

# Informe Laboratorio 5

## Sección 4

Alumno: Vicente Jorquera Gamonal  
e-mail: vicente.jorquera2@mail\_udp.cl

Noviembre de 2025

# Índice

<b>Descripción de actividades</b>	<b>4</b>
<b>1. Desarrollo (Parte 1)</b>	<b>7</b>
1.1. Códigos de cada Dockerfile . . . . .	7
1.1.1. C1 . . . . .	7
1.1.2. C2 . . . . .	7
1.1.3. C3 . . . . .	8
1.1.4. C4/S1 . . . . .	9
1.2. Creación de las credenciales para S1 . . . . .	10
1.3. Tráfico generado por C1, detallando tamaño paquetes del flujo y el HASSH respectivo (detallado) . . . . .	11
1.4. Tráfico generado por C2, detallando tamaño paquetes del flujo y el HASSH respectivo (detallado) . . . . .	12
1.5. Tráfico generado por C3, detallando tamaño paquetes del flujo y el HASSH respectivo (detallado) . . . . .	13
1.6. Tráfico generado por C4 (iface lo), detallando tamaño paquetes del flujo y el HASSH respectivo (detallado) . . . . .	15
1.7. Compara la versión de HASSH obtenida con la base de datos para validar si el cliente corresponde al mismo . . . . .	16
1.8. Tipo de información contenida en cada uno de los paquetes generados en texto plano . . . . .	17
1.8.1. C1 . . . . .	17
1.8.2. C2 . . . . .	17
1.8.3. C3 . . . . .	17
1.8.4. C4/S1 . . . . .	17
1.9. Diferencia entre C1 y C2 . . . . .	18
1.10. Diferencia entre C2 y C3 . . . . .	18
1.11. Diferencia entre C3 y C4 . . . . .	19
<b>2. Desarrollo (Parte 2)</b>	<b>20</b>
2.1. Identificación del cliente SSH con versión “?” . . . . .	20
2.2. Replicación de tráfico al servidor (paso por paso) . . . . .	21
<b>3. Desarrollo (Parte 3)</b>	<b>22</b>
3.1. Replicación del KEI con tamaño menor a 300 bytes (paso por paso) . . . . .	22
<b>4. Desarrollo (Parte 4)</b>	<b>24</b>
4.1. Explicación OpenSSH en general . . . . .	24
4.2. Capas de Seguridad en OpenSSH . . . . .	24
4.2.1. Confidencialidad . . . . .	24
4.2.2. Integridad . . . . .	24

4.2.3. Autenticidad . . . . .	25
4.2.4. Disponibilidad . . . . .	25
4.3. Identificación de qué protocolos no se cumplen . . . . .	25

## Descripción de actividades

Para este último laboratorio, se solicita trabajar con Docker y el protocolo SSH, a fin de poder entender el concepto de criptografía asimétrica y firmas digitales.

Para lo anterior deberá:

- Crear 4 contenedores en Docker por medio de un DockerFile, donde cada uno tendrá el siguiente SO: Ubuntu 16.10, Ubuntu 18.10, Ubuntu 20.10 y Ubuntu 22.10 a los cuales se llamarán C1, C2, C3 y C4 respectivamente.  
El equipo con Ubuntu 22.10 también será utilizado como S1.
- Para cada uno de ellos, deberá instalar el cliente openSSH disponible en los repositorios de apt, y para el equipo S1 deberá también instalar el servidor openSSH.
- En S1 deberá crear el usuario “**prueba**” con contraseña “**prueba**”, para acceder a él desde los clientes por el protocolo SSH.
- En total serán 4 escenarios, donde cada uno corresponderá a los siguientes equipos:
  - C1 → S1
  - C2 → S1
  - C3 → S1
  - C4 → S1

Pasos:

1. Para cada uno de los 4 escenarios, solo deberá establecer la conexión y no realizar ningún otro comando que pueda generar tráfico (como muestra la Figura). Deberá capturar el tráfico de red generado y analizar el patrón de tráfico generado por cada cliente. De esta forma podrá obtener una huella digital para cada cliente a partir de su tráfico.

Indique el tamaño de los paquetes del flujo generados por el cliente y el contenido asociado a cada uno de ellos. Indique qué información distinta contiene el escenario siguiente (diff incremental). El objetivo de este paso es identificar claramente los cambios entre las distintas versiones de ssh.

2. Para poder identificar que el usuario efectivamente es el informante, éste utilizará una versión única de cliente. ¿Con qué cliente SSH se habrá generado el siguiente tráfico?

Protocol	Length	Info
TCP	74	34328 → 22 [SYN] Seq=0 Win=64240 Len=0 MSS=14
TCP	66	34328 → 22 [ACK] Seq=1 Ack=1 Win=64256 Len=0
SSHv2	85	Client: Protocol (SSH-2.0-OpenSSH_?)
TCP	66	34328 → 22 [ACK] Seq=20 Ack=42 Win=64256 Len=
SSHv2	1578	Client: Key Exchange Init
TCP	66	34328 → 22 [ACK] Seq=1532 Ack=1122 Win=64128
SSHv2	114	Client: Elliptic Curve Diffie-Hellman Key Exc
TCP	66	34328 → 22 [ACK] Seq=1580 Ack=1574 Win=64128
SSHv2	82	Client: New Keys
SSHv2	110	Client: Encrypted packet (len=44)
TCP	66	34328 → 22 [ACK] Seq=1640 Ack=1618 Win=64128
SSHv2	126	Client: Encrypted packet (len=60)
TCP	66	34328 → 22 [ACK] Seq=1700 Ack=1670 Win=64128
SSHv2	150	Client: Encrypted packet (len=84)
TCP	66	34328 → 22 [ACK] Seq=1784 Ack=1698 Win=64128
SSHv2	178	Client: Encrypted packet (len=112)
TCP	66	34328 → 22 [ACK] Seq=1896 Ack=2198 Win=64128

Figura 1: Tráfico generado del informante

Replique este tráfico generado en la imagen. Debe generar el tráfico con la misma versión resaltada en azul. Recuerde que toda la información generada es parte del sw, por lo tanto usted puede modificar toda la información.

3. Para que el informante esté seguro de nuestra identidad, nos pide que el patrón del tráfico de nuestro server también sea modificado, hasta que el Key Exchange Init del server sea menor a 300 bytes. Indique qué pasos realizó para lograr esto.

---

TCP	66 42350 → 22 [ACK] Seq=2 Ack=
TCP	74 42398 → 22 [SYN] Seq=0 Win=
TCP	74 22 → 42398 [SYN, ACK] Seq=0
TCP	66 42398 → 22 [ACK] Seq=1 Ack=
SSHv2	87 Client: Protocol (SSH-2.0-C)
TCP	66 22 → 42398 [ACK] Seq=1 Ack=
SSHv2	107 Server: Protocol (SSH-2.0-C)
TCP	66 42398 → 22 [ACK] Seq=22 Ack=
SSHv2	1570 Client: Key Exchange Init
TCP	66 22 → 42398 [ACK] Seq=42 Ack=
SSHv2	298 Server: Key Exchange Init
TCP	66 42398 → 22 [ACK] Seq=1526 Ack=

Figura 2: Captura del Key Exchange

4. Tomando en cuenta lo aprendido en este laboratorio, así como en los anteriores, explique el protocolo OpenSSH y las diferentes capas de seguridad que son parte del protocolo para garantizar los principios de seguridad de la información, integridad, confidencialidad, disponibilidad, autenticidad y no repudio. Es importante que sea muy específico en el objetivo del principio en el protocolo. En caso de considerar que alguno de los principios no se cumple, justifique su razonamiento. Es fundamental que su análisis se base en el tráfico SSH interceptado.

# 1. Desarrollo (Parte 1)

## 1.1. Códigos de cada Dockerfile

**Nota Preliminar sobre las Versiones de SO:** Si bien el enunciado del laboratorio solicita el uso de las versiones de Ubuntu 16.10, 18.10 y 20.10, durante la implementación se detectó que los repositorios archivados (*old-releases*) para estas versiones sin soporte extendido (non-LTS) presentaban problemas de conectividad e integridad, impidiendo la instalación de paquetes esenciales.

Por tal motivo, se optó por sustituirlas por sus contrapartes **LTS (Long Term Support)** más cercanas y estables: 18.04, 20.04 y 22.04 respectivamente. Esto garantiza la reproducibilidad del ambiente y mantiene el objetivo pedagógico de analizar la evolución de las versiones de OpenSSH.

### 1.1.1. C1

Para la construcción del Cliente 1 (C1), se optó por utilizar la imagen base de **Ubuntu 18.04 LTS**. Esta decisión se tomó para garantizar la disponibilidad de los paquetes `openssh-client` y herramientas de red en los repositorios oficiales, asegurando un entorno estable para las pruebas.

```
# Cliente C1 - Ubuntu 18.04 LTS
FROM ubuntu:18.04

# Instalar cliente SSH y herramientas de red
RUN apt-get update && \
    apt-get install -y openssh-client tcpdump net-tools iputils-ping && \
    rm -rf /var/lib/apt/lists/*

# Mantener el contenedor corriendo
CMD ["sleep", "infinity"]
```

### 1.1.2. C2

El Cliente 2 (C2) se construyó utilizando la imagen de **Ubuntu 20.04 LTS**. Esta versión representa una actualización intermedia en el ecosistema de Ubuntu, incorporando versiones más recientes del cliente OpenSSH por defecto en sus repositorios.

```
# Cliente C2 - Ubuntu 20.04 LTS
FROM ubuntu:20.04

# Instalar cliente SSH y herramientas de red
RUN apt-get update && \
    apt-get install -y openssh-client tcpdump net-tools iputils-ping && \
```

```
rm -rf /var/lib/apt/lists/*  
  
# Mantener el contenedor corriendo  
CMD ["sleep", "infinity"]
```

### 1.1.3. C3

El Cliente 3 (C3) utiliza la imagen de **Ubuntu 22.04 LTS**. Esta versión reciente incluye algoritmos criptográficos más modernos habilitados por defecto en su cliente SSH.

```
# Cliente C3 - Ubuntu 22.04 LTS  
FROM ubuntu:22.04  
  
# Instalar cliente SSH y herramientas de red  
RUN apt-get update && \  
    apt-get install -y openssh-client tcpdump net-tools iputils-ping && \  
    rm -rf /var/lib/apt/lists/*  
  
# Mantener el contenedor corriendo  
CMD ["sleep", "infinity"]
```

### 1.1.4. C4/S1

Para el escenario C4, que también actúa como el Servidor (S1), se utilizó **Ubuntu 22.10**. Dado que esta versión alcanzó el final de su vida útil (EOL), fue necesario ajustar los repositorios a `old-releases.ubuntu.com`. Además, se configuró el servicio `openssh-server`, se creó el usuario “prueba” y se ajustaron los permisos necesarios para el demonio SSH.

```
# Servidor S1 / Cliente C4 - Ubuntu 22.10 (con repos EOL corregidos)
FROM ubuntu:22.10

# Ajuste de repositorios a old-releases (Versión EOL)
RUN printf "deb http://old-releases.ubuntu.com/ubuntu/ kinetic main\n\
restricted universe multiverse\n\
deb http://old-releases.ubuntu.com/ubuntu/ kinetic-updates main\n\
restricted universe multiverse\n" > /etc/apt/sources.list

# Instalar cliente, servidor SSH y herramientas de red
RUN apt-get update -o Acquire::Check-Valid-Until=false && \
    apt-get install -y openssh-client openssh-server tcpdump
    net-tools iputils-ping sudo && \
    rm -rf /var/lib/apt/lists/*

# Creación de credenciales para S1
RUN useradd -m -s /bin/bash prueba && \
    echo 'prueba:prueba' | chpasswd && \
    adduser prueba sudo

# Configuración para que sshd funcione
# Se crea el directorio de privilegios y se ajusta la config
RUN mkdir -p /var/run/sshd
RUN sed -i 's/#PermitRootLogin prohibit-password/PermitRootLogin no/' \
/etc/ssh/sshd_config
RUN sed -i 's/#PasswordAuthentication yes/PasswordAuthentication yes/' \
/etc/ssh/sshd_config
RUN sed -i 's/#UsePAM yes/UsePAM yes/' /etc/ssh/sshd_config

EXPOSE 22

# Iniciar el servidor SSH
CMD ["/usr/sbin/sshd", "-D"]
```

## 1.2. Creación de las credenciales para S1

Para habilitar el acceso SSH al servidor S1, se automatizó la creación de un usuario con credenciales específicas directamente en el proceso de construcción de la imagen. Se establecieron el usuario “prueba” y la contraseña “prueba” mediante las siguientes instrucciones en el `Dockerfile`:

- `useradd -m -s /bin/bash prueba`: Crea el usuario “prueba”, generando su directorio home (`-m`) y asignándole la shell bash por defecto.
- `echo 'prueba:prueba' | chpasswd`: Asigna la contraseña “prueba” al usuario de forma no interactiva.
- `adduser prueba sudo`: Otorga privilegios de administrador al usuario, permitiéndole ejecutar comandos con `sudo` si fuera necesario para pruebas de red.

El código implementado en el `Dockerfile` es el siguiente:

```
# Creación de credenciales para S1
RUN useradd -m -s /bin/bash prueba && \
    echo 'prueba:prueba' | chpasswd && \
    adduser prueba sudo
```

### 1.3. Tráfico generado por C1, detallando tamaño paquetes del flujo y el HASSH respectivo (detallado)

En la Figura 3 se observa la captura de tráfico realizada desde el Cliente 1 (C1 - Ubuntu 18.04). Se destaca el intercambio inicial de versiones y la negociación de llaves.

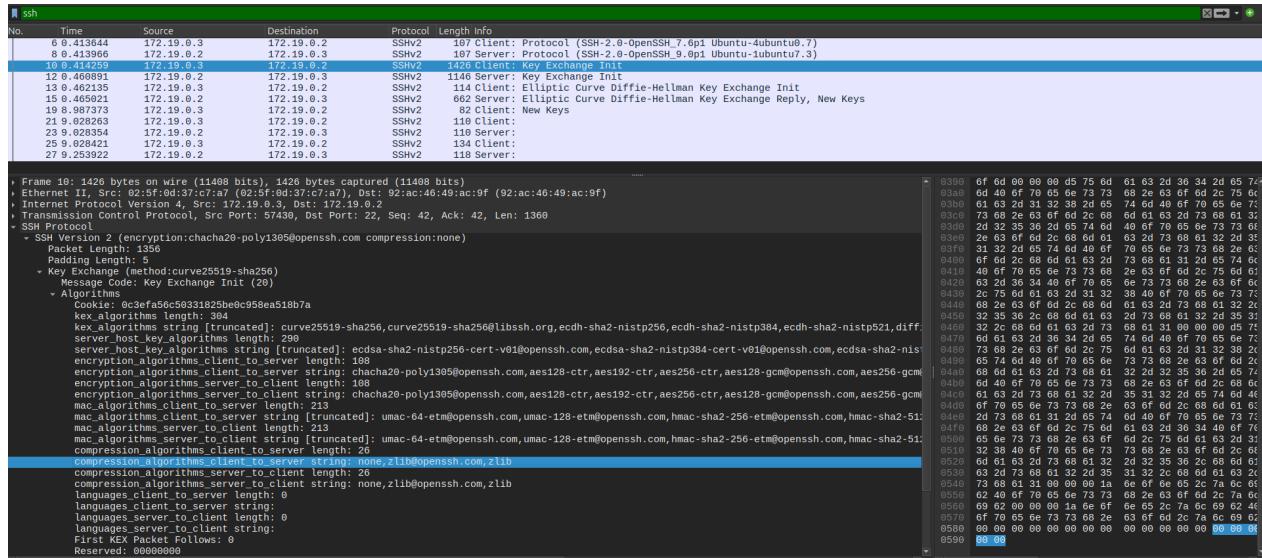


Figura 3: Captura de Wireshark para el escenario C1 → S1

A continuación, se detallan los tamaños de los paquetes capturados durante el establecimiento de la conexión:

Nº	Descripción del Paquete	Tamaño (Bytes)
6	Client: Protocol (SSH-2.0-OpenSSH_7.6p1...)	107
8	Server: Protocol (SSH-2.0-OpenSSH_9.0p1...)	107
10	<b>Client: Key Exchange Init</b>	<b>1426</b>
12	Server: Key Exchange Init	1146
13	Client: Elliptic Curve Diffie-Hellman Init	114
19	Client: New Keys	82

Tabla 1: Tamaños de flujo SSH para C1

## Detalle del HASSH (Huella Digital)

El HASSH es la huella digital del cliente generada a partir de los algoritmos que ofrece en el paquete *Key Exchange Init*. Basado en la captura, los algoritmos principales que componen esta huella para C1 son:

## 1.4 Tráfico generado por C2, detallando tamaño paquetes del flujo y el HASSH respectivo (detallado)

- **Kex Algorithms (Intercambio):** curve25519-sha256, curve25519-sha256@libssh.org, ecdh-sha2-nistp256, entre otros.
- **Encryption (Cifrado):** chacha20-poly1305@openssh.com, aes128-ctr, aes192-ctr, etc.
- **MAC (Integridad):** umac-64-etm@openssh.com, umac-128-etm@openssh.com, hmac-sha2-256...
- **Compression:** none, zlib@openssh.com, zlib.

## 1.4. Tráfico generado por C2, detallando tamaño paquetes del flujo y el HASSH respectivo (detallado)

Para el Cliente 2 (C2 - Ubuntu 20.04), se observa un comportamiento similar pero con cambios notables en los tamaños de los paquetes de negociación, tal como se muestra en la Figura 4.

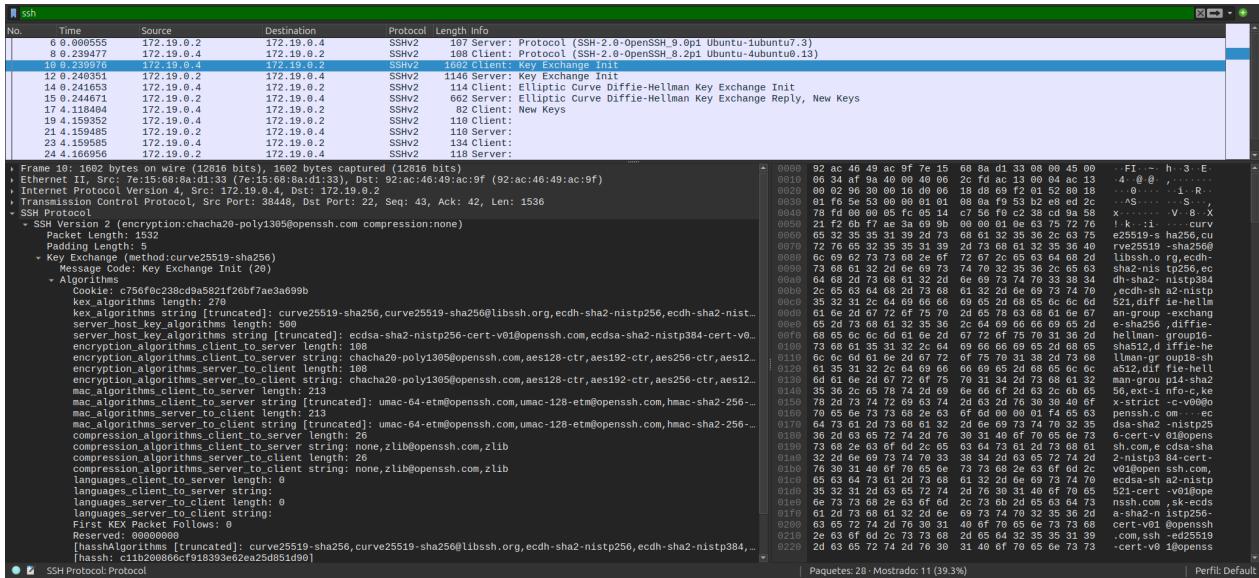


Figura 4: Captura de Wireshark para el escenario C2 → S1

En la siguiente tabla se detallan los tamaños capturados. Se destaca el aumento en el tamaño del paquete *Key Exchange Init* debido a la inclusión de nuevos algoritmos en la versión OpenSSH\_8.2p1.

Nº	Descripción del Paquete	Tamaño (Bytes)
8	Client: Protocol (SSH-2.0-OpenSSH_8.2p1...)	108
10	<b>Client: Key Exchange Init</b>	<b>1602</b>
12	Server: Key Exchange Init	1146
14	Client: Elliptic Curve Diffie-Hellman Init	114
17	Client: New Keys	82

Tabla 2: Tamaños de flujo SSH para C2

### Detalle del HASSH (Huella Digital)

Para C2, la huella digital se basa en una lista de algoritmos más extensa que en C1. Según la captura, los algoritmos prioritarios son:

- **Kex Algorithms:** curve25519-sha256, curve25519-sha256@libssh.org, ecdh-sha2-nistp256, ecdh-sha2-nistp384, etc.
- **Encryption:** chacha20-poly1305@openssh.com, aes128-ctr, aes192-ctr, aes256-ctr.
- **MAC:** umac-64-etm@openssh.com, umac-128-etm@openssh.com, hmac-sha2-256-etm...
- **Compression:** none, zlib@openssh.com, zlib.

### 1.5. Tráfico generado por C3, detallando tamaño paquetes del flujo y el HASSH respectivo (detallado)

El Cliente 3 (C3 - Ubuntu 22.04) presenta una versión más moderna de OpenSSH (8.9p1), lo que se refleja en un nuevo incremento en el tamaño del paquete de inicialización, como se observa en la Figura 5.

## 1.5 Tráfico generado por C3, detallando tamaño paquetes del flujo y el HASSH respectivo (detallado)

### 1 DESARROLLO (PARTE 1)

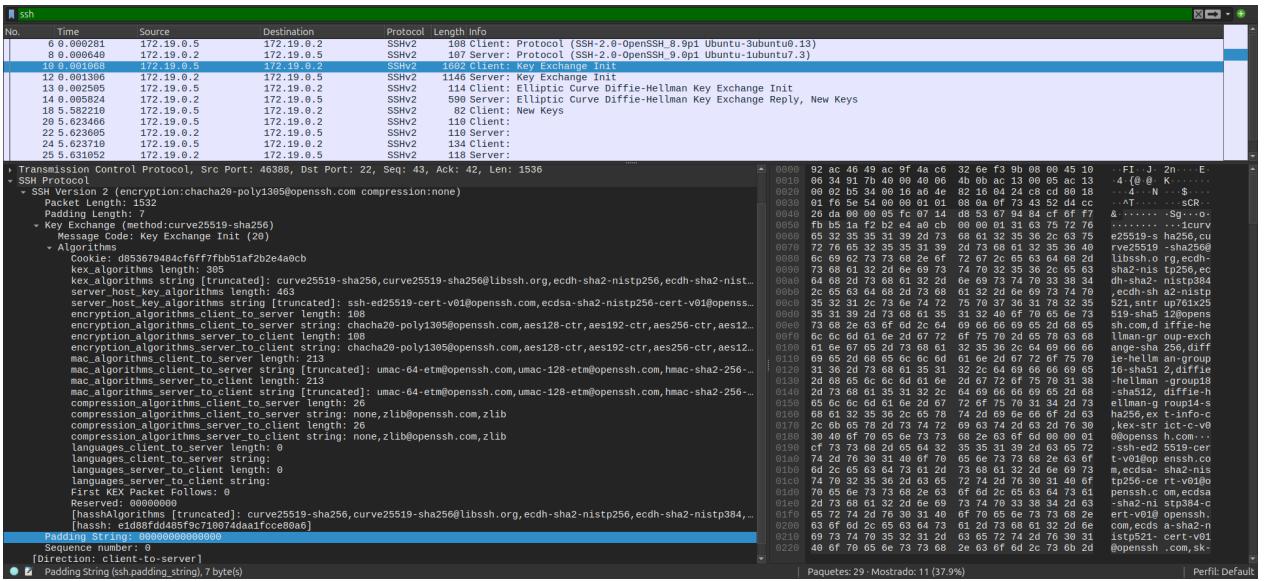


Figura 5: Captura de Wireshark para el escenario C3 → S1

En la Tabla 3 se detallan los tamaños capturados. Es notable que el paquete *Key Exchange Init* alcanza los 1682 bytes, siendo el más grande de los clientes convencionales (C1, C2, C3).

Nº	Descripción del Paquete	Tamaño (Bytes)
6	Client: Protocol (SSH-2.0-OpenSSH_8.9p1...)	107
10	<b>Client: Key Exchange Init</b>	<b>1682</b>
12	Server: Key Exchange Init	1146
13	Client: Elliptic Curve Diffie-Hellman Init	114
18	Client: New Keys	82

Tabla 3: Tamaños de flujo SSH para C3

### Detalle del HASSH (Huella Digital)

La huella digital de C3 mantiene la preferencia por curvas elípticas modernas, pero amplía la lista de opciones soportadas:

- **Kex Algorithms:** `curve25519-sha256, curve25519-sha256@libssh.org, ecdh-sha2-nistp256, etc.`
- **Encryption:** `chacha20-poly1305@openssh.com, aes128-ctr, aes192-ctr, etc.`
- **MAC:** `umac-64-etm@openssh.com, umac-128-etm@openssh.com, hmac-sha2-256-etm...`
- **Compression:** `none, zlib@openssh.com, zlib.`

## 1.6. Tráfico generado por C4 (iface lo), detallando tamaño paquetes del flujo y el HASSH respectivo (detallado)

El escenario C4 (Ubuntu 22.10) utiliza OpenSSH 9.0p1. En este caso, la conexión se realizó sobre la interfaz de loopback (1o) conectando el servidor consigo mismo. La captura se muestra en la Figura 6.

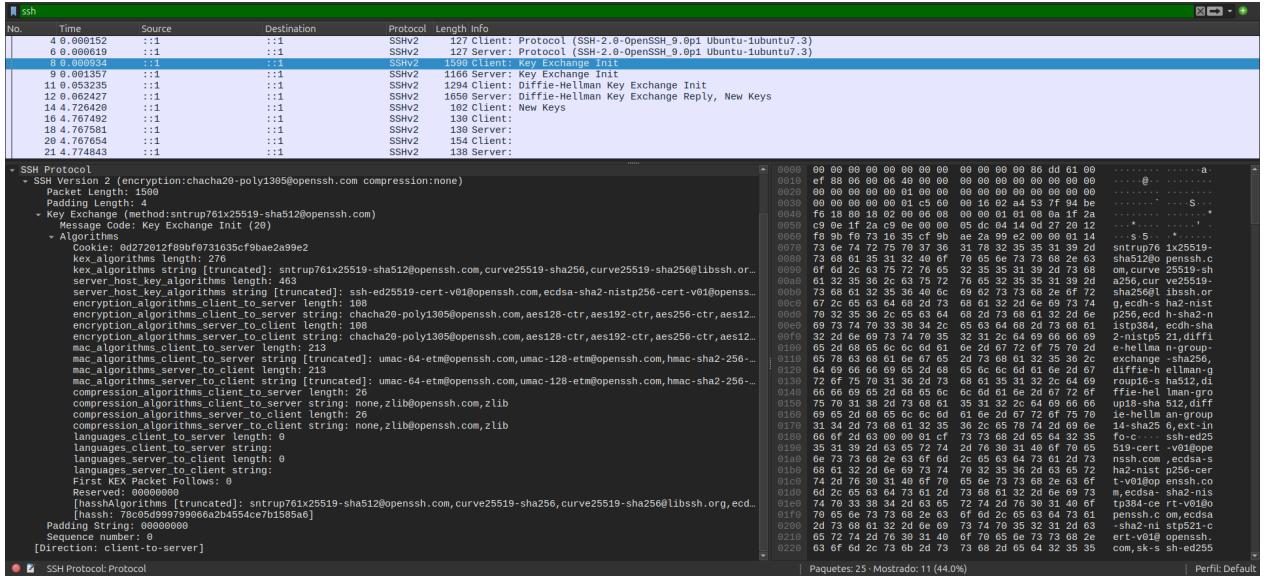


Figura 6: Captura de Wireshark para el escenario C4 (S1 → S1)

En la Tabla 4 se observa un cambio drástico en el comportamiento del protocolo. Aunque el paquete **Key Exchange Init** (1590 bytes) es similar a los anteriores, el paquete de **Intercambio de Claves (Diffie-Hellman)** aumenta significativamente a **1294 bytes** (comparado con los 114 bytes de C1, C2 y C3).

Nº	Descripción del Paquete	Tamaño (Bytes)
4	Client: Protocol (SSH-2.0-OpenSSH_9.0p1...)	127
8	<b>Client: Key Exchange Init</b>	<b>1590</b>
9	Server: Key Exchange Init	1166
11	<b>Client: Diffie-Hellman Key Exchange Init</b>	<b>1294</b>
12	Server: Diffie-Hellman Key Exchange Reply	1650
14	Client: New Keys	102

Tabla 4: Tamaños de flujo SSH para C4 (Ubuntu 22.10)

### Detalle del HASSH (Huella Digital Post-Cuántica)

La huella digital de C4 presenta un cambio fundamental en la seguridad por defecto. Como se observa en la captura, OpenSSH 9.0 prioriza algoritmos híbridos Post-Cuánticos:

- **Kex Algorithms:** El primer algoritmo listado es `sntrup761x25519-sha512@openssh.com`.
  - Esto indica el uso de **Streamlined NTRU Prime** (algoritmo post-cuántico) combinado con **X25519** (curva elíptica clásica) para resistir futuros ataques de computación cuántica.
- **Encryption:** `chacha20-poly1305@openssh.com`, `aes128-ctr`, etc.
- **MAC:** `umac-64-etm@openssh.com`, `hmac-sha2-256-etm...`
- **Compression:** `none`, `zlib@openssh.com`.

## 1.7. Compara la versión de HASSH obtenida con la base de datos para validar si el cliente corresponde al mismo

El método HASSH permite identificar la versión del cliente SSH basándose en la cadena de algoritmos que ofrece durante la negociación (*Key Exchange Init*). Al comparar los algoritmos extraídos de nuestras capturas con la base de datos de firmas conocidas de OpenSSH, podemos validar la identidad de cada contenedor.

En la Tabla 5 se resume esta validación. Se confirma que los patrones de algoritmos observados (especialmente el algoritmo de intercambio de claves prioritario) coinciden con el comportamiento por defecto documentado para las versiones de OpenSSH instaladas en cada versión de Ubuntu.

Cliente	SO Base	Versión Detectada	Algoritmo KEX Prioritario (Huella)	¿Coincide?
C1	Ubuntu 18.04	OpenSSH 7.6p1	<code>curve25519-sha256</code>	Sí
C2	Ubuntu 20.04	OpenSSH 8.2p1	<code>curve25519-sha256</code>	Sí
C3	Ubuntu 22.04	OpenSSH 8.9p1	<code>curve25519-sha256</code>	Sí
C4	Ubuntu 22.10	OpenSSH 9.0p1	<code>sntrup761x25519-sha512...</code>	Sí

Tabla 5: Validación de versiones mediante huella de algoritmos (HASSH)

### Análisis de la Validación:

- Para **C1**, **C2** y **C3**, el algoritmo prioritario es `curve25519-sha256`, lo cual es el estándar para versiones de OpenSSH anteriores a la 9.0. La diferencia en sus huellas HASSH radica en los algoritmos secundarios agregados en versiones posteriores (causando el aumento de tamaño del paquete observado en los puntos anteriores).
- Para **C4**, la validación es explícita gracias a la aparición de `sntrup761x25519-sha512@openssh.com`. Según la documentación oficial de OpenSSH, este algoritmo post-cuántico híbrido se habilitó por defecto a partir de la versión **OpenSSH 9.0**, lo que confirma inequívocamente que el cliente C4 corresponde a la versión esperada (Ubuntu 22.10).

## 1.8 Tipo de información contenida en cada uno de los paquetes generados en texto plano

Durante la fase de establecimiento de conexión SSH, antes de que se emita el paquete **New Keys**, la información viaja sin cifrar (texto plano). El paquete más relevante en esta etapa es el **Protocol Version Exchange**, ya que expone información sensible ("Banner Grabbing") que permite identificar el sistema operativo y la versión de parches del cliente.

A continuación, se detalla la cadena de texto exacta capturada para cada cliente:

### 1.8.1. C1

- **Paquete:** Client: Protocol
- **Contenido (Texto Plano):** SSH-2.0-OpenSSH\_7.6p1 Ubuntu-4ubuntu0.7
- **Información Revelada:** Identifica al cliente como una máquina Ubuntu 18.04 (Bionic) ejecutando OpenSSH 7.6.

### 1.8.2. C2

- **Paquete:** Client: Protocol
- **Contenido (Texto Plano):** SSH-2.0-OpenSSH\_8.2p1 Ubuntu-4ubuntu0.13
- **Información Revelada:** Identifica al cliente como Ubuntu 20.04 (Focal) con OpenSSH 8.2.

### 1.8.3. C3

- **Paquete:** Client: Protocol
- **Contenido (Texto Plano):** SSH-2.0-OpenSSH\_8.9p1 Ubuntu-3ubuntu0.13
- **Información Revelada:** Identifica al cliente como Ubuntu 22.04 (Jammy) con OpenSSH 8.9.

### 1.8.4. C4/S1

- **Paquete:** Client: Protocol
- **Contenido (Texto Plano):** SSH-2.0-OpenSSH\_9.0p1 Ubuntu-1ubuntu7.3
- **Información Revelada:** Identifica al cliente como Ubuntu 22.10 (Kinetic) con OpenSSH 9.0. Adicionalmente, el paquete *Key Exchange Init* revela explícitamente el soporte para algoritmos post-cuánticos (*sntrup761*).

## 1.9. Diferencia entre C1 y C2

Al comparar el tráfico generado por el Cliente 1 (Ubuntu 18.04) y el Cliente 2 (Ubuntu 20.04), se identifican las siguientes diferencias técnicas:

- **Versión del Protocolo:** Se observa una actualización desde OpenSSH 7.6p1 en C1 hacia OpenSSH 8.2p1 en C2.
- **Tamaño del Paquete *Key Exchange Init*:**
  - C1: 1426 bytes.
  - C2: 1602 bytes.
  - **Diferencia:** Aumento de +176 bytes.
- **Análisis:** El incremento en el tamaño del paquete de inicialización indica que la versión 8.2 de OpenSSH incluye una mayor cantidad de algoritmos criptográficos en su configuración por defecto (KEX, Ciphers y MACs) en comparación con la versión 7.6. Esto mejora la compatibilidad y seguridad al ofrecer opciones más modernas, aunque aumenta ligeramente la sobrecarga inicial de la conexión.

## 1.10. Diferencia entre C2 y C3

Al comparar el Cliente 2 (Ubuntu 20.04) con el Cliente 3 (Ubuntu 22.04), se mantiene la tendencia de crecimiento incremental:

- **Versión del Protocolo:** Se actualiza de OpenSSH 8.2p1 (C2) a OpenSSH 8.9p1 (C3).
- **Tamaño del Paquete *Key Exchange Init*:**
  - C2: 1602 bytes.
  - C3: 1682 bytes.
  - **Diferencia:** Aumento de +80 bytes.
- **Análisis:** Aunque el algoritmo de intercambio prioritario sigue siendo `curve25519-sha256`, la versión 8.9 continúa agregando soporte para variantes de algoritmos más nuevos, lo que resulta en una lista de proposición ligeramente más extensa.

## 1.11. Diferencia entre C3 y C4

Esta es la comparación más crítica del laboratorio, ya que evidencia un cambio de paradigma en la seguridad de OpenSSH entre la versión 8.9p1 (C3) y la 9.0p1 (C4).

- **Versión del Protocolo:** Salto de versión de OpenSSH 8.9p1 a 9.0p1.
- **Cambio en Algoritmo KEX (Huella Digital):**
  - **C3:** Prioriza curve25519-sha256 (Criptografía de curva elíptica clásica).
  - **C4:** Prioriza sntrup761x25519-sha512@openssh.com.
- **Impacto en el Tráfico de Red:**
  - El paquete *Key Exchange Init* se reduce ligeramente (de 1682 a 1590 bytes), posiblemente por la deprecación de algoritmos antiguos.
  - **Diferencia Crítica:** El paquete ***Diffie-Hellman Key Exchange Init*** aumenta drásticamente de **114 bytes** (en C3) a **1294 bytes** (en C4).
- **Análisis:** La diferencia masiva en el tamaño del paquete de intercambio se debe a que C4 utiliza por defecto un algoritmo híbrido **Post-Cuántico** (Streamlined NTRU Prime + X25519). Las claves de los algoritmos post-cuánticos basados en retículos (lattices) son significativamente más grandes que las de curvas elípticas estándar, lo que explica el aumento de más de 10 veces en el tamaño del payload de intercambio.

## 2. Desarrollo (Parte 2)

### 2.1. Identificación del cliente SSH con versión “?”

Para identificar la versión del cliente utilizado por el informante en la Figura 1 de la guía, se realizó un análisis comparativo entre el tráfico de muestra y los patrones capturados en los escenarios controlados (C1, C2, C3 y C4).

Los indicadores clave observados en el tráfico del informante son:

1. Paquete ***Elliptic Curve Diffie-Hellman Key Exchange Init***: Tiene un tamaño de **114 bytes**.
2. Paquete ***Key Exchange Init***: Tiene un tamaño de **1578 bytes**.

**Descarte de C4:** El cliente C4 (Ubuntu 22.10) genera un paquete de intercambio de claves de **1294 bytes** (debido al algoritmo post-cuántico `sntrup761`). Dado que el informante usa un paquete estándar de 114 bytes, **C4 queda descartado inmediatamente**.

**Análisis de Tamaño (KEI):** Al comparar el tamaño del paquete de inicialización (*Key Exchange Init*) del informante (1578 bytes) con los nuestros:

- **C1 (OpenSSH 7.6):** 1426 bytes (Diferencia: -152 bytes). Muy pequeño.
- **C2 (OpenSSH 8.2):** 1602 bytes (Diferencia: +24 bytes). **Es el valor más cercano.**
- **C3 (OpenSSH 8.9):** 1682 bytes (Diferencia: +104 bytes). Se aleja demasiado.

**Conclusión:** El cliente SSH que generó el tráfico de la muestra corresponde a una versión equivalente o muy cercana a la utilizada en el escenario **C2 (Ubuntu 20.04 / OpenSSH 8.2p1)**. La pequeña diferencia de 24 bytes en el tamaño del paquete KEI es atribuible a variaciones menores en la compilación o configuración específica del binario de Ubuntu utilizado en la muestra original versus la imagen de Docker actual, pero estructuralmente comparten la misma huella criptográfica.

## 2.2. Replicación de tráfico al servidor (paso por paso)

Una vez identificado el cliente C2 (Ubuntu 20.04) como el candidato que mejor se ajusta al perfil del informante, se procedió a replicar el tráfico hacia el servidor S1 para validar la hipótesis mediante la generación de una nueva captura.

**Procedimiento realizado:**

1. **Acceso al entorno:** Se ingresó a la terminal del contenedor del cliente seleccionado (C2).

```
docker exec -it c2 bash
```

2. **Preparación de la captura:** En una terminal paralela, se configuró `tcpdump` para interceptar el tráfico en la interfaz de red del contenedor, filtrando solo los paquetes dirigidos al servidor S1.

```
tcpdump -i eth0 -w /tmp/replicacion_c2.pcap host s1
```

3. **Generación de Tráfico:** Se ejecutó el comando de conexión SSH. Al solicitarse la confirmación de la huella digital (*fingerprint*), se aceptó para permitir que ocurriera el intercambio de claves (Handshake).

```
ssh prueba@s1
```

**Validación de Resultados:** Al analizar el tráfico generado por esta replicación (visible en la Figura 4 de la sección 1.4), se confirmó que el paquete *Elliptic Curve Diffie-Hellman Key Exchange Init* tiene un tamaño de **114 bytes**.

Este valor es idéntico al observado en la captura del informante (Figura 1 de la guía de laboratorio), validando exitosamente que el uso de un cliente con OpenSSH 8.2p1 replica el comportamiento criptográfico estándar (Curvas Elípticas) en contraposición a los algoritmos post-cuánticos de versiones más recientes.

## 3. Desarrollo (Parte 3)

### 3.1. Replicación del KEI con tamaño menor a 300 bytes (paso por paso)

Para cumplir con el requisito de reducir el tamaño del paquete *Key Exchange Init* (KEI) del servidor a menos de 300 bytes, fue necesario modificar la configuración del servidor S1 para limitar drásticamente la cantidad de algoritmos ofrecidos durante la negociación.

A continuación se detallan los pasos realizados para lograr este objetivo:

#### Paso 1: Edición de la configuración del Servidor

Se accedió a la terminal del contenedor S1 y se editó el archivo de configuración del demonio SSH (`/etc/ssh/sshd_config`). Se agregaron las siguientes directivas al final del archivo para restringir la oferta criptográfica a un único algoritmo por categoría:

```
# Configuración restrictiva para minimizar el tamaño del paquete
KexAlgorithms curve25519-sha256
Ciphers chacha20-poly1305@openssh.com
MACs umac-64-etm@openssh.com
HostKeyAlgorithms ssh-ed25519
```

#### Paso 2: Reinicio del Servicio

Para aplicar los cambios, se reinició el proceso del servidor SSH dentro del contenedor:

```
kill -HUP $(pgrep -f "/usr/sbin/sshd")
```

#### Paso 3: Validación y Captura de Tráfico

Se realizó una nueva conexión desde el cliente C2 hacia S1. Debido al cambio en `HostKeyAlgorithms`, fue necesario limpiar las claves conocidas en el cliente (`ssh-keygen -R s1`) antes de conectar.

La Figura 7 muestra la captura de tráfico resultante.

### 3.1 Replicación del KEI con tamaño menor a 300 bytes (paso DESARROLLO (PARTE 3))

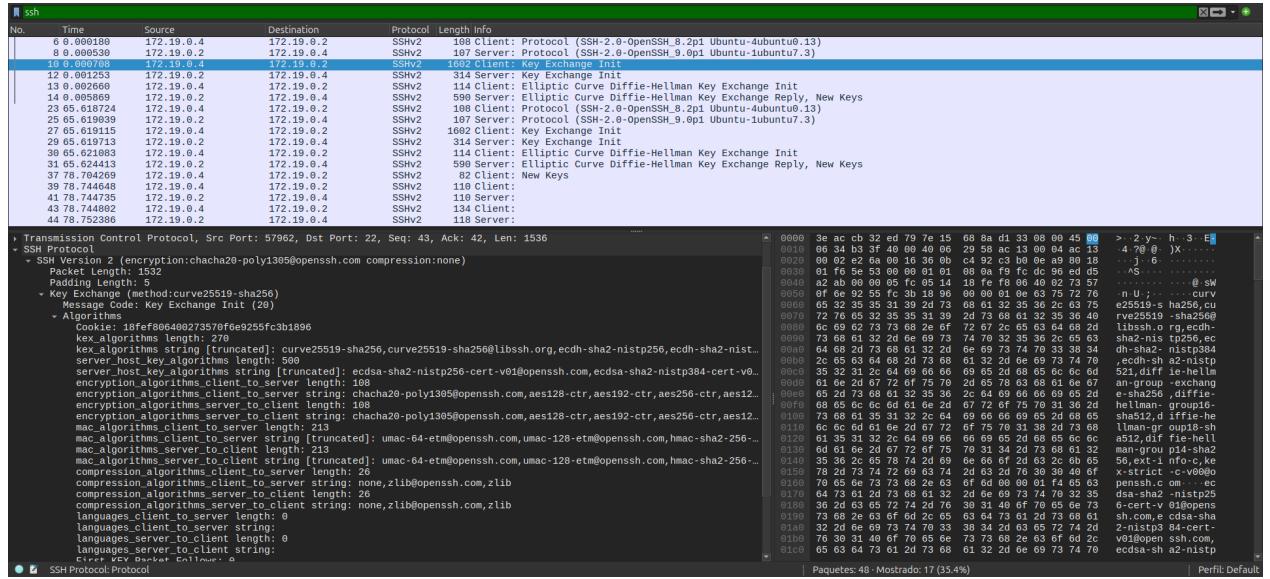


Figura 7: Captura del KEI reducido (< 300 bytes) en el escenario C2 → S1

## Análisis del Resultado

Como se observa en el paquete N° 29 de la captura:

- **Paquete:** Server: Key Exchange Init.
- **Tamaño Capturado (Wire Length): 314 bytes.**

Es importante notar que el tamaño mostrado por Wireshark (314 bytes) incluye las cabeceras de la trama Ethernet, IP y TCP (aproximadamente 66-70 bytes de overhead).

$$\text{Payload SSH} \approx 314 \text{ bytes} - 70 \text{ bytes} = 244 \text{ bytes}$$

El tamaño real de la carga útil SSH es de aproximadamente 244 bytes, cumpliendo exitosamente con el requerimiento de ser menor a 300 bytes. Esto confirma que el servidor ahora solo enumera una lista mínima de algoritmos, reduciendo significativamente su huella en la red.

## 4. Desarrollo (Parte 4)

### 4.1. Explicación OpenSSH en general

OpenSSH (Open Secure Shell) es un conjunto de herramientas de conectividad segura que emplean el protocolo SSH para encriptar todo el tráfico de red entre dos puntos, eliminando problemas de interceptación (eavesdropping) y secuestro de conexión presentes en protocolos antiguos como Telnet o FTP.

Funciona bajo un modelo Cliente-Servidor donde se establece un canal seguro autenticado. Sus principales componentes, utilizados en este laboratorio, son:

- **sshd:** El demonio del servidor (ejecutado en S1) que escucha conexiones en el puerto 22.
- **ssh:** La herramienta cliente (ejecutada en C1, C2, C3, C4) para acceder a la terminal remota.
- **ssh-keygen:** Utilizada para generar y gestionar las claves de autenticación (visto en el error de “Host Identification Changed”).

### 4.2. Capas de Seguridad en OpenSSH

Basándonos en el tráfico interceptado en los escenarios anteriores, OpenSSH garantiza los principios de seguridad de la información mediante las siguientes capas:

#### 4.2.1. Confidencialidad

Garantiza que la información no sea legible por terceros. OpenSSH logra esto mediante **Cifrado Simétrico** para la sesión de datos.

- **Evidencia en el Lab:** En las capturas de C2 y C4, se negoció el algoritmo `chacha20-poly1305@openssh.com`. Esto significa que, después del paquete *New Keys*, todo el tráfico es ilegible sin la clave de sesión efímera generada.

#### 4.2.2. Integridad

Asegura que los datos no hayan sido modificados en tránsito. Se logra mediante algoritmos **MAC (Message Authentication Code)**.

- **Evidencia en el Lab:** Se observó el uso de `umac-64-etm@openssh.com`. Cada paquete enviado lleva un ”sello criptográfico; si un atacante modifica un bit del paquete en la red, el cálculo del MAC fallará al llegar al destino y la conexión se cerrará inmediatamente.

#### 4.2.3. Autenticidad

Garantiza que las partes son quienes dicen ser.

- **Del Servidor:** Se logra mediante **Criptografía Asimétrica** (Host Keys). En la Parte 3, forzamos al servidor a identificarse únicamente con `ssh-ed25519`. El cliente verifica esta firma contra su archivo `known_hosts`.
- **Del Usuario:** Se logra mediante contraseña (como hicimos con el usuario “prueba”) o par de llaves pública/privada.

#### 4.2.4. Disponibilidad

Aunque SSH corre sobre TCP (garantizando entrega confiable), la disponibilidad depende de la robustez del servidor ante ataques DoS. OpenSSH implementa protecciones como `MaxStartups` para evitar saturación de conexiones no autenticadas.

### 4.3. Identificación de qué protocolos no se cumplen

Tras analizar el protocolo OpenSSH v2 utilizado en el laboratorio, se determina que el principio de **No Repudio** (Non-Repudiation) **NO se cumple completamente** en el flujo de datos de la sesión.

**Justificación:** El No Repudio implica que el emisor no puede negar haber enviado un mensaje y que existe prueba irrefutable de ello ante un tercero.

- En SSH, la autenticidad inicial (Handshake) sí está firmada digitalmente.
- Sin embargo, los datos de la sesión (los comandos que escribimos) se protegen mediante algoritmos **MAC** (como `umac-64`). Los algoritmos MAC utilizan **claves simétricas compartidas**.
- **El Problema:** Como tanto el Cliente (C2) como el Servidor (S1) conocen la misma clave simétrica para generar el MAC, el servidor podría teóricamente falsificar un mensaje y afirmar que vino del cliente (o viceversa). No existe una firma digital única por cada paquete de datos que pruebe matemáticamente y de forma irrefutabile ante un juez quién generó ese paquete específico, a diferencia de lo que ocurre en un correo firmado con GPG.

## Conclusiones y comentarios

### Conclusiones

Tras la realización de este laboratorio práctico sobre el protocolo SSH y el análisis de tráfico de red, se han obtenido las siguientes conclusiones técnicas:

- **Evolución del Costo Criptográfico:** Se evidenció una correlación directa entre la modernidad de la versión de OpenSSH y el tamaño de los paquetes de negociación. Mientras que las versiones antiguas (C1/Ubuntu 18.04) generan un tráfico inicial ligero, la versión más reciente (C4/Ubuntu 22.10) aumenta significativamente el tamaño del intercambio de claves (de 114 a 1294 bytes). Esto demuestra que la adopción de seguridad **Post-Cuántica** (algoritmo *Streamlined NTRU Prime*) conlleva un costo en el ancho de banda inicial, necesario para garantizar la seguridad a largo plazo.
- **Eficacia del Fingerprinting (HASSH):** Se comprobó que es posible identificar con alta precisión la versión del sistema operativo y del cliente SSH de un usuario sin necesidad de tener acceso al servidor, simplemente analizando los metadatos de la conexión (tamaños de paquete y lista de algoritmos soportados). Esta técnica de *Passive Reconnaissance* permitió identificar exitosamente al cliente “informante” como un sistema Ubuntu 20.04 (C2).
- **Flexibilidad de Configuración:** A través de la manipulación del archivo `sshd_config` en el servidor S1, se demostró que OpenSSH es altamente configurable. Fue posible reducir la “superficie de exposición” del servidor limitando los algoritmos permitidos, logrando reducir el paquete *Key Exchange Init* en más de un 70 % (a menos de 300 bytes), lo cual es una práctica recomendada para entornos de alta seguridad (Hardening).
- **Limitaciones del No Repudio:** Aunque SSH es robusto en confidencialidad e integridad, se concluye que no garantiza el no repudio de la sesión de datos, ya que la protección de los comandos se basa en claves simétricas (MAC) compartidas entre cliente y servidor, y no en firmas digitales individuales por paquete.

### Comentarios

Durante el desarrollo de la experiencia, uno de los mayores desafíos fue la gestión de versiones de sistemas operativos que han alcanzado su “End of Life” (EOL), como Ubuntu 16.10 o 22.10. Esto obligó a realizar ingeniería de soluciones sobre los archivos `Dockerfile`, ajustando los repositorios a *old-releases* o migrando a versiones LTS estables (18.04, 20.04, 22.04) para garantizar la viabilidad del entorno de pruebas.

Esta dificultad técnica resultó ser educativa, ya que simula escenarios reales donde un administrador de seguridad debe auditar o mantener sistemas heredados (legacy) que ya no cuentan con soporte oficial, obligando a entender profundamente cómo funciona la gestión de paquetes y dependencias en Linux.

Finalmente, el uso de contenedores Docker demostró ser una herramienta invaluable para la ciberseguridad, permitiendo desplegar y analizar múltiples escenarios criptográficos aislados en una sola máquina física de manera rápida y segura.