

Protocolo de transferencia de hipertexto (HTTP)

Evaluación Continua 1 (EC1)

Luis Golac, Sebastian Loza, Paolo Vásquez
Redes y Comunicaciones (CS4054) - Laboratorio 1.01 - 2025 - 1
UTEC - Universidad de Ingeniería y Tecnología, Lima-Perú

I. INTRODUCIÓN

Actualmente estamos, por no decir siempre, enviando, recibiendo y solicitando información; es decir, constantemente estamos comunicándonos. A un nivel más alto, por ejemplo, de redes o sistemas, esto también sucede. El modelo de interconexión de sistemas abiertos (OSI, por sus siglas en inglés), el cual detallaremos un poco más adelante, define básicamente cómo se comunican los sistemas de redes; cómo envían datos y también los reciben. Este modelo define diferentes capas o niveles, dentro de las cuales aparece la capa de aplicación, el foco de este estudio. A través de esta experiencia, dentro de la capa de aplicación, estaremos explorando el protocolo de transferencia de hipertexto (HTTP), experimentando cómo este solicita y recibe información a través del analizador de protocolos Wireshark. También, aunque no es el foco de este estudio, se verá en alguna sección la capa de transporte, con su protocolo de control de transmisión (TCP).

I-A. Marco Tórico

Previo a la experiencia, es necesario tener claras ciertas nociones teóricas. A manera de síntesis, revisaremos los siguientes contenidos: **Capa de Aplicación y HTTP**, **Capa de Transporte y TCP**, **Wireshark** y **Adicionales**.

1. Capa de Aplicación y HTTP

El modelo OSI [1], como se detalló anteriormente, define ciertas capas, específicamente siete, que se apilan de abajo hacia arriba. En orden de menor a mayor, estas capas son: física, enlace de datos, red, transporte, sesión, presentación y aplicación. En el nivel más alto aparece la capa de aplicación [2], la cual se comunica directamente con el usuario. Aunque una aplicación puede interactuar con otras capas del modelo OSI, finalmente la interfaz se ejecuta en esta capa. Cuando un usuario promedio recibe un mensaje en cierto software, es esta capa la que se lo presenta. Dentro de esta, el protocolo más común es el HTTP [3], que veremos a continuación.

El protocolo de transferencia de hipertexto (HTTP) se emplea para la comunicación entre navegadores y servidores web. Tiene una estructura cliente-servidor y se basa en la noción de hacer peticiones (requests) y recibir respuestas (responses). En sí, el cliente hace solicitudes de información, datos o demás a un servidor, y este último le envía una respuesta, que puede ser afirmativa, negativa o de otras maneras. Para tener una noción, básicamente los requests y responses siguen el formato que se puede apreciar en la Figura 1, donde en cada uno tendremos un encabezado (header), en el que se detallan aspectos del mensaje. Cabe recalcar que el servidor puede asignarle un identificador al cliente (cookie) para recordarlo en futuras ocasiones. Este, si existe, puede verse en el encabezado.

The diagram illustrates the structure of an HTTP request message and an HTTP response message. It highlights various components: the request line (method, URL, version), header lines (Content-Type, Accept, User-Agent, etc.), and the body (data). The response message shows the status line (version, code, reason), header lines (Content-Type, Content-Length, etc.), and the body (data). Arrows point from labels to specific parts of the messages, such as 'request line' to the first line of the request, 'header lines' to the lines following it, and 'body' to the final part of the message.

Figura 1: Formato de request y response de HTTP [4].

2. Capa de Transporte y TCP

De forma breve, porque no es el foco de esta experiencia, esta capa, la número cuatro [2], se encarga de tomar los datos y dividirlos en partes más pequeñas; ello para poder transferirlos de

un lugar a otro. Dentro de esta capa, tenemos el protocolo de control de transmisión [5] (TCP), el cual permite intercambiar mensajes a través de una red. HTTP utiliza TCP por debajo para poder transferir mensajes. Es importante tener noción de lo que son los segmentos TCP; se pueden entender como la cantidad de partes en las que TCP dividió un mensaje para transmitirlo, y en muchos casos, esa cantidad para un mensaje específico dependerá de la tarjeta de red del dispositivo empleado.

3. Wireshark

Este software es un analizador de paquetes de código abierto [6]. Básicamente, nos permite analizar la red de manera muy minuciosa, dándonos herramientas y comandos para filtrar y examinar con detalle el tráfico de la red. A manera de ejemplo, nos podría permitir observar paquetes caídos o incluso actividad maliciosa en alguna red. También, y de forma general, permite aprender sobre protocolos como HTTP y TCP, analizar y comprender los encabezados de paquetes, su enrutamiento y demás aspectos.

Sobre su utilización, de manera breve: al instalar la aplicación podrá ver el entorno que se observa en la Figura 2. Para iniciar la captura de paquetes, seleccione la alternativa que deseé (pruebe con Wi-Fi: en0) y podrá ver cómo se capturan paquetes como en la Figura 3. En esta última podrá observar las partes relevantes. Para detener la captura, presione el botón cuadrado rojo, y para iniciar nuevamente la captura, presione el botón azul con forma de aleta de tiburón.

En esta experiencia, como ya debe suponerse, haremos uso de Wireshark para monitorear la comunicación entre nosotros (cliente) y diferentes servicios (servidores), con ello viendo y entendiendo de manera práctica un poco de lo explicado anteriormente. Se comprenderá también un poco más la utilidad de este software, aunque también veremos más adelante cómo se ha usado en diferentes estudios.

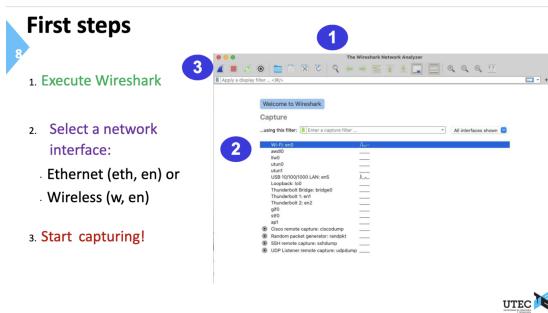


Figura 2: Vista general de Wireshark [7].

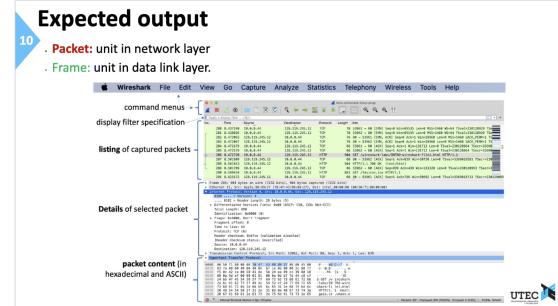


Figura 3: Vista de capturador de paquetes de Wireshark [7].

4. Adicionales

Teniendo hasta el momento una noción clara de los conceptos, podemos ver y discutir algunos tipos de protocolo HTTP. Primero, tenemos **HTTP/2** [8], que es capaz de enviar varios flujos de datos a la vez, usando una sola conexión TCP, a diferencia de **HTTP/1.1** [8], que envía los recursos uno después del otro. Luego, tenemos **HTTP/3** [8], cuya diferencia clave es la utilización de QUIC, un protocolo de la capa de transporte más rápido y seguro, en vez de TCP. Finalmente, **HTTPS** [9], básicamente es una versión encriptada de HTTP, que permite aumentar la seguridad al momento de transferir datos y mensajes. Estas nociones serán útiles más adelante, ya que se distinguirá el tipo de HTTP utilizado.

I-B. Estado del Arte

A manera de entender la importancia y utilidad de HTTP, y también del software Wireshark, revisaremos un poco la bibliografía, enfocándonos en estudios académicos que emplearon Wireshark para el análisis de HTTP, principalmente para temas de seguridad y detección de actividad maliciosa.

1. Enhancing LAN Security by Mitigating Credential Threats via HTTP Packet Analysis with Wireshark

Este estudio [10] explora y explica cómo atacantes pueden aprovecharse de las vulnerabilidades de HTTP para extraer credenciales de redes de área local (LANs). De forma específica, se trata un escenario en el que, en una red de área local, un atacante puede estar conectado a esta red, y mediante el uso de Wireshark explotar las vulnerabilidades de HTTP, obteniendo credenciales de ingreso (login). Pueden obtener direcciones IP, direcciones de correo electrónico y demás. Y de forma general, se muestra cómo, empleando esta herramienta, se pueden capturar y analizar protocolos como el Hypertext Transfer Protocol (HTTP), Address Resolution Protocol y Transmission Control Protocol (TCP). Lo que se propone en este estudio, además de hacer

un monitoreo regular de las redes, es implementar diferentes medios de seguridad como firewalls, IDS/IPS, segmentación de red, HTTPS, entre otros, para evitar esta explotación en las redes locales.

2. *Analyzing Vulnerabilities in Network Protocols Using Wireshark: A Case Study on HTTP and HTTPS*

Dicho trabajo [11] detalla y examina vulnerabilidades comunes en los protocolos HTTP y HTTPS, como por ejemplo el secuestro de sesión (session hijacking) y ataques de tipo man-in-the-middle. Explica que, aunque HTTPS provee un nivel adicional de seguridad debido a la encriptación TLS, tanto este como HTTP poseen vulnerabilidades. Se explora cómo, empleando Wireshark, se pueden detectar estas y finalmente mitigarlas, utilizando diferentes dominios y páginas web como sustento. Mediante todo ello, este estudio concluye en la imperiosa necesidad de robustas prácticas de encriptación, auditorías de seguridad constantes y una adecuada gestión para asegurar la comunicación segura en redes.

3. *Malicious Traffic Analysis Using Wireshark by Collection of Indicators of Compromise*

Esta investigación [12] se centra en el uso de Wireshark para reconocer indicadores que podrían comprometer la comunicación en el tráfico HTTP. Mediante Wireshark se puede realizar el análisis de paquetes o protocolos, conocido como sniffing, que se trata del proceso de capturar e interpretar datos en tiempo real mientras estos circulan por una red. De esta manera, se puede detectar actividad maliciosa, como filtraciones de datos, accesos no autorizados, infecciones por malware, entre otros. En dicha investigación, se aplica Wireshark para realizar ese análisis, diagnosticar protocolos de red y utilizarlo para detectar actividad maliciosa.

II. DESARROLLO Y RESULTADOS

El desarrollo de esta experiencia se dividió en 5 fases principales, las cuales iremos detallando una por una en este informe. Para cada sección mostraremos por ítems las preguntas del laboratorio y luego el detalle de la experiencia y las respuestas.

II-A. *The Basic HTTP GET/response interaction*

En esta primera parte, lo que se hizo fue capturar los paquetes enviados y recibidos por nuestro navegador al acceder a esta dirección URL <http://gaia.cs.umass.edu/wireshark-labs/HTTP-wireshark-file1.html>. Luego de esto, se analizaron los resultados obtenidos para responder las

preguntas.

1. *Is your browser running HTTP version 1.0, 1.1, or 2? What version of HTTP is the server running?*

En este caso todos obtuvimos que nuestro browser y el servidor que nos envía la respuesta a nuestra petición utilizaba la versión 1.1 de HTTP. Esto lo podemos observar en la Figura 4:

Time	Source	Destination	Protocol	Length	Info
1. 14:41:47.000000000	192.168.0.157	128.119.245.12	HTTP	501	GET /wireshark-labs/HTTP-wireshark-file1.html HTTP/1.1
1. 14:41:47.000000000	192.168.0.157	128.119.245.12	HTTP	552	HTTP/1.1 200 OK (text/html)

Figura 4: Versión HTTP del servidor y de nuestro browser.

2. *What is the IP address of your computer? What is the IP address of the gaia.cs.umass.edu server?*

En este ítem detectamos cuál era la dirección ip de nuestra computadora y cuál era la dirección ip de sel servidor de respuesta. Vimos que la ip del servidor de respuesta era: 128.119.245.12, mientras que la de nuestra computadora variaba para cada uno ya que cada máquina tiene su propia dirección ip. Eso lo pudimos notar en base a la siguiente información que nos mostraba Wireshark (Figura 5).

Source	Destination	Protocol	Length	Info
192.168.0.157	128.119.245.12	HTTP	501	GET /wireshark-labs/HTTP-wireshark-file1.html HTTP/1.1
128.119.245.12	192.168.0.157	HTTP	552	HTTP/1.1 200 OK (text/html)

Figura 5: Direcciones ip del servidor y de nuestra máquina.

Como vimos la primera petición es el GET que hace nuestro browser hacia el servidor para poder obtener el html de la página, por lo que la fuente (Source) sería la dirección ip de nuestra máquina y el destino (Destination) sería la dirección ip del servidor. Además en la segunda línea se muestra la respuesta 200 OK y el html del servidor hacia nuestro browser, por lo que la fuente de dicha respuesta tendría que ser el servidor y el destino sería nuestra computadora.

3. *What is the status code returned from the server to your browser?*

Para el 3er ítem nos basamos en lo discutido en la sección anterior donde explicamos que la segundo línea es la respuesta del servidor y se puede observar que la respuesta del servidor es 200 OK.

4. *When was the HTML file that you are retrieving last modified at the server?*

Para responder esta pregunta tuvimos que acceder a la sección de *Packet-content window*, específicamente a *Hypertext Transfer Protocol* del paquete referente a la respuesta del servidor. En esta sección pudimos obtener la información de cuando fue modificado el html por última vez en el servidor. Se observa esto en la Figura 6.

5. *How many bytes of content are being returned to your browser?*

```
Server: Apache/2.4.6 (CentOS) OpenSSL/1.0.2k-fips PHP/7.4.33 m
Last-Modified: wed, 02 apr 2025 05:59:01 gmt\r\n
ETag: "80-631c55998a879"\r\n
```

Figura 6: Fecha y hora de la última modificación del html en el servidor.

En el 5to ítem observamos que la cantidad de bytes de contenido que manda el servidor al navegador es de 128 bytes (Figura 7) y esta información se encuentra en la misma sección que accedimos en el ítem previo.

Full request URI: http:
File Data: 128 bytes

Figura 7: Cantidad de bytes del contenido.

6. *By inspecting the raw data in the packet content window, do you see any headers within the data that are not displayed in the packet-listing window? If so, name one.*

Para este último ítem analizamos la sección de la raw data y vimos que los headers que aparecen ahí son los mismos que aparecen en el *packet-content window*; sin embargo, en el *packet content window* muestra la información de la cantidad de bytes del archivo de respuesta (Figura 7) y en el *raw data* no lo muestra como se puede ver en la Figura 24

```
] .HTTP/1 .1 200 0
K · Date: Wed, 16
Apr 2025 05:45:
35 GMT · Server:
Apache/2 .4.6 (Ce
ntOS) OpenSSL/1.
0.2k-fip s PHP/7.
4.33 mod _perl/2.
0.11 Per l/v5.16.
3 · Last- Modified
: Wed, 16 Apr 20
25 05:45 :01 GMT·
· ETag: " 80-632de
c94b1992 " · Accep
t-Ranges : bytes·
· Content -Length:
128 · Ke ep-Alive
: timeout t=5, max
=100 · Co nnection
: Keep-A live · Co
nnect-Ty pe: text
/html; c harset=U
TF-8 · · <html>· C
```

Figura 8: Raw data de la respuesta del servidor.

II-B. The HTTP CONDITIONAL GET/response interaction

Para esta sección, el proceso fue el mismo que el anterior, sólo que el link usado fue distinto: <http://gaia.cs.umass.edu/wireshark-labs/HTTP-wireshark-file2.html>.

En esta experiencia hubo una variación y es que, una vez que ya estábamos capturando los paquetes http, luego de haber ingresado la url en el navegador, se hizo un refresh para que se volviera a hacer la petición al servidor que almacena dicho sitio web.

1. *Inspect the contents of the first HTTP GET request from your browser to the server. Do you see an “IF_MODIFIED_SINCE” line in the HTTP GET?*

En este primer ítem observamos la sección de *Hypertext Transfer Protocol* de la primera petición y vimos de que no aparece el *IF-MODIFIED-SINCE* para ninguno de los 3, lo cual lo puedes observar en la Figura 9. Esto nos dimos cuenta que se debía a que como era la primera vez que se hacía una petición a este servidor para obtener el archivo html, dicho archivo aún no estaba en la caché.

```
> Frame 316: 517 bytes on wire (4136 bits), 517 bytes captured (4136 bits) on interface en0, i
> Ethernet II, Src: MitraStarTec/de:7c:68 (02:00:00:00:00:00), Dst: MitraStarTec/de:7c:68 (02:00:00:00:00:00)
> Internet Protocol Version 4, Src: 192.168.1.38, Dst: 128.119.245.12
> Transmission Control Protocol, Src Port: 56251, Dst Port: 80, Seq: 1, Ack: 1, Len: 451
< Hypertext Transfer Protocol
  < GET /wireshark-labs/HTTP-wireshark-file2.html HTTP/1.1\r\n
    Request Method: GET
    Request URI: /wireshark-labs/HTTP-wireshark-file2.html
    Request Version: HTTP/1.1
    Host: gaia.cs.umass.edu\r\n
    Connection: keep-alive\r\n
    Upgrade-Insecure-Requests: 1\r\n
    User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/81.0.4044.138 Safari/537.36
    Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
    Sec-GPC: 1\r\n
    Accept-Language: en-US,en;q=0.7\r\n
    Accept-Encoding: gzip, deflate\r\n
  \r\n
  [Response in frame: 320]
  [Full request URI: http://gaia.cs.umass.edu/wireshark-labs/HTTP-wireshark-file2.html]
```

Figura 9: Sección de Hypertext Transfer Protocol.

2. *Inspect the contents of the server response. Did the server explicitly return the contents of the file? How can you tell?*

Para este ítem vimos que si devuelve la data de manera explícita al ver la sección *Line-based text data* ya que en esta sección se muestra la información que es renderizada en el browser. Se puede observar en la Figura 17:

```
- Line-based text data: text/html (10 lines)
  \n
  <html>\n
  \n
  Congratulations again! Now you've downloaded the file lab2-2.html. <br>\n
  This file's last modification date will not change. <p>\n
  Thus if you download this multiple times on your browser, a complete copy <br>\n
  will only be sent once by the server due to the inclusion of the IF-MODIFIED-SINCE<br>\n
  field in your browser's HTTP GET request to the server.\n
  \n
  </html>\n
```

Figura 10: Sección de Line-based text data.

3. *Now inspect the contents of the second HTTP GET request from your browser to the server. Do you see an “IF_MODIFIED_SINCE:” line in the HTTP GET? If so, what information follows the “IF_MODIFIED_SINCE:” header?*

En este caso, al analizar la segunda petición GET que hizo nuestro browser al servidor vimos que si se aparecía el header *IF-MODIFIED-SINCE*. En dicho

header aparece una fecha, como se muestra en la Figura 18. Luego de investigar más a fondo vimos que este header ayuda en cierta forma a que esta petición sea "condicional", es decir, dicha petición va a preguntar al servidor si el archivo ha sido modificado después de la fecha que aparece en este header, si no es así entonces no me mandes el archivo de vuelta. Esto es porque el browser ya tiene ese archivo en la caché, por lo que no es necesario mandarlo de vuelta.

```
> Frame 716: 629 bytes on wire (5032 bits), 629 bytes captured (5032 bits) on interface en0, j
> Ethernet II, Src: f6:72:83:74:16:78 (f6:72:83:74:16:78), Dst: MitraStarTec_de7c:68 (2c:96:8
> Internet Protocol Version 4, Src: 192.168.1.38, Dst: 128.119.245.12
> Transmission Control Protocol, Src Port: 56252, Dst Port: 80, Seq: 1, Ack: 1, Len: 563
> Hypertext Transfer Protocol
  <-- GET /wireshark-labs/HTTP-wireshark-file2.html HTTP/1.1\r\n
    Request Method: GET
    Request URI: /wireshark-labs/HTTP-wireshark-file2.html
    Request Version: HTTP/1.1
    Host: gaia.cs.umass.edu\r\n
    Connection: keep-alive\r\n
    Cache-Control: max-age=0\r\n
    Upgrade-Insecure-Requests: 1\r\n
    User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like
    Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/
    Sec-GPC: 1\r\n
    Accept-Language: en-US;q=0.7\r\n
    Accept-Encoding: gzip, deflate\r\n
    If-None-Match: "173-632a2ad1d4a3a3"\r\n
    If-Modified-Since: Sun, 13 Apr 2025 05:59:01 GMT\r\n
  \r\n
  [Response in frame: 720]
  [Full request URI: http://gaia.cs.umass.edu/wireshark-labs/HTTP-wireshark-file2.html]
```

Figura 11: Sección de Hypertext Transfer Protocol.

- What is the HTTP status code and phrase returned from the server in response to this second HTTP GET? Did the server explicitly return the contents of the file? Explain.

Para este ítem vimos que el *status code* que nos mandó el servidor fue el 304 junto con un mensaje que decía *Not Modified*. Esto se puede observar en la Figura 20

Protocol Length Info					
	No.	Time	Source	Destination	Protocol
HTTP	1	19:46:40...	192.168.0...	128.119.24...	HTTP
HTTP	2	19:46:40...	128.119.24...	192.168.0...	HTTP
HTTP	3	19:46:41...	192.168.0...	128.119.24...	HTTP
HTTP	4	19:46:41...	128.119.24...	192.168.0...	HTTP
					501 GET /wireshark-labs/HTTP-wireshark-file2.html HTTP/1.1
					796 HTTP/1.1 200 OK (text/html)
					613 GET /wireshark-labs/HTTP-wireshark-file2.html HTTP/1.1
					305 HTTP/1.1 304 Not Modified

Figura 12: Segunda respuesta del servidor al browser.

Además al analizar el *packet-content window* de la respuesta no había la sección **Line-based text data** y en la sección de *Hypertext Transfer Protocol* no se encuentra el header *Content-Length* (Figura 21). Esto nos indica que la data no ha sido enviada de vuelta. Como mencionamos en el anterior ítem, esto se debe a que como el Browser ya tiene el archivo en el caché y no ha sido modificado luego de haber sido obtenido del servidor, no es necesario volver a mandar el archivo.

```
> Hypertext Transfer Protocol
> HTTP/1.1 304 Not Modified\r\n
> Date: Sun, 13 Apr 2025 23:02:19 GMT\r\n
> Server: Apache/2.4.33 (CentOS) OpenSSL/1.0.2k-fips PHP/7.4.33 mod_perl/2.0.11 Perl/v5.16.3\r\n
> Connection: Keep-Alive\r\n
> Keep-Alive: timeout=5, max=99\r\n
> ETag: "173-632a2ad1d4a3a3"\r\n
> \r\n
> [Request in frame: 1861]
> [Time since request: 0.127340817 seconds]
> [Request URI: /wireshark-labs/HTTP-wireshark-file2.html]
> [Full request URI: http://gaia.cs.umass.edu/wireshark-labs/HTTP-wireshark-file2.html]
```

Figura 13: Sección de Hypertext Transfer Protocol de la segunda respuesta del servidor.

II-C. Retrieving Long Documents

Para esta sección trabajamos con el siguiente link <http://gaia.cs.umass.edu/wireshark-labs/HTTP-wireshark-file3.html> y en este caso sólo hicimos la búsqueda en el browser una vez.

- How many HTTP GET request messages did your browser send?

Para este ítem, luego de haber hecho la búsqueda en el browser y filtrar por http en Wireshark, vimos que se hicieron 2 peticiones GET desde nuestro browser (Figura 19). Sin embargo, como se puede observar, la segunda petición es de un favicon.ico, lo cuál no es relevante así que podemos ignorar esa petición quedándonos con sólo 1 petición.

No.	Time	Source	Destination	Protocol	Length/Info
1	19:46:40...	192.168.0...	128.119.24...	HTTP	501 GET /wireshark-labs/HTTP-wireshark-file3.html HTTP/1.1
2	19:46:40...	128.119.24...	192.168.0...	HTTP	583 HTTP/1.1 200 OK (text/html)
3	19:46:41...	192.168.0...	128.119.24...	HTTP	486 GET /favicon.ico HTTP/1.1
4	19:46:41...	128.119.24...	192.168.0...	HTTP	550 HTTP/1.1 404 Not Found (text/html)

Figura 14: Packet-List window de wireshark.

- What is the status code and phrase in the response?

El *status code* y la frase de respuesta del servidor es 200 Ok, lo cuál nos indica que no ha habido ningún problema para enviar la información hacia nuestro navegador. Esto también lo podemos ver en la misma Figura 19.

- How many data-containing TCP segments were needed to carry the single HTTP response and the text of the Bill of Rights?

Para este último ítem ocurrió algo interesante, para 2 miembros del grupo que realizaron la experiencia usando laptops con sistema operativo Linux se necesitaron sólo 2 segmentos TCP para recuperar la información solicitada (Figura 22). Por otro lado, para el 3er miembro que estaba usando una MAC se necesitaron 4 segmentos TCP para traer el mismo archivo (Figura 23).

```
> TCP segment data (703 bytes)
> [Segment count: 2]
> [Reassembled Segments (486 bytes): #103(4158), #104(703)]
> [Frame: 103, payload: 0-4157 (4158 bytes)]
> [Frame: 104, payload: 4158-4660 (703 bytes)]
> [Segment count: 2]
> [Reassembled TCP Data (...): 48545402f31e312032303204f4b0d0a446174653a285468752c20303204170722032352030313a3513a3]
```

Figura 15: Cantidad de segmentos TCP en una máquina Linux.

```
[Frame 387]: 607 bytes on wire (4856 bits), 607 bytes captured (4856 bits) on interface en0, id 0
Ethernet II [SRC: MitMStarfire.de:fc:de:0c (12:96:82:de:c7:68)], Dst: f6:72:83:74:16:78 (f6:72:83:74:16:78)
Internet Protocol Version 4, Src: 128.119.245.12, Dst: 192.168.1.38
Transmission Control Protocol, Src Port: 80, Dst Port: 56262, Seq: 4321, Ack: 452, Len: 541
[HTTP/1.1]
[Frame 384, payload: 0-1439 (1448 bytes)]
[Frame 385, payload: 1448-2679 (1448 bytes)]
[Frame 386, payload: 2688-4319 (1448 bytes)]
[Frame 387, payload: 4328-5956 (591 bytes)]
[Segment count: 4]
[Reassembled TCP length: 4861]
[Reassembled TCP Data ...]
[Frame 38454502f312e3128323028024f4bb08a446174653a20537562c2031332841772023283252032303a34343a32392847d5a...]
HyperText Transfer Protocol
[HTTP/1.1 200 OK\r\n]
Response Version: HTTP/1.1
Status Code: 200
Status Code Description: OK
Headers: \r\n
Date: Sun, 13 Apr 2025 20:44:29 GMT\r\n
Server: Apache/2.4.6 (CentOS) OpenSSL/1.8.2-fips PHP/7.4.33 mod_perl/2.0.11 Perl/5.16.3\r\n
Content-Type: text/html; charset=UTF-8\r\n
ETag: "114e-532a2e1446f5"\r\n
Accept-Ranges: bytes\r\n
Content-Length: 45800\r\n
Keep-Alive: timeout=10, max=100\r\n
Connection: keep-alive\r\n
Content-Type: text/html; charset=UTF-8\r\n
\r\n
```

Figura 16: Cantidad de segmentos TCP en una máquina MAC.

En primera instancia podemos ver que en la laptop con Linux en un segmento TCP se pueden enviar hasta 4158 bytes, mientras que en la MAC, permite un máximo de 1440 bytes, lo que hace necesario una mayor cantidad de segmentos. Esto tiene que ver mucho con el tipo de sistema operativo ya que estos tienen distintas configuraciones de MSS (*Maximun Segment Size*). Como explica Huston [13], “convencionalmente, el valor de MSS para una conexión es establecido por la plataforma en lugar de la aplicación, y dicho valor se aplica a todas las conexiones TCP”.

II-D. HTML Documents with Embedded Objects

Para esta sección trabajamos con el siguiente link <http://gaia.cs.umass.edu/wireshark-labs/HTTP-wireshark-file4.html> el cual se realizó la búsqueda en el browser una vez.

1. How many HTTP GET request messages did your browser send? To which Internet addresses were these GET requests sent?

En este ítem se obtuvieron inicialmente resultados distintos, que sin embargo llevan a la misma conclusión. Dos miembros del grupo que realizaron la experiencia usando laptops con sistema operativo Linux, tras aplicar el filtro HTTP, identificaron un total de 4 peticiones GET (Figura 17). Por otro lado, el tercer miembro, que utilizaba macOS, registró solo 3 solicitudes GET (Figura 18). La diferencia de GETs entre ambos casos se debió a una solicitud adicional al archivo favico.ico, archivo que no afecta al contenido principal de la página. Dado que esta petición no es relevante para el análisis, es descartado. Por lo tanto, todos los miembros realizaron 3 solicitudes GET.

Figura 17: *Solicitudes GET en Linux*

88	2025-04-13	15:16:09	90-564146	192.168.1.18	192.168.245.12	HTTP	517 GET /firewall-test-wireshark/file4.html HTTP/1.1
89	2025-04-13	15:16:10	7970315	192.168.1.18	192.168.1.18	HTTP	2897 GET /index.html 200 OK (text/html)
90	2025-04-13	15:16:10	8363561	192.168.1.18	192.168.1.18	HTTP	582 GET /session.png 200 OK (PNG)
94	2025-04-13	15:16:10	8935027	192.168.1.18	192.168.1.18	HTTP	797 THT /index.html 200 OK (HTML)
131	2025-04-13	15:16:10	883797	192.168.1.18	178.79.137.164	HTTP	469 GET //RE_server.cgi 200 OK (HTML)

Figura 18: Solicitudes GET en macOS

Las solicitudes GET fueron enviadas a dos direcciones IP diferentes. Las siguientes dos solicitudes:

- GET /wireshark.../...-file4.html
 - GET /pearson.png

fueron enviadas a la dirección 128.119.245.12, mientras que la solicitud:

- GET /8E_cover_small.jp

fue dirigida a la dirección 178.79.137.164

2. Can you tell whether your browser downloaded the two images serially, or whether they were downloaded from the two websites in parallel? Explain

En este ítem, inicialmente se obtuvo una interpretación distinta entre los miembros del grupo. Dos miembros concluyeron que las imágenes se descargaban en paralelo, mientras que otro consideró que la descarga era secuencial. Esta diferencia se basó en la observación de las solicitudes GET en los Wireshark de cada miembro, en el caso de dos miembros, el segundo GET para una imagen se realizaba antes de que finalizara la primera (Figura 20), lo que sugería una ejecución paralela. Sin embargo, en el caso de un miembro, la segunda solicitud solo se realizaba después de completarse la primera (Figura19). lo que indicaría un comportamiento secuencial.

	IP	Port	Protocol	Method	Path	HTTP Version	Response Code	Content Type	Content Length	File Extension	Request Headers
1+	138.20.18.27	673257835	18, 19, 224, 237		128.119.245.12	HTTP	486	GET /pearson.php	HTTP/1.1		
142	10.20.18.27	130.91840777	128, 119, 245, 12		10.100.224.237	HTTP	905	HTTP/1.1	200	(PNG)	
146	20.20.18.27	21.93714312	100, 18, 224, 237		178.79.137.164	HTTP	453	GET /BE_cover_small.jpg	HTTP/1.1		
152	10.20.18.28	231813401	178, 79, 137, 164		10.100.224.237	HTTP	237	HTTP/1.1	301	Moved Permanently	

Figura 19: *Solicitudes GET que parecían en paralelo*

Figura 20: *Solicitudes GET que parecían en secuencial*

No obstante, tras un análisis más detallado del tráfico TCP, se descubrió que todas las solicitudes se ejecutaban de manera secuencial. Específicamente, el navegador completaba por completo la primera solicitud antes de iniciar la siguiente. Si bien hacía la petición antes de mostrar la confirmación en nuestros navegadores, esta se realizaba después de completar todas sus acciones necesarias (Figura 21).

Figura 21: *Solicitudes TCP*

Por tanto, se concluyó que, en todos los casos, las solicitudes fueron realizadas de forma secuencial, y

la deducción de paralelismo se debió a la cercanía temporal entre el fin de una solicitud y el inicio de la siguiente.

II-E. HTTP Authentication

Para esta sección trabajamos con el siguiente link http://gaia.cs.umass.edu/wireshark-labs/protected_pages/HTTP-wireshark-file5.html y en este caso se uso credenciales para poder realizar la experiencia completa. Las credenciales utilizadas fueron:

- username: wireshark-students
- password: network

1. ***How many HTTP GET request messages did your browser send? To which Internet addresses were these GET requests sent?***

Al realizar la primera petición, todos los miembros del grupo recibieron un código de estado **401** acompañado del mensaje Unauthorized (Figura 22). Además, se abrió una alerta con dos campos de entrada de texto, solicitando las credenciales proporcionadas para acceder al recurso (Figura 23).

Figura 22: Respuesta 401 Unauthorized

Figura 23: Alerta de ingreso de credenciales

2. ***When your browser sends the HTTP GET message for the second time, what new field is included in the HTTP GET message?***

Al ingresar las credenciales respectivas, la página se actualiza y se realiza nuevamente la petición. En esta ocasión, se recibe un código de respuesta **200** con el mensaje OK (Figura 24). En la nueva solicitud, se incluye el campo Authorization, el cual contiene un tipo de token, así como un subcampo denominado Credentials, que almacena las credenciales ingresadas por el usuario (Figura 25).

Figura 24: Respuesta 200 OK

```
Cache-Control: max-age=0\r\n
Authorization: Basic d2lyZXNoYXJrLXN0dWRlbz0m5ldHdvcm=\r\n
Credentials: wireshark-students:network
```

Figura 25: Authorization

3. ***What's the content of this field? Now go to <http://www.motobit.com/util/base64-decoder-encoder.asp> and paste the content (without the "basic") of the field.***

El valor del token utilizado en el campo Authorization con el esquema Basic es: d2lyZXNoYXJrLXN0dWRlbz0m5ldHdvcm= (Figura 25).

Al ingresar este valor en la página indicada y seleccionar la opción *convert the source data*, se obtiene otra combinación de caracteres similar al token generado inicialmente: ZDJseVpYTm9ZWEpyTFhOMGRXUmxb1J6T201bGRIZHZjbXM9... Figura (26).

Figura 26: token convertido

4. ***Is the information above encrypted or encoded?***

Esta información está codificada, no encriptada. La diferencia principal radica en que la encriptación implica el uso de una clave para acceder a la información original, mientras que en este caso no se empleó ninguna llave. Por lo tanto, no se trata de un proceso de cifrado, sino únicamente de una codificación.

La codificación utilizada en este caso es Base64. La cual convierte los bytes del mensaje en caracteres ASCII imprimibles para facilitar su transmisión en protocolos como HTTP. Sin embargo, esta transformación no ofrece seguridad, ya que cualquier persona puede decodificar el contenido sin necesidad de una clave.

Además, dado que el token viaja sin encriptación, un atacante que intercepte el tráfico podría capturar y decodificar el valor del token con facilidad, exponiendo así las credenciales del usuario.

III. CONCLUSIONES

- En la sección II-A se pudo observar el funcionamiento básico del protocolo HTTP.. En primera instancia, pudimos ver que ambos operan con HTTP/1.1 y que la respuesta del servidor siempre viene con un código de respuesta, en este caso 200OK. También pudimos ver cuáles eran las ip's de nuestras máquinas y del servidor de respuesta. Con esta sección pudimos familiarizarnos un poco más con la forma en como el navegador solicita recursos a un servidor y como este responde.
- Con la sección II-B pudimos entender un poco más acerca del comportamiento eficiente del protocolo HTTP. Esto lo pudimos entender al ver como opera el formato “condicional” de una petición a un servidor usando el encabezado “IF_MODIFIED_SINCE”. Como pudimos ver, la primera vez que se hace la petición de un recurso este encabezado no aparece ya que el recurso aún no está en caché; sin embargo, en una 2da petición dicho encabezado si se encontraba con la última fecha de modificación lo que permitió al servidor no tener que enviar todo el recurso completo ya que ya se encontraba en caché. Esto muestra como el protocolo http maneja ciertas factores que permiten que la comunicación sea más eficiente.
- En la sección II-C se analizó como se comporta la transferencia de un archivo más grande y se pudo observar que esto sea posible se requieren más de 1 segmento TCP. Aquí obtuvimos un diferencia entre 2 sistemas operativos, mientras los sistemas con Linux requirieron de sólo 2 segmentos TCP, el sistema con macOS necesitó 4. Esta diferencia se atribuye a las variaciones en el valor de MSS (Maximum Segment Size) determinado por cada plataforma, como lo explica Huston (2019). Esto nos permitió darnos cuenta que cada sistema con sus propias configuraciones puede influir en el comportamiento de los protocolos de comunicación como TCP.
- En la sección II-D se trabajó con un documento HTML que contenía recursos secundarios, lo cual permitió analizar cómo el navegador realiza múltiples solicitudes HTTP GET en una sola carga de página. Inicialmente se observaron diferencias entre los miembros del grupo: los usuarios con Linux registraron 4 solicitudes GET, mientras que el usuario con macOS solo 3. Esta diferencia se debió a una petición adicional del archivo favicon.ico, la cual no es relevante para el contenido principal y fue descartada. Además, se identificó que las solicitudes GET se dirigían a dos direcciones IP distintas, lo que permitió entender cómo el navegador gestiona

múltiples conexiones. Por otro lado, al analizar si las imágenes se descargaban en paralelo o de manera secuencial, se concluyó tras una inspección del tráfico TCP que las solicitudes se ejecutaban de forma secuencial en todos los casos. Este análisis permitió reforzar la comprensión sobre la relación entre el protocolo HTTP y el comportamiento subyacente del protocolo TCP, así como la influencia del navegador y del sistema operativo en la ejecución de las solicitudes.

- En la sección II-E se exploró el funcionamiento del mecanismo de autenticación HTTP básico. Al acceder por primera vez al recurso protegido sin credenciales, todos los miembros del grupo recibieron un código de estado **401** con el mensaje Unauthorized 22, lo que generó una alerta emergente en el navegador solicitando nombre de usuario y contraseña (Figura 23). Una vez ingresadas las credenciales proporcionadas (wireshark-students / network), el navegador reenvió la solicitud GET, esta vez obteniendo una respuesta satisfactoria con el código **200 OK** (Figura 24). En esta nueva petición, se añadió un encabezado Authorization bajo el esquema Basic, el cual incluía un token en formato Base64. Este token, mostrado en la Figura 25, está compuesto por la concatenación del usuario y la contraseña en el formato usuario:contraseña. Posteriormente, al decodificar este token utilizando una herramienta en línea (como se muestra en la Figura 26), se evidenció que el contenido era simplemente el texto plano de las credenciales originales. Esto permitió concluir que la información fue codificada, no encriptada, ya que no se utilizó ninguna clave secreta para proteger el contenido. Este análisis resalta que la autenticación básica en HTTP, si bien funcional, no ofrece seguridad por sí sola; y que para proteger las credenciales transmitidas, es fundamental que se utilice en conjunto con protocolos seguros como HTTPS.

REFERENCIAS

- [1] Cloudflare, “¿Qué es el modelo OSI?”, *Cloudflare*. [En línea]. Disponible en: <https://www.cloudflare.com/es-es/learning/ddos/glossary/open-systems-interconnection-model-osi/> [Accedido: 15-abr-2025].
- [2] Proofpoint, “¿Qué es el modelo OSI? Definición, capas y más”, *Proofpoint*. [En línea]. Disponible en: <https://www.proofpoint.com/es/threat-reference/osi-model> [Accedido: 15-abr-2025].
- [3] MDN Web Docs, “Generalidades del protocolo HTTP”, *Mozilla*. [En línea]. Disponible en: <https://developer.mozilla.org/es/docs/Web/HTTP/Guides/Overview> [Accedido: 15-abr-2025].
- [4] C. Williams, *Internet Protocols*. Lima, Perú: Universidad de Ingeniería y Tecnología (UTEC), 2025. Basado en Kurose y Peterson & Davie.
- [5] Khan Academy, “Protocolo de control de transmisión (TCP)”, *Khan Academy*. [En línea]. Disponible en: <https://es.khanacademy.org/computing/ap-computer-science-principles/the-internet/xd2f703b37b450a3:transporting-packets/a/transmission-control-protocol--tcp> [Accedido: 15-abr-2025].

- [6] R. Altube, “Wireshark: Qué es y ejemplos de uso”, *OpenWebinars*, 07-ene-2021. [En línea]. Disponible en: <https://openwebinars.net/blog/wireshark-que-es-y-ejemplos-de-uso/> [Accedido: 15-abr-2025].
- [7] Garias, *Wireshark Presentation*, Lima, Perú: Universidad de Ingeniería y Tecnología (UTEC), Curso CS4054 – Redes y Comunicaciones, 2025.
- [8] Cloudflare, “HTTP/2 vs. HTTP/1.1: ¿Cuál es la diferencia?”, *Cloudflare*. [En línea]. Disponible en: <https://www.cloudflare.com/es-es/learning/performance/http2-vs-http1.1/> [Accedido: 15-abr-2025].
- [9] Cloudflare, “¿Qué es HTTPS?”, *Cloudflare*. [En línea]. Disponible en: <https://www.cloudflare.com/es-es/learning/ssl/what-is-https/> [Accedido: 15-abr-2025].
- [10] A. Hussain, A. Hussain, S. Qadri, A. Razzaq, H. Nazir, y M. S. Ullah, “Enhancing LAN security by mitigating credential threats via HTTP packet analysis with Wireshark”, *J. Comput. Biomed. Inform.*, vol. 6, no. 02, pp. 433–440, 2024. [En línea]. Disponible en: <https://jcbi.org/index.php/Main/article/view/417> [Accedido: 15-abr-2025].
- [11] I. O. Ibraheem, S. K. Kayode, y N. O. Ibrahim, “Analyzing vulnerabilities in network protocols using Wireshark: A case study on HTTP and HTTPS”, *Int. J. Adv. Res. Multidiscip. Stud.*, vol. 4, no. 1, pp. 737–744, 2024. [En línea]. Disponible en: https://www.researchgate.net/publication/385744283_ANALYZING_VULNERABILITIES_IN_NETWORK_PROTOCOLS_USING_WIRESHARK_A_CASE_STUDY_ON_HTTP_AND_HTTPS [Accedido: 15-abr-2025].
- [12] B. Dodiya y U. K. Singh, “Malicious traffic analysis using Wireshark by collection of Indicators of Compromise,” *Int. J. Comput. Appl.*,
- [13] Huston, G. (2019). “TCP MSS values – what's changed?” Blog de APNIC. Disponible en: <https://blog.apnic.net/2019/07/31/tcp-mss-values-whats-changed/>. Último acceso: 16 de abril de 2025.