

**HTML5 – CORS**

**Sécuriser les communications applicatives inter-domaines**



|  |
| --- |
| **Historique** |

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Opération** | **Acteur(s)** | **Visa** |
| 19/01/2015 | Création | Jean-Luc LEBOUTET | Jean-Luc LEBOUTET |
| 10/02/2015 | Mise à jour | Paul SIMILOWSKI |  |

|  |  |
| --- | --- |
| **Description** | |
| Résumé | Ce document :   * Rappel les grands principes autour de la *Same Origin Poilcy (SOP)* * Présente les mécanismes de contournement *JSON Padding* et fichier de politique *Cross Domain* « crossdomain.xml » * Présente *Cross Origin Resource Sharing (CORS)* * Décrit les impacts de sécurité des requêtes *CORS* ; * Donne des indications sur la meilleure façon de sécuriser ces requêtes *CORS* dans les développements. |

# Introduction

Les requêtes *Ajax* sont limitées par la politique *Same Origin Policy (SOP)* mise en œuvre dans les navigateurs, qui n’autorise l’émission de requêtes qu’à des ressources d’un même domaine.

Cependant, il a toujours été possible pour un navigateur d’effectuer des requêtes *Cross Origin* en spécifiant une ressource à partir d'un domaine étranger dans les balises *HTML* « img », « script », « iframe », etc. Mais en utilisant ces requêtes, un script client n'a pas accès au contenu de ces ressources et ne peut donc les manipuler : celles-ci peuvent seulement être visualisées/exécutées par le navigateur.

*HTML5* casse cette restriction et permet aux requêtes *Ajax* d’effectuer des requêtes inter-domaines.

Ce document décrit les impacts de sécurité des requêtes *Cross-Domain* et donne des indications sur la meilleure façon de sécuriser ces appels *Cross Origin (COR)* dans les développements.

# Same Origin Policy

## Rappel des principes

*SOP* limite la manière dont un document ou un script chargé depuis une origine (un site), peut interagir avec une ressource issue d’une autre origine (un autre site). La première mise en œuvre de ce concept date de 1996.

Ce contrôle s’effectue sur l’origine (URI, informations de chemin exclues) en tenant compte du protocole (HTTP, HTTPS, etc.) et du port.

Par conséquent : http://monsite.fr est différent de https://monsite.fr ou de http://monsite.fr:8080

## Applications

Avec *SOP*, une origine est autorisée à envoyer des requêtes vers une autre origine, mais pas à lire les réponses.

On peut donc classer les interactions entre origines dans 3 catégories :

* Cross Origin Writes (redirect, liens, form action, etc.) ;
* Cross Origin Embedding (html tags avec src/hrefs, iFrame, plug-ins, etc.) ;
* Cross Origin Read (non autorisé sans autres mesures – CORS).

## Limitation des accès Cross Origin

Pour limiter les accès :

* Cross Origin Writes : il faut utiliser un token anti-CSRF ;
* Cross Origin Embedding : il faut contrôler que les ressources chargées dans la page ne sont pas des types vus précédemment ;
* Cross Origin Read : il faut contrôler que la ressource n’est pas encapsulée (embeded).

A noter que dans le cas des iFrame, l’utilisation des en-têtes *X-Frame-Options* est nécessaire pour contrôler les accès à la ressource.

# Contournements SOP avant CORS

## JSON Padding (JSONP)

Le but premier de toute requête cross-domaine est de pouvoir récupérer des données sur un autre domaine, et d'exécuter une fonction quand la réponse est renvoyée.

*JSONP*, est une technique détournée qui permet de faire des requêtes *Cross Domain*, et exécuter un callback quand la réponse revient. Il tire parti du fait que les navigateurs n’appliquent pas la politique *SOP* sur les balises « script ».

Pour fonctionner, *JSONP* ajoute donc une balise « script » au *Document Object Model (DOM)*, et demande au serveur d'afficher du *JavaScript* qui exécutera une fonction globale.

Afin de pouvoir passer des paramètres au serveur distant, et puisque seules des requêtes GET peuvent être faites (inhérent à l’inclusion d’une balise « script »), les paramètres sont passées en *Query String*. En général, un des paramètres s'appelle « callback », et il s'agit du nom de la fonction globale que l'on veut que le serveur exécute.

Pour donner un exemple, si l’API accessible sur l'URI [http://exemple.com/jsonp](http://example.com/jsonp) renvoie les dates de naissance de personnes, on déclare en premier lieu une fonction de « callback » :

window.jsonpcallback = function(birthdate) {

console.log(birthdate);

};

On injecte ensuite la balise « script » dans le *DOM* :

<script src="http://example.com/jsonp?name=paul&callback=jsonpcallback"></script>

Pour obtenir une réponse du serveur du type :

jsonpcallback("20/10/2010");

Le navigateur charge donc la page et exécute le *JavaScript* contenu dans la balise « script », permettant d’effectuer un appel à l’API sur le serveur et lorsque la réponse est reçue d’appeler la fonction locale de « callback ».

Cette technique détourne donc de leur fonction première des mécanismes non conçus pour effectuer des requêtes *Cross Domain* et ne peut être apparentée à un véritable standard.

De réelles considérations de sécurité sont par ailleurs soulevées par la capacité du serveur distant à injecter n'importe quel code *JavaScript* sur le site d’origine à son insu. Cette technique n’est donc à utiliser qu’en dernier recours, et avec une confiance complète du serveur partenaire.

Une protection anti-CSRF basée sur des jetons et la méthode POST est par ailleurs complexe à mettre via *JSONP* (requêtes GET), et expose une surface d’attaque au même titre que les éléments potentiellement sensibles passés en clair dans les paramètres de l’URI.

## Politique Cross Domain « crossdomain.xml »

Des plug-ins comme *Flash* et *Silverlight* sont également capables d’effectuer des requêtes *Cross* *Domain* et de lire le contenu des réponses.

Pour ce faire, un fichier « crossdomain.xml » est créé sur le serveur, celui-ci :

* Est récupéré et analysé avant toute autre action par le navigateur ;
  + Si un site externe n’est pas autorisé à faire des requêtes *Cross* *Domain* dans ce fichier, le navigateur n’effectuera pas de requêtes vers ce site externe.
* Permet de préciser quels domaines peuvent effectuer des requêtes *Cross Domain* vers le site ;
  + Si http://ton.site autorise http://mon.site dans le fichier de politique, alors http://mon.site peut accéder à tous les fichiers dans http://ton.site. L’accès est donc contrôlé par site en mode « tout ou rien ».

**Différences de ce mécanisme avec CORS**

A contrario, *CORS* :

* effectue la requête d’emblée et contrôle ensuite la politique dans un header de la réponse ;
  + Si cet en-tête positionné par le serveur autorise le site externe, alors le navigateur client peut lire la réponse, sinon la réponse est inaccessible aux *JavaScript*.
  + Dans le cas d’une requête complexe (méthodes ou type de contenu peu communs, en-têtes spécifiques, etc.), une pré-requête peut être émise par le navigateur afin de vérifier coté serveur la faisabilité technique du traitement de la requête réelle.
* fonctionne sur un mode de contrôle d’accès par page ;
  + Chaque page doit répondre avec un en-tête spécial pour être accessible par un site externe, permettant ainsi d’exposer des parties limitées d’un site Web, vers des sites externes, tout en gardant le reste inaccessible.
  + La responsabilité de contrôle d'accès est ainsi dans les mains des développeurs et non pas des administrateurs de serveur[[1]](#footnote-1). Les développeurs doivent écrire une portion de code spéciale dans les pages qui devront être accessibles par des sites étrangers.

# Cross Origin Resource Sharing

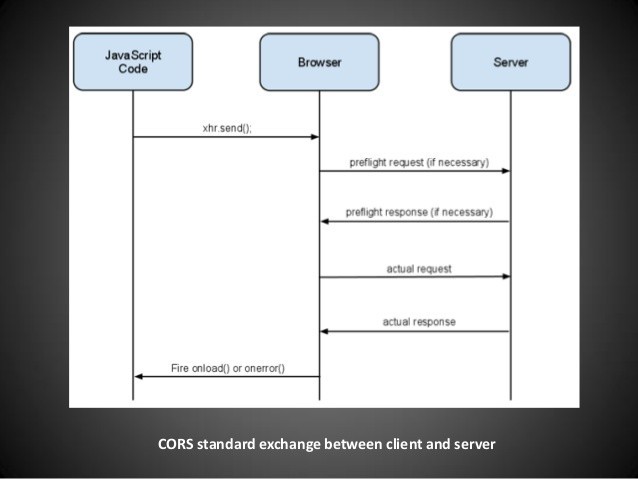
Afin de faciliter les échanges licites entre des domaines différents dans le navigateur, le W3C a introduit le *Cross Origin Ressources Sharing (CORS)*.

Cette spécification, encore à l’état de Draft, décrit comment un navigateur et un serveur doivent communiquer quand ils accèdent à des ressources d’origines différentes.

Cette spécification fonctionne via de nouveaux en-têtes, listant un ensemble de domaines ayant le droit de lecture inter-domaine. Ils sont positionnés par le serveur et contrôlés par le navigateur.

Deux types de requêtes peuvent être catégorisés :

* Les requêtes « simples » ;
* Les requêtes nécessitant un « preflight » (requête initiale pour effectuer une demande de contrôle en amont).



*Principe général des échanges CORS*

## Usages

CORS est nécessaire dans les cas déterminés suivants :

* Pour effectuer des requêtes légitimes avec des sites de confiance, permettant la consultation de données à accès contrôlé ;
* Pour le partage de données inter-applications ;
* Pour l’accès à des Web Services sur des domaines de confiance.

## Requêtes CORS simples

Elles sont utilisées s’il n’est pas nécessaire d’effectuer de contrôles préalables.

* Les requêtes sont de type HEAD/GET/POST via XMLHttpRequest ;
* Il n’y a pas d’en-tête spécifique à transmettre avec la requête ;
* Dans le cas d’une requête POST, le corps de la page est parmi les types :
  + text/plain ;
  + application/x-www-form-urlencoded ;
  + multipart/form-data.

Le navigateur utilise l’en-tête « Origin » dans la requête *CORS*, en le positionnant à la valeur du domaine qui effectue la requête.

Le serveur répond en positionnant l’en-tête CORS « Access-Control-Allow-Origin », le navigateur détermine sur cette base d’autoriser ou non l’exploitation de la réponse.

Lee contenu du site externe est le cas échéant accessible à travers la propriété spécifique : « **responseText** ».

// Fonction présente dans une page de http://mondomaine.com/index.php

function makeXMLRequest() {

xhr = new XMLHttpRequest();

xhr.open("GET", "http://www.autredomaine.com/index.php”, true);

xhr.onreadystatechange = function() {

if (xhr.readyState == 4) {

alert(xhr.**responseText**);

}

}

xhr.send();

}

*Fonction JavaSript effectuant un appel CORS*

GET /index.php HTTP/1.1

Host: autredomaine.com

User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US; rv:1.9.1b3pre) Gecko/20081130 Minefield/3.1b3pre

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8

Accept-Language: en-us,en;q=0.5

Accept-Encoding: gzip,deflate

Accept-Charset: ISO-8859-1,utf-8;q=0.7,\*;q=0.7

Connection: keep-alive

Referer: http://mondomaine.com/index.php

**Origin: http://mondomaine.com**

HTTP/1.1 200 OK

Date: Mon, 01 Dec 2008 00:23:53 GMT

Server: Apache/2.0.61

**Access-Control-Allow-Origin: http://mondomaine.com**

Keep-Alive: timeout=2, max=100

Connection: Keep-Alive

Transfer-Encoding: chunked

Content-Type: application/xml

*Exemple d’échange client/serveur pour une requête CORS simple*

## Requêtes CORS avec « preflight »

Une pré-requête ou requête « preflight » est nécessaire pour vérifier coté serveur la faisabilité technique du traitement de la requête réelle si :

* La requête est d’un type différent de HEAD/GET/POST (PUT, DELETE, etc.) ;
* Des en-têtes spécifiques sont présents (X-PINGBACK, X-PINGOTHER, etc.) ;
* La requête est de type POST mais le corps de la page n’est pas parmi les types suscités pour les requêtes simples.

Une requête préalable est donc effectuée pour récupérer les informations d’accès auprès du serveur, le RFC recommande une requête « preflight » avec la méthode OPTIONS.

A l’instar d’une requête simple, le navigateur positionne l’en-tête « Origin » mais également l’en-tête « Access-Control-Request-Method » et optionnellement l’en-tête « Access-Control-Request-Headers »[[2]](#footnote-2)

Le serveur répond en positionnant l’en-tête CORS « Access-Control-Allow-Origin  à l’instar d’une réponse simple, mais également les en-têtes « Access-Control-Allow-Methods », « Access-Control-Allow-Headers »,« Access-Control-Max-Age », et optionnellement l’en-tête « Access-Control-Allow-Credentials »2.

Si les informations de la requête et de la réponse correspondent, alors le navigateur envoie la requête réelle.

// Requête “preflight”

OPTIONS /resources/not-simple HTTP/1.1

Host: autredomaine.com

User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US; rv:1.9.1b3pre) Gecko/20081130 Minefield/3.1b3pre

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8

Accept-Language: en-us,en;q=0.5

Accept-Encoding: gzip,deflate

Accept-Charset: ISO-8859-1,utf-8;q=0.7,\*;q=0.7

Connection: keep-alive

Referer: http://mondomaine.com/index.php

**Origin: http://mondomaine.com**

**Access-Control-Request-Method: POST**

**Access-Control-Request-Headers: X-PINGOTHER**

HTTP/1.1 200 OK

Date: Mon, 01 Dec 2008 01:15:39 GMT

Server: Apache/2.0.61 (Unix)

**Access-Control-Allow-Origin: http://mondomaine.com**

**Access-Control-Allow-Methods: POST, GET, OPTIONS**

**Access-Control-Allow-Headers: X-PINGOTHER**

**Access-Control-Max-Age: 1800**

Vary: Accept-Encoding, Origin

Content-Encoding: gzip

Content-Length: 0

Keep-Alive: timeout=2, max=100

Connection: Keep-Alive

Content-Type: text/plain

// Requête réelle

POST /resources/not-simple HTTP/1.1

Host: autredomaine.com

User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US; rv:1.9.1b3pre) Gecko/20081130 Minefield/3.1b3pre

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8

Accept-Language: en-us,en;q=0.5

Accept-Encoding: gzip,deflate

Accept-Charset: ISO-8859-1,utf-8;q=0.7,\*;q=0.7

Connection: keep-alive

**X-PINGOTHER: pingpong**

Content-Type: text/xml; charset=UTF-8

Referer: http://mondomaine.com/index.phpContent-Length: 55

**Origin: http://mondomaine.com**

Pragma: no-cache

Cache-Control: no-cache

<?xml version="1.0"?><person><name>Paul</name></person>

HTTP/1.1 200 OK

Date: Mon, 01 Dec 2008 01:15:40 GMT

Server: Apache/2.0.61 (Unix)

**Access-Control-Allow-Origin: http://mondomaine.com**

**Access-Control-Expose-Header: Custom-Complementary-Header**

Vary: Accept-Encoding, Origin

Content-Encoding: gzip

Content-Length: 235

Keep-Alive: timeout=2, max=99

Connection: Keep-Alive

Content-Type: text/plain

**Custom-Complementary-Header: ae5f680gc**

*Exemple d’échange client/serveur pour une requite CORS avec « preflight »*

N.B : les specifications CORS indiquent que seuls 6 en-têtes peuvent être exposés au client par le navigateur (« Cache-Control », « Content-Language », « Content-Type », « Expires », « Last-Modified » et « Pragma »), tout en-tête complémentaire doit être spécifié comme dans l’exemple précédent à l’aide de « Access-Control-Expose-Headers ».

## Requêtes CORS avec « credentials »

Par défaut, les navigateurs ne transmettent pas les cookies et informations d’authentifications HTTP lorsque des requêtes *CORS* sont émises. Une requête unitaire peut cependant être tagguée avec la propriété « withCredentials » dans le navigateur, qui exposera ainsi ces données dans la requête au moment de l’envoi.

Le serveur doit par suite répondre avec l’en-tête « Access-Control-Allow-Credentials », si l’en-tête n’est pas présent ou que sa valeur est à « true », alors le navigateur n’expose pas le corps de la réponse au client.

// Fonction présente dans une page de http://mondomaine.com/index.php

function makeXMLRequest() {

xhr = new XMLHttpRequest();

xhr.open("GET", "http://www.autredomaine.com/priv/index.php”, true);

xhr.**withCredentials** = true;

xhr.onreadystatechange = handler;

xhr.send();

}

*Fonction JavaSript effectuant un appel CORS avec credentials (envoi des cookies)*

GET /resources/with-credentials/ HTTP/1.1

Host: autredomaine.com

User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US; rv:1.9.1b3pre) Gecko/20081130 Minefield/3.1b3pre

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8

Accept-Language: en-us,en;q=0.5

Accept-Encoding: gzip,deflate

Accept-Charset: ISO-8859-1,utf-8;q=0.7,\*;q=0.7

Connection: keep-alive

Referer: http://mondomaine.com/index.php

**Origin: http://mondomaine.com**

**Cookie: authID=2f5g6ui49jnvkl**

HTTP/1.1 200 OK

Date: Mon, 01 Dec 2008 01:34:52 GMT

Server: Apache/2.0.61 (Unix) PHP/4.4.7 mod\_ssl/2.0.61 OpenSSL/0.9.7e mod\_fastcgi/2.4.2 DAV/2 SVN/1.4.2

X-Powered-By: PHP/5.2.6

**Access-Control-Allow-Origin: http://mondomaine.com**

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

Cache-Control: no-cache

Pragma: no-cache

Set-Cookie: authID=234rjhkn908E; expires=Wed, 31-Dec-2008 01:34:53 GMT

Vary: Accept-Encoding, Origin

Content-Encoding: gzip

Content-Length: 106

Keep-Alive: timeout=2, max=100

Connection: Keep-Alive

Content-Type: text/plain

*Exemple d’échange client/serveur pour une requête CORS avec « credentials »*

## Détails de l’API CORS

L’API *CORS* est implémentée dans tous les moteurs de navigateurs majeurs (Gecko 1.9.1, Webkit, Trident 4.0, Presto)[[3]](#footnote-3) ainsi que les principales plateformes (Apache, IIS, Tomcat, etc.)[[4]](#footnote-4) et fournit différentes directives via de nouveaux en-têtes HTTP (headers) :

**En-tête HTTP de requête**

* « Origin » : une URI indiquant le serveur à partir duquel la demande est initiée (informations de chemin exclues) ;
* « Access-Control-Request-Method » : utilisé lors de l'émission d'une demande de contrôle en amont pour spécifier au serveur quelle méthode HTTP sera utilisée lors de la demande réelle ;
* « Access-Control-Request-Headers » : utilisé lors de l'émission d'une demande de contrôle en amont pour spécifier au serveur quels en-têtes HTTP seront utilisés lors de la demande réelle.

**En-tête HTTP de réponse**

* « Access-Control-Allow-Origin » : spécifie une URI (et une seule) pouvant accéder à la ressource, le contrôle étant effectué par le navigateur (pour une requête sans authentification et sur une page non sensible, le serveur peut spécifier "\*") ;
* « Access-Control-Expose-Headers » : spécifie la liste des en-têtes retournés que le navigateur est autorisé à mettre à disposition du script client ;
* « Access-Control-Max-Age » : indique combien de temps (en secondes) les résultats d'une demande de contrôle en amont peuvent être mis en cache ;
* « Access-Control-Allow-Credentials » : indique si la réponse peut être exposée par le navigateur (booléen), dans le cas où la requête a été émise avec la propriété « withCredentials » (ne transparait pas dans la requête, l’information n’étant connue que du navigateur). Dans le cas d'une réponse à une demande de contrôle en amont, cet en-tête indique si la demande réelle doit être faite en transmettant des informations d'authentification.
* « Access-Control-Allow-Methods» : indique en réponse à une demande de contrôle en amont la ou les méthodes autorisées lors de l'accès à la ressource.
* « Access-Control-Allow-Headers» : indique en réponse à une demande de contrôle en amont la liste des en-têtes HTTP pouvant être utilisés lors de la demande réelle.

Notons que, par mesure de sécurité, il n’est pas possible de positionner les en-têtes mentionnés précédemment aux valeurs suivantes dans une même réponse :

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

# Considérations de sécurités CORS

L’échange de données via CORS entre deux domaines est basé sur un certain niveau de confiance entre les domaines. Le domaine client attend que le domaine serveur fournisse une réponse valide. Le domaine serveur attend lui que le domaine client soit bien autorisé à accéder à ses données.

Même si dans beaucoup de cas, les deux parties sont légitimes et du même fournisseur, la confiance placée dans le partenaire sur Internet doit toujours être minimale. En effet, le partenaire, bien que légitime, peut avoir été compromis, et profiter de la relation de confiance pour usurper l’identité du site partenaire.

D’une manière générale, il est donc très important pour le domaine client de valider les données reçues et pour le domaine serveur d’exposer le minimum de service possible. La compromission d’un des deux sites ne doit pas résulter dans la compromission des deux sites.

De manière plus spécifique, plusieurs risques sont identifiés à l’utilisation de *CORS* :

* Absence de restriction des accès *CORS* (« Universal Allow ») ;
* Gestion éparse des accès *CORS* ;
* Traitement des requêtes *CORS* illégitimes ;
* Altération des en-têtes *CORS* ;
* Outrepassement protocolaire ou prolongation de la mise en cache ;
* Sensibilité aux attaques CSRF.

## Absence de restriction des accès CORS

Il s’agit de l’erreur la plus communément mise en œuvre : « Universal Allow ». L’entête « Access-Control-Allow-Origin » doit contenir la liste des sites qui peuvent effectuer des requêtes *CORS*, mais cet en-tête peut également être retournée par le serveur avec un wildcart « \* » pour valeur, permettant ainsi à tous les sites sans distinction d’effectuer des requêtes *CORS* sur la page et de lire et exploiter les réponses.

header("Access-Control-Allow-Origin: \*");

*Exemple de directive PHP positionné dans une page et à l’origine de l’« Universal Allow »*

Cela peut être problématique dans de nombreux cas :

* Si une partie d’un site interne, interdisant l’accès depuis l’extérieur, a un « Universal Allow » positionné, un utilisateur interne qui visiterait un site malicieux pourrait voir ce site effectuer des requêtes *CORS* vers des ressources internes afin de dérober le contenu des réponses ;
* Une variante de ce problème se présente quand un site est accessible depuis Internet mais a un comportement légèrement différent quand il est accédé depuis l’intranet. Par exemple, l’exposition en interne de nouvelles fonctionnalités accessibles aux utilisateurs internes uniquement ;
* A noter qu’il est aussi possible d’attaquer directement l’utilisateur à contrario du serveur. Un utilisateur malicieux peut pour l’exemple écrire un code *JavaScript* exploitant une faille quelconque (*SQL Injection* par exemple) sur le site autorisant les requêtes CORS en mode « Universal Allow », et installer celui-ci sur un site tiers qu’il maitrise puis inciter un utilisateur à naviguer sur celui-ci. L’attaque semblera finalement venir de l’utilisateur ainsi piégé (scénario CSRF)

## Gestion éparse des accès CORS

Par conception, *CORS* nécessite un contrôle d’accès sur chaque page. Une page ne peut donc être accédée depuis un site étranger que si celle-ci est autorisée explicitement.

Cela peut être un inconvénient si, pour l’exemple, un site présentant X pages autorisées en accès avec *CORS* doit faire évoluer la liste des sites autorisés à effectuer des requêtes inter-domaines vers ces X pages. Le développeur devra le cas échéant répercuter manuellement le changement ou utiliser une directive d’inclusion pour permettre de regrouper ces autorisations.

Le risque d’erreur humaine ou de généralisation des accès sans restriction fine pour chaque page est décuplé (dans le pire de cas, accès total au site).

## Traitement des requêtes CORS illégitimes

Des requêtes *CORS* peuvent être traitées illégitimement, même si :

* le serveur / l’application web n’est pas configuré pour permettre ces requêtes sur la ressource / page sollicitée ;
* la ressource / page est configurée pour permettre ces requêtes mais depuis une autre origine.

Dans ces cas d’usage, la réponse du serveur n’est théoriquement pas visible du client de par l’absence de positionnement de l’en-tête « Access-Control-Allow-Origin » par le serveur. Il est cependant possible :

* d’ainsi effectuer une attaque DDoS au niveau applicatif ;
* d’exploiter de potentielles vulnérabilités coté serveur, la requête étant effectivement traitée et la réponse transmise bien que non exposée au client par le navigateur ;
* d’intercepter la réponse et d’altérer les en-têtes afin de tout même exposer celle-ci au client.

Les parties du site qui doivent être accessibles à des sites externes et qui peuvent être la cible de ce type d’attaque, doivent être protégées en conséquence.

## Altération des en-têtes CORS

Par conception, *CORS* effectue la majorité de ces contrôles sur le contenu des en-têtes explicités précédemment, qu’ils soient positionnés par le navigateur client ou le serveur.

Ces contrôles sont cependant effectués avec un niveau de confiance non maitrisé, il en effet possible d’altérer ces en-têtes volontairement ou à l’insu des différents protagonistes (scénario d’attaque Man In the Middle).

Pour l’exemple, la page ci-dessous affiche deux informations différentes selon l’origine de la requête. Le contrôle d’accès à cette page est effectué en fonction de l’en-tête « Origin » positionnée par le navigateur.

if (isset($\_SERVER["HTTP\_ORIGIN"])) {  
 if ($\_SERVER['HTTP\_ORIGIN'] == "http://A") {  
 header('Access-Control-Allow-Origin: http://A');  
 echo "Confidential Data";  
 }  
 else {  
 echo "Denied".  
 }  
}

Cette protection peut être contournée puisque l’en-tête « Origin » est, comme tout en-tête HTTP reçu par le serveur, falsifiable.

De la même manière, les en-têtes « Access-Control-Allow-Origin » et « Access-Control-Allow-Credentials » reçu dans la réponse par le navigateur sont falsifiables et ne doivent pas être considérés avec un niveau de confiance élevée, d’autant plus si le traitement des requêtes CORS illégitimes est possible.

## Outrepassement protocolaire et prolongation de la mise en cache

Les requêtes « preflight » ont pour l'objectif de s’assurer que la requête CORS réelle n’aura pas un impact négatif sur les données en contrôlant la méthode, le type de contenu et les headers utilisés.

Un risque réside ainsi dans le fait que la décision d’émettre une requête « preflight » est à la charge du navigateur uniquement, aucune garantie n’étant donnée au serveur que ces requêtes « preflight » seront systématiquement émises.

Il est ainsi possible de créer / envoyer (en utilisant des outils comme Curl, OWASP Zap Proxy, etc.) une requête HTTP finale sans préalablement envoyer la pré-requête.

Par ailleurs toujours dans le cas d’usage ou une requête « preflight » est effectuée, la réponse du serveur peut être mise en cache afin d’anticiper de potentiels problèmes de performance liés à l’envoi répété de ces pré-requêtes. Le temps maximum de mise en cache peut alors être spécifié dans la réponse du serveur via l’en-tête « Access-Control-Max-Age».

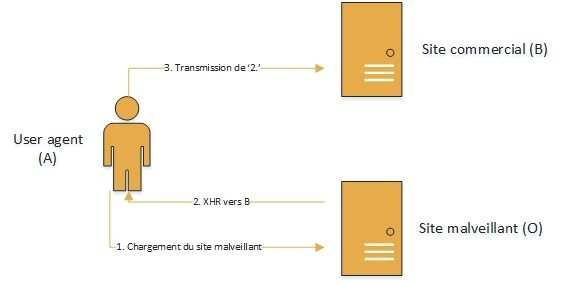
Dans le cas où la politique de contrôle d’accès *CORS* viendrait à être modifiée sur le serveur, une valeur trop importante de durée de conservation en cache de ces réponses pourrait permettre d’effectuer des traitements non autorisés par l’intermédiaire de requêtes CORS devenues illégitimes.

## Sensibilité aux attaques CSRF

Soient trois protagonistes : un utilisateur A ayant un compte sur un site B (et par conséquent un cookie de session) et un site malveillant O. Le site malveillant contient le script suivant :

request = new XMLHttpRequest();  
request.open("GET","http://URL\_B/service.php?buy=apple",true);  
request.withCredentials = "true";  
request.onreadystatechange = function() {  
 if (request.readyState == 4) {  
 var response = request.responseText;  
 document.getElementById("response").textContent = response;  
 }  
}  
request.send();

Dans ce scénario, dès que l’utilisateur visite le site malveillant O, une requête sera envoyée à B par le biais de A et à l’aide du script ci-dessus. L’attribut « withCredentials » fixé à true indique que le cookie de A sera transmis à B. En exécutant l’action « buy », il y a donc eu CSRF sans que l’utilisateur ne la déclenche de lui-même.

[](https://securite.intrinsec.com/wp-content/uploads/2013/04/article_COR_exemple2.jpg)

*Illustration d’un scénario CSRF*

Une protection d’un point de vue serveur contre ce type d’attaques est donc nécessaire.

# ****Préconisations****

**Préconisations à mettre en œuvre au niveau serveur de manière prioritaire**

1. Rejeter les requêtes reçues en HTTP avec une origine en HTTPS, afin d’éviter les problèmes de contenus mixtes ;
2. Préférer une liste blanche de domaines autorisés à effectuer des requêtes inter-domaines à une liste noire ou un « Universal Allow ». En cas de refus d’accès, ne pas positionner l’en-tête « Access-Control-Allow-Origin » dans la réponse ;
3. Si l’utilisation d’un « Universal Allow » est indispensable, veiller à ce que les URI ainsi retournés avec « Access-Control-Allow-Origin: \* » n’incluent pas des contenus sensibles ou des informations qui pourraient aider un attaquant ;
4. Ne positionner l'en-tête « Access-Control-Allow-Origin » que sur les URI devant être disponibles inter-domaines. Ne pas l’utiliser pour l'ensemble du domaine et réécrire systématiquement l’en-tête dans la réponse (i.e ne pas faire confiance à un en-tête déjà positionné dans la requête) ;
5. Vérifier l’unicité (non vide) des en-têtes « Origin » et « Host » ;
6. Rejeter sans autre traitement les requêtes *CORS* émises sur des pages non accessibles inter-domaines ;

Exemple :

<?php  
 if(isset($\_SERVER['HTTP\_ORIGIN'])) {  
 exit;  
 }  
 else {  
 // process request;  
 }

?>

1. Rejeter sans autre traitement les requêtes CORS émises sur des pages accessibles inter-domaines depuis un domaine non autorisé ;

Exemple :

<?php  
 if($\_SERVER['HTTP\_ORIGIN'] == 'http://trusted.site') {  
 header('Access-Control-Allow-Origin: http://trusted.site');  
 //process request  
 }  
 else {  
 exit;  
 }  
?>

1. Chaque fois qu’une fonctionnalité importante ou que des données sensibles sont exposées, accorder un niveau de confiance minimal aux en-têtes *CORS* et authentifier l’ensemble des requêtes. Réécrire systématiquement l’en-tête « Access-Control-Allow-Credentials » dans la réponse (i.e ne pas faire confiance à un en-tête déjà positionné dans la requête)  et conditionner également le contenu de la réponse en fonction du résultat de l’authentification : rejeter sans autre traitement une requête impossible à authentifier.
2. Pour toute requête *CORS,* mettre en œuvre un mécanisme an-CSRF à jeton (aléatoire, non prédictible et unique) et rejeter sans autre traitement une requête présentant un jeton invalide. (Dans une cinématique formulaire, le jeton doit être généré et inséré dans la page au moment où le client effectue la requête pour récupérer celui-ci).
3. Limiter le temps de mise en cache des réponses « preflight » à 24 heures.

**Préconisations à mettre en œuvre au niveau serveur de manière secondaire**

1. Mettre en cache l’adresse IP (éventuellement couplée au UserAgent pour les utilisateurs accédant par l’intermédiaire d’un proxy) pour 1 heure. Rejeter les requêtes de tout utilisateur ainsi identifié ayant émis une requêtes *CORS* depuis un domaine non autorisé ;
2. Utiliser le cache serveur afin de conserver un état des étapes de « preflight » et rejeter sans autre traitement toute requête nécessitant une pré-requête non effectuée au préalable ;
3. Industrialiser la gestion (BDD, etc.) et définir un mécanisme de suivi des accès *CORS* positionnés sur chaque ressource du serveur afin d’éviter les erreurs de configuration humaines et le positionnement d’« autorisations génériques » ;

**Préconisations à mettre en œuvre au niveau applicatif (exécution dans le navigateur client) de manière prioritaire**

1. Anticiper de potentiels détournements du code applicatif (par injection par exemple) et valider les URI invoquées à l’aide de la fonction « XMLHttpRequest.open » en portant une attention particulière aux URI absolues.

**Préconisations à mettre en œuvre au niveau applicatif (exécution dans le navigateur client) de manière secondaire**

1. De la même manière, s’appuyer sur des contrôles applicatifs (jetons, etc.) pour exposer le contenu d’une réponse au client et ne pas s’appuyer uniquement sur les en-têtes « Access-Control-Allow-Origin » et « Access-Control-Allow-Credentials » à l‘instar du navigateur ;
2. S’assurer qu’un un niveau de confiance minimal est accordé au comportement du navigateur client (pouvant être compromis ou obsolète) et vérifier ainsi la présence des en-têtes *CORS* attendus ainsi que leurs valeurs vis-à-vis de ce qui est attendu conformément au protocole.

# Références

* <http://code.google.com/p/html5security/wiki/CrossOriginRequestSecurity>
* <http://securite.intrinsec.com/2013/06/14/html-5-cross-origin-resource-sharing-securiser-communications-interdomaines/>
* <http://fr.slideshare.net/michaelneale/cors-michael-webdirections?next_slideshow=1>
* <http://margaine.com/2014/06/28/jsonp-vs-cors.html>
* <https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS>
* <https://developer.mozilla.org/en-US/docs/Web/HTTP/Server-Side_Access_Control>
* <https://www.owasp.org/index.php/HTML5_Security_Cheat_Sheet>
* <https://www.owasp.org/index.php/CORS_RequestPreflighScrutiny>
* <https://www.owasp.org/index.php/CORS_OriginHeaderScrutiny>

1. Une configuration est possible niveau serveur, mais ne permet d’adresser que des besoins simplistes. [↑](#footnote-ref-1)
2. Voir signification et utilité en section 4.4 Détails de l’API CORS [↑](#footnote-ref-2)
3. http://enable-cors.org/client.html [↑](#footnote-ref-3)
4. http://enable-cors.org/server.html [↑](#footnote-ref-4)