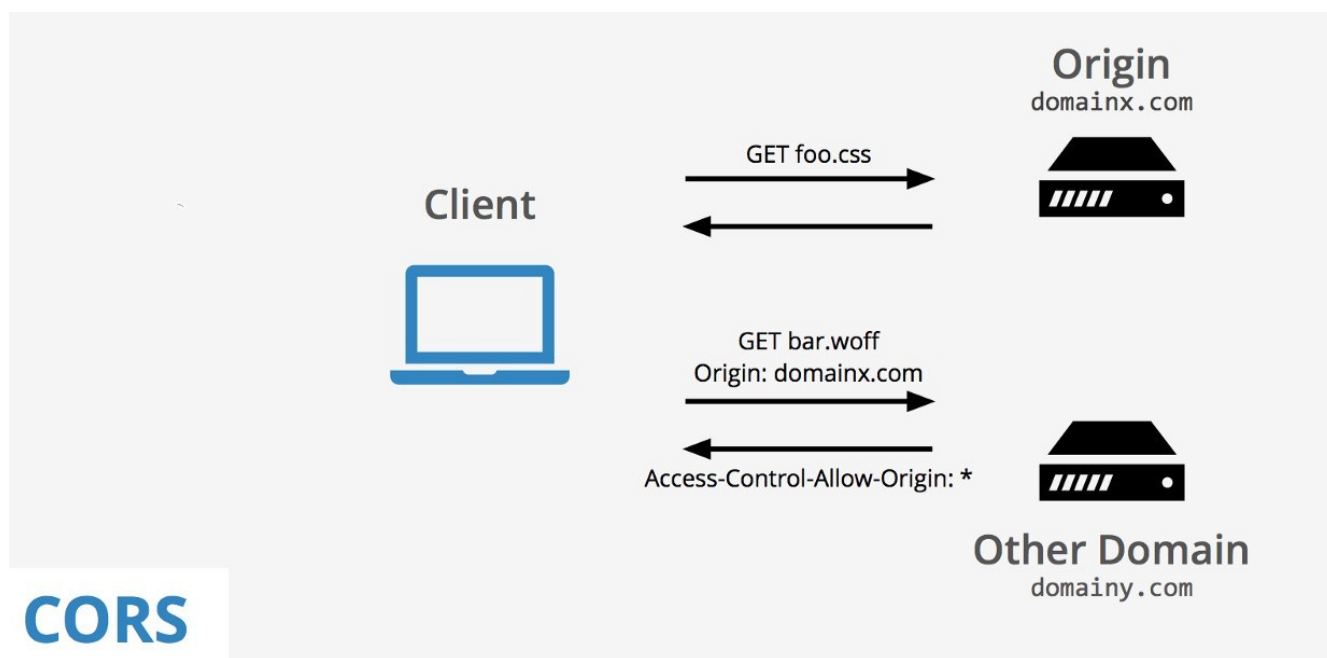
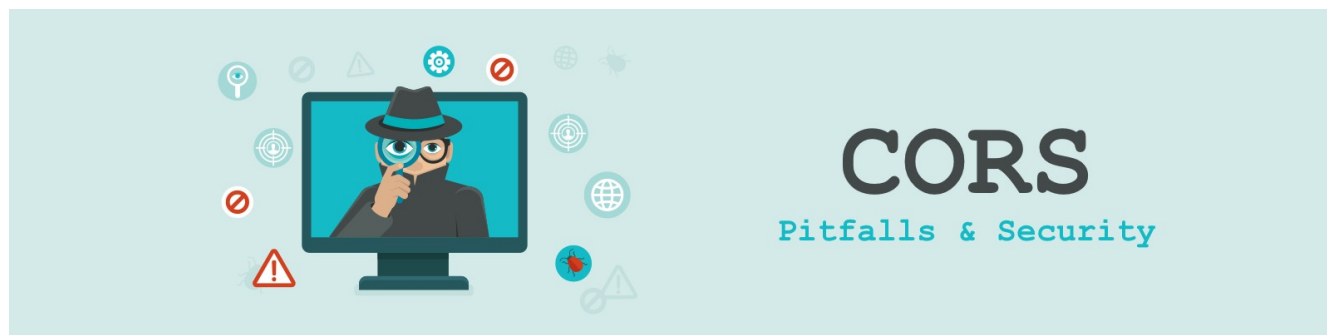


HACKING HTTP CORS



Sumario

1. Introducción.....	3
Simple Request.....	3
Preflight Request.....	4
Cabeceras básicas en CORS.....	5
3. Caso práctico.....	6
Setup del laboratorio.....	6
Estructura del proyecto.....	6
Ejecución del escenario.....	8
Comprendiendo las peticiones CORS.....	10
Configuración de un proxy web para interceptación de las peticiones HTTP.....	11
CORS bypass mediante HTTP Response tampering.....	13
3. Ejercicios propuestos.....	14
4. Bibliografía.....	16

1. Introducción

HTTP CORS es un mecanismo que permite a una página web situada en un dominio A, acceder a recursos situados en un dominio B distinto. Por defecto, esto no es posible realizarlo debido a la política Same Origin Policy (SOP) propuesta en 1995 por Netscape.

¿Qué se entiende por origen? El origen consta del protocolo o esquema, el host y el puerto por lo que dos URL representan el mismo origen si comparten estos mismos elementos. Por ejemplo, para el dominio http://hacking.cors.com/mr_robots.txt, la siguiente tabla muestra dominios que cumplen el mismo origen o no.

URL	Same Origin?	Description
http://hacking.cors.com/path/file.txt	YES	Another path, but the same tuple.
http://hacking.cors.com/	YES	Another path, but the same tuple.
https://hacking.cors.com/mr_robots.txt	NO	Different protocol (HTTPS)
http://noevil.cors.com/mr_robots.txt	NO	Different subdomain/hostname
http://hacking.cors.com:8080/mr_robots.txt	NO	Different port (8080)

Para poder relajar estas restricciones estrictas se creó CORS. De este modo, se permite que páginas de un dominio puedan obtener recursos de otro dominio. El responsable de autorizar el acceso a un recurso es responsabilidad de su dueño.

La relación de confianza entre el navegador y el servidor se establece a través de cabeceras HTTP específicas para CORS. Las cabeceras de respuesta del servidor informan al navegador si confiar o no en el origen y confiar en la petición para continuar con ella. **Lo importante para nosotros es que el navegador confía siempre en la respuesta del servidor.**

Hay diferentes tipos de peticiones CORS, veremos cómo funcionan las peticiones simples (*Simple Request*) y las peticiones preflight (*Preflight Request*).

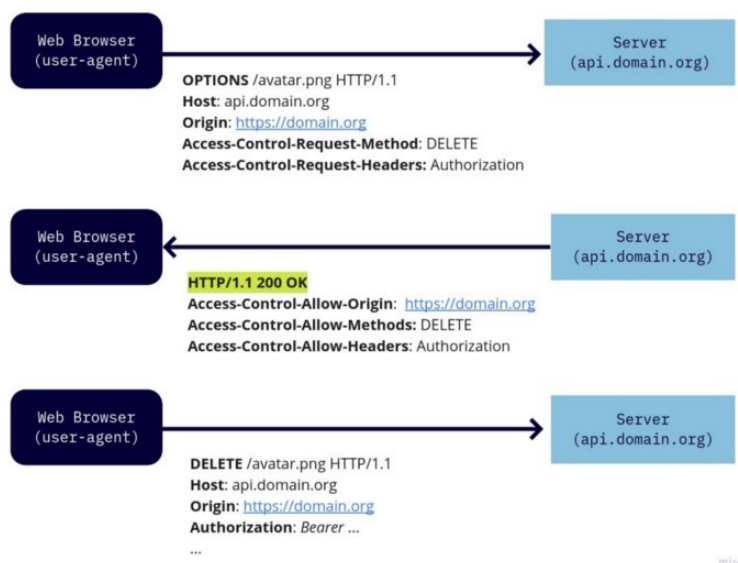
Simple Request.

Estas peticiones cumplen las siguientes condiciones en cuanto a los métodos y cabeceras empleados en la petición.

HTTP Methods	Headers*	Content-Type Values**
GET	Accept	application/x-www-form-urlencoded
HEAD	Accept-Language	multipart/form-data
POST	Content-Language	text/plain
	Content-Type**	
	DPR	
	Downlink	
	Save-Data	
	Viewport-Width	
	Width	

Preflight Request.

Estas peticiones son lanzadas por el navegador para comprobar de qué modo está autorizada la petición y si es segura. En caso de que sea así, se realiza la petición para obtener el recurso. La petición preflight utiliza la cabecera HTTP OPTIONS.



Cabeceras básicas en CORS.

Las siguientes cabeceras son las que se emplean para establecer los mecanismos de control.

Header	Used for <i>Preflight</i> HTTP?	HTTP Type
Access-Control-Allow-Origin	NO	Response
Access-Control-Allow-Credentials	NO	Response
Access-Control-Allow-Headers	YES	Response
Access-Control-Allow-Methods	YES	Response
Access-Control-Expose-Headers	NO	Response
Access-Control-Max-Age	YES	Response
Access-Control-Request-Headers	YES	Request
Access-Control-Request-Method	YES	Request
Origin	NO	Request
Timing-Allow-Origin	NO	Response

Las cabeceras marcadas con un YES juegan un papel crucial en las peticiones preflight:

- **Access-Control-Allow-Headers.** Establece qué cabeceras están permitidas.
- **Access-Control-Allow-Methods.** Establece qué métodos HTTP están permitidos.
- **Access-Control-Max-Age.** Establece la cantidad máxima de segundos que el navegador debe guardar en caché la petición.

De forma general, la cabecera más importante es **Access-Control-Allow-Origin**, ya que con ella el servidor indica al navegador en qué dominios debe confiar.

3. Caso práctico

Setup del laboratorio.

Usaremos un laboratorio docker que clonaremos de GitHub.

```
git clone https://github.com/lvrosa/hacking-cors
```

Una vez descargado, accedemos al directorio raíz *hacking-cors* y ejecutamos **npm install** para instalar las dependencias de Javascript necesarias, en caso de no tenerlas.

```
jose@jose-Aspire-VN7-791:~/vmcompartido/docker/hacking-cors$ npm install
hacking-cors@1.0.0 /home/jose/vmcompartido/docker/hacking-cors
├── http@0.0.1-security
├── http-proxy@1.18.1
├── require@2.4.20
├── url@0.11.0
└── xmlhttprequest@1.8.0
```

También es necesario tener instalado en el sistema **docker** y **docker-compose**.

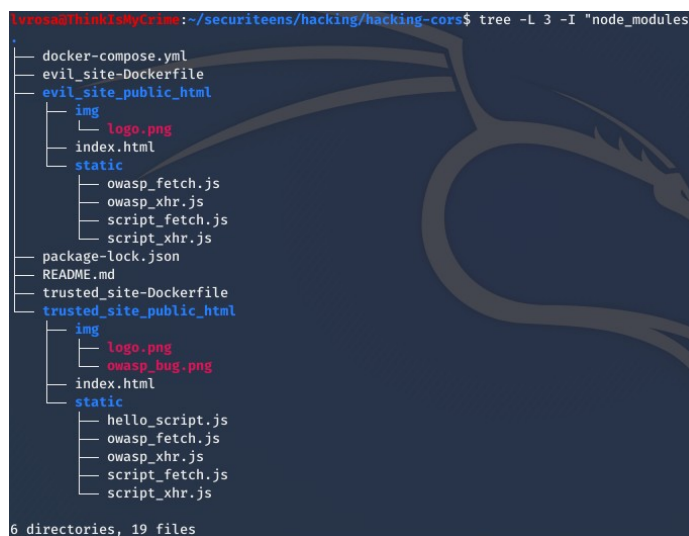
```
jose@jose-Aspire-VN7-791:~/vmcompartido/docker/hacking-cors$ docker-compose -v
docker-compose version 1.25.5, build unknown
jose@jose-Aspire-VN7-791:~/vmcompartido/docker/hacking-cors$ docker -v
Docker version 19.03.13, build cd8016b6bc
```

Repositorio GIT con el escenario de práctica.

<https://github.com/lvrosa/hacking-cors>

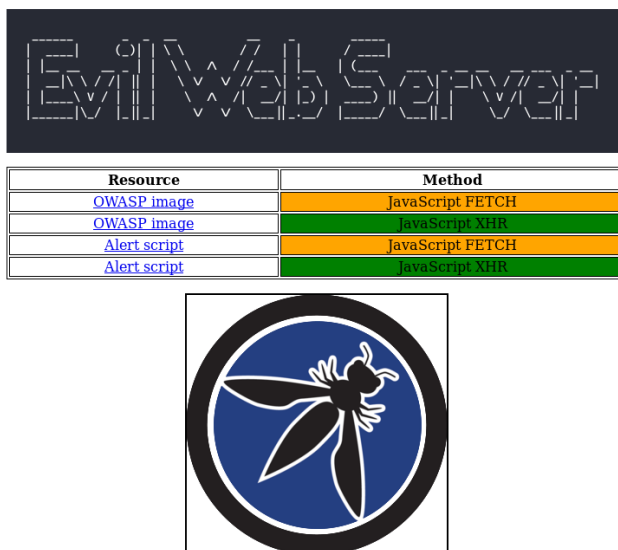
Estructura del proyecto.

La estructura de directorios se muestra en la siguiente imagen.



```
lvrosa@ThinkIsMyCrime:~/securiteens/hacking/hacking-cors$ tree -L 3 -I "node_modules"
.
├── docker-compose.yml
├── evil_site-Dockerfile
├── evil_site_public_html
│   ├── img
│   │   └── logo.png
│   ├── index.html
│   └── static
│       ├── owasp_fetch.js
│       ├── owasp_xhr.js
│       ├── script_fetch.js
│       └── script_xhr.js
├── package-lock.json
├── README.md
├── trusted_site-Dockerfile
├── trusted_site_public_html
│   ├── img
│   │   ├── logo.png
│   │   └── owasp_bug.png
│   ├── index.html
│   └── static
│       ├── hello_script.js
│       ├── owasp_fetch.js
│       ├── owasp_xhr.js
│       ├── script_fetch.js
│       └── script_xhr.js
└── 6 directories, 19 files
```

Este laboratorio consta de dos sitios web: **EvilSite** y **TrustedSite**.



Cada uno de los sitios web dispone de un fichero de configuración de docker y dos carpetas **img** y **static**. Ambas almacenan los recursos que podrán ser solicitados por otras páginas web. Los ficheros javascript se encuentran en el directorio static.

Project name	Docker configuration file
Trusted Site	trusted_site-Dockerfile
Evil Site	evil_site-Dockerfile

En el directorio *img* de *TrustedSite* se encuentra la imagen **owasp_bug.png** que será solicitada por el *EvilSite* para tratar de cargarla. De igual modo, *EvilSite* tratará de cargar/ejecutar el fichero **static/hello_script.js**.

La información de configuración de las imágenes docker y la red se encuentra en el fichero **docker-compose.yml**, donde observamos que se crean dos contenedores: **evil_site (10.5.0.3)** y **trusted_site (10.5.0.2)** enlazados a la red **cors_hack_net (10.5.0.0/16)**.

```
1  version: '3'
2
3  services:
4    trusted_app:
5      container_name: trusted_site
6      image: trusted_site_img
7      build:
8        context: .
9        dockerfile: ./trusted_site-Dockerfile
10     ports:
11       - "8080:80"
12     networks:
13       cors_hack_net:
14         ipv4_address: 10.5.0.2
15
16     evil_app:
17       container_name: evil_site
18       image: evil_site_img
19       build:
20         context: .
21         dockerfile: ./evil_site-Dockerfile
22     ports:
23       - "8081:80"
24     networks:
25       cors_hack_net:
26         ipv4_address: 10.5.0.3
27
28   networks:
29     cors_hack_net:
30       driver: bridge
31       ipam:
32         config:
33           - subnet: 10.5.0.0/16
```

Ejecución del escenario.

Para poder acceder a los diferentes sitios de forma sencilla es recomendable añadir las siguientes entradas a nuestro fichero **/etc/hosts**:

10.5.0.2	www.trustedsite.com trustedsite.com
10.5.0.3	www.evilsite.com evilsite.com

En primer lugar, debemos construir los contenedores. Los comandos que siguen a continuación se ejecutan desde el directorio raíz del proyecto.

```
$ sudo docker-compose build --no-cache
Building trusted_app
Step 1/4 : FROM httpd:2.4
--> d5995e280a0e
Step 2/4 : COPY ./trusted_site_public_html/ ./htdocs/
--> a19ed4dd3c56
Step 3/4 : RUN sed -i 's/AllowOverride None/AllowOverride All/g' ./conf/httpd.conf
--> Running in 8bd77eelfe14
Removing intermediate container 8bd77eelfe14
--> b6c4f140750f
Step 4/4 : RUN sed -i '1 i\ServerName localhost' ./conf/httpd.conf
```



```
---> Running in 1c1147f383e6
Removing intermediate container 1c1147f383e6
---> dde76b0cb233
Successfully built dde76b0cb233
Successfully tagged trusted_site_img:latest
Building evil_app
Step 1/3 : FROM httpd:2.4
---> d5995e280a0e
Step 2/3 : COPY ./evil_site_public_html/ ./htdocs/
---> 51d461a3ad5d
Step 3/3 : RUN sed -i '1 i\ServerName localhost' ./conf/httpd.conf
---> Running in 75f7e2f73352
Removing intermediate container 75f7e2f73352
---> 6017883465db
Successfully built 6017883465db
Successfully tagged evil_site_img:latest
```

Para arrancar los contenedores usamos el comando ***docker-compose up***.

```
$ sudo docker-compose up
Creating network "hacking-cors_cors_hack_net" with driver "bridge"
Creating evil_site ... done
Creating trusted_site ... done
Attaching to evil_site, trusted_site
evil_site | [Thu Apr 08 09:44:37.184611 2021] [mpm_event:notice] [pid 1:tid 140626936083584]
AH00489: Apache/2.4.46 (Unix) configured -- resuming normal operations
evil_site | [Thu Apr 08 09:44:37.184753 2021] [core:notice] [pid 1:tid 140626936083584] AH00094:
Command line: 'httpd -D FOREGROUND'
trusted_site | [Thu Apr 08 09:44:38.253925 2021] [mpm_event:notice] [pid 1:tid 139943087670400]
AH00489: Apache/2.4.46 (Unix) configured -- resuming normal operations
trusted_site | [Thu Apr 08 09:44:38.254029 2021] [core:notice] [pid 1:tid 139943087670400] AH00094:
Command line: 'httpd -D FOREGROUND'
```

En otra terminal, podemos consultar los contenedores creados con ***docker-compose ps***.

```
$ sudo docker-compose ps
  Name                Command             State      Ports
  -----
evil_site             httpd-foreground    Up         0.0.0.0:8081->80/tcp
trusted_site          httpd-foreground    Up         0.0.0.0:8080->80/tcp
```

Podemos detener los contenedores en cualquier momento con ***docker-compose down***.

En este momento podemos abrir nuestro navegador y acceder a los dos sitios web para comprobar que se muestran correctamente. El sitio ***TrustedSite*** inicialmente muestra la imagen en blanco pero puede ser cargada mediante Javascript con el método *XMLHttpRequest* o la *API Fetch* con los enlaces que tenemos en la parte superior. Asimismo, se pueden cargar los mensajes *alert* de Javascript pulsando los enlaces correspondientes. En cambio, en ***EvilSite***, estos enlaces no funcionan, vemos que CORS está haciendo su trabajo correctamente.

Nos conectaremos ahora al contenedor ***trusted_site*** para comprobar la configuración del fichero ***.htaccess***.

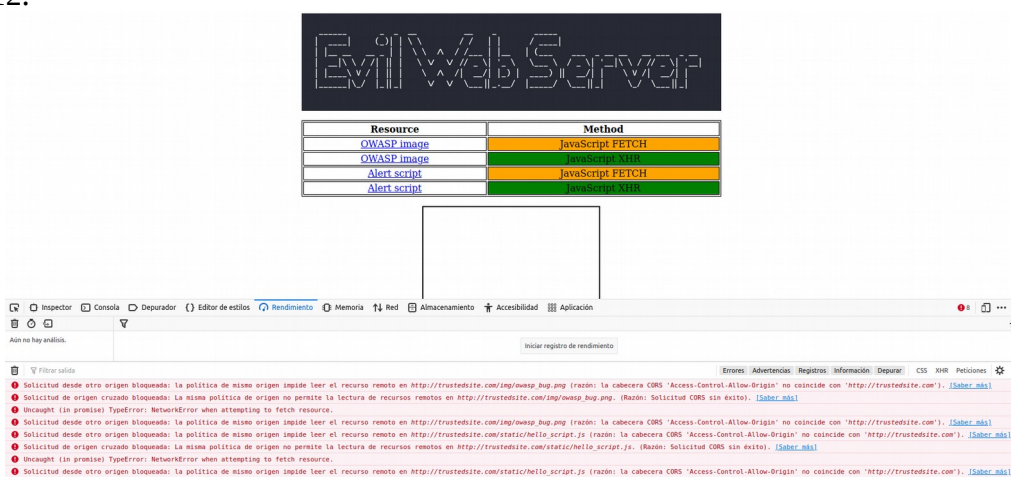
```
$ sudo docker exec -it trusted_site /bin/bash
root@e7e2e77482a9:/usr/local/apache2# cat htdocs/.htaccess
<IfModule mod_setenvif.c>
```

```
<IfModule mod_headers.c>
  <FilesMatch "\.(cur|gif|ico|jpe?g|png|svgz?|webp|js)$">
    Header add Access-Control-Allow-Origin http://trustedsite.com
    Header merge Vary Origin
  </FilesMatch>
</IfModule>
</IfModule>
```

Como se puede observar, el servidor Apache devuelve la cabecera **Access-Control-Allow-Origin** cuando se realiza una petición a alguno de los ficheros indicados por la regla **FilesMatch**. De este modo, el recurso solo se podrá autorizar a peticiones cuyo origen coincidan con **http://trustedsite.com**. Esto explica porqué los enlaces sí funcionan para *TrustedSite* y no para *EvilSite*.

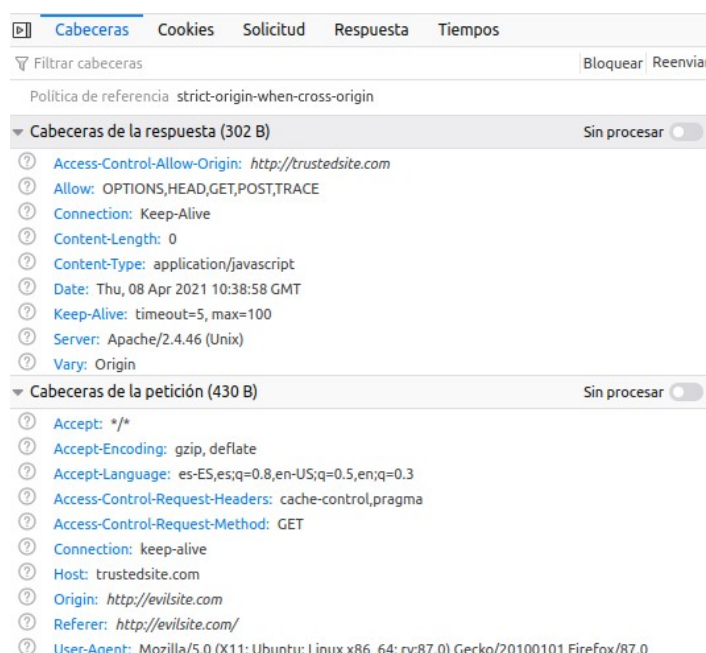
Comprendiendo las peticiones CORS.

Tras solicitar los recursos desde EvilSite, abre las herramientas de desarrollo web del navegador pulsando F12.



Vemos como se obtiene el mensaje de error “*Solicitud desde otro origen bloqueada: la política de mismo origen impide leer el recurso remoto...*”.

Vamos a analizar en más detalle la petición al recurso **hello_script.js**. Pinchamos en la pestaña **Red** y seleccionamos la **petición OPTIONS**, en la parte derecha de la ventana se mostrarán las cabeceras de las peticiones **HTTP Response** y **HTTP Request**.



Observamos cómo la **cabecera Origin** en la petición tiene el valor `http://evilsite.com`. Si estudiamos el código fuente de `script_fetch.js` de `EvilSite` vemos como este ha intentado falsificar la cabecera `Origin`. Sin embargo, la implementación del navegador ha prevenido este intento de modificación maliciosa.

```
1 function load_script_via_fetch() {
2   fetch('http://trustedsite.com/static/hello_script.js', {
3     method: 'GET',
4     headers: {
5       'pragma': 'no-cache',
6       'cache-control': 'no-cache',
7       'origin': 'http://trustedsite.com',
8     })
9   .then(function(response) {
10     return response.text().then(function(text) {
11       eval(text);
12     });
13   });
14 }
```

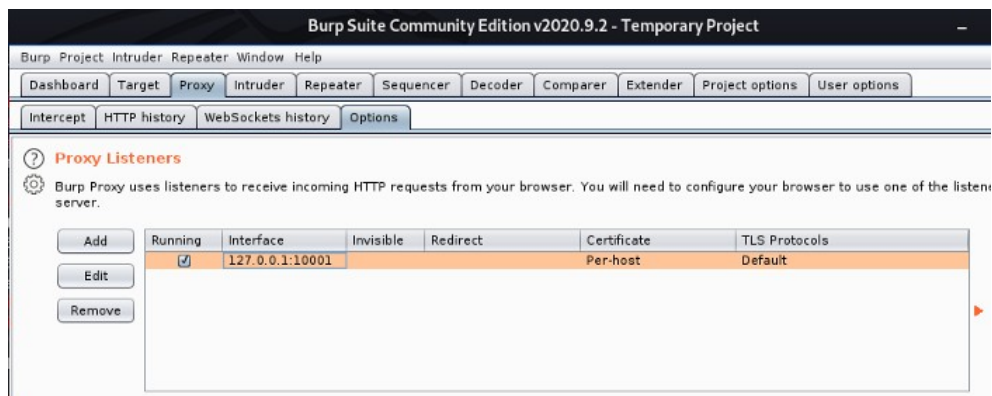
A continuación veremos diferentes técnicas para saltarnos la protección del navegador.

Configuración de un proxy web para interceptación de las peticiones HTTP.

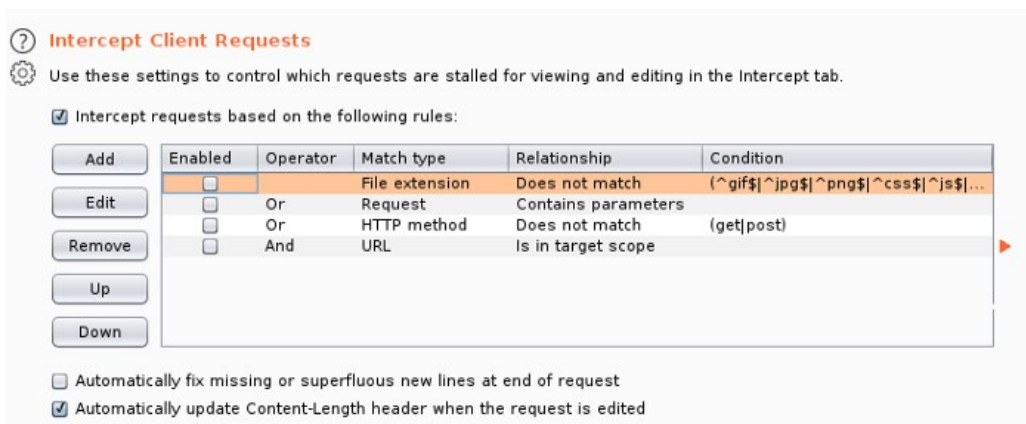
Para realizar la interceptación de las cabeceras HTTP usamos un proxy como ZAP o Burp Suite. Para este caso guiado lo haremos con el segundo.

En la pestaña **Proxy – Options** editamos el puerto del interfaz que aparece por defecto en la

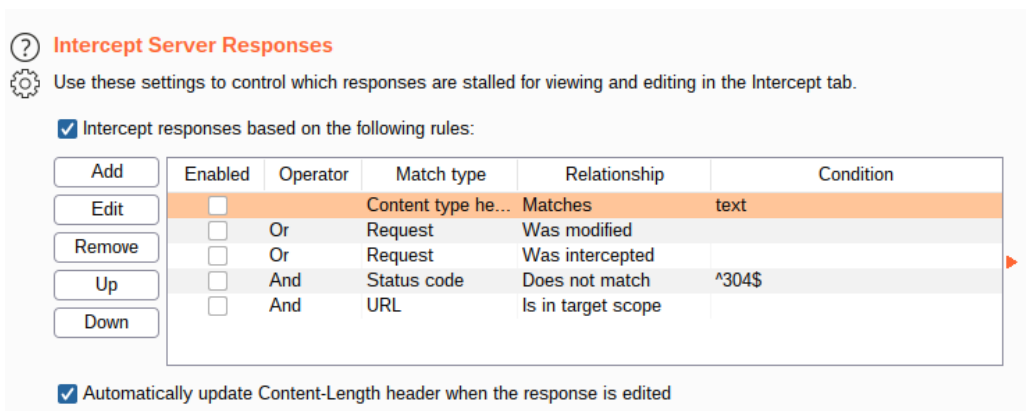
configuración de Proxy Listeners para establecerlo a **127.0.0.1:10001** y marcamos la casilla **running**.



A continuación, en la misma pestaña establecemos la configuración adecuada en **Intercept Client Request** como se muestra en la imagen. Nos cercioramos de no tener marcada ninguna casilla **enabled**.



Seguidamente, configuramos las opciones de **Intercept Server Responses** como se muestra en la imagen siguiente, cerciorándonos de tener marcada la casilla **Intercept responses based on the following rules**, y no activar ninguna casilla **enabled** para capturar todas las respuestas.



Por último, comprobamos que en la pestaña **Proxy – Intercept** está activada la intercepción (*Intercept is on*).

Una vez que hemos configurado el proxy, es necesario indicar a nuestro navegador que redirija las peticiones a través de nuestro proxy. Podemos hacerlo con las opciones de red del propio navegador o usar una extensión como **FoxyProxy** (esta última opción es más cómoda para realizar labores continuas de auditoria o pentesting).

Nota

Según el tipo y versión del navegador podemos tener problemas a la hora de capturar las cabeceras de las peticiones y respuesta de tipo Javascript XHR. En tal caso, como alternativa se puede emplear el navegador que incorpora Burp Suite en Proxy – Intercept – Open Browser.

Proxy Type
HTTP

Proxy IP address or DNS name ★
127.0.0.1

Port ★
10001

Username (optional)
username

Password (optional) 👁

Cancel Save & Add Another Save

Web de FoxyProxy

<https://getfoxyproxy.org/>

CORS bypass mediante HTTP Response tampering.

Ya estamos listos. Abrimos el sitio EvilSite y activamos la redirección configurada en FoxyProxy. Seguidamente, pulsamos sobre el enlace **OWASP Image** del tipo **Javascript XHR**.

Nota

Es posible que Burp Suite esté capturando conexiones a detectportal.firefox.com la primera vez que lo usemos. En el siguiente enlace se explica cómo desactivarlo.

<https://medium.com/@ArtsSEC/burp-suite-desactivar-detectportal-firefox-com-8ec6a0e55417>

En la pestaña **Proxy – Intercept** nos aparecerá la petición realizada que debe ser como la siguiente:

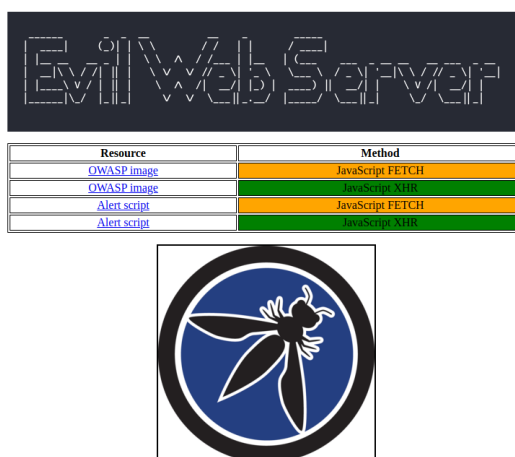
```
GET /img/owasp_bug.png HTTP/1.1
Host: trustedsite.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/89.0.4389.90 Safari/537.36
Accept: */*
Origin: http://evilsite.com
Referer: http://evilsite.com/
```

```
Accept-Encoding: gzip, deflate
Accept-Language: es-ES,es;q=0.9
Connection: close
```

Reenviamos la petición pulsando el botón *Forward* e inmediatamente se presenta la respuesta del servidor con las cabeceras HTTP CORS.

```
HTTP/1.1 200 OK
Date: Mon, 12 Apr 2021 10:00:19 GMT
Server: Apache/2.4.46 (Unix)
Last-Modified: Thu, 08 Apr 2021 09:12:04 GMT
ETag: "9540-5bf7270d90100"
Accept-Ranges: bytes
Content-Length: 38208
Access-Control-Allow-Origin: http://trustedsite.com
Vary: Origin
Connection: close
Content-Type: image/png
```

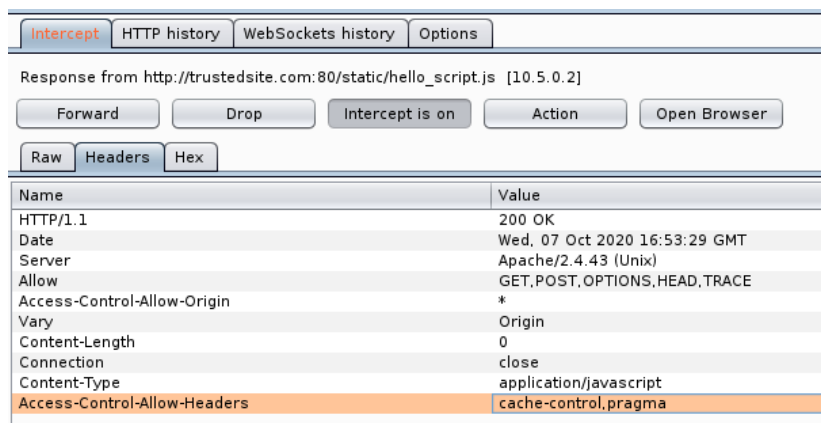
Modificamos la cabecera **Access-Control-Allow-Origin** al valor ***** y enviamos la respuesta al navegador pulsando el botón **Forward**. En este momento la imagen se cargará en la página y el bypass habrá funcionado.



3. Ejercicios propuestos

En el caso práctico se ha analizado el ciclo de peticiones-respuestas HTTP relacionadas con CORS y se ha demostrado cómo es posible realizar un bypass de seguridad. Ahora se proponen una serie de ejercicios para seguir aprendiendo y mejorando estos conocimientos.

1. Realiza el bypass del resto de enlaces del menu de la web EvilSite. En este caso, se realiza una petición *preflighted* por lo que tendrás que añadir la cabecera **Access-Control-Allow-Headers** en la petición HTTP OPTIONS.



2. Estudia la posibilidad de crear un proxy automático para saltar las restricciones CORS. El proxy será el intermediario para filtrar las peticiones y respuestas y realizar los cambios que se hayan configurado en el mismo.

Recurso [ENG]

Proyecto CORS-Anywhere. Esta API permite crear y configurar un proxy CORS automático.

<https://cors-anywhere.herokuapp.com/>

<https://github.com/Rob--W/cors-anywhere/>

3. Estudia las peticiones y respuesta de las demos creadas en el siguiente proyecto y utiliza Burp Suite para modificar las peticiones y obtener diferentes respuestas.

Recurso

Proyecto con numerosas demos sobre el funcionamiento de CORS.

https://digi.ninja/blog/cors_demos.php

4. Bibliografía

Recursos y enlaces utilizados para elaborar este documento.

- <https://infosecwriteups.com/hacking-http-cors-from-inside-out-512cb125c528>
- Authoritative guide to CORS for REST APIs.
<https://www.moesif.com/blog/technical/cors/Authoritative-Guide-to-CORS-Cross-Origin-Resource-Sharing-for-REST-APIs/>
- Brower Security Handbook. <https://code.google.com/archive/p/browsersec/wikis/Main.wiki>
- The Tangled Web: A Guide to Securing Modern Web Applications. <https://lcamtuf.coredump.cx/tangled/>