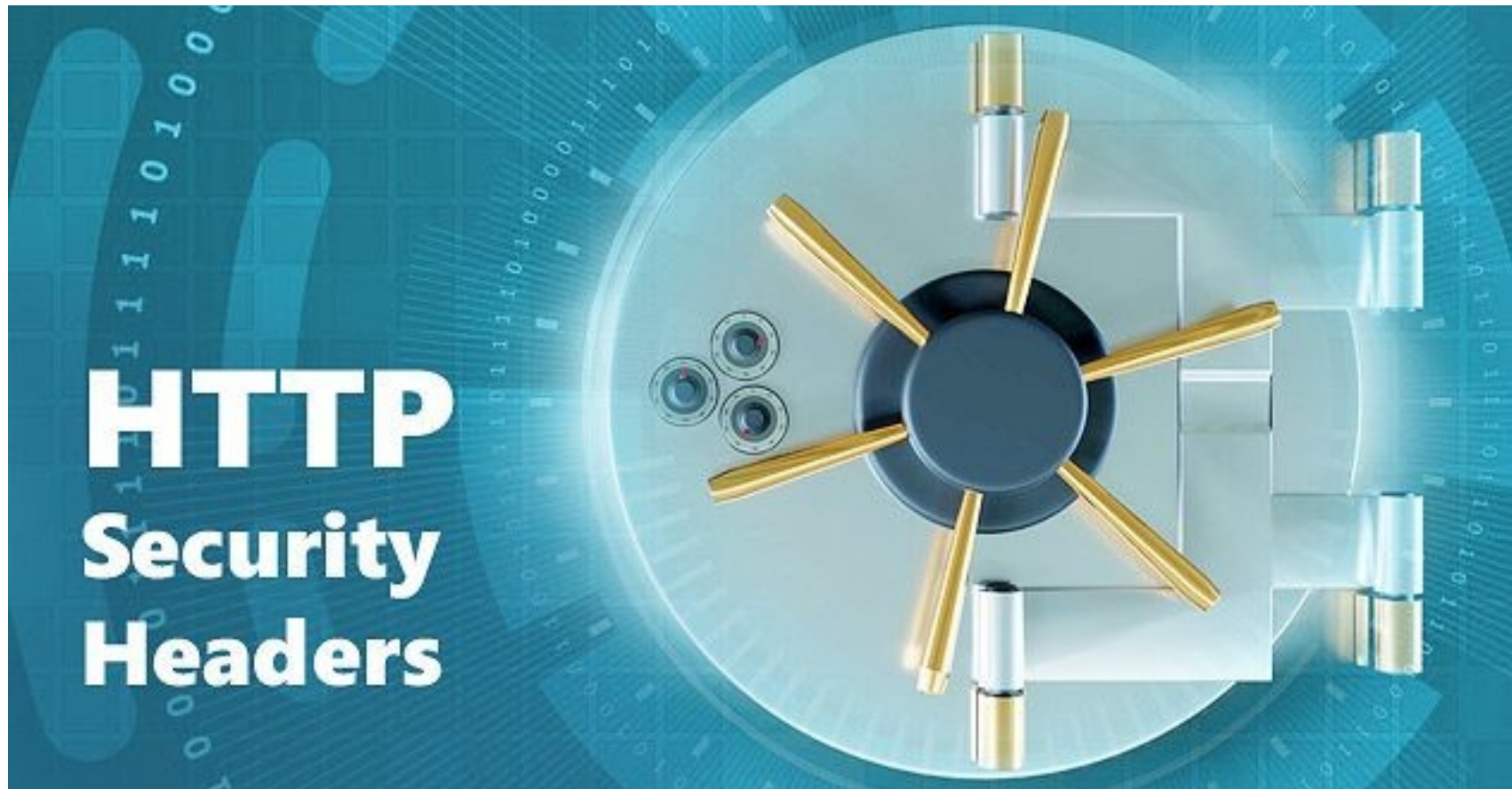


Seguridad en HTTP



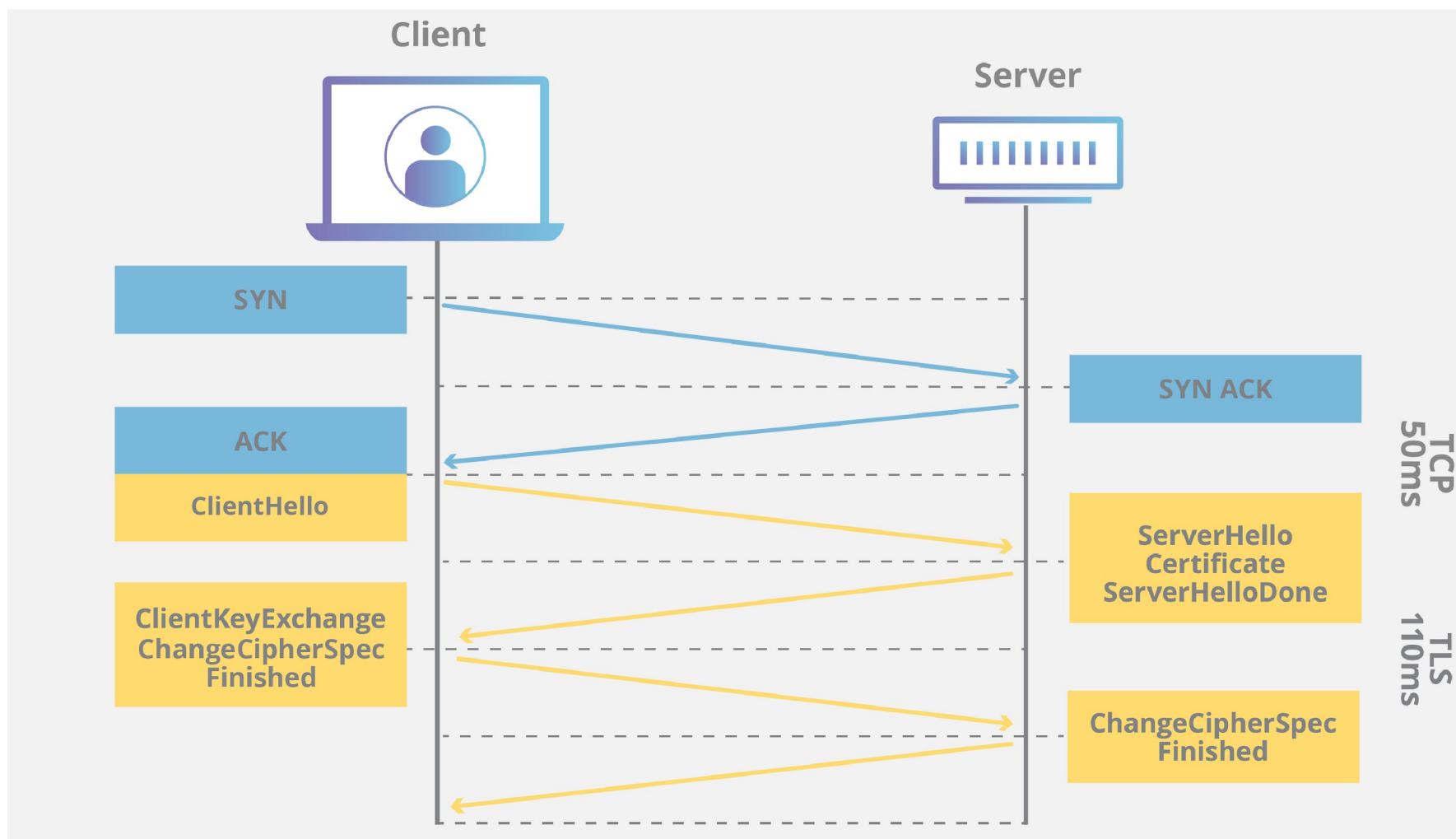
Contenidos

1. Introducción.
2. Cabeceras de seguridad.
 1. Content Security Policy (CSP).
 2. HTTP Strict Transport Security (HSTS).
 3. Uso seguro de HTTP Cookies.
 4. Cabeceras X-Content-Type-Options, X-Frame-Options y X-XSS-Protection.
3. HTTP Authentication.
4. HTTP Access Control (CORS).
 1. Peticiones simples.
 2. Peticiones preflighted o verificadas.
5. Ataques a HTTP.
 1. HTTP Host Header Injection Attack.
 2. HTTP Request/Response Splitting.
 3. HTTP Request/Smuggling.

Introducción

- ▶ Principal iniciativa de seguridad es **HTTPS**.
 - **Cifrado** de las comunicaciones.
 - Certificación de la **identidad** del servidor.
 - **Integridad** de los datos transmitidos.
- ▶ Se emplea **TLS** para negociar el intercambio de claves y verificar los certificados ante una autoridad de certificación (**PKI** – **Public Key Infrastructure**).

Introducción



<https://www.cloudflare.com/es-es/learning/ssl/what-happens-in-a-tls-handshake/>

Cabeceras de seguridad

► Content Security Policy (CSP).

- Permite detectar y mitigar ataques del tipo *Cross Site Scripting* (XSS) y ataques de inyección de datos.
- Para usar CSP, hay que activar la **cabecera *Content-Security-Policy*** (la cabecera *X-Content-Security-Policy* es más antigua y ya no se usa).
- Alternativamente, se puede emplear el elemento ***<meta>*** para configurarlo.

Content-Security-Policy: default-src 'self'

```
<meta http-equiv="Content-Security-Policy" content="default-src 'self'; img-src https://*; child-src 'none';">
```

Cabeceras de seguridad

► Content Security Policy (CSP).

- Formato de las directivas: ***nombre-directiva valores***
 - ***default-src***: acción por defecto para todos los contenidos.
 - ***script-src***: especifica el origen de los scripts que pueden ejecutarse. El valor '***unsafe-inline***' previene la ejecución de Javascript inline (habitual en XSS).
 - ***img-src***: especifica el origen de las imágenes que pueden insertarse.
 - ***media-src***: lo mismo para audio y vídeo.
 - ***frame-ancestors***: especifica quién puede incluir la página en un frame (**ver cabecera X-Frame-Options**).

Cabeceras de seguridad

► Content Security Policy (CSP).

– Ejemplo de directivas:

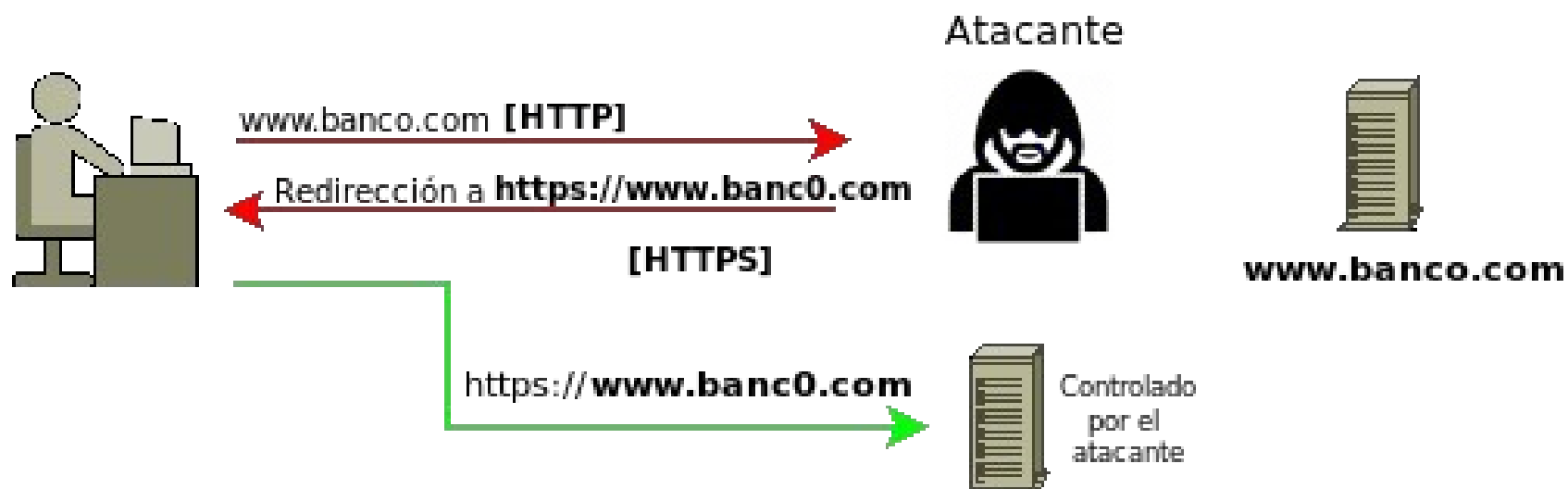
- 1) El contenido solo puede provenir del propio sitio web.
- 2) Permitir todo el contenido proveniente del propio servidor y de un dominio de confianza (incluyendo sus subdominios).
- 3) Permitir solo contenido proveniente del propio servidor, scripts provenientes de un dominio de confianza (sin subdominios) e imágenes de cualquier sitio.
- 4) Forzar la conexión por HTTPS para todos los recursos.

- 1) `Content-Security-Policy: default-src 'self'`
- 2) `Content-Security-Policy: default-src 'self' *.midominio.com`
- 3) `Content-Security-Policy: default-src 'self'; script-src scripts.midominio.com; img-src *`
- 4) `Content-Security-Policy: default-src https://www.midominio.com`

Cabeceras de seguridad

► HTTP Strict Transport Security (HSTS):

- La cabecera **Strict-Transport-Security** establece que el sitio web solo puede ser accedido mediante HTTPS.
- Si un sitio web permite la conexión a HTTP y redirige a HTTPS, esto crea la oportunidad de un ataque MitM. La redirección podría ser explotada llevando a los visitantes a un sitio malicioso (**SSL Stripping Attack**).



<https://cheapsslsecurity.com/blog/what-is-hsts-a-brief-overview-of-hsts-technology/>

Cabeceras de seguridad

► HTTP Strict Transport Security (HSTS):

- Cuando el navegador se conecta a un servidor con HSTS recordará que las peticiones futuras deben hacerse por HTTPS.
- El ataque visto anteriormente solo podría ocurrir durante la primera conexión al sitio web. Para evitarlo existen listas precargadas como la mantenida por Google (<https://hstspreload.org/>).

```
HTTP/2 200 OK
...
server: ATS/9.1.4
strict-transport-security: max-age=106384710; includeSubDomains;
...
```

Cabeceras de seguridad

► Uso seguro de HTTP cookies.

- HTTP es un protocolo sin estado, no distingue conexiones ni sesiones.
- Para mantener sesiones de usuario se introdujeron las **cookies** que se gestionan mediante cabeceras HTTP.
 - **Set-Cookie**. Cabecera enviada por el servidor para establecer el valor de una cookie la primera vez que se inicia la conexión.
 - El navegador la almacena y en peticiones sucesivas al mismo servidor es enviada en la cabecera **Cookie**.



Cabeceras de seguridad

► Uso seguro de HTTP cookies.

- En la cabecera **Set-Cookie** se pueden incluir una serie de atributos opcionales.
 - **Fecha de expiración (*Expires*)**. Indica cuándo caduca la *cookie*. En el momento que ocurre, el cliente deja de enviarla al servidor. Dos tipos:
 - **De sesión**: no incluyen *Expires* o *Max-Age* y se eliminan cuando se cierra el cliente.
 - **Permanentes**: Se eliminan cuando expira la fecha indicada por *Expires* o tras el periodo de tiempo indicado por *Max-Age*.
 - Un dominio (***Domain***). Especifica a qué dominios y subdominios el cliente debe enviar la *cookie*.
 - Un ***Path***. Ruta que debe existir en la URL para enviar la *cookie*.
 - Atributos de seguridad: ***SameSite***, ***HttpOnly***, ***Secure***.

Set-Cookie: <cookie>=<valor>; Expires=<fecha>; Domain=<dominio>; Path=<path>; atributos de seguridad

Cabeceras de seguridad

► Uso seguro de HTTP cookies.

- Debe tenerse en cuenta que toda la información almacenada en una *cookie* es visible y modificable por el usuario.
- Dependiendo de la aplicación, puede ser deseable usar identificadores opacos del tipo **JSON Web Token**.
- Usar los atributos ***Path*** y ***Domain*** para restringir dónde se enviará la cookie.
- Usar el atributo ***HttpOnly*** para prevenir el acceso a la *cookie* vía Javascript (Ej. *document.cookie*).
- Usar el atributo ***Secure*** para enviar la *cookie* solo por conexiones HTTPS.

```
Set-Cookie: session=a4e3d266f75543dd; Secure  
Set-Cookie: session=a4e3d266f75543dd; HttpOnly
```

Cabeceras de seguridad

► Uso seguro de HTTP cookies.

- El navegador incluía las *cookies* de un sitio independientemente de que la solicitud se haya hecho desde otro dominio distinto (**ataque CSRF**).
- El atributo **SameSite** permite controlar este comportamiento con tres valores posibles:
 - **None**. No hay control sobre el origen de la petición y siempre se incluyen las *cookies* asociadas al destino.
 - **Lax**. Se envían las *cookies* en **peticiones cross-site** si se cumplen dos condiciones: que la petición sea GET y que se inicie por una acción de usuario como hacer clic en un enlace.
 - **Strict**. Las *cookies* no se envían en **peticiones cross-site**. Valor **recomendado para cookies de sesión autenticadas**.

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie/SameSite>

Cabeceras de seguridad

► Uso seguro de HTTP cookies.

- Una petición **cross-site** es aquella en la que el dominio, el protocolo o el puerto desde el que se hace la petición es distinto al del destino de la solicitud.

Origen	Destino	¿Cross-site?
https://www.example.com	https://backend.example.com	No. Es SameSite. En este caso se enviará la cookie.
https://www.example.com	https://backend.foo.com	Sí. TLD + 1 no coincide (example.com y foo.com).
http://www.example.com	https://www.example.com	Sí. Los Protocolos no coinciden (http y https).
https://www.example.com	https://www.example.com:8080	Sí. Puertos no coinciden.

https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/06-Session_Management_Testing/02-Testing_for_Cookies_Attributes

Cabeceras de seguridad

► Cabecera X-Content-Type-Options.

- Es una cabecera **HTTP Response** usada por el servidor para indicar que el tipo MIME indicado en la cabecera *Content-Type* no debe cambiarse por el cliente.
- Para evitar **ataques *MIME Type Sniffing* o *Content Sniffing*** [1].
 - Mecanismo usado por los navegadores para averiguar el tipo MIME del recurso inspeccionando la secuencia de bytes del contenido.
 - Empezó a realizarse ante la falta de especificación del tipo MIME en las respuestas del servidor.
 - En caso de duda entre lo indicado por la cabecera *Content-Type* (si estaba presente) y lo determinado por MIME Sniffing se escoge este último (deshabilitado con Firefox 72).

```
X-Content-Type-Options: nosniff
```


Cabeceras de seguridad

► Cabecera X-Frame-Options.

- Es una cabecera usada para indicar si se permite al navegador renderizar una página en un **<frame>**, **<iframe>**, **<embed>** u **<object>**.
- Es un mecanismo para evitar **ataques de click-jacking** [1] [2].
- Hay dos posibles valores:
 - **DENY**: la página no puede ser mostrada en un marco.
 - **SAMEORIGIN**: la página solo puede mostrarse en un marco del mismo origen.
- La directiva **frame-ancestors** de CSP deja obsoleta esta cabecera por lo que conviene emplear CSP en su lugar.

```
X-Frame-Options: DENY  
X-Frame-Options: SAMEORIGIN
```

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/frame-ancestors>

Cabeceras de seguridad

► Cabecera X-XSS-Protection.

- Es una cabecera **HTTP Response** disponible en navegadores antiguos. **Puede crear vulnerabilidades XSS en sitios web seguros.**
- Los navegadores modernos no lo implementan ya que es suficiente con una buena implementación de *Content-Security-Policy* que deshabilite la ejecución de Javascript inline (*'unsafe-inline'*).
- Valores posibles:
 - **0**: indica al navegador que desactive el filtro XSS.
 - **1**: indica al navegador que active el filtro XSS. Si se detecta un ataque, el navegador desactiva las partes no seguras.
 - **1; mode=block**: activa el filtro, pero si se detecta un ataque la página no se renderiza.
 - **1; report=<reporting-uri>**: activa el filtro y si se detecta un ataque informará de

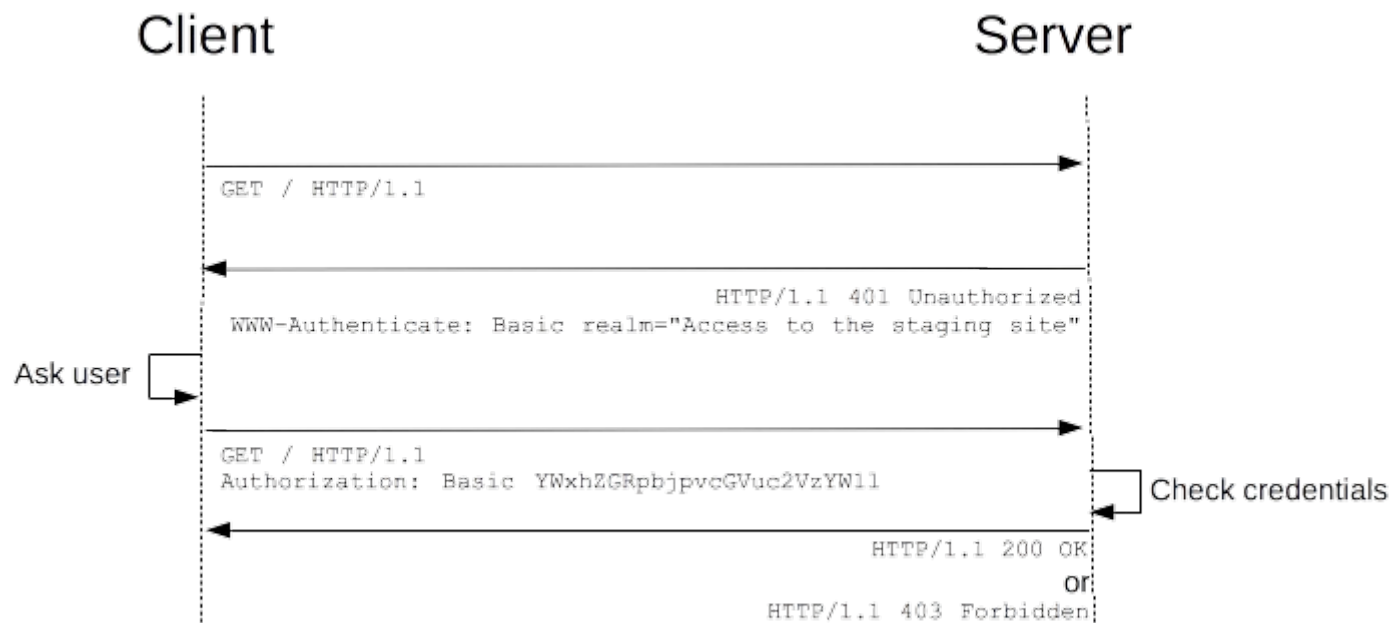
```
X-XSS-Protection: 0
X-XSS-Protection: 1
X-XSS-Protection: 1; mode=block
X-XSS-Protection: 1; report=<reporting-uri>
```

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-XSS-Protection>

HTTP Authentication

► Introducción al framework.

- Definido en el RFC 7235, ofrece un mecanismo de desafío para autenticar a los clientes.
 - El servidor responde con el código 401 (*unauthorized*) y ofrece información sobre cómo realizar el desafío (**WWW-Authenticate**).
 - El cliente debe responder con una cabecera **Authorization** con las credenciales para acceder al recurso o sitio web.
 - El servidor comprueba la validez de los datos recibidos.



HTTP Authentication

- ▶ Esquemas de autenticación.
 - **Basic**. Definido en el RFC 7617. Usa credenciales codificadas en base64. Inseguro salvo que se use con HTTPS.
 - **Bearer**. Definido en el RFC 6750. Se emplea un token para acceder a recursos protegidos por OAuth 2.0.
 - **Digest**. Definido en el RFC 7616. Se emplea un algoritmo de resumen para el desafío.
 - **HOBA** (*HTTP Origin-Bound Authentication*). Definido en el RFC 7486. Basado en firma digital.
 - **Mutual**. Definido en el RFC 8120.
 - **AWS4-HMAC-SHA256**. Esquema de autenticación de Amazon AWS.

HTTP Authentication

► Esquema de autenticación *Basic*.

- Se utilizan credenciales usuario/contraseña que se transmiten en claro por la red (codificadas en base64) por lo que no es un esquema seguro. Se debe emplear en conjunción con HTTPS/TLS.
- **Restricción de acceso con Apache.**
 - Para proteger un directorio, se requiere un fichero *.htaccess* y *.htpasswd*.

```
AuthType Basic
AuthName "Access to the staging site"
AuthUserFile /path/to/.htpasswd
Require valid-user
```

```
aladdin:$apr1$ZjTqBB3f$IF9gdYAGIMrs2fuINjHsz.
user2:$apr1$O04r.y2H$/vEkesPhVInBBYJUkXitA/
```

HTTP Authentication

► Esquema de autenticación *Basic*.

- Se utilizan credenciales usuario/contraseña que se transmiten en claro por la red (codificadas en base64) por lo que no es un esquema seguro. Se debe emplear en conjunción con HTTPS/TLS.
- **Restricción de acceso con nginx.**
 - Se indica la localización del recurso a proteger y la directiva *auth_basic*. La directiva *auth_basic_user_file* apunta al fichero que contiene las credenciales.

```
location /status {  
    auth_basic "Access to the staging site";  
    auth_basic_user_file /etc/apache2/.htpasswd;  
}
```

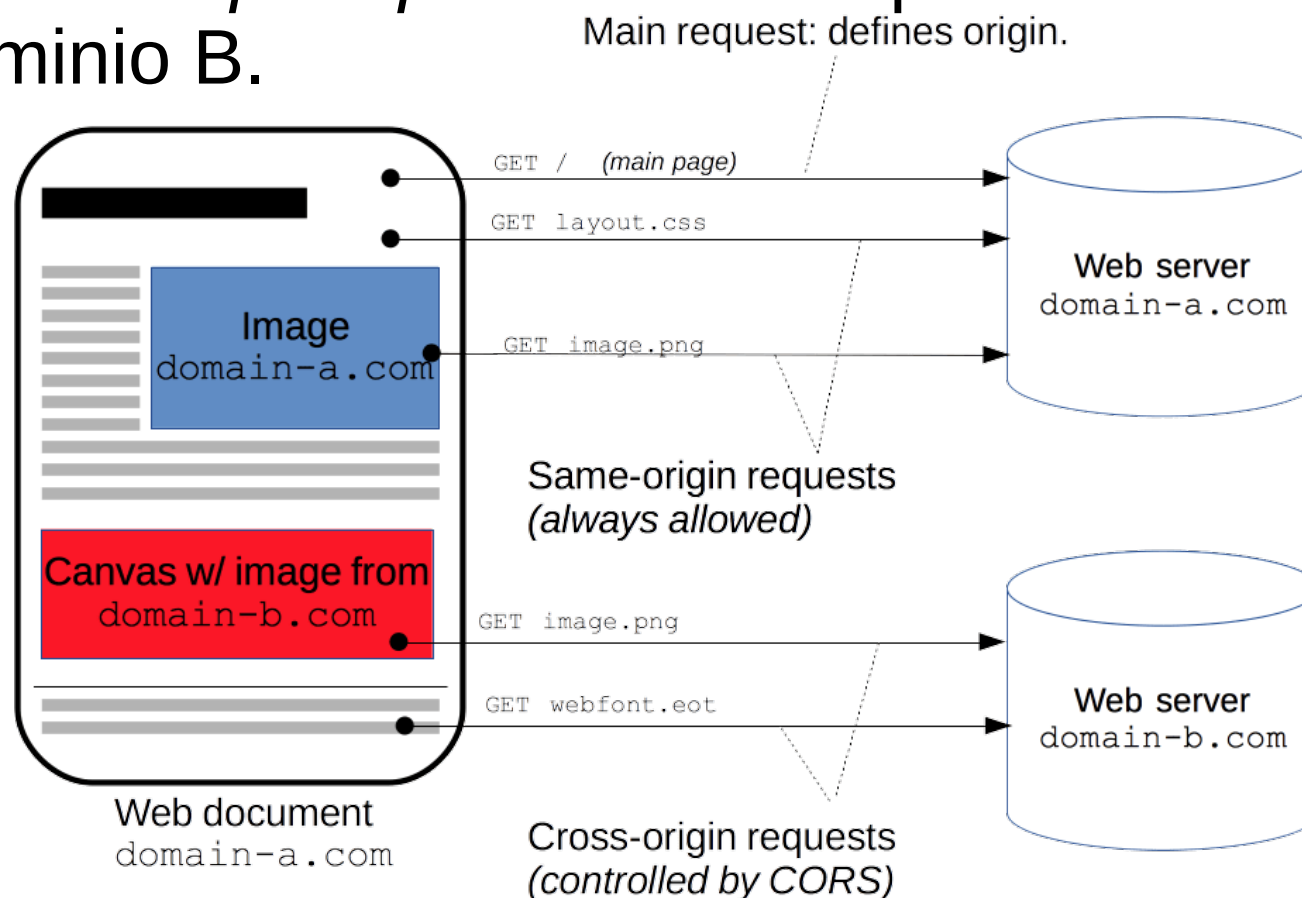
HTTP Access Control (CORS)

► **CORS** (***Cross-Origin Resource Sharing***).

- Es un mecanismo que permite a un servidor indicar otro origen (dominio, esquema, puerto) distinto al suyo, desde el cual el navegador puede obtener recursos.
- Se basa en otro mecanismo por el cual los navegadores hacen una petición previa ("*preflight*") al servidor que almacena el recurso *cross-origin* para comprobar si se permite dicha petición.
- En esa petición *preflight*, el navegador envía cabeceras que indican el método HTTP y las cabeceras que se emplearán en la petición.
- Por motivos de seguridad, los navegadores restringen las peticiones *cross-origin* iniciadas por scripts (***same-origin policy***) a menos que se incluyan las cabeceras CORS apropiadas.

HTTP Access Control (CORS)

- ▶ Ejemplo. El frontend en Javascript del dominio A usa *XMLHttpRequest* o *Fetch* para solicitar datos al dominio B.



HTTP Access Control (CORS)

- ▶ **Elementos que realizan peticiones CORS.**
 - Invocaciones de scripts *XMLHttpRequest* o el *API Fetch*.
 - Fuentes para la web a través de dominios mediante CSS y *@font-face*.
 - Texturas WebGL.
 - Imágenes y vídeo dibujados en un canvas mediante *drawImage()*.
 - Formas CSS desde imágenes.

HTTP Access Control (CORS)

► Posibles escenarios: Peticiones simples

- No realizan el *preflight*. Cumplen todos estos requisitos:
 - El método de la petición es **GET, POST** o **HEAD**.
 - Además de las cabeceras automáticas establecidas por el navegador (*Connection, User-Agent, Origin, ...*), las únicas cabeceras permitidas son: **Accept, Accept-Language, Content-Language, Content-Type** y **Range**.
 - Los únicos valores permitidos para **Content-Type**: ***application/x-www-form-urlencoded, multipart/form-data, text/plain***.
 - Otras condiciones que no vamos a mencionar.

HTTP Access Control (CORS)

► Posibles escenarios: Peticiones simples

- El navegador envía la cabecera HTTP **Origin** en la solicitud indicando el origen del recurso principal y comprueba la cabecera de respuesta **Access-Control-Allow-Origin**. Posibles valores:
 - *: indica que el recurso puede ser accedido desde cualquier origen.
 - **Origen específico**: restringe desde qué orígenes se puede consumir el recurso.
- El siguiente script alojado en el dominio <https://foo.example> quiere invocar contenido alojado en <https://bar.other>.
- El navegador generará una petición HTTP donde incluirá la cabecera **Origin: https://foo.example**

```
const xhr = new XMLHttpRequest();
const url = 'https://bar.other/resources/public-data/';

xhr.open('GET', url);
xhr.onreadystatechange = someHandler;
xhr.send();
```

HTTP Access Control (CORS)

► Posibles escenarios: Peticiones simples

GET /resources/public-data/ HTTP/1.1

Host: bar.other

User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:71.0) Gecko/20100101 Firefox/71.0

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

Connection: keep-alive

Origin: https://foo.example

HTTP/1.1 200 OK

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

Server: Apache/2

Access-Control-Allow-Origin: *

Keep-Alive: timeout=2, max=100

Connection: Keep-Alive

Transfer-Encoding: chunked

Content-Type: application/xml

[...XML Data...]

HTTP Access Control (CORS)

► Posibles escenarios: Peticiones simples



HTTP Access Control (CORS)

► Posibles escenarios: Peticiones *preflighted*

- Son aquellas peticiones en las que no se cumple alguna de las condiciones establecidas para las peticiones simples, es decir:
 - Se usa un método HTTP distinto a GET, POST o HEAD, como **PUT** o **DELETE**.
 - Se incluye una cabecera que no está permitida o hay cabeceras personalizadas.
 - La cabecera Content-Type no tiene un valor de los permitidos, por ejemplo *application/xml* o *application/json*.
 - Se adjuntan credenciales en la solicitud, por ejemplo, mediante *cookies*.
- El proceso de solicitud es más complejo. El navegador envía primero una petición usando el método **OPTIONS** al recurso en el otro dominio para determinar si la petición es segura.
- En el siguiente ejemplo tenemos un script alojado en **foo.example** que quiere acceder a un recurso en **bar.other**. Dado que el *Content-Type* es *application/xml* y se utiliza una cabecera personalizada (**X-PINGOTHER**), la petición es de tipo *preflighted*.

```
const xhr = new XMLHttpRequest();
xhr.open('POST', 'https://bar.other/resources/post-here/');
xhr.setRequestHeader('X-PINGOTHER', 'pongpong');
xhr.setRequestHeader('Content-Type', 'application/xml');
xhr.onreadystatechange = handler;
xhr.send('<person><name>Arun</name></person>');
```


HTTP Access Control (CORS)

► Posibles escenarios: Peticiones *preflighted*

```
OPTIONS /doc HTTP/1.1
Host: bar.other
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:71.0) Gecko/20100101
Firefox/71.0
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
Connection: keep-alive
Origin: http://foo.example
Access-Control-Request-Method: POST
Access-Control-Request-Headers: X-PINGOTHER, Content-Type

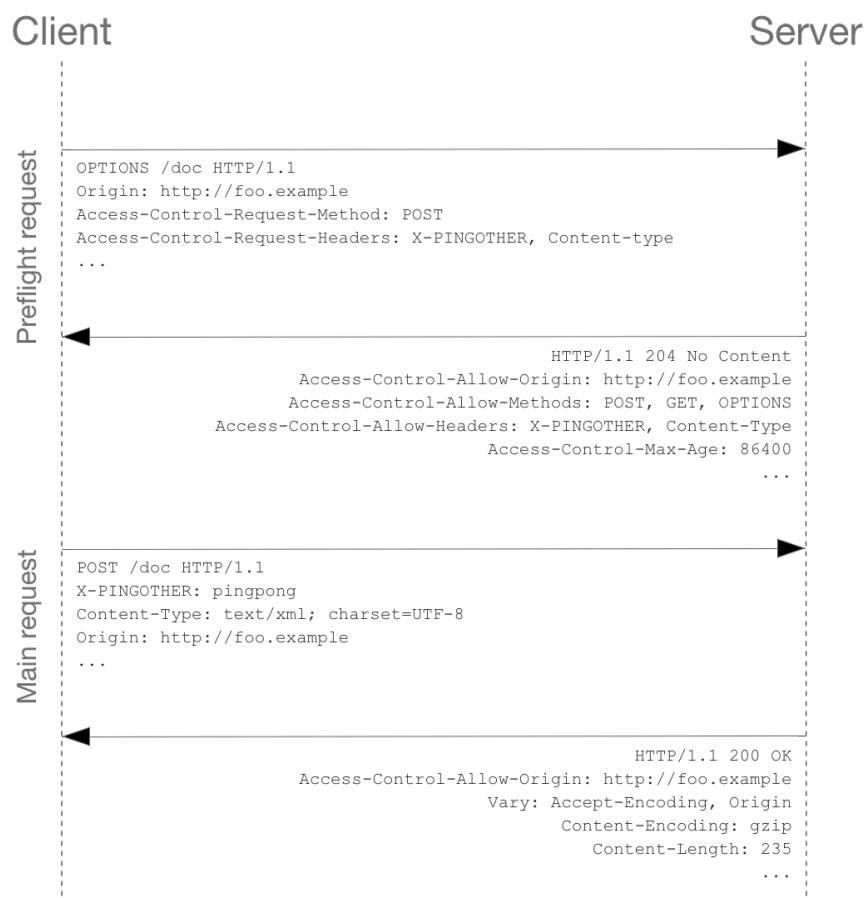
HTTP/1.1 204 No Content
Date: Mon, 01 Dec 2008 01:15:39 GMT
Server: Apache/2
Access-Control-Allow-Origin: https://foo.example
Access-Control-Allow-Methods: POST, GET, OPTIONS
Access-Control-Allow-Headers: X-PINGOTHER, Content-Type
Access-Control-Max-Age: 86400
Vary: Accept-Encoding, Origin
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
```

HTTP Access Control (CORS)

► Posibles *preflighted*

escenarios:

Peticiones



HTTP Access Control (CORS)

► Posibles escenarios: Peticiones *preflighted*

```
POST /doc HTTP/1.1
Host: bar.other
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:71.0) Gecko/20100101 Firefox/71.0
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
Connection: keep-alive
X-PINGOTHER: pongpong
Content-Type: text/xml; charset=UTF-8
Referer: https://foo.example/examples/preflightInvocation.html
Content-Length: 55
Origin: https://foo.example
Pragma: no-cache
Cache-Control: no-cache

<person><name>Arun</name></person>

HTTP/1.1 200 OK
Date: Mon, 01 Dec 2008 01:15:40 GMT
Server: Apache/2
Access-Control-Allow-Origin: https://foo.example
Vary: Accept-Encoding, Origin
Content-Encoding: gzip
Content-Length: 235
Keep-Alive: timeout=2, max=99
Connection: Keep-Alive
Content-Type: text/plain

[Some XML payload]
```

HTTP Access Control (CORS)

- **Posibles escenarios: Peticiones con credenciales**
 - Emplean **HTTP Cookies** y **HTTP Authentication**.
 - Por defecto, no se emplean credenciales, hay que establecer una *flag* en la petición.

```
const invocation = new XMLHttpRequest();
const url = 'http://bar.other/resources/credentialed-content/';

function callOtherDomain() {
  if (invocation) {
    invocation.open('GET', url, true);
    invocation.withCredentials = true;
    invocation.onreadystatechange = handler;
    invocation.send();
  }
}
```

HTTP Access Control (CORS)

► Posibles escenarios: Peticiones con credenciales

Client

Server

```
GET /doc HTTP/1.1  
Origin: foo.example  
Cookie: pageAccess=2
```

```
HTTP/1.1 200 OK  
Access-Control-Allow-Origin: http://foo.example  
Access-Control-Allow-Credentials: true
```

HTTP Access Control (CORS)

► Posibles escenarios: Peticiones con credenciales

```
GET /resources/credentialed-content/ HTTP/1.1
Host: bar.other
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:71.0) Gecko/20100101 Firefox/71.0
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
Connection: keep-alive
Referer: http://foo.example/examples/credential.html
Origin: http://foo.example
Cookie: pageAccess=2

HTTP/1.1 200 OK
Date: Mon, 01 Dec 2008 01:34:52 GMT
Server: Apache/2
Access-Control-Allow-Origin: https://foo.example
Access-Control-Allow-Credentials: true
Cache-Control: no-cache
Pragma: no-cache
Set-Cookie: pageAccess=3; 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

[text/plain payload]
```

Ataques a HTTP

► HTTP Host Header Injection Attack.

- La cabecera **Host** es una cabecera de petición obligatoria desde HTTP/1.1.
- Especifica el dominio al que el cliente dirige la petición (la IP no es suficiente).
 - **Virtual Hosting.** Con esta técnica puede haber más de un dominio alojado en el mismo servidor web. La cabecera Host se utiliza para saber a cuál de ellos va dirigida la petición.
 - **Uso de intermediarios.** La dirección IP puede ser un proxy inverso o un balanceador de carga, por ejemplo, el dominio está protegido por un CDN.

Ataques a HTTP

► HTTP Host Header Injection Attack.

- La cabecera **Host** puede ser manipulada por un atacante por lo que el back-end no debe confiar en su valor.
- Posibles ataques:
 - **Password Reset Poisoning.** El atacante puede manipular la cabecera para generar un link de reseteo de la contraseña que apunte a un servidor controlado por él ([vídeo de lppsec](#)).
 - **Web Cache Poisoning.** El atacante puede manipular un proxy caché (u otro sistema intermediario) para almacenar información falsa o maliciosa que será servida al resto de usuarios.
 - **Authentication Bypass.** Por ejemplo, un panel de administración que solo permita accesos desde la máquina local y que para ello confíe en el valor de la cabecera Host.

ACTIVIDAD: LABORATORIOS DE HTTP HOST HEADER ATTACKS (PORTSWIGGER ACADEMY)

<https://portswigger.net/web-security/host-header>

Ataques a HTTP

► HTTP Request/Response Splitting.

- Se alteran las cabeceras de respuesta HTTP para que manipular cómo el navegador interpreta la petición.
- En HTTP, cada línea en una petición o respuesta acaba con los caracteres `\r` (retorno de carro, *Carriage Return* o CR) y `\n` (nueva línea, *Line Feed* o LF), llamándose al conjunto **CRLF** (`\r\n`).
- Si el atacante puede insertar un carácter CRLF y escribir HTML estará modificando el comportamiento de la petición/respuesta.

```
HTTP/1.1 200 OK\r\n
Server: nginx/1.19.0\r\n
Content-Type: image/png\r\n
...
```

Ataques a HTTP

► HTTP Request/Response Splitting.

- Las cabeceras comúnmente afectadas son ***Set-Cookie*** y ***Location***.
- El siguiente ejemplo hace una redirección 302 a la página de perfil de un usuario, dado este valor.
- Los datos obtenidos de la cabecera no se filtran adecuadamente lo que permite la inyección de código por parte del atacante.

```
HTTP/1.1 302 FOUND
Server: nginx/1.19.0
Location: /username=bob
Content-Type: text/html
...
```

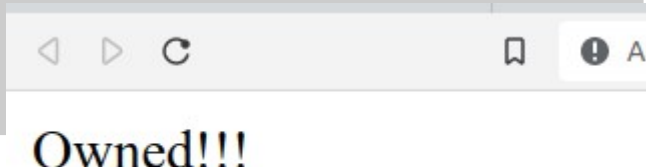
```
String username = request.getParameter(USER_PARAM);
...
response.addHeader("Location", "/username="+username);
...
```

Ataques a HTTP

► HTTP Request/Response Splitting.

- El atacante puede inyectar nuevas cabeceras y código HTML para generar una respuesta maliciosa.
- En negrita se muestra el contenido insertado por el atacante. La cabecera Location, al estar vacía será ignorada y el navegador renderiza el HTML.

```
HTTP/1.1 302 FOUND
Server: nginx/1.19.0
...
Location: 
Content-Type: text/html\r\n\r\n
<html><body>Owned!!!</body></html>
Content-Type: text/html
...
```



Owned!!!

Ataques a HTTP

► HTTP Request/Response Splitting.

- Los ataques que se pueden realizar con esta técnica son variados: ***Cross-User Defacement***, ***Cache Poisoning***, ***Cross-Site Scripting*** (XSS), etc.
- Cómo mitigar o evitar el ataque:
 - No usar directamente datos procedentes de entradas de usuario.
 - Codificar siempre los caracteres que procedan de entradas de usuarios para evitar que se interpreten como código.
 - Tener especial cuidado con las cabeceras susceptibles de ser explotadas como *Location* o *Set-Cookie*.

https://owasp.org/www-community/attacks/HTTP_Response_Splitting

Ataques a HTTP

► HTTP Request Smuggling.

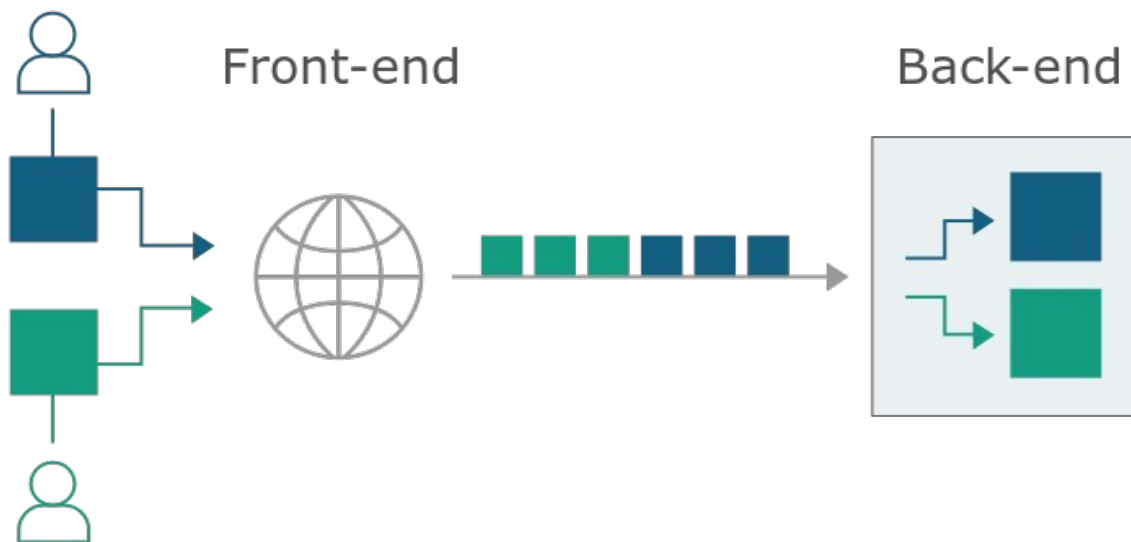
- Este ataque fue propuesto por primera vez en 2005 por Chaim Linhart, Amit Klein, Ronen Heled y Steve Orrin y publicado por [WatchFire](#).
- [James Kettle](#) (@albinowax), Director de Investigación de PortSwigger, presentó en 2019 en BlackHat y DEF CON 27 '**HTTP Desync Attacks: Request Smuggling Reborn**', ofreciendo un nuevo enfoque para llevar acabo este tipo de ataques.
- Nuevamente en 2022, James Kettle, presentó un nuevo ataque en BlackHat USA 2022 y DEF CON 30, '**Browser-Powered Desync Attacks: A new frontier in HTTP Request Smuggling**', donde se comprometían objetivos como Apache, Akamai o Amazon.

<https://portswigger.net/web-security/request-smuggling>

Ataques a HTTP

► HTTP Request Smuggling.

- Entre el cliente y el destino final pueden existir varios servidores HTTP (WAF, proxys inversos, balanceadores de carga, ...). A los servidores intermedios los llamaremos **servidores front-end**, y al servidor final, **servidor back-end**.
- Los servidores front-end reutilizan la misma conexión de red para enviar varias peticiones de distintos clientes al servidor back-end.



Ataques a HTTP

► HTTP Request Smuggling.

- HTTP/1.1 usa dos mecanismos distintos para delimitar una petición de otra.
 - Usando la cabecera de petición **Content-Length**, que indica el tamaño del cuerpo del mensaje en bytes.

```
POST /userinfo.php HTTP/1.1
Host: testphp.vulnweb.com
User-Agent: Mozilla/5.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 20

uname=test&pass=test
```


Ataques a HTTP

► HTTP Request Smuggling.

- HTTP/1.1 usa dos mecanismos distintos para delimitar una petición de otra.
 - Usando la cabecera de petición ***Transfer-Encoding***, que indica que el cuerpo puede contener uno o más ***chunks*** (pedazos de datos). Cada ***chunk*** consta de:
 - una primera línea que indica en hexadecimal el tamaño de los datos a enviar,
 - los propios datos a enviar y,
 - una línea con un 0 indicando el final de ese *chunk*.

```
POST /userinfo.php HTTP/1.1
Host: testphp.vulnweb.com
User-Agent: Mozilla/5.0
Content-Type: application/x-www-form-urlencoded
Transfer-Encoding: chunked
```

```
14
uname=test&pass=test
0
```

Ataques a HTTP

► HTTP Request Smuggling.

- Para realizar el ataque se necesita que:
 - Exista al menos un servidor front-end entre el cliente y el servidor back-end.
 - El atacante envíe ambas cabeceras al mismo tiempo.
 - El servidor front-end y el back-end procesen las cabeceras de forma diferente.
- Existen diferentes situaciones según cómo se comporten los servidores front-end y back-end:
 - El servidor front-end usa la cabecera **Content-Length** y el servidor back-end **Transfer-Encoding**.
 - El servidor front-end usa la cabecera **Transfer-Encoding** y el servidor back-end **Content-Length**.

Ataques a HTTP

► HTTP Request Smuggling.

- El servidor front-end usa la cabecera Content-Length y el servidor back-end Transfer-Encoding.
 - El servidor front-end procesa la cabecera Content-Length y determina que el cuerpo del mensaje, al tener 52 bytes, llega hasta el final de la línea 'Host: testphp.vulnweb.com'.
 - Cuando esta misma petición llega al servidor back-end procesa la cabecera Transfer-Encoding. Al encontrarse un solo chunk de tamaño 0, considera que aquí termina la petición e interpreta los datos siguientes como parte de la segunda petición.

```
POST /userinfo.php HTTP/1.1
Host: testphp.vulnweb.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 52
Transfer-Encoding: chunked
```

```
0
```

```
GET /admin HTTP/1.1
Host: testphp.vulnweb.com
CabeceraX: whatever
```

Ataques a HTTP

► HTTP Request Smuggling.

- El servidor front-end usa la cabecera Transfer-Encoding y el servidor back-end Content-Length.
 - El servidor front-end, procesa la cabecera Content-Length con el cuerpo del mensaje de solo 3 bytes (El 0 y \r\n). Los siguientes bytes se envían sin llegar a procesarse.
 - El servidor back-end, procesando la cabecera Transfer-Encoding, y viendo un solo chunk de 0 bytes, considerará el resto del cuerpo como una nueva petición.

```
POST /userinfo.php HTTP/1.1
Host: testphp.vulnweb.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 3
Transfer-Encoding: chunked
```

```
0
```

```
GET /admin HTTP/1.1
Host: testphp.vulnweb.com
CabeceraX: whatever
```

FIN