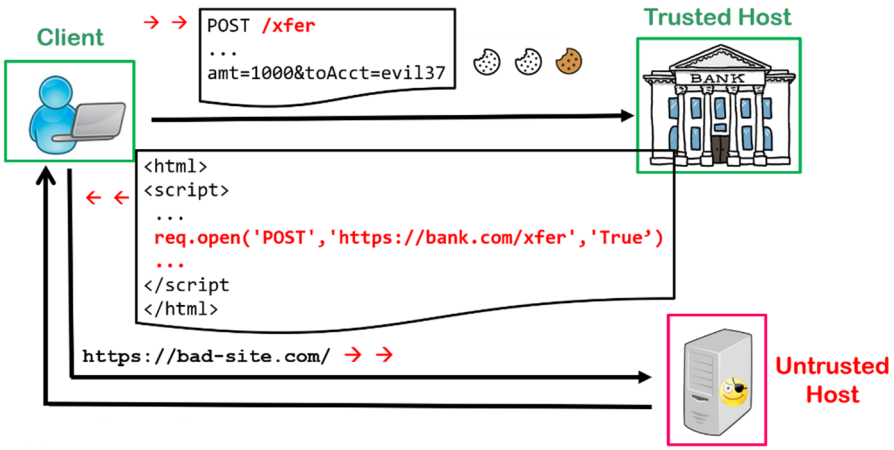

Figure 1: The CORS Trust Model

In a web request, when there is an attempt to make a cross origin attack, there are three parties involved, the client, the host that the client is trying to access (e.g., their bank), and a malicious host that will try to take advantage of the fact that the client is already authenticated to the trusted host. The client, of course, trusts themselves and trusts the service that it is trying to access. The client definitely does not (or definitely should not) trust random servers with which they do not have an established relationship.

The client might access the untrust host because of bad judgement or by accident, such as by mistyping a web address. The untrusted host might be operated by a malicious actor or just be a poorly administered host that was successfully exploited.

CORS tries to prevent attempts at cross origin attacks by giving the decision making process to client based on information from the trusted host.

The goal of CORS is not to replace existing security mechanisms like secure session management; the goal is to provide an extra layer of security to try to prevent attacks that you might not have anticipated. Remember that one of our secure design principles is defense in depth.

To allow the client to determine if a web request is safe, such as for a request that originates from JavaScript contained in a questionable web page, the

client needs to provide the trusted host with information about what it plans to request; the trusted host then needs to respond with advice as to whether that might be safe.

This means that the client needs to tell the trusted host the domain name of the untrust host that provided the web page with the JavaScript. This information gives the trusted host a chance to decide if the untrust host is a reasonable source of an embedded web request.

24.2 Simple vs. Complex Requests

As with most security standards, especially those related to the web, there are a **lot** of details involved with CORS and we are not going to present all of them here[3]. Our goal is for you to understand the basics of how CORS works (and why), and position you to be able to understand the more detailed aspects of CORS on your own.

As a starting point, CORS classifies web requests as "simple" or "complex". Simple requests can be handled, well, simply. This means that some extra information, in the form of request headers, is used by the browser to tell the server about the request, and the server responds with information in its headers to say whether this request is an acceptable cross domain reference.

A request is simple if it is a GET, POST, or HEAD method call with basic headers. It might be surprising that POST is included in the list of simple headers because it can change values on the web server, but note that CORS is designed to protect the user from cross origin (site) attacks, not protect the server's data. However, this reasoning gets more complicated because PUT requests are always categorized as complex.

If the request includes authentication credentials – including those in cookies, TLS client certificates, and HTTP headers – then it is not a simple request. And simple requests can only include a few header types beyond those automatically inserted by the browser. These headers that describe basic content of the web content: `Accept`, `Accept-Language`, `Content-Language`, and Content-Type (only `application/x-www-form-urlencoded`, `multipart/form-data`, or `text/plain`)

If the request originates from a web form, then it is defined as simple. The thought process here was that the old code that uses forms should already have had some protective mechanism in place, like the secure sessions we described in the previous chapter, so CORS was not necessary. Note that request from web forms are restricted by the browser to include only a few programmer specified header types (`application/x-www-form-`

[3] For example, the CORS standard document from 2009 is 21 pages long and the new Fetch standard document is 131 page.

urlencoded, multipart/form-data, and text/plain), so fit within the header limitations of a simple request.

Any request that is not classified as simple is classified as complex. Complex requests can be thought of as approximately (and inaccurately) as ones that make non-idempotent changes to the server. These complex requests require something called "preflighting", which we will describe shortly.

The key feature that allows CORS to work is the Origin header in the request. We will see how this header plays an important role the examples that follow.

24.2.1 Details of a Simple Request

Let's look at a simple request. In this case, the client sends a GET request to bank.com and specifies that the request originated from a web page that came from bad-site.com.

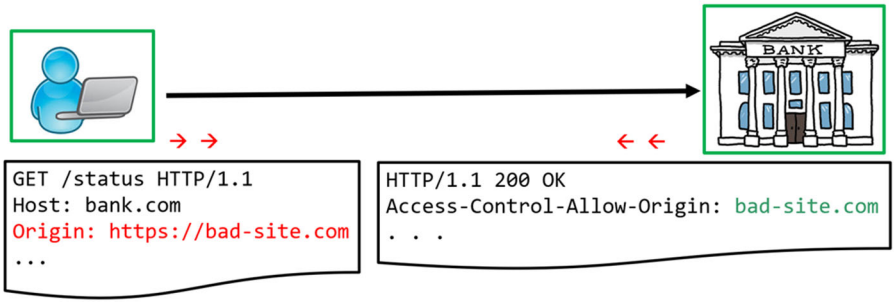

Figure 2: CORS Simple Request and Response

In Figure 2, we see the Origin request header that triggers the CORS processing in the trusted host, the bank. The question that the client is asking the bank is: is the web page that originated from bad-site.com allowed to contain a request to bank.com?

When the server at bank.com responds, it will include an Access-Control-Allow-Origin header to indicate whether this request from bad-site.com is allowable.

If bad-site.com is included in the hosts listed in this header, such as it is in this example, then the request is allowable and will proceed as normal.

Alternatively, the server, bank.com can respond that **any** host can include a cross domain request to that server. The server indicates this liberal permission by including the wildcard character on the Access-Control-Allow-Origin header:

```
Access-Control-Allow-Origin: *
```

Any request that gets such a response will proceed as usual.

If the server determines that the request is NOT allowable, then it can list a fake domain name, like `does.not.exist` in this `Access-Control-Allow-Origin` header. Such a name will never match a real domain name. The browser will compare the name in the `Origin` request header to the name provided by the server in the `Access-Control-Allow-Origin` header. If these do not match, then request is not allowed and the browser will report an error.

Again, note that the server has provided the advice, and the browser has made the decision. In this case, the requested operation may have been executed on the server, but the browser will refuse to display the result. Personally, I find this to be a strange behavior, but it is one that you need to be aware of.

You will also notice that the HTTP response code is still `200 OK`. The way to think about this is that the server successfully responded to the access request, even though the origin was not allowed.

24.2.2 Details of a Complex Request and Preflighting

Now, we move on to complex requests and the preflighting mechanism. Remember that any request that is not simple is considered complex.

The basic idea of preflighting is that the browser sends an extra message before the actual request. This extra message describes the origin of the request, type of request, headers to be used, and server that is being accessed. The server then has a chance to respond as to whether the request is an allowable cross domain request. This preflight response happens before the actual request and before any state would be changed on the server.

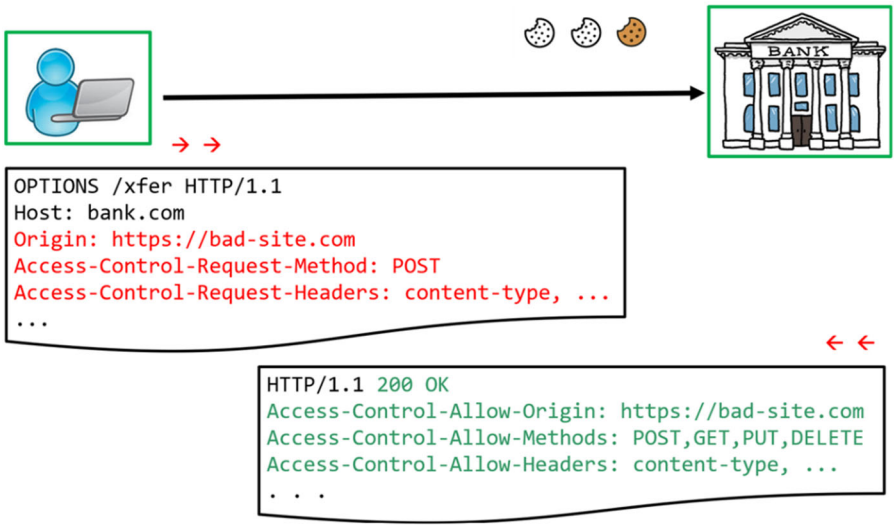

Figure 3: CORS Complex Request Preflight and Response

Remember that the server does not allow or disallow a request. It merely provides enough information to the browser to refuse the request. If an actual request reaches the server, then the server will execute it.

Figure 3 shows an example of a preflight request. Here, the client is again making a request to `bank.com`. We know that this is a preflight request and not a regular request because the method type is OPTIONS. That is not the most intuitive name for the method, but it is the standard.

The headers in the request describe the details. This request originated in a web page from `bad-site.com` and the method type is POST. We also see that client is telling the server that the actual request would contain some specific header types, such as `content-type` (and others).

The server's response to the preflight request include several headers that provide the client's browser with enough information to make a decision as to whether the actual request should be sent. We see that the allowed origins include `bad-site.com` and the allowed methods include POST (as well as GET, PUT, and DELETE). We also see the list of allowed request headers.

Again, note that the response code is `200 OK`, which simply means that the server successfully responded to the preflight request, not that the request should be allowed. The browser will decide if the request is allowed based on the data in headers returned from the server.

As with the simple request, the server could have responded with the allowed origin specified as a wildcard, meaning that any host is allowed.

```
Access-Control-Allow-Origin: *
```

Remember that there are a lot of technical details that we are skipping over in this introduction to CORS. For example, if the request contained credentials, then the standard says that the server is not allowed to respond with a wildcard.

And, if the request should not be allowed, then the server will typically respond with a fake domain name that will not match any valid domain name.

```
Access-Control-Allow-Origin: does.not.exist
```

The server could have responded with an actual valid domain name, which would not have matched `bad-site.com`. However, it is common practice to respond with a fake domain name so as to not leak any valid information to an unauthorized party. This follows the secure design principle of least information.

24.3 Summary

- Analyzed the trust model involved in cross origin attacks
- Learned about when CORS is used

- Understood the role of HTTP request and response options and headers
- Learned what it means to “Preflight” web requests

24.4 Exercises

1. What are the three parties (hosts) involved in the CORS trust model?
2. Describe the requirement that a web request much satisfy to be considered “simple”.
3. Do some research to understand the background and thinking behind POST being acceptable as a simple request and PUT not being acceptable.
4. Why does the server response “`200 OK`”, even for web CORS requests that are being denied?
5. When a CORS response is intended to deny a request, why is it good security practice for the `Access-Control-Allow-Origin` header to include a fake host name instead of an actual allowed host name?