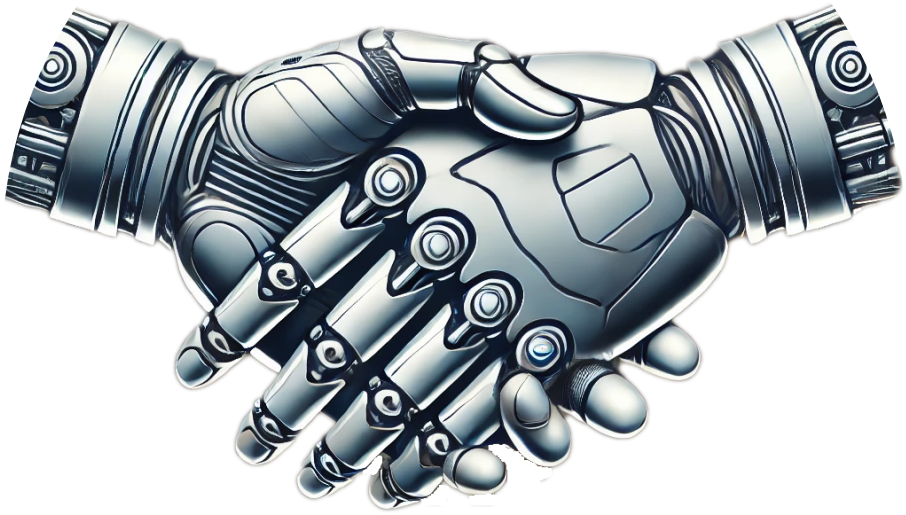
SSH Handshake



- Federico Williamson
- Leonel Castinelli

Seguridad Informática 2024

Índice

[1] RFC4253-4.2
[2] RFC4253-7.1
[3] RFC4253-8
[4] RFC4252
[5] RFC4252
[6] RFC4254

I. Introducción

I.I. ¿Qué es SSH?

SSH (o **Secure SHell**) es el nombre un protocolo y del programa que lo implementa cuya principal función es el **acceso remoto** a un servidor por medio de un **canal seguro** en el que toda la información está cifrada. El puerto TCP asignado de forma predeterminada es el puerto **22**, asignado por la IANA [1].

I.II. Descripción técnica del problema que soluciona.

Ante la comunicación entre dos equipos informáticos, existe la posible amenaza de que un equipo malicioso esté escuchando la comunicación legítima y busque robar información que se esté transmitiendo o incluso intente robar la identidad de uno de los actores en la comunicación para inducir mensajes maliciosos en la misma.

I.III. Principales aplicaciones.

Ya que este protocolo prueba ser de extrema utilidad, se aplica en las siguientes ocasiones:

- Acceso Remoto Seguro: Permite controlar equipos informáticos de forma segura a través de internet.
- Túneles SSH: Redirección de tráfico en la red a través de un canal seguro, útil para evitar restricciones de red y proteger datos sensibles.
- Transferencia segura de archivos: Los protocolos SCP y SFTP utilizan el protocolo SSH para la transferencia de archivos de forma s

I.IV. ¿Qué es SSH Handshake?

Es un proceso por el cual se establece una conexión segura entre un cliente y un servidor. Se produce tras la primera comunicación con el servidor. Consta de **5 pasos**:

- **Paso 1:** Intercambio de Versiones.
- **Paso 2:** Intercambio de Claves (KEX).
- **Paso 3:** Inicialización de Elliptic Curve Diffie-Hellman (ECDH).
- **Paso 4:** Respuesta de ECDH.
- **Paso 5:** Nuevas Claves (NewKeys).

II. Intercambio de Versiones.

II.I. Cadena de Version[7]

La cadena de versión de un participante en la comunicación se encuentra en el siguiente formato:

```
<SSH-protocolversion>-<softwareversion> <comments>CRLF
```

Con esta cadena en este formato ambos participantes podrán intercambiar la versión del protocol que están utilizando y la versión de software que están utilizando. Y siempre deben terminar con CRLF[8].[2]

Ambas partes, **cliente** y **servidor** deben enviar sus cadenas de versión. Las cuales llamaremos V_C a la cadena de version del **cliente** y V_S a la del **servidor**.

```
INFO     SSHClient: Connected to server
DEBUG    SSHClient: My SSH version: SSH-2.0-LeoFedeSsh0.1 Un trabajo para un tp
DEBUG    SSHSocketWrapper: c >> s [46]: 5353482d322e302d4c656f4665646553 7368302e3120556e2074726162616a6f 207061726120756e2074700d0d0a
DEBUG    SSHSocketWrapper: s >> c [21]: 5353482d322e302d4f70656e5353485f 392e300d0a
DEBUG    SSHClient: Remote SSH version: SSH-2.0-OpenSSH_9.0
```

Acá podemos observar como enviamos nuestra versión y recibimos la versión del servidor.

II.II. Protocolo binario de paquetes

Cada paquete se encuentra en el siguient

```
uint32    packet_length
byte      padding_length
byte[n1]  payload; n1 = packet_length - padding_length - 1
byte[n2]  random padding; n2 = padding_length
byte[m]   mac (Message Authentication Code - MAC); m = mac_length
```

Donde cada parte representa:

- **packet_length**: Es la longitud del paquete en bytes, sin contar “mac” o el mismo campo “packet_length”
- **padding_length**: longitud del “random padding” en bytes.
- **payload**: La parte útil del paquete.
- **random_padding**: padding de longitud arbitraria, para que la longitud total (packet_length + padding_length + payload + random_padding) sea un múltiplo de 8, el padding tiene que ser cómo mínimo de 4 y máximo de 255. Además sirve para introducr ruido al mensaje, confundiendo a receptores ilegítimos de la comunicación.
- **MAC**: Código de Autorización de Mensajes, si se negoció la autenticación de mensajes, este campo contiene los bytes de MAC. Inicialmente el algoritmo de MAC es `none`.

II.II.I. MAC

Una funcionalidad de proteccion, de extrema utilidad en lo que concierne a la seguridad de la comunicación, que ofrece SSH en su modelo de paquetes es MAC: Message Authorization Code (Código de Autorizacón de Mensajes).

Este codigo es un algoritmo de hash del contenido del mensaje no encriptado, con un número de secuencia del paquete.

Esto es extremadamente util en evitar replay attacks.

[7] RFC4253-4.2

[8] CRLF es un `\r` seguido de `\n`

III. Intercambio de algoritmos (KEXInit)[9]

```
Local key:
SSHKEXInitPacket(
    cookie=b'$\x1a\x7fM\x86E!\xafH\x14X\xfd<\xf8\x90\x93',
    kex_algorithms=[b'diffie-hellman-group14-sha256'],
    server_host_key_algorithms=[b'rsa-sha2-256'],
    encryption_algorithms_client_to_server=[b'aes128-ctr'],
    encryption_algorithms_server_to_client=[b'aes128-ctr'],
    mac_algorithms_client_to_server=[b'hmac-sha1'],
    mac_algorithms_server_to_client=[b'hmac-sha1'],
    compression_algorithms_client_to_server=[b'none'],
    compression_algorithms_server_to_client=[b'none'],
    languages_client_to_server=[b''],
    languages_server_to_client=[b''],
    first_kex_packet_follows=False,
    reserved=0
)
```

Nosotros mandamos este paquete de tipo KEX init al servidor.

Aca manifestamos los algoritmos que soportamos para cada categoria, en orden descendente de preferencia. Para forzar el uso de ciertos algoritmos y simplificar la implementación, enviamos listas de un único elemento.

```
Remote key:
SSHKEXInitPacket(
    cookie=b'\x81\xf8\xff\xfa\x01\xc6\xdd\x88\x8b6Q\x1eG\x12\x81/',
    kex_algorithms=[
        b'sntrup761x25519-sha512@openssh.com',
        b'curve25519-sha256',
        b'curve25519-sha256@libssh.org',
        b'ecdh-sha2-nistp256',
        b'ecdh-sha2-nistp384',
        b'ecdh-sha2-nistp521',
        b'diffie-hellman-group-exchange-sha256',
        b'diffie-hellman-group16-sha512',
        b'diffie-hellman-group18-sha512',
        b'diffie-hellman-group14-sha256'
    ],
    server_host_key_algorithms=[b'rsa-sha2-512', b'rsa-sha2-256', b'ecdsa-sha2-nistp256', b'ssh-ed25519'],
    encryption_algorithms_client_to_server=[b'chacha20-poly1305@openssh.com', b'aes128-ctr', b'aes192-ctr', b'aes256-ctr', b'aes128-gcm@openssh.com', b'aes256-gcm@openssh.com'],
    encryption_algorithms_server_to_client=[b'chacha20-poly1305@openssh.com', b'aes128-ctr', b'aes192-ctr', b'aes256-ctr', b'aes128-gcm@openssh.com', b'aes256-gcm@openssh.com'],
    mac_algorithms_client_to_server=[
        b'umac-64-etm@openssh.com',
        b'umac-128-etm@openssh.com',
        b'hmac-sha2-256-etm@openssh.com',
        b'hmac-sha2-512-etm@openssh.com',
        b'hmac-sha1-etm@openssh.com',
        b'umac-64@openssh.com',
        b'umac-128@openssh.com',
        b'hmac-sha2-256',
        b'hmac-sha2-512',
        b'hmac-sha1'
    ],
    mac_algorithms_server_to_client=[
        b'umac-64-etm@openssh.com',
        b'umac-128-etm@openssh.com',
        b'hmac-sha2-256-etm@openssh.com',
        b'hmac-sha2-512-etm@openssh.com',
        b'hmac-sha1-etm@openssh.com',
        b'umac-64@openssh.com',
        b'umac-128@openssh.com',
        b'hmac-sha2-256',
        b'hmac-sha2-512',
        b'hmac-sha1'
    ],
    compression_algorithms_client_to_server=[b'none', b'zlib@openssh.com'],
    compression_algorithms_server_to_client=[b'none', b'zlib@openssh.com'],
    languages_client_to_server=[b''],
    languages_server_to_client=[b''],
    first_kex_packet_follows=False,
    reserved=0
)
```

La respuesta del servidor, mostrando todos los algoritmos que soporta para cada categoria.

El protocolo SSH define que se usa el primero en comun, llendo en orden de izquierda a derecha.

Es posible que en cada direccion se usen algirtmos distintos

En este momentto, hambos cliente y servidor saben que algoritmo van a usar para cada caso (lo pueden inferir por separado)

A los **payload** de **cliente** y **servidor** los llamaremos I_C y I_S respectivamente

[9] RFC4253-7.1

IV. Intercambio de claves Diffie Hellman[10]

Nosotros, elegimos usar el algoritmo de `diffie-hellman-group14-sha256` como algoritmo de intercambio.

Esto significa que:

- Vamos a hacer un intercambio de tipo DH. (que usará Curvas Elípticas Diffie-Hellman, ECDH)
- Vamos a usar primos del grupo 14.
- Vamos a usar sha256 como algoritmo de hash.

> El objetivo del KEX DH es que ambos participantes puedan mutuamente acordar en una clave igual, sin que esta sea derivable por un tercero que esta observando la comunicación.

Como funciona esto?

1. Se acuerdan p y g. P es un primo publico muy grande, y g su generador.

Como estamos usando group14, usamos el primos del grupo 14, que esta definido en RFC3526-3

C (cliente) genera un numero aleatorio x entre 1 y q[11] y computa el valor:

$$e = g^x \bmod p$$

El cual es conocido como **clave pública del cliente** y a x lo llamaremos **clave privada del cliente**.

S (servidor) genera su propio numero aleatorio y entre 0 y q y computa:

$$f = g^y \bmod p$$

Acá se puede observar la **clave pública del servidor** f y la **clave privada del servidor** y.

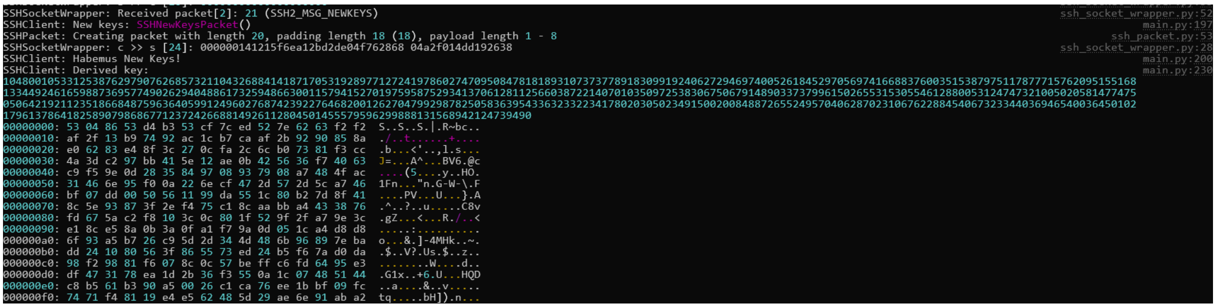
[10] RFC4253-8

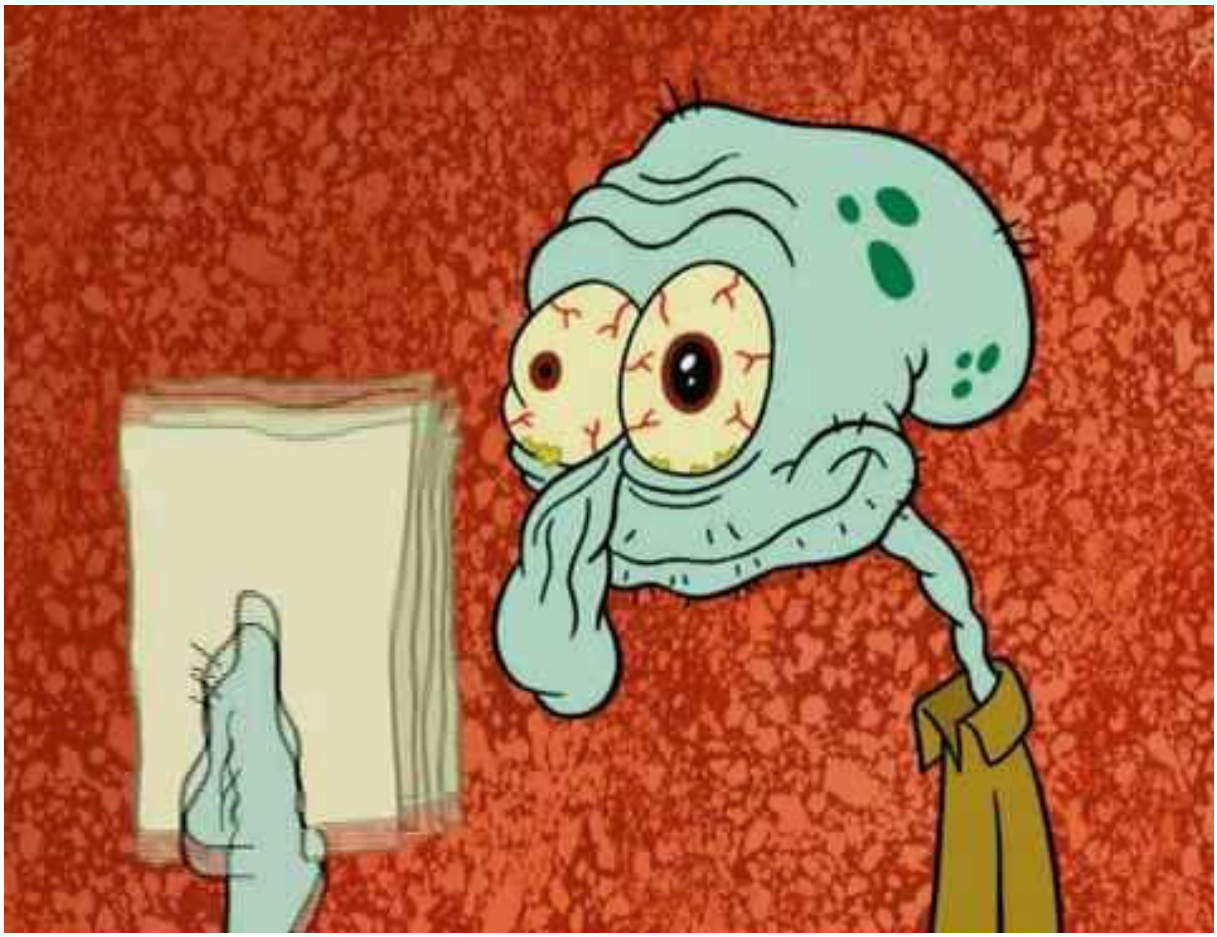
[11] q hace referencia al orden del subgrupo, que no se necesita clacular explicitamente, es aproximadamente $\lfloor \frac{p}{2} \rfloor$

S al recibir e, computa el **secreto compartido** K de la siguiente forma:

$$K = e^y \bmod p$$

Una vez que ves. recibe x del C, ya esta listo y manda `SSH_NEW_KEYS`. Cuando C recibe f de S, puede emitir `SSH_NEW_KEYS`.





En este momento ambos tienen suficiente información para terminar de derivar la clave compartida K, pero únicamente K es insuficiente para la comunicación

y tambien computa el hash H:

$$H = \text{hash}(V_C \parallel V_S \parallel I_C \parallel I_S \parallel K_S \parallel e \parallel f \parallel K)$$

Donde:

- hash es sha256, por el algoritmo de intercambio que elegimos,
- V_C Es la version de ssh cliente (sin CRLF)
- V_S Version de ssh del seervidor (sin CRLF)
- I_C El Payload del `KEX_INIT` enviado por el cliente
- I_S El Payload del `KEX_INIT` enviado por el servidor
- K_S La Host-Key del servidor[12]
- e La clave publica del cliente
- f La clave privada del servidor
- K el secreto compartido entre las partes.

Un dato interesante de esto, es que al incluir el payload de I_C e I_S, que como anteriormente cubrimos en Sección III, contienen un cookie que contiene un valor random. Esto hace que sea completamente imposible para un atacante determinar H en sesiones repetidas.

En base a H, K y la session id[13] se calculan los siguientes valores:

[12]Este es un valor adicional que envia el servidor cuando envia f, con el proposito de autenticarse.

[13]La session id es el primer H que se acuerda entre C y S, y no cambia cuando hay un reset de keys.

```
o  Initial IV client to server: HASH(K || H || "A" || session_id)
   (Here K is encoded as mpint and "A" as byte and session_id as raw
   data.  "A" means the single character A, ASCII 65).
o  Initial IV server to client: HASH(K || H || "B" || session_id)
o  Encryption key client to server: HASH(K || H || "C" || session_id)
o  Encryption key server to client: HASH(K || H || "D" || session_id)
o  Integrity key client to server: HASH(K || H || "E" || session_id)
o  Integrity key server to client: HASH(K || H || "F" || session_id)
```

IV.I. Markus

Para probar el cliente, con el servidor de openssh, es posible ejecutarlo en modo debug. Una de las funcionalidades del modo debug, es una vez que se inicia la comunicacion encriptada, se envia un paquete de tipo 02 (`SSH_IGNORE`) con texto: `markus`. Este mensaje conocido es extremadamente util para debuggear problemas.

`markus` en este caso hace referencia a Markus Friedl, un contribuidor original de SSH.

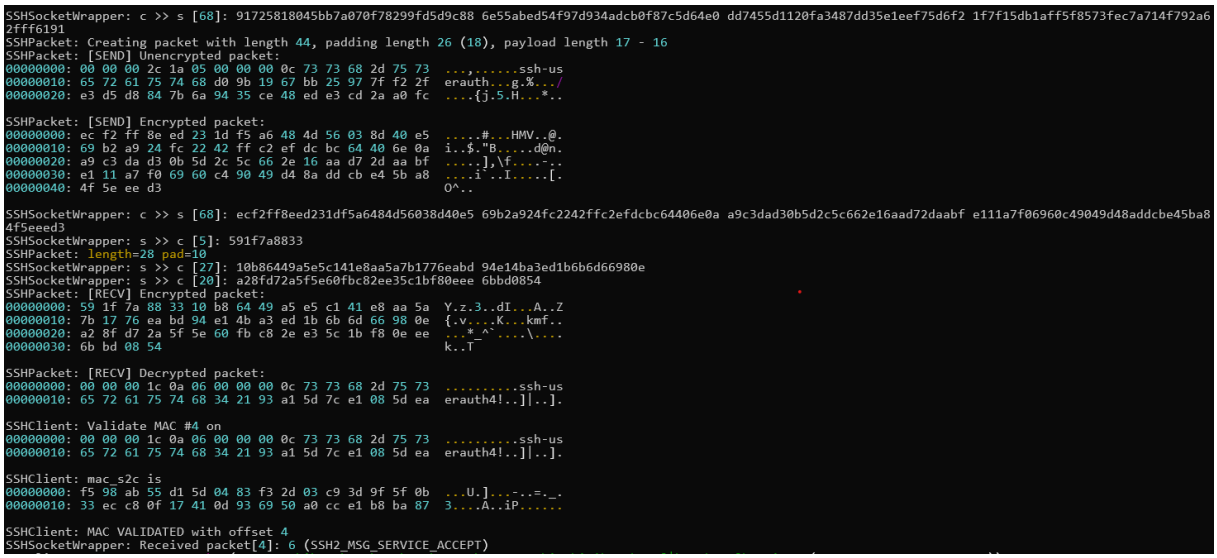
```
          DEBUG    SSHSocketWrapper: s >> c [5]: 6cfd30b2ea
          DEBUG    SSHPacket: length=28 pad=16
          DEBUG    SSHSocketWrapper: s >> c [27]: bcd48930491313f1eb620d2bbc2927ed448207c49e8170a51b09f2
          DEBUG    SSHSocketWrapper: s >> c [20]: 140c6f551723c2d21cc27d215b72857d4cc113f0
          DEBUG    SSHPacket: [RECV] Encrypted packet:
                   00000000: 6c fd 30 b2 ea bc d4 89 30 49 13 13 f1 eb 62 0d  l.0.....0I....b.
                   00000010: 2b bc 29 27 ed 44 82 07 c4 9e 81 70 a5 1b 09 f2  +.)'.D.....p....
                   00000020: 14 0c 6f 55 17 23 c2 d2 1c c2 7d 21 5b 72 85 7d  ..oU.#....}![r.}
                   00000030: 4c c1 13 f0                                      L...

          DEBUG    SSHPacket: [RECV] Decrypted packet:
                   00000000: 00 00 00 1c 10 02 00 00 00 06 6d 61 72 6b 75 73  ..........markus
                   00000010: 09 fe 9b d4 72 b0 fd a2 0b 2c 3a 17 fa c5 21 96  ....r....,:...!.

          INFO     SSHClient: Validate MAC #3 on
                   00000000: 00 00 00 1c 10 02 00 00 00 06 6d 61 72 6b 75 73  ..........markus
                   00000010: 09 fe 9b d4 72 b0 fd a2 0b 2c 3a 17 fa c5 21 96  ....r....,:...!.

          INFO     SSHClient: mac_s2c is
                   00000000: f5 98 ab 55 d1 5d 04 83 f3 2d 03 c9 3d 9f 5f 0b  ...U.]...-..=._.
                   00000010: 33 ec c8 0f 17 41 0d 93 69 50 a0 cc e1 b8 ba 87  3....A..iP......

          INFO     SSHClient: MAC VALIDATED with offset 3
          DEBUG    SSHSocketWrapper: Received packet[3]: 2 (SSH2_MSG_IGNORE)
SSHPacket(28, 16, b'\x02\x00\x00\x00\x06markus', b'\t\xfe\x9b\xd4r\xb0\xfd\xa2\x0b,:\x17\xfa\xc5!\x96', 2 (SSH2_MSG_IGNORE))
          DEBUG    SSHPacket: Creating packet with length 44, padding length 32 (16), payload length 11 - 16
          DEBUG    SSHPacket: [SEND] Unencrypted packet:
                   00000000: 00 00 00 2c 20 02 00 00 00 06 6d 61 72 6b 75 73  ...,  .....markus
                   00000010: 6a 5e f2 1d 35 d7 f7 87 e5 96 f2 95 8e 5a eb f7  j^..5........Z..
                   00000020: 80 eb e8 44 7f c6 43 69 4f db 15 0d 59 5e c2 ea  ...D..CiO...Y^..

          DEBUG    SSHPacket: [SEND] Encrypted packet:
                   00000000: 91 72 58 18 04 5b b7 a0 70 f7 82 99 fd 5d 9c 88  .rX..[..p....]..
                   00000010: 6e 55 ab ed 54 f9 7d 93 4a dc b0 f8 7c 5d 64 e0  nU..T.}.J...|]d.
                   00000020: dd 74 55 d1 12 0f a3 48 7d d3 5e 1e ef 75 d6 f2  .tU....H}.^..u..
                   00000030: 1f 7f 15 db 1a ff 5f 85 73 fe c7 a7 14 f7 92 a6  ......_.s.......
                   00000040: 2f ff 61 91                                      /.a.
```

Aca se puede observar como tambien enviamos un paquete ignorable con `markus` al servidor para verificar que no produce errores.

IV.II. Solicitud de servicio[14]

Antes de proceder, requerimos que el servidor nos permita autenticar usuarios.

[14] RFC4252



IV.III. Solicitud de Conexion[15]

Una vez que tenemos permisos para autenticar, podemos solicitar una conexion

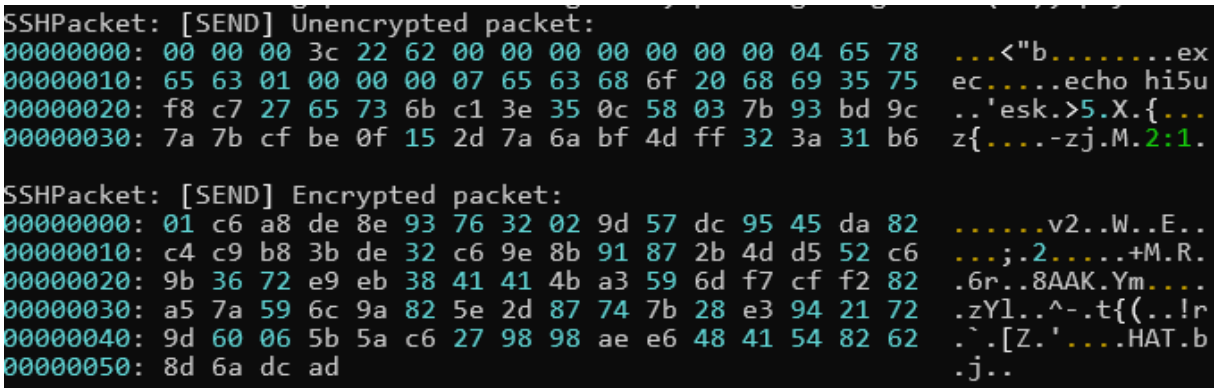
Una vez que realizamos esto, si es exitosa, recibimos varios mensajes del servidor:

1. HostKeys: Autenticacion del servidor
2. Confirmacion de la apertura del canal

Y estamos en posicion de enviar un comando

IV.IV. Ejecucion[16]

[15] RFC4252
[16] RFC4254

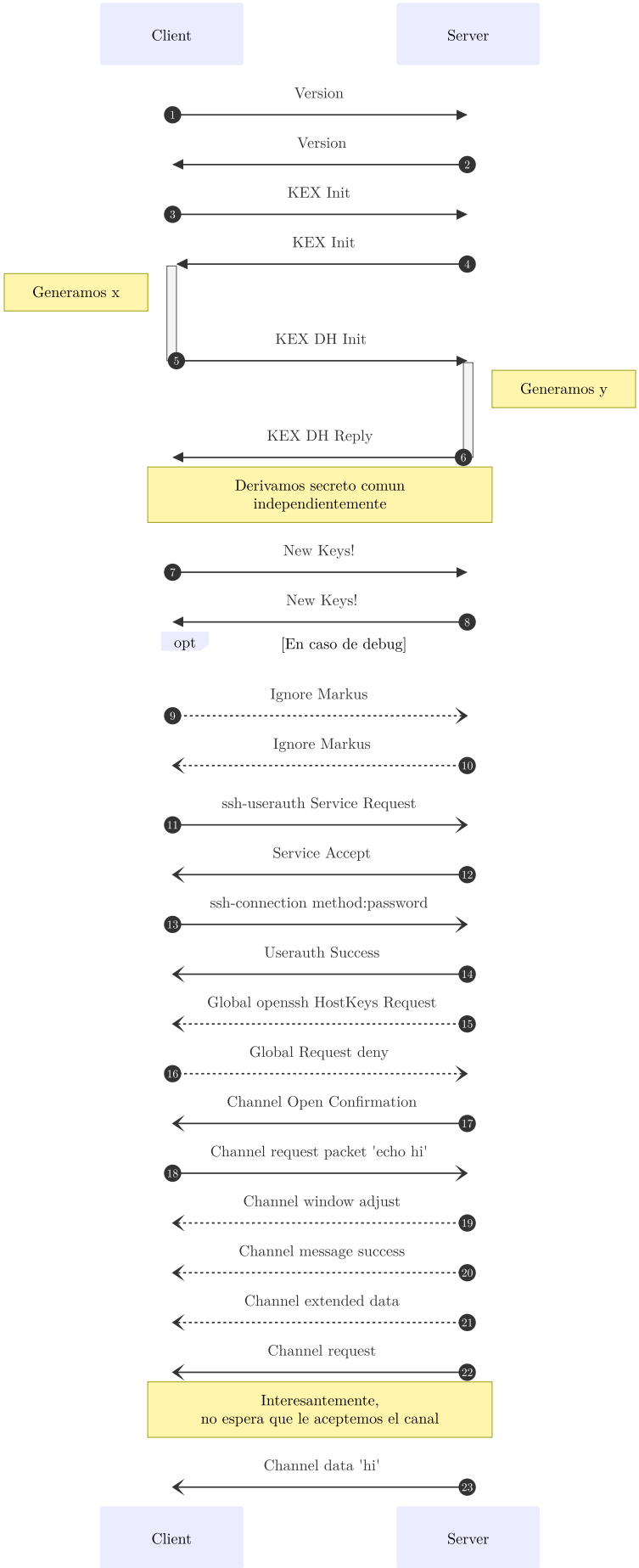


Enviamos un comando de EXEC con payload `echo hi`

Y recibimos varios mensajes del servidor:

- `SSH2_MSG_CHANNEL_WINDOW_ADJUST`: Incrementando el tamaño maximo de un paquete.
- `SSH2_MSG_CHANNEL_EXTENDED_DATA`: Con informacion del path, y otras variables de entorno
- `SSH2_MSG_CHANNEL_REQUEST`: Pide abrir un canal de comunicacion hacia nosotros
- `SSH2_MSG_CHANNEL_DATA`: Optimisticamente envia la respuesta a nuestra peticion antes que aceptemos abrir el canal.

V. Apéndice II


Client
Server
Version
1
Version
2
KEX Init
3
KEX Init
4
Generamos x
KEX DH Init
5
Generamos y
KEX DH Reply
6
Derivamos secreto comun
independientemente
New Keys!
7
New Keys!
8
opt
[En caso de debug]
Ignore Markus
9
Ignore Markus
10
ssh-userauth Service Request
11
Service Accept
12
ssh-connection method:password
13
Userauth Success
14
Global openssh HostKeys Request
15
Global Request deny
16
Channel Open Confirmation
17
Channel request packet 'echo hi'
18
Channel window adjust
19
Channel message success
20
Channel extended data
21
Channel request
22
Interesantemente,
no espera que le aceptemos el canal
Channel data 'hi'
23
Client
Server

Bibliografía

[1] «The story of the SSH port is 22. — ssh.com». 2017.

[2] «ietf.org». 2006.

Los RFC mencionados tienen links.