Protocolo HTTP

Generalidades del protocolo HTTP y HTTPS

HTTP, de sus siglas en inglés: "*Hypertext Transfer Protocol*", es el nombre de un protocolo el cual nos permite realizar una petición de datos y recursos, como pueden ser documentos HTML. Es la base de cualquier intercambio de datos en la Web, y un protocolo de estructura cliente-servidor, esto quiere decir que una petición de datos es iniciada, por el elemento que recibirá los datos (el cliente), normalmente un navegador Web. Así una página web completa, resulta de la unión de distintos sub-documentos recibidos, como por ejemplo: un documento que especifique el estilo de maquetación de la página web (CSS), el texto, las imágenes, vídeos, scripts …

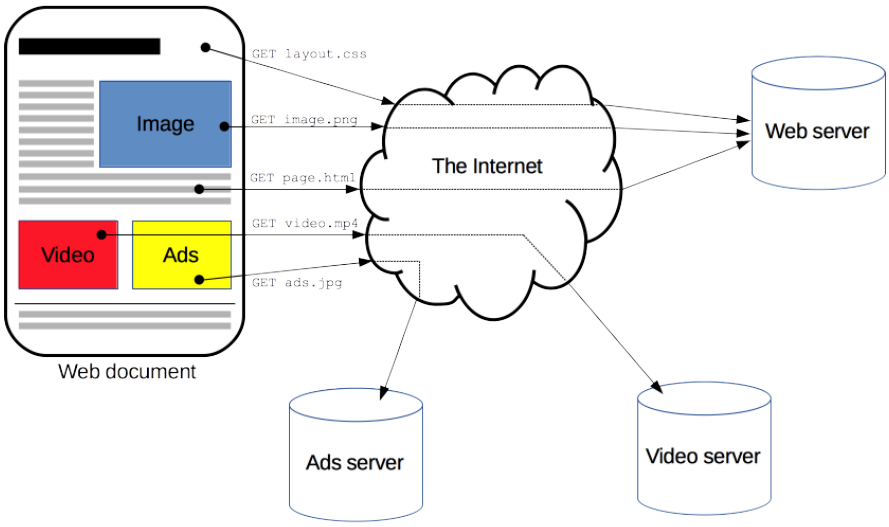


Figure 1: Internet

Clientes y servidores se comunican intercambiando mensajes individuales (en contraposición a las comunicaciones que utilizan flujos continuos de datos). Los

mensajes que envía el cliente, normalmente un navegador Web, se llaman peticiones, y los mensajes enviados por el servidor, se llaman respuestas.

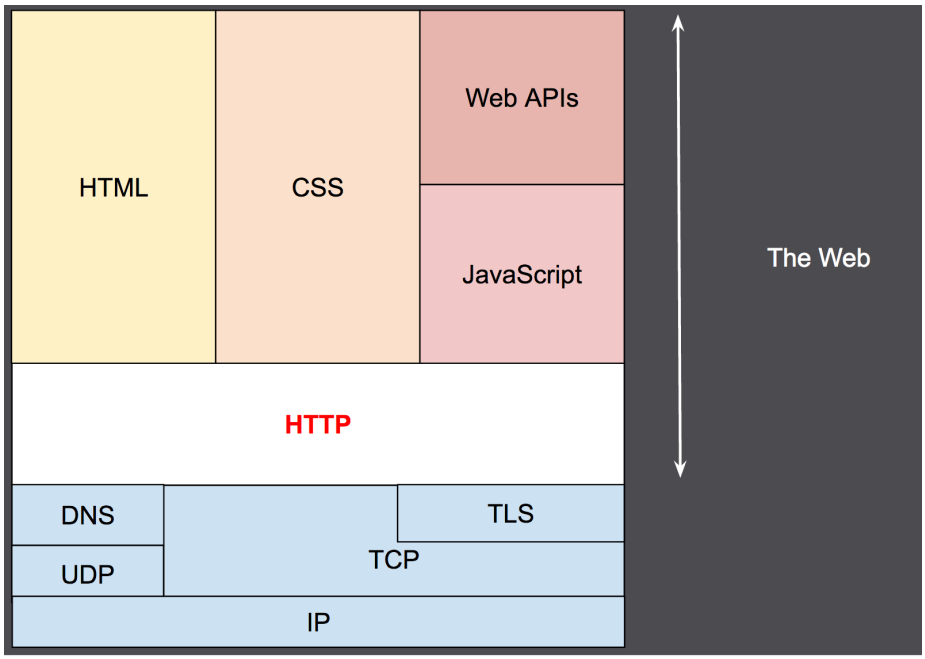


Figure 2: Capas HTTP

Diseñado a principios de la década de 1990, HTTP es un protocolo ampliable, que ha ido evolucionando con el tiempo. Es lo que se conoce como un protocolo de la capa de aplicación, y se transmite sobre el protocolo TCP, o el protocolo encriptado TLS, aunque teóricamente podría usarse cualquier otro protocolo fiable. Gracias a que es un protocolo capaz de ampliarse, se usa no solo para transmitir documentos de hipertexto (HTML), si no que además, se usa para transmitir imágenes o vídeos, o enviar datos o contenido a los servidores, como en el caso de los formularios de datos. HTTP puede incluso ser utilizado para transmitir partes de documentos, y actualizar páginas Web en el acto.

TCP

TCP (*Transmission Control Protocol*) es un importante protocolo de red que permite a dos hosts conectarse e intercambiar flujos de datos. TCP garantiza la entrega de datos y paquetes en el mismo orden en que fueron enviados. Vint Cerf y Bob Kahn, que eran científicos de DARPA en ese entonces, diseñaron TCP en los años 70.

TLS

TLS (*Tansport Layer Security*), anteriormente conocido como Secure Sockets Layer (SSL), es un protocolo utilizado por las aplicaciones para comunicarse de

forma segura a través de una red, evitando la manipulación indebida de correo electrónico, navegación web, mensajería y otros protocolos.

Todos los navegadores modernos soportan el protocolo TLS, requiriendo que el servidor proporcione un certificado digital válido que confirme su identidad para establecer una conexión segura. Es posible que tanto el cliente como el servidor se autentiquen mutuamente, si ambas partes proporcionan sus propios certificados digitales individuales.

Arquitectura de los sistemas basados en HTTP

HTTP es un protocolo basado en el principio de cliente-servidor: las peticiones son enviadas por una entidad: el agente del usuario (o un proxy a petición de uno). La mayoría de las veces el agente del usuario (cliente) es un navegador Web, pero podría ser cualquier otro programa, como por ejemplo un programa-robot, que explore la Web, para adquirir datos de su estructura y contenido para uso de un buscador de Internet.

Cada petición individual se envía a un servidor, el cuál la gestiona y responde. Entre cada petición y respuesta, hay varios intermediarios, normalmente denominados proxies, los cuales realizan distintas funciones, como: gateways o cachés.

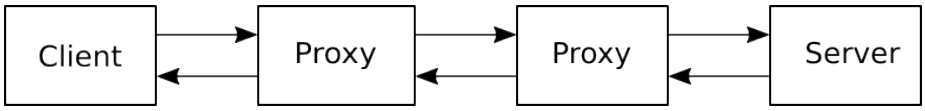


Figure 3: Arquitectura HTTP

En realidad, hay más elementos intermedios, entre un navegador y el servidor que gestiona su petición: hay otros tipos de dispositivos: como routers, modems … Es gracias a la arquitectura en capas de la Web, que estos intermediarios, son transparentes al navegador y al servidor, ya que HTTP se apoya en los protocolos de red y transporte. HTTP es un protocolo de aplicación, y por tanto se apoya sobre los anteriores. Aunque para diagnosticar problemas en redes de comunicación, las capas inferiores son irrelevantes para la definición del protocolo HTTP .

Gateway

La pasarela (en inglés gateway) o puerta de enlace es el dispositivo que actúa de interfaz de conexión entre aparatos o dispositivos, y también posibilita compartir recursos entre dos o más computadoras.

Su propósito es traducir la información del protocolo utilizado en una red inicial, al protocolo usado en la red de destino.

La pasarela es normalmente un equipo informático configurado para dotar a las máquinas de una red de área local (Local Area Network, LAN) conectadas a él de un acceso hacia una red exterior, generalmente realizando para ello operaciones de traducción de direcciones de red (Network Address Translation, NAT). Esta capacidad de traducción de direcciones permite aplicar una técnica llamada "enmascaramiento de IP" (IP Masquerading), usada muy a menudo para dar acceso a Internet a los equipos de una LAN compartiendo una única conexión a Internet, y por tanto, una única dirección IP externa.

La dirección IP de una pasarela a menudo se parece a 192.168.1.1 o 192.168.0.1 y utiliza algunos rangos predefinidos, 127.x.x.x, 10.x.x.x, 172.x.x.x, 192.x.x.x. Puedes averiguar la puerta de enlace de tu router ejecutando el comando ipconfig desde el cmd de Windows, o ejecutando la orden ip route desde la terminal macOS o GNU/Linux (ver más).

Un equipo que haga de puerta de enlace en una red debe tener necesariamente dos tarjetas de red (Network Interface Card, NIC).

La puerta de enlace predeterminada (default gateway) es la ruta predeterminada o ruta por defecto que se le asigna a un equipo y tiene como función enviar cualquier paquete del que no conozca por cuál interfaz enviarlo y no esté definido en las rutas del equipo, enviando el paquete por la ruta predeterminada.

En entornos domésticos, se usan los routers ADSL como puertas de enlace para conectar la red local doméstica con Internet; aunque esta puerta de enlace no conecta dos redes con protocolos diferentes, sí que hace posible conectar dos redes independientes haciendo uso de NAT.

Cliente: el agente del usuario

El agente del usuario, es cualquier herramienta que actúe en representación del usuario. Esta función es realizada en la mayor parte de los casos por un navegador Web. Hay excepciones, como el caso de programas específicamente usados por desarrolladores para desarrollar y depurar sus aplicaciones.

El navegador es siempre el que inicia una comunicación (petición), y el servidor nunca la comienza (hay algunos mecanismos que permiten esto, pero no son muy habituales).

Para poder mostrar una página Web, el navegador envía una petición de documento HTML al servidor. Entonces procesa este documento, y envía más peticiones para solicitar scripts, hojas de estilo (CSS), y otros datos que necesite (normalmente vídeos y/o imágenes). En navegador, une todos estos documentos y datos, y compone el resultado final: la página Web. Los scripts, los ejecuta también el navegador, y también pueden generar más peticiones de datos en el tiempo, y el navegador, gestionará y actualizará la página Web en consecuencia.

Una página Web, es un documento de hipertexto (HTTP), luego habrá partes del texto en la página que puedan ser enlaces (links) que pueden ser activados

(normalmente al hacer click sobre ellos) para hacer una petición de una nueva página Web, permitiendo así dirigir su agente de usuario y navegar por la Web. El navegador, traduce esas direcciones en peticiones de HTTP, e interpretara y procesará las respuestas HTTP, para presentar al usuario la página Web que desea.

El servidor Web

Al otro lado del canal de comunicación, está el servidor, el cual "sirve" los datos que ha pedido el cliente. Un servidor conceptualmente es una única entidad, aunque puede estar formado por varios elementos, que se reparten la carga de peticiones (*load balancing*), u otros programas, que gestionan otros computadores (como caché, bases de datos, servidores de correo electrónico, …), y que generan parte o todo el documento que ha sido pedido.

Un servidor no tiene que ser necesariamente un único equipo físico, aunque si que varios servidores pueden estar funcionando en un único computador. En el estándar HTTP/1.1 y `Host` , pueden incluso compartir la misma dirección de IP.

Proxies

Un servidor proxy es un programa intermedio o una computadora utilizada al navegar a través de diferentes redes de Internet. Facilitan el acceso al contenido en la World Wide Web. Un apoderado intercepta las solicitudes y responde las respuestas; puede enviar las solicitudes, o no (por ejemplo, en el caso de una caché), y puede modificarlo (por ejemplo, cambiar sus encabezados, en el límite entre dos redes).

Un proxy puede estar en el equipo local del usuario o en cualquier lugar entre el equipo del usuario y un servidor de destino en Internet.

Estos pueden ser transparentes, o no (modificando las peticiones que pasan por ellos), y realizan varias funciones:

- caching (la caché puede ser pública o privada, como la caché de un navegador)
- filtrado (como un anti-virus, control parental, …)
- balanceo de carga de peticiones (para permitir a varios servidores responder a la carga total de peticiones que reciben)
- autentificación (para el control al acceso de recursos y datos)
- registro de eventos (para tener un histórico de los eventos que se producen)

Características clave del protocolo HTTP

HTTP es sencillo

Incluso con el incremento de complejidad, que se produjo en el desarrollo de la versión del protocolo HTTP/2, en la que se encapsularon los mensajes, HTTP esta pensado y desarrollado para ser leído y fácilmente interpretado por las personas, haciendo de esta manera más facil la depuración de errores, y reduciendo la curva de aprendizaje para las personan que empieza a trabajar con él.

HTTP es extensible

Presentadas en la versión HTTP/1.0, las cabeceras de HTTP, han hecho que este protocolo sea fácil de ampliar y de experimentar con él. Funcionalidades nuevas pueden desarrollarse, sin más que un cliente y su servidor, comprendan la misma semántica sobre las cabeceras de HTTP.

HTTP es un protocolo con sesiones, pero sin estados

HTTP es un protocolo sin estado, es decir: no guarda ningún dato entre dos peticiones en la misma sesión. Esto plantea la problemática, en caso de que los usuarios requieran interactuar con determinadas páginas Web de forma ordenada y coherente, por ejemplo, para el uso de "cestas de la compra" en páginas que utilizan en comercio electrónico. Pero, mientras HTTP ciertamente es un protocolo sin estado, el uso de HTTP cookies, sí permite guardar datos con respecto a la sesión de comunicación. Usando la capacidad de ampliación del protocolo HTTP, las cookies permiten crear un contexto común para cada sesión de comunicación.

HTTP y conexiones

Una conexión se gestiona al nivel de la capa de trasporte, y por tanto queda fuera del alcance del protocolo HTTP. Aún con este factor, HTTP no necesita que el protocolo que lo sustenta mantenga una conexión continua entre los participantes en la comunicación, solamente necesita que sea un protocolo fiable o que no pierda mensajes (como mínimo, en todo caso, un protocolo que sea capaz de detectar que se ha perdido un mensaje y reporte un error). De los dos protocolos más comunes en Internet, TCP es fiable, mientras que UDP, no lo es. Por lo tanto HTTP, se apoya en el uso del protocolo TPC, que está orientado a conexión, aunque una conexión continua no es necesaria siempre.

¿Qué se puede controlar con HTTP?

La característica del protocolo HTTP de ser ampliable, ha permitido que durante su desarrollo se hayan implementado más funciones de control y funcionalidad sobre la Web: caché o métodos de identificación o autentificación fueron temas que se abordaron pronto en su historia. Al contrario la relajación de la restricción de origen solo se ha abordado en los años de la década de 2010.

Se presenta a continuación una lista con algunos de los elementos que se pueden controlar con el protocolo HTTP:

- **Caché**. El como se almacenan los documentos en la caché, puede ser especificado por HTTP. El servidor puede indicar a los proxies y clientes,

que quiere almacenar y durante cuanto tiempo. Aunque el cliente, también puede indicar a los proxies de caché intermedios que ignoren el documento almacenado.

- **Flexibilidad del requisito de origen x-frame-options**. Para prevenir invasiones de la privacidad de los usuarios, los navegadores Web, solamente permiten a páginas del mismo origen, compartir la información o datos. Esto es una complicación para el servidor, asi que mediante cabeceras HTTP, se puede flexibilizar o relajar esta división entre cliente y servidor
- **Autentificación**. Hay páginas Web, que pueden estar protegidas, de manera que solo los usuarios autorizados puedan acceder. HTTP provee de servicios básicos de autentificación, por ejemplo mediante el uso de cabeceras como: WWW-Authenticate, o estableciendo una sesión especifica mediante el uso de HTTP cookies.
- **Sesiones**. El uso de HTTP cookies permite relacionar peticiones con el estado del servidor. Esto define las sesiones, a pesar de que por definición el protocolo HTTP es un protocolo sin estado. Esto es muy útil no sólo para aplicaciones de comercio electrónico, sino también para cualquier sitio que permita configuración al usuario.

Tarea 1

Crea una página html con el código a continuación. Guárdala y ábrela con el navegador

```
<!DOCTYPE html>
<html>
 <head>
     <title>Wikipedia Insertada</title>
    </head>
    <body>
        <iframe src='https://es.wikipedia.org/wiki/Wikipedia:Portada' width='100%' heig
    </body>
   </html>
```

Ahora, crea otra con el siguiente código:

```
<!DOCTYPE html>
<html>
 <head>
     <title>Wikipedia Insertada</title>
    </head>
    <body>
        <iframe src='https://www.google.es/' width='100%' height='800px'></iframe>
```

```
  </body>
</html>
```

Ahora responde, ¿por qué no se ve la página con el iframe a https://www.google.es/?

Flujo de HTTP

Cuando el cliente quiere comunicarse con el servidor, tanto si es directamente con él, o a través de un proxy intermedio, realiza los siguientes pasos:

1. Abre una conexión TPC: la conexión TPC se usará para hacer una petición, o varias, y recibir la respuesta. El cliente puede abrir una conexión nueva, reusar una existente, o abrir varias a la vez hacia el servidor.
2. Hacer una petición HTTP: Los mensajes HTTP (previos a HTTP/2) son legibles en texto plano. A partir de la versión del protocolo HTTP/2, los mensajes se encapsulan en franjas, haciendo que no sean directamente interpretables, aunque el principio de operación es el mismo.

```
GET / HTTP/1.1
Host: developer.mozilla.org
Accept-Language: es
```

3.- Leer la respuesta enviada por el servidor

```
HTTP/1.1 200 OK
Date: Sat, 09 Oct 2010 14:28:02 GMT
Server: Apache
Last-Modified: Tue, 01 Dec 2009 20:18:22 GMT
ETag: "51142bc1-7449-479b075b2891b"
Accept-Ranges: bytes
Content-Length: 29769
Content-Type: text/html

<!DOCTYPE html... \(here comes the 29769 bytes of the requested web page\)
```

4.- Cierre o reuso de la conexión para futuras peticiones.

Mensajes HTTP

Los mensajes HTTP son la forma en que se intercambian los datos entre un servidor y un cliente.

Hay dos tipos de mensajes: las solicitudes enviadas por el cliente para activar una acción en el servidor y las respuestas, la respuesta del servidor.

Los mensajes HTTP se componen de información textual codificada en ASCII, y se extienden sobre varias líneas. En HTTP / 1.1 y versiones anteriores del protocolo, estos mensajes se enviaron abiertamente a través de la conexión. En HTTP / 2, el mensaje antes legible por humanos ahora se divide en marcos HTTP, proporcionando mejoras de optimización y rendimiento. Los desarrolladores web, o webmasters, raramente elaboran estos mensajes HTTP textuales ellos mismos: un software, un navegador Web, proxy o servidor Web, realiza esta acción. Proporcionan mensajes HTTP a través de archivos de configuración (para proxies o servidores), API (para navegadores) u otras interfaces.

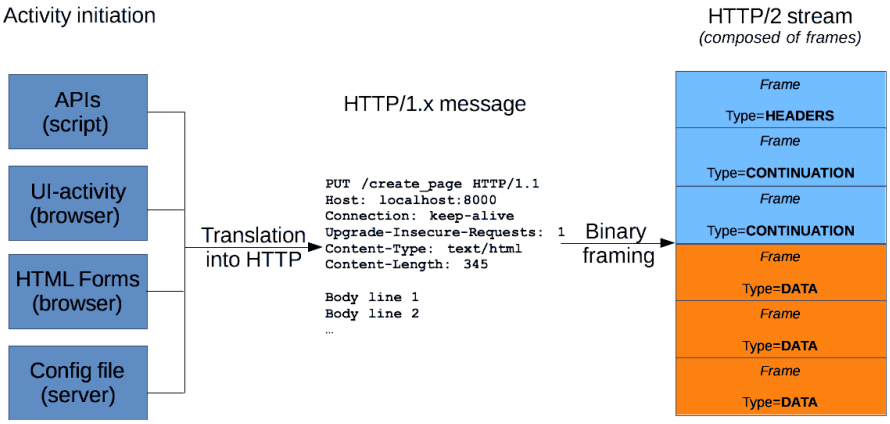


Figure 4: Mensajes HTTP

Las solicitudes HTTP y las respuestas comparten una estructura similar y están compuestas por:

1. Una línea de inicio que describe las solicitudes que se van a implementar, o su estado de éxito o fallo. Esta línea de inicio es siempre una sola línea.
2. Un conjunto opcional de encabezados HTTP que especifica la solicitud o describe el cuerpo incluido en el mensaje.
3. Una línea en blanco indicando que toda la meta-información para la solicitud se ha enviado.
4. Un cuerpo opcional que contiene datos asociados con la solicitud (como el contenido de un formulario HTML) o el documento asociado a una respuesta. La presencia del cuerpo y su tamaño se especifica por la línea de inicio y los encabezados HTTP.

Los encabezados de línea de inicio del mensaje HTTP se conocen colectivamente como la cabecera (*header*) de las solicitudes, mientras que su carga útil (*payload*) se conoce como el cuerpo (*body*).

Si deseas obtener las cabeceras de respuesta desde línea de comandos, puedes usar:

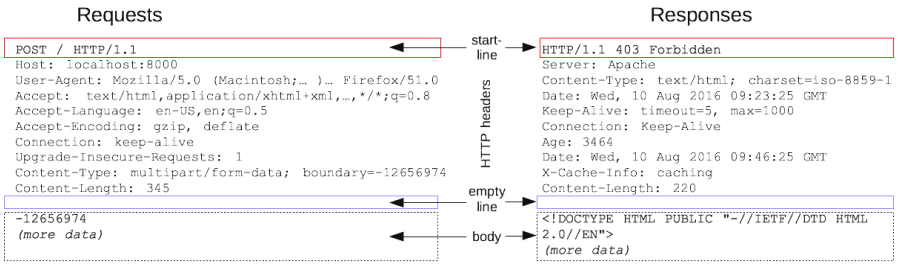


Figure 5: Request - Response

```
curl -I google.es
```

cuya salida será la siguiente:

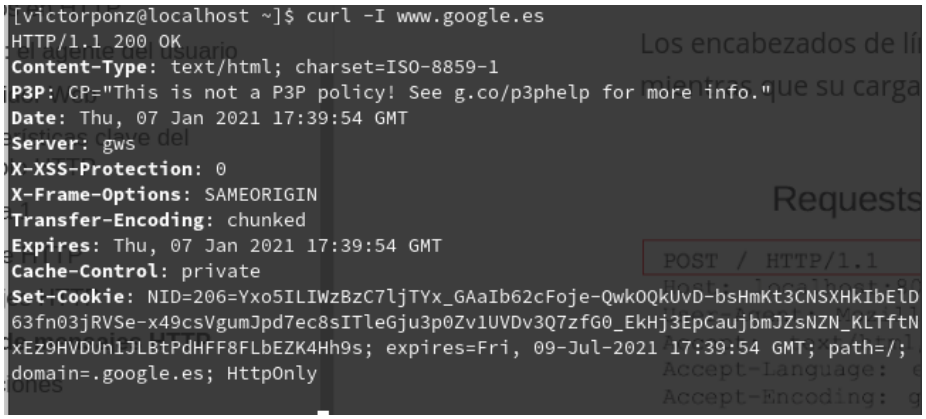


Figure 6: Curl

o bien

```
wget -q --server-response www.google.es
```

cuya salida será la siguiente

Tipos de mensajes HTTP

Existen dos tipos de mensajes HTTP: peticiones y respuestas, cada uno sigue su propio formato.

Figure 7: wget

Peticiones

Un ejemplo de petición HTTP:

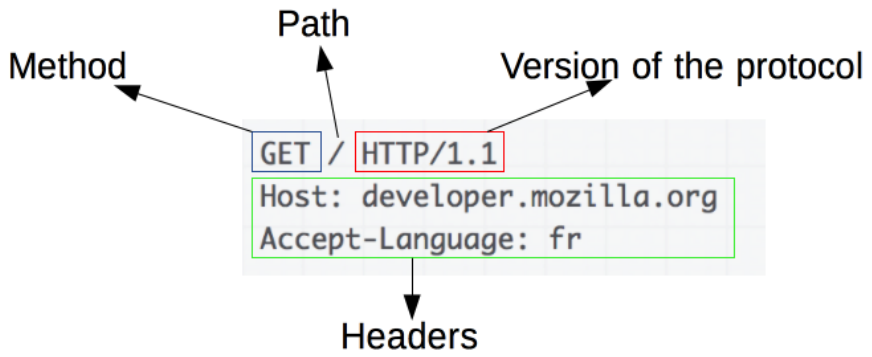


Figure 8: Ejemplo de petición HTTP

Una petición de HTTP, está formado por los siguientes campos:

- **Un método HTTP,** normalmente pueden ser un verbo, como: `GET`, `POST` o un nombre como: `OPTIONS` o `HEAD`, que defina la operación que el cliente quiera realizar. El objetivo de un cliente, suele ser una petición de recursos, usando `GET`, o presentar un valor de un formulario HTML, usando `POST`, aunque en otras ocasiones puede hacer otros tipos de peticiones.

- **La dirección del recurso pedido**; la URL del recurso, sin los elementos obvios por el contexto, como pueden ser: sin el protocolo (http://), el dominio , o el puerto TCP (por defecto, el 80).
- La **versión** del protocolo HTTP.
- **Cabeceras HTTP opcionales**, que pueden aportar información adicional a los servidores.
- **O un cuerpo de mensaje**, en algún método, como puede ser POST, en el cual envía la información para el servidor.

Respuestas

Un ejemplo de repuesta:

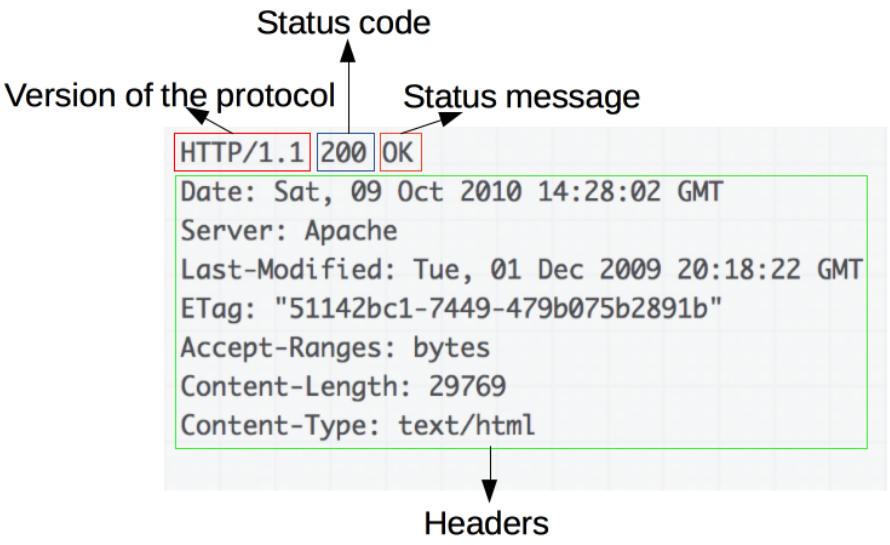


Figure 9: Respuestas

Las respuestas están formadas por los siguentes campos:

- La **versión** del protocolo HTTP que están usando.
- Un **código de estado**, indicando si la petición ha sido exitosa, o no, y debido a que.
- Un **mensaje de estado**, una breve descriptión del código de estado.
- **Cabeceras HTTP**, como las de las peticiones.
- Opcionalmente, **el recurso que se ha pedido**.

Tarea 2

Abre en el Firefox una dirección web.

Activa Firebug (tecla F12)

Selecciona la opción Red

Recarga la página y revisa las cabeceras

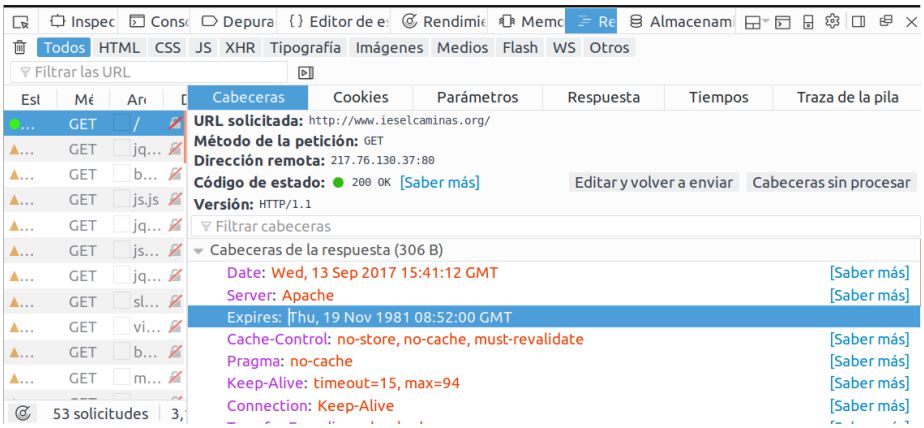


Figure 10: Firebug

Método HTTP

HTTP define 8 métodos (algunas veces referido como "verbos") que indica la acción que desea que se efectúe sobre el recurso identificado. Lo que este recurso representa, es decir, si son datos preexistentes o datos que se generan de forma dinámica, depende de la aplicación del servidor. A menudo, el recurso corresponde a un archivo o la salida de un ejecutable que reside en el servidor. Los métodos principales son:

- **GET** Sirve para obtener la entidad correspondiente a la URI especificado en la línea de petición. Normalmente el servidor traducirá el camino del URI a un nombre de fichero o programa:

  1. En el primer caso, el cuerpo de la entidad será el contenido del fichero.
  2. En el segundo caso, el servidor ejecutará el programa y la entidad será el resultado que genere.

  Por seguridad no se tendría que utilizar por aplicaciones que causen efectos puesto que transmite información a través de la URI agregando parámetros a la URL.

Ejemplo:

`GET /images/logo.png HTTP/1.1` obtiene un recurso llamado logo.png

Ejemplo con parámetros:

`GET /index.php?page=main&ln=es`

- **HEAD**

  Pide una respuesta idéntica a la que correspondería a una petición GET, pero en la petición no se devuelve el cuerpo. Esto es útil para poder recuperar los metadatos de los encabezados de respuesta, sin tener que transportar todo el contenido.

- **POST** Sirve para enviar una entidad que el servidor tiene que incorporar en el recurso identificado por el URI de la línea de petición. La semántica de este método depende del tipo de recurso. Por ejemplo, se puede utilizar para:

  - Añadir contenido a un recurso existente,
  - Mandar un mensaje a un grupo de noticias
  - Crear un registro nuevo en una base de datos
  - Pasar datos a un programa que se tiene que ejecutar al servidor. Un caso típico de este último ejemplo son los datos de un formulario HTML.

- **PUT** y **DELETE** Se usan principalmente en *endpoints* en **RESTful APIs**. El método `PUT` se usa para actualizar contenido mientras que `DELETE` par borrarlo

Cabeceras HTTP en peticiones

Las principales cabeceras son:

- `Host` El encabezado de solicitud Host especifica el nombre de dominio del servidor (para alojamiento virtual) y (opcional mente) el número de puerto TCP en el que el servidor está escuchando. Si no se da ningún puerto, el puerto predeterminado para el servicio solicitado (por ejemplo, "80" para una URL HTTP) está implícito. En PHP se puede obtener con `$_SERVER['HTTP_HOST'] or $_SERVER['SERVER_NAME']`

- `User-Agent` Contiene una cadena característica que permite a los pares del protocolo de red identificar el tipo de aplicación, el sistema operativo, el proveedor de software o la versión de software del agente de usuario de software solicitante. Por ejemplo,

  ```
  Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:55.0) Gecko/20100101
  Firefox/55.0
  ```

`Mozilla/5.0 (Android 4.4; Mobile; rv:41.0) Gecko/41.0 Firefox/41.0`

Podéis acceder a un listado exhaustivo en https://developers.whatismybrowser.com/useragents/explore/

Se puede utilizar para comprobar la versión del navegador y redirigir a una página de descarga si este está obsoleto. También se puede usar para condicionar el comportamiento de la página dependiendo del tipo de `user-agent`. Ahora mismo hay una especificación de Google para eliminar esta cabecera.

- `Cookie` Contiene HTTP cookies previamente enviadas por el servidor mediante la cabecera `Set-cookie`. En este manual se explica cómo crear cookies en PHP.

Cabeceras HTTP en respuestas

- `Cache-control` El campo `header` general `Cache-Control` es usado para especificar directivas las cuales DEBEN de ser obedecidas por todos los mecanismos de cacheo junto a la cadena petición/respuesta

```php
<?php
  //set headers to NOT cache a page
  header("Cache-Control: no-cache, must-revalidate"); //HTTP 1.1
  header("Pragma: no-cache"); //HTTP 1.0
  header("Expires: Sat, 26 Jul 1997 05:00:00 GMT"); // Date in the past

  //or, if you DO want a file to cache, use:
  header("Cache-Control: max-age=2592000"); //30days (60sec * 60min * 24hours * 30days)
?>
```

- `Content-type` Esta cabecera indica el "`mime-type`" del documento. El navegador entonces decide como interpretar los contenidos basado en esto. Por ejemplo, una página html (o un script PHP con salida html) puede devolver esto:

  `Content-Type: text/html; charset=UTF-8`

- Principales cabeceras para proteger contra hackeos (las iremos viendo a lo largo del módulo)

  - `X-Frame-Options`
  - `X-Content-Type-Options`
  - `Access-Control-Allow-Origin`
  - `HTTP Strict Transport Security`
  - `Content Security Policy`

Para una lista exhaustiva de todas las cabeceras, consultad https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers

La siguiente presentación, en inglés, explica las headers desde el punto de vista de los hackers

Principales códigos de estado (`status codes`)

Los códigos de estado de respuesta HTTP indican si se ha completado satisfactoriamente una solicitud HTTP específica. Las respuestas se agrupan en cinco clases: respuestas informativas, respuestas satisfactorias, redirecciones, errores de los clientes y errores de los servidores.

Están formadas por un código de tres dígitos, indicando el primero a qué tipo se corresponde:

- `1xx` son de tipo informativas. En general, no son usadas
- `2xx` son satisfactorias. Por ejemplo, el código `200 OK` indica que todo ha ido bien
- `3xx` son redirecciones, `301 Moved Permanently` indica que la página se ha movido a otra dirección temporalmente
- `4xx` error en cliente, `404 Not Found` significa que la página no existe.
- `5xx` error en el servidor, `500 Internal Server` Error significa que el servidor ha sido incapaz de procesar la petición tal vez por un error de programación o falta de recursos.

Podéis acceder a una lista completa en este artículo.

HTTP/2

Hasta ahora hemos hablado del protocolo `HTTP/1`.

Sin embargo, hoy en día se está empezando a utilizar `HTTP/2` para resolver algunos de los inconvenientes del protocolo anterior. `HTTP/2` se basa en SPDY, un protocolo de Google para mejora la velocidad,

`HTTP/2` hace que nuestras apps sean más veloces, más simples y más sólidas — una combinación extraña— al permitirnos deshacernos de los numerosos métodos alternativos de `HTTP/1.1` aplicados anteriormente dentro de nuestras apps y abordar estas inquietudes dentro de la capa de transporte. Mejor aún, también abre un abanico de oportunidades totalmente nuevas para optimizar nuestras apps y mejorar el rendimiento.

Los objetivos principales de `HTTP/2` son reducir la **latencia** al permitir una **multiplexación** completa de solicitudes y respuestas, minimizar la sobrecarga de protocolo mediante una compresión eficiente de campos de encabezados de `HTTP` y agregar soporte para priorización de solicitudes y servidor `push`. Con el fin de implementar estos requisitos, existe una vasta serie de otras mejoras de protocolo tales como nuevos mecanismos de control de flujo, manejo de errores y actualizaciones, pero estas son las funciones más importantes que todo programador web debe entender y aprovechar en sus apps.

`HTTP/2` de ningún modo modifica la semántica de app de `HTTP`. Todos los conceptos centrales, como los métodos de `HTTP`, códigos de estado, URIs y campos de encabezados, permanecen vigentes. Por otra parte, `HTTP/2` modifica el modo en que los datos se formatean (entraman) y se transportan entre el cliente y el servidor (ambos administran el proceso completo) y oculta toda complejidad de nuestras apps dentro de la nueva capa de entramado. En consecuencia, todas las apps existentes pueden proporcionarse sin ninguna modificación.

En el centro de todas las mejoras de rendimiento de `HTTP/2` se encuentra la nueva capa de entramado binario, que impone la forma en que los mensajes de `HTTP` se encapsulan y se transfieren entre el cliente y el servidor.

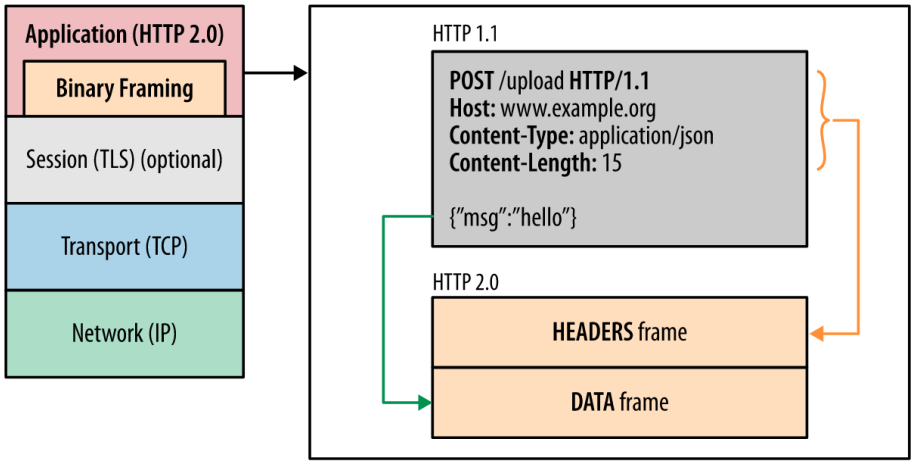


Figure 11: Capa de entramado binario de HTTP/2

La "*capa*" se refiere a una elección de diseño para introducir un nuevo mecanismo de codificación optimizado entre la interfaz del socket y la API de `HTTP` superior expuesta a nuestras apps: la semántica de `HTTP`, como verbos, métodos y encabezados, no se ven afectados, lo que difiere es la forma en que se codifican mientras están en tránsito. A diferencia del protocolo `HTTP/1.x` de texto plano delimitado por línea nueva, toda la comunicación de `HTTP/2` se divide en mensajes y tramas más pequeños, cada uno de los cuales está codificado en formato binario.

En consecuencia, tanto el cliente como el servidor deben usar el nuevo mecanismo de codificación binaria para entenderse entre sí: un cliente de `HTTP/1.x` **no entenderá un servidor únicamente compatible con** `HTTP/2` y viceversa. Afortunadamente, nuestras apps desconocen todos estos cambios, ya que el cliente y el servidor realizan todo el trabajo de entramado necesario en nuestro nombre.

¿Cómo se consigue ese aumento de velocidad tan significativo? Pues **multiplexando** las peticiones que reciben los servidores por parte de los usuarios y sus navegadores web. Es decir: que esos servidores puedan atender varias peticiones al mismo tiempo. Eso también ahorra en cantidad de conexiones, liberando de trabajo a los servidores. Este gráfico lo explica bien

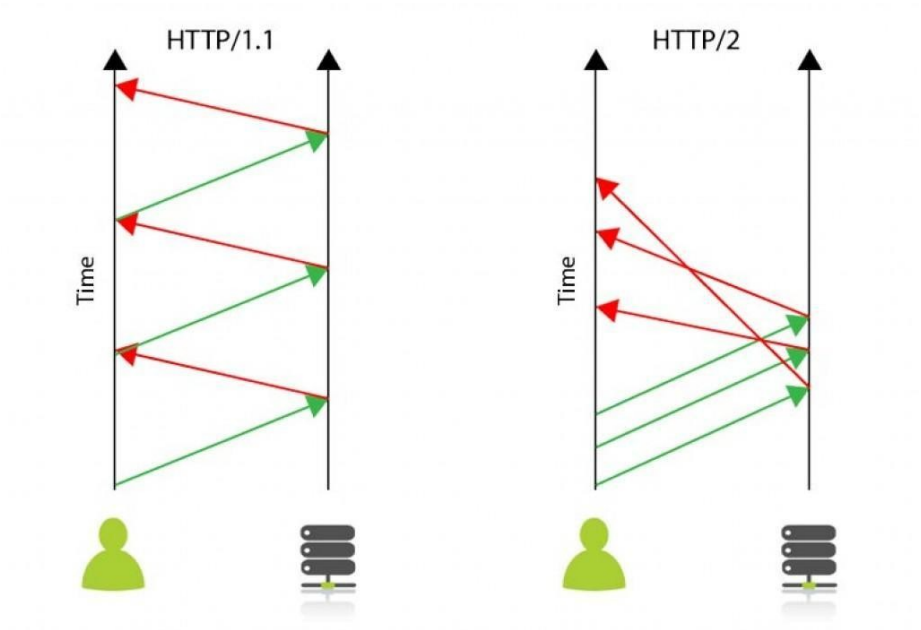


Figure 12: HTTP 1.1 vs HTTP/2

Además los servidores podrán ser proactivos: reconocerán qué tipo de cliente (navegador web) ha enviado una petición y, además de enviar la respuesta que necesita, enviará también respuestas con datos que ya sabe que el navegador va a necesitar antes de que éste los pida en una nueva petición. Por ejemplo: mientras que con HTML tenemos que cargar primero todo el HTML de la web para después cargar su contenido (CSS, imágenes), con `HTTP/2` podemos cargar todo ese contenido al mismo tiempo que el mismo HTML base. También se implementa `Server Push`, de esta forma el servidor puede enviar datos al cliente sin que este los solicite para que sean cacheados en el navegador.

En la siguiente página oficial de HTTP/2 podéis consultar las ventajas sobre

`HTTP/1`.

Podéis ver un ejemplo de esta mejora en las siguiente páginas

- https://http2.akamai.com/demo
- https://www.cloudflare.com/website-optimization/http2/

Desde el punto de vista de la ciberseguridad

Como el protocolo HTTP no está cifrado a nivel de la capa de red, se pueden realizar ataques de **Man-in-the-middle** (MITM). Es decir, un atacante puede interceptar las comunicaciones entre el cliente y el servidor (por ejemplo, conocer estados financieros, contraseñas) e incluso modificar dicha comunicación, tal vez, inyectado algún script para realizar criptominado o otro tipo de ataque. Ocurre lo mismo que en las WIFIs abiertas. Un hacker puede estar en una estación de tren y crear un punto de acceso WIFI pudiendo realizar cualquier tipo de ataque MITM.

Es por ello que hoy en día se recomienda el uso del protocolo `HTTPS` no sólo en aquellas webs que manejan datos sensibles.

HTTPS

HTTPS (HyperText Transfer Protocol Secure, Protocolo de transferencia de hipertexto seguro) es un protocolo de comunicación de Internet que protege la integridad y la confidencialidad de los datos de los usuarios entre sus ordenadores y el sitio web.

El envío de datos mediante el protocolo HTTPS está protegido mediante el protocolo de seguridad de la capa de transporte(TLS), que proporciona las tres capas clave de seguridad siguientes:

1. **Cifrado:** se cifran los datos intercambiados para mantenerlos a salvo de miradas indiscretas. Ello significa que cuando un usuario está navegando por un sitio web, nadie puede "escuchar" sus conversaciones, hacer un seguimiento de sus actividades por las diferentes páginas ni robarle información.
2. **Integridad de los datos:** los datos no pueden modificarse ni dañarse durante las transferencias, ni de forma intencionada ni de otros modos, sin que esto se detecte.
3. **Autenticación:** garantiza que tus usuarios se comunican con el sitio web previsto. Proporciona protección frente a los ataques "man-in-the-middle".

Las dos primeras capas se consiguen mediante la creación de un par de claves: una pública y una privada, en lo que se conoce como Criptografía de clave pública.

La autenticación se consigue mediante la participación de una autoridad de confianza de certificados que garantiza que el certificado ha sido emitido "para y sólo para" el sitio certificado.

Figure 13: Certificado digital

Hoy en día, este protocolo se usa en la mayoría de sitios en los que el visitante puede proporcionar información sensible, como contraseñas, transacciones bancarias, correo electrónico, etc.

De hecho es ya tan habitual, que los navegadores muestran una alerta cuando se intenta informar un campo de un formulario de tipo password.

También los navegadores han empezado a marcar aquellas webs que no usan el protocolo https:

¿Cómo funciona?

Una transacción segura `TSL` se realiza de acuerdo al siguiente modelo (resumido):

1. El servidor envía su clave pública al cliente (navegador)
2. El cliente verifica la validez de este certificado (y por consiguiente, la autenticidad del servidor), luego crea una clave secreta al azar, cifra esta

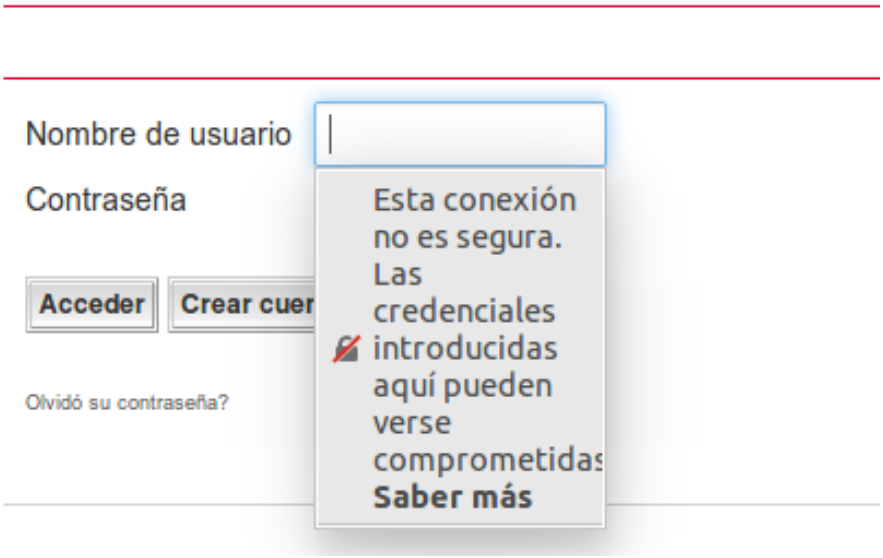


Figure 14: Aviso de conexión no segura

clave con la clave pública del servidor y envía el resultado del servidor (clave de sesión).

3. El servidor ahora es capaz también de descifrar los mensajes enviados por el cliente a partir de esta clave de sesión y su clave privada. De esta manera, hay dos entidades que comparten una clave que sólo ellos conocen. Las transacciones restantes pueden realizarse utilizando la clave de sesión, garantizando la integridad y la confidencialidad de los datos que se intercambian.

Ejemplo de cifrado de mensaje: Ana envía un mensaje a David

1. Ana redacta un mensaje
2. Ana cifra el mensaje con la clave pública de David
3. Ana envía el mensaje cifrado a David a través de Internet, ya sea por correo electrónico, mensajería instantánea o cualquier otro medio
4. David recibe el mensaje cifrado y lo descifra con su clave privada
5. David ya puede leer el mensaje original que le mandó Ana

Para activar el protocolo HTTPS en nuestro sitio web debemos obtener un certificado de seguridad, que lo emite una Autoridad de Certificación, aunque si sólo queremos encriptar nuestro sitio, podemos instalar un certificado autofirmado.

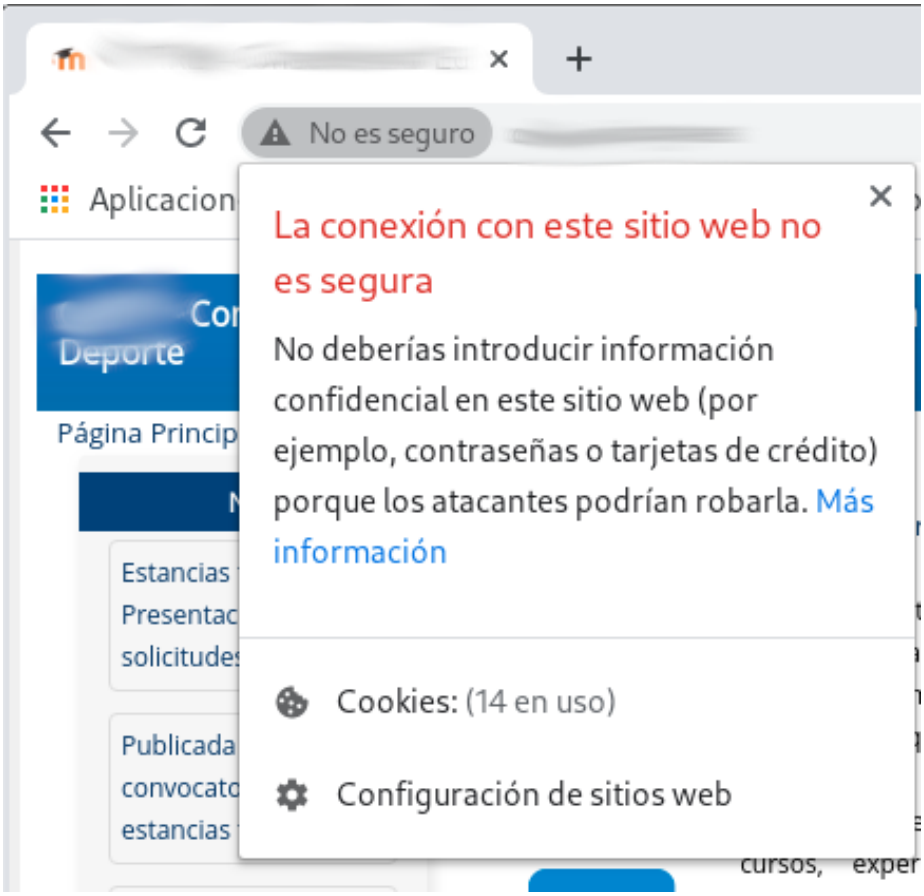


Figure 15: Conexión no segura

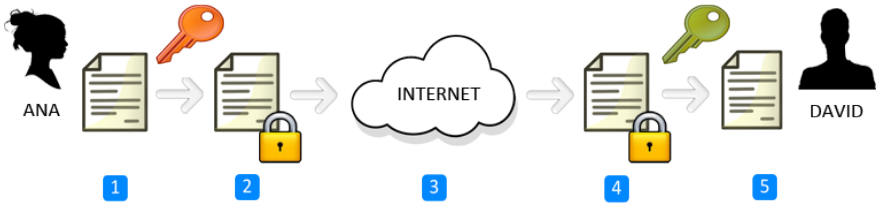


Figure 16: Cifrado asimétrico

Desde el punto de vista de la ciberseguridad

Aunque la información viaje encriptada entre el emisor y el receptor, esto no nos libra de un ataque `MITM` cuando estamos accediendo a una web segura mediante una WIFI abierta. Por lo que nos hemos de asegurar, además, de que el certificado está emitido para la web que estamos visitando.

Ejemplo de ataque

En primer lugar, Alice le pregunta a Bob por su clave pública. Si Bob envía su clave pública a Alice, pero Mallory es capaz de interceptarla, un ataque de intermediario puede comenzar. Mallory envía un mensaje falsificado a Alice que dice ser de Bob, pero en cambio incluye la clave pública de Mallory. Alice, creyendo que esta clave pública sea de Bob, cifra su mensaje con la clave de Mallory y envía el mensaje cifrado de nuevo a Bob. Mallory intercepta otra vez, descifra el mensaje utilizando su clave privada, posiblemente lo altera si quiere, y vuelve a cifrar con la clave pública de Bob que fue enviada originalmente a Alice. Cuando Bob recibe el nuevo mensaje cifrado, él cree que vino de Alice.

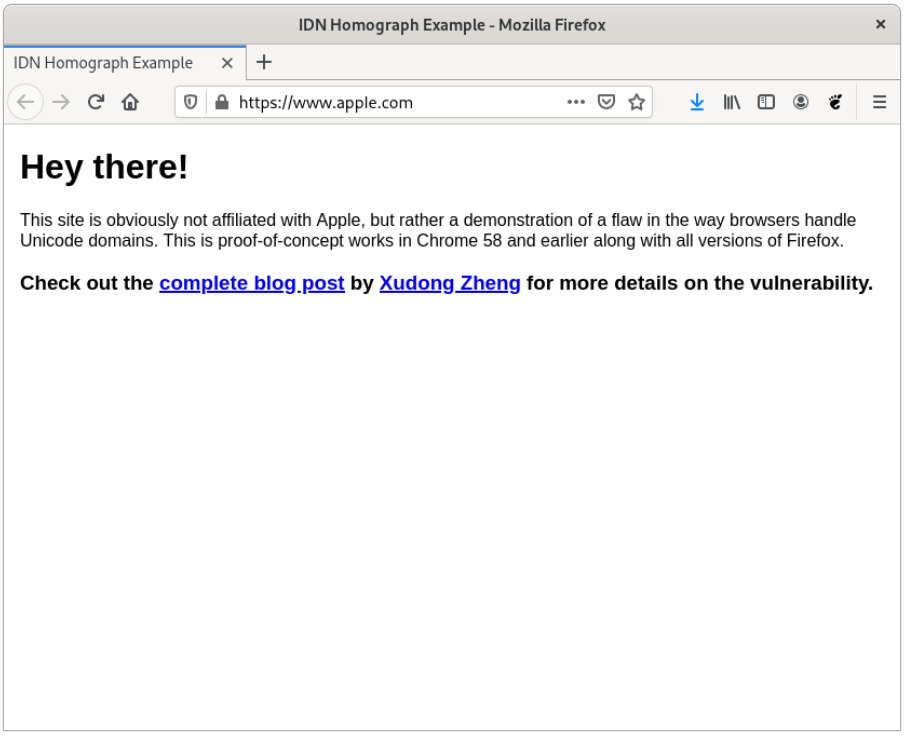
1. Alice envía un mensaje a Bob, que es interceptado por Mallory:
   Alice "*Hola Bob, soy Alice. Dame tu clave.*" → Mallory Bob
2. Mallory reenvía este mensaje a Bob; Bob no puede decir que no es realmente de Alice:
   Alice Mallory "*Hola Bob, soy Alice. Dame tu clave.*" → Bob
3. Bob responde con su clave de cifrado:
   Alice Mallory ← *[clave de Bob]* Bob
4. Mallory reemplaza la clave de Bob con la suya, y transmite esto a Alice, afirmando que es la clave de Bob:
   Alice ← *[clave de Mallory]* Mallory Bob
5. Alicia encripta un mensaje con lo que ella cree que es la clave de Bob, pensando que sólo Bob puede leer:
   Alice "*¡Nos vemos en la parada de autobús!*" *[Cifrada con la clave de Mallory]* → Mallory Bob
6. Sin embargo, debido a que en realidad estaba cifrada con la clave de Mallory, Mallory puede descifrarlo, leerlo, modificarlo (si se desea), volver a cifrar con la clave de Bob, y lo remitirá a Bob:
   Alice Mallory "*¡Nos vemos en la furgoneta de al lado del río!*" *[Cifrada con la clave de Bob]* → Bob
7. Bob cree que este mensaje es una comunicación segura de Alice.
8. Bob va a la furgoneta sin ventanas y Mallory le atraca.

Figure 17: Ejemplo de ataque

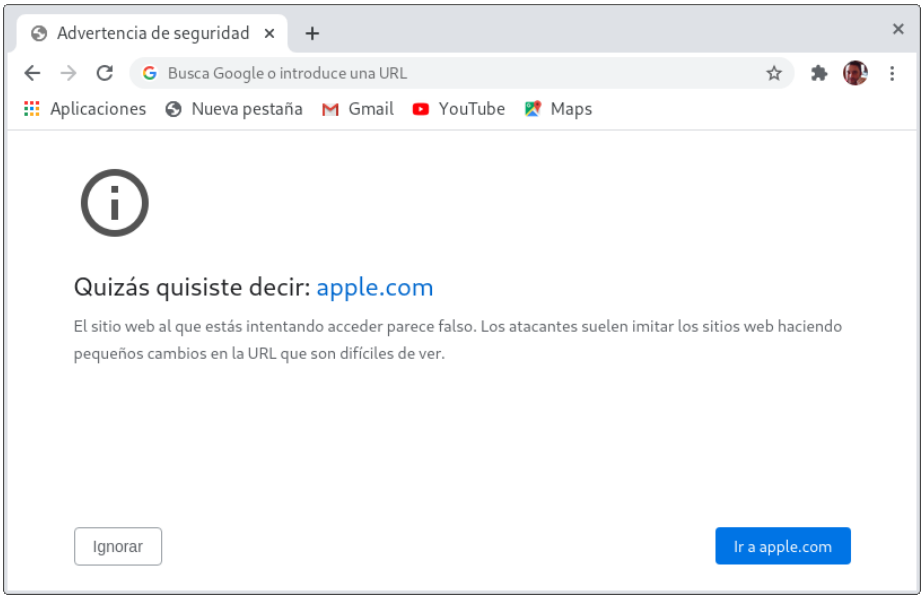
PunyCodes

Y también cuidado con los PunyCodes. Por ejemplo si visitas esta web que parece legítima ocurre lo siguiente:

En Firefox


IDN Homograph Example - Mozilla Firefox
IDN Homograph Example
https://www.apple.com
Hey there!
This site is obviously not affiliated with Apple, but rather a demonstration of a flaw in the way browsers handle Unicode domains. This is proof-of-concept works in Chrome 58 and earlier along with all versions of Firefox.
Check out the complete blog post by Xudong Zheng for more details on the vulnerability.

En Chrome



Antes y después de la renderización: todo lo demás que hace el navegador

Un navegador es mucho más que una canalización de renderizado y un motor JavaScript. Además de renderizar HTML y ejecutar JavaScript, los navegadores modernos contienen lógica para muchas otras responsabilidades. Los navegadores se conectan con el sistema operativo para resolver y almacenar en caché direcciones `DNS`, interpretar y verificar certificados de seguridad, codificar solicitudes en HTTPS si es necesario y almacenar y transmitir `cookies` de acuerdo con las instrucciones del servidor web. Para comprender cómo encajan estas responsabilidades, echemos un vistazo entre bastidores a un usuario que inicia sesión en Amazon:

1. El usuario visita www.amazon.es en su navegador favorito.
2. El navegador intenta resolver el dominio (amazon.es) en una dirección `IP`. Primero, el navegador consulta la caché de `DNS` del sistema operativo. Si no encuentra resultados, le pide al proveedor de servicios de Internet que busque en la caché de `DNS` del proveedor. En el improbable caso de que nadie del `ISP` haya visitado el sitio web de Amazon antes, el `ISP` resolverá el dominio en un servidor `DNS` autorizado.
3. Ahora que ha resuelto la dirección `IP`, el navegador intenta iniciar un protocolo de enlace `TCP` con el servidor correspondiente a la dirección `IP`

para establecer una conexión segura.

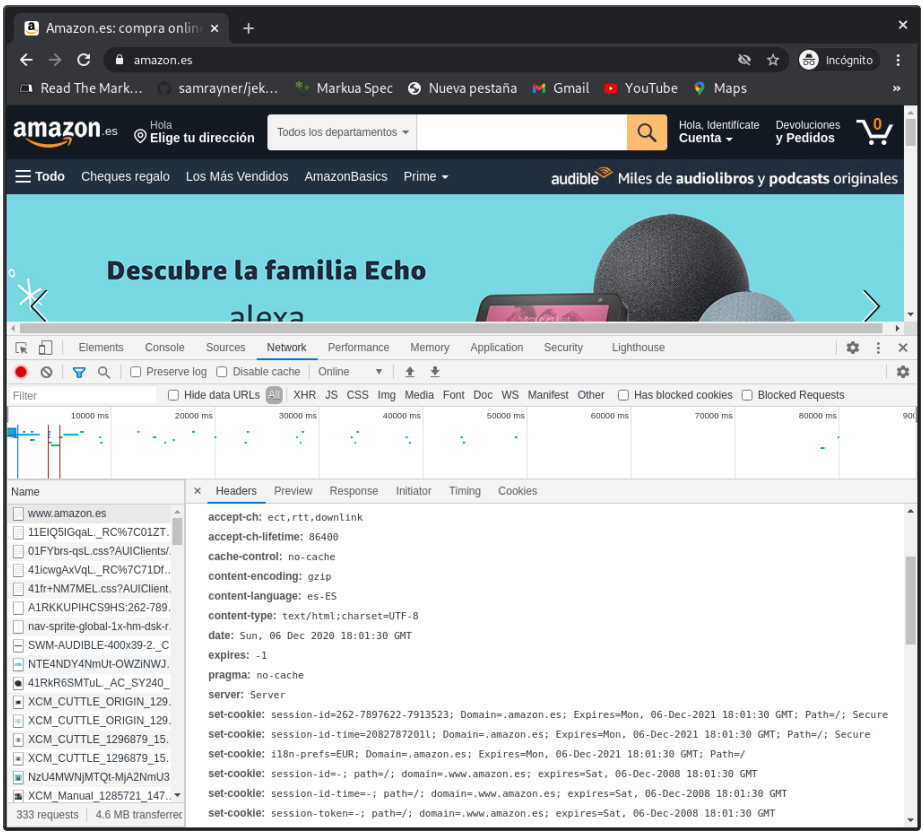
4. Una vez que se ha establecido la sesión de `TCP`, el navegador construye una solicitud `HTTP GET` a www.amazon.es. `TCP` divide la solicitud `HTTP` en paquetes y los envía al servidor para ser reensamblados.

5. En este punto, la conversación `HTTP` se actualiza a `HTTPS` para garantizar una comunicación segura. El navegador y el servidor realizan un protocolo de enlace `TLS`, acuerdan un cifrado de cifrado e intercambian claves de cifrado.

6. El servidor utiliza el canal seguro para enviar una respuesta `HTTP` que contiene `HTML` de la página principal de Amazon. El navegador analiza y muestra la página, lo que generalmente activa muchas otras solicitudes `HTTP GET`.

7. El usuario navega a la página de inicio de sesión, ingresa sus credenciales de inicio de sesión y envía el formulario de inicio de sesión, que genera una solicitud `POST` al servidor.

8. El servidor valida las credenciales de inicio de sesión y establece una sesión devolviendo un encabezado `Set-Cookie` en la respuesta. El navegador almacena la `cookie` durante el tiempo prescrito y la devuelve con solicitudes posteriores a Amazon.



Después de que todo esto suceda, el usuario puede acceder a su cuenta de Amazon.

Basado en:

https://developers.google.com/web/fundamentals/performance/http2/?hl=es

https://developer.mozilla.org/es/docs/Web/HTTP/Overview

https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers

https://es.wikipedia.org/wiki/Ataque_de_intermediario