

Analisi del protocollo SSH

Progetto di Network Security

Francesco Iannaccone, Matteo Conti, Alfonso Conte, Daniele Fazzari

2022-2023

Indice

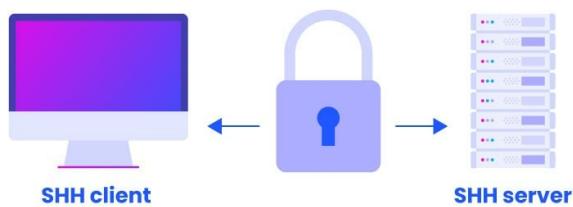
1	SSH	3
1.1	Descrizione del protocollo	3
1.1.1	Il pacchetto SSH	4
1.1.2	Convenzioni sui codici di messaggio	5
1.2	SSH Transport Layer Protocol	6
1.2.1	Messaggi per lo scambio delle chiavi	7
1.2.2	Richieste di servizio	9
1.2.3	Messaggi aggiuntivi	10
1.3	SSH Authentication Protocol	11
1.3.1	Richiesta di autenticazione	12
1.3.2	Messaggio di banner	15
1.3.3	Messaggi aggiuntivi	15
1.4	SSH Connection Protocol	16
1.4.1	Port forwarding	17
1.4.2	Local port forwarding	18
1.4.3	Remote port forwarding	18
1.4.4	Apertura di canale	19
1.4.5	Richieste channel-specific	21
2	Il Client SSH	26
2.1	Modifica di libssh	26
2.2	L'implementazione del client	27
2.2.1	Autenticazione Server-Client	27
2.2.2	Autenticazione Client-Server	28

2.2.3	Implementazione Remote Shell	28
2.2.4	Implementazione direct forwarding	28
2.2.5	Implementazione exec di un comando remoto	29
3	SSH Wireshark Post Dissector	30
3.1	Descrizione del plugin	30
3.2	Catture di esempio	31
3.2.1	Autenticazione	31
3.2.2	Remote shell con canale di sessione	36
3.2.3	Direct forwarding	38

1 SSH

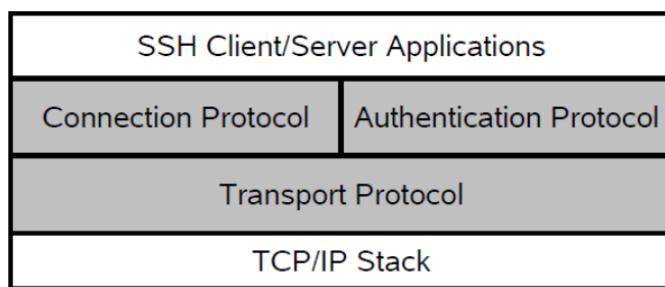
1.1 Descrizione del protocollo

SSH è un protocollo crittografico che consente di utilizzare servizi di rete in modo sicuro su una rete non protetta. Le applicazioni più utilizzate attualmente del protocollo sono il login remoto e l'esecuzione a linea di comando remota. SSH si basa su un'architettura client-server, che permette a un'istanza di client SSH di connettersi con un server SSH.



Nel modello OSI, il protocollo avviene a livello applicazione, posizionandosi al di sopra del protocollo TCP. In particolare è organizzato come una suite protocollare a livelli, la quale comprende tre componenti principali:

- SSH Transport Layer Protocol
- SSH User Authentication Protocol
- SSH Connection Protocol



Storicamente, questo protocollo fu progettato su sistemi operativi Unix-like, come sostituto al posto di Telnet e protocolli di shell remoti non sicuri di

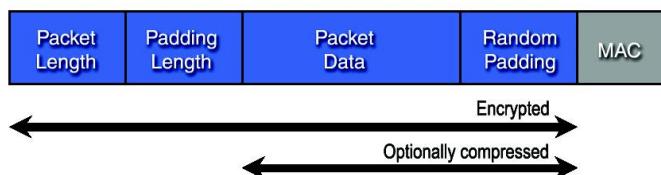
Unix, che prevedevano la trasmissione in chiaro dei token di autenticazione e dei comandi remoti.

L'implementazione più utilizzata di SSH è OpenSSH, rilasciata come software open-source dagli sviluppatori di OpenBSD. Le implementazioni sono distribuite per tutti i tipi di sistemi operativi di uso comune, a partire da Windows e Linux.

1.1.1 Il pacchetto SSH

Ogni pacchetto nella versione SSH2 è nel seguente formato:

- Packet length, intero rappresentato su 32 bit, che indica la lunghezza del pacchetto in byte, escludendo i campi del MAC e del packet length stesso;
- Padding length, un byte che rappresenta la lunghezza in byte del random padding;
- Payload: codificato su una sequenza di byte lunga $packet_length - padding_length - 1$, è il contenuto utile del pacchetto; se è stata negoziata la compressione, questo campo viene compresso;
- Random padding: padding di lunghezza arbitraria, tale che la lunghezza totale di (packet_length, padding_length, payload, random_padding) sia un multiplo della dimensione del blocco cifrato o di 8, a seconda di quale sia il maggiore.
- MAC: se l'autenticazione del messaggio è stata negoziata, questo campo contiene i byte del MAC; viene calcolato sull'intero pacchetto (MAC escluso), più un numero di sequenza.



Il sequence number è un intero a 32 bit inizializzato a 0 per il primo pacchetto e incrementato di 1 per ogni pacchetto successivo.

1.1.2 Convenzioni sui codici di messaggio

Il primo byte del payload è sempre un codice identificativo, che rappresenta il tipo di messaggio ed identifica a quale protocollo di SSH questo appartiene. In figura 1, sono specificati tutti i possibili identificativi [1]:

Message ID	Value
SSH_MSG_DISCONNECT	1
SSH_MSG_IGNORE	2
SSH_MSG_UNIMPLEMENTED	3
SSH_MSG_DEBUG	4
SSH_MSG_SERVICE_REQUEST	5
SSH_MSG_SERVICE_ACCEPT	6
SSH_MSG_KEXINIT	20
SSH_MSG_NEWKEYS	21
SSH_MSG_USERAUTH_REQUEST	50
SSH_MSG_USERAUTH_FAILURE	51
SSH_MSG_USERAUTH_SUCCESS	52
SSH_MSG_USERAUTH_BANNER	53
SSH_MSG_GLOBAL_REQUEST	80
SSH_MSG_REQUEST_SUCCESS	81
SSH_MSG_REQUEST_FAILURE	82
SSH_MSG_CHANNEL_OPEN	90
SSH_MSG_CHANNEL_OPEN_CONFIRMATION	91
SSH_MSG_CHANNEL_OPEN_FAILURE	92
SSH_MSG_CHANNEL_WINDOW_ADJUST	93
SSH_MSG_CHANNEL_DATA	94
SSH_MSG_CHANNEL_EXTENDED_DATA	95
SSH_MSG_CHANNEL_EOF	96
SSH_MSG_CHANNEL_CLOSE	97
SSH_MSG_CHANNEL_REQUEST	98
SSH_MSG_CHANNEL_SUCCESS	99
SSH_MSG_CHANNEL_FAILURE	100

Figura 1

Per quanto riguarda il transport layer protocol, i codici sono distribuiti nel seguente modo [4]:

- da 1 a 19: codici generici del livello trasporto (come disconnect, ignore, debug)
- da 20 a 29: utili per la negoziazione dell'algoritmo
- da 30 a 49: specifici per il metodo di scambio delle chiavi

Per il protocollo di autenticazione si ha:

- 50 a 59: generici per l'autenticazione dell'utente

- 60 a 69: specifici per il metodo di autenticazione dell’utente

Infine, riguardo al protocollo di connessione, si avrà:

- 80 a 89: generici per il protocollo di connessione
- 90 a 127: messaggi relativi al canale

1.2 SSH Transport Layer Protocol

Il protocollo utilizza TCP, riservando la porta 22 come porta di ascolto del server.

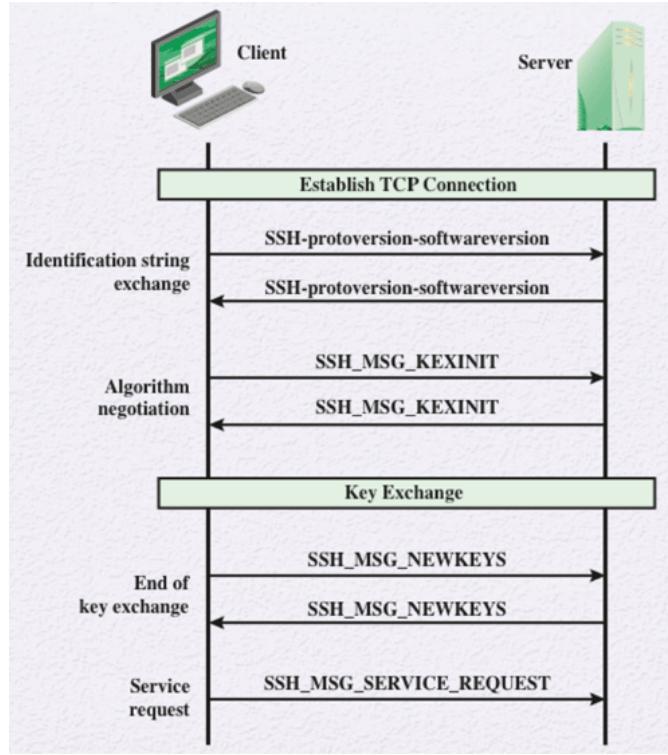
Principalmente si occupa dello scambio iniziale di chiavi e dell’ autenticazione del server; inoltre, imposta la crittografia, la compressione e la verifica dell’integrità [5].

Al livello superiore espone un’interfaccia per l’invio e la ricezione di pacchetti con una dimensione massima di 32,768 byte ciascuno, ma ogni implementazione opzionalmente può consentirne di più.

Per quanto riguarda l’autenticazione del server, esso può utilizzare molteplici chiavi di host, chiamate ”host key”, tante quante sono i diversi algoritmi di crittografia implementati. Quindi, la host key del server viene sfruttata durante il processo di scambio delle chiavi per verificare che il client stia veramente comunicando col server corretto. Affinchè ciò sia possibile, il client deve conoscere a priori l’host key pubblica del server prima di iniziare a contattarlo.

Si può quindi dividere il protocollo in 5 fasi distinte:

- la fase di connessione TCP tra server e client;
- la fase di identificazione del protocollo: per decidere quale versione di SSH (SSH1 o SSH2) si sta utilizzando;
- la fase di negoziazione dell’algoritmo: per accordarsi su quale algoritmo verrà usato per lo scambio delle chiavi;
- la fase di scambio delle chiavi: per eseguire l’effettivo scambio delle chiavi usando quest’algoritmo;
- la fase di richiesta di servizio, che dà inizio ai protocolli successivi.



A seguito della creazione del canale tcp, durante la seconda fase, il client invia un messaggio al server di tipo SSH-protoversion-softwareversion, dove specifica la versione di SSH utilizzata, e poi il server replica con la sua versione supportata.

Nella fase successiva il client invia un messaggio SSH-MSGKEXINIT, in cui stabilisce una lista di algoritmi in ordine preferenziale (in ordine crescente) che intende utilizzare, e successivamente il server gli risponde con l'algoritmo scelto.

1.2.1 Messaggi per lo scambio delle chiavi

Lo scambio di chiavi inizia da entrambi i lati mandando il seguente pacchetto:

```

byte      SSH_MSG_KEXINIT
byte[16]  cookie (random bytes)
name-list kex_algorithms
name-list server_host_key_algorithms
name-list encryption_algorithms_client_to_server
name-list encryption_algorithms_server_to_client
name-list mac_algorithms_client_to_server
name-list mac_algorithms_server_to_client
name-list compression_algorithms_client_to_server
name-list compression_algorithms_server_to_client
name-list languages_client_to_server
name-list languages_server_to_client
boolean   first_kex_packet_follows
uint32    0 (reserved for future extension)

```

Le liste di nomi dei vari campi presenti in figura dovranno essere una lista di nomi di algoritmi separati dalla virgola e ordinati in ordine di preferenza. In particolare, i campi del pacchetto più significativi sono:

- cookie: valore casuale generato dal mittente; il suo scopo è rendere impossibile per ambo le parti di determinare completamente le chiavi e l'identificatore di sessione;
- kex_algorithms: algoritmi di scambio delle chiavi;
- server_host_key_algorithms: algoritmi supportati per la host key del server; in questo contesto, il server elenca gli algoritmi per cui possiede host key e il client elenca gli algoritmi che è disposto ad accettare;
- encryption_algorithms: algoritmi di crittografia simmetrica in ordine di preferenza;
- languages: lista di tag di linguaggio in ordine di preferenza; è da notare come entrambe le parti potrebbero ignorare questo campo;
- first_kex_packet_follows: indica se a seguire vi sarà un pacchetto di guess di scambio delle chiavi;

Per quanto riguarda l'ultimo campo, si noti che dopo aver ricevuto il messaggio SSH_MSG_KEXINIT dall'altra parte, ogni nodo saprà se il proprio

guess era giusto. Se il guess dell’altro nodo era sbagliato, e il campo era true, il pacchetto successivo deve essere ignorato e l’algoritmo continua come stabilito. Altrimenti, lo scambio delle chiavi deve continuare utilizzando il pacchetto guessed.

Lo scambio delle chiavi termina mandando un messaggio di SSH_MSG_NEWKEYS da ambo le parti. Il messaggio ha come payload solo il byte identificativo, pari a 21. Tutti i messaggi inviati dopo questo messaggio devono usare i nuovi algoritmi e chiavi.

Lo scopo di questo messaggio finale è di assicurarsi che una delle due parti possa rispondere con un messaggio di disconnessione che l’altro endpoint può comprendere se qualcosa va male durante lo scambio delle chiavi.

1.2.2 Richieste di servizio

I messaggi di questa tipologia sono inoltrati dal client per richiedere uno specifico servizio al server SSH. Il servizio richiesto dal client è costituito da un nome simbolico. Il payload del messaggio SSH_MSG_SERVICE_REQUEST è molto semplice ed è rappresentato in figura 2:

```
byte      SSH_MSG_SERVICE_REQUEST  
string    service name
```

Figura 2

I service name implementati di default da tutti i server sono:

- ssh-userauth
- ssh-connection

Come si può intuire dai nomi, il primo è utilizzato per richiedere l’utilizzo dell’ SSH Authentication Protocol e il secondo per il Connection Protocol. Il protocollo consente di definire ulteriori tipologie di servizi nelle implementazioni.

Se il server supporta il servizio richiesto, deve rispondere con il seguente messaggio:

```
byte      SSH_MSG_SERVICE_ACCEPT  
string    service name
```

1.2.3 Messaggi aggiuntivi

Sono messaggi che entrambi gli host nella comunicazione possano inviare in qualsiasi momento. Tra questi, si hanno:

- SSH_MSG_DISCONNECT, messaggio utilizzato per segnalare la disconnessione di uno degli endpoint e la terminazione della sessione corrente.

```
byte      SSH_MSG_DISCONNECT
uint32    reason code
string   description in ISO-10646 UTF-8 encoding [RFC3629]
string   language tag [RFC3066]
```

- SSH_MSG_IGNORE, messaggio che può essere inoltrato ma che sarà ignorato.

```
byte      SSH_MSG_IGNORE
string   data
```

- SSH_MSG_DEBUG, Questo messaggio è usato per trasmettere informazioni che potrebbero aiutare la fase di debugging. Se il valore booleano always_display è settato a true, il messaggio deve essere mostrato; altrimenti, non deve essere mostrato se non richiesto esplicitamente dall'utente

```
byte      SSH_MSG_DEBUG
boolean  always_display
string   message in ISO-10646 UTF-8 encoding [RFC3629]
string   language tag [RFC3066]
```

- SSH_MSG_UNIMPLEMENTED

```
byte      SSH_MSG_UNIMPLEMENTED
uint32   packet sequence number of rejected message
```

1.3 SSH Authentication Protocol

Il livello di autenticazione gestisce l'autenticazione del client, necessaria per utilizzare i servizi offerti dal connection protocol [2]. L'autenticazione avviene a seguito dell'istituzione della connessione SSH, dell'autenticazione del server e dello scambio delle chiavi. I dati trasmessi sono quindi protetti dai meccanismi crittografici e di controllo di integrità offerti dal transport layer protocol.

I metodi di autenticazione dell'utente tipicamente supportati sono:

- password: metodo di autenticazione semplice basato sull'invio di una password da parte del client;
- publickey: autenticazione in cui il client invia un messaggio al server, con la propria chiave pubblica, firmato con la propria chiave privata; quindi, il server verifica che la chiave pubblica sia valida per l'autenticazione e, nel caso lo sia, verifica che la firma sia corretta;
- hostbased: metodo simile a quello basato su chiave pubblica, ma effettuata sull'host che ospita il client.
- none: metodo di autenticazione tipicamente non supportato dai server SSH. Il metodo viene spesso utilizzato dai client come meccanismo di discovery dei metodi di autenticazione previsti dal server.

Una tipica sessione di autenticazione prevede i seguenti passi:

1. il client invia un SSH MSG USER AUTH REQUEST con un requested method nullo
2. Il server controlla la validità dell'username. Se l'username non è valido, invia un messaggio SSH MSG USER AUTH FAILURE con il valore partial success settato a FALSE
3. Il server replica con un SSH MSG USER AUTH FAILURE con una lista di uno o più metodi di autenticazione supportati
4. Il client seleziona uno dei metodi e invia il messaggio SSH MSG USER AUTH REQUEST con quel method name e i campi specifici del metodo.
5. Il server invia un messaggio SSH MSG USER AUTH SUCCESS e l'authentication protocol finisce. Se l'autenticazione dovesse fallire, il

server tornerebbe al passo 3, con un valore di partial success pari a FALSE.

Tra il passo 4 e il passo 5 potrebbero esservi dei messaggi aggiuntivi, dipendentemente dal metodo di auth scelto. La sequenza dettagliata dei messaggi scambiati sarà discussa successivamente a partire dall'analisi delle catture. L'autenticaction protocol prevede la possibilità di combinare più metodi di autenticazione. In questo caso, a seguito di ogni fase dell'autenticazione si ritornerà allo step 3 ed il server imposerà il flag partial success a true fin quando tutti i metodi saranno eseguiti con successo.

1.3.1 Richiesta di autenticazione

La sequenza ed il formato dei messaggi USER_AUTH_REQUEST è differente in base alla tecnica di autenticazione utilizzata. Questi messaggi sono in ogni caso identificati dallo stesso codice e da una stringa identificativa che specifica la modalità .

Autenticazione con chiave pubblica L'autenticazione con chiave pubblica è l'unico metodo obbligatorio per qualsiasi server. Il client può, precedentemente alla vera fase di autenticazione inviare un messaggio come quello mostrato in figura 3 per verificare se la tipologia di chiave pubblica ed eventualmente il proprio cerficate è accettato dal server.

```
byte      SSH_MSG_USERAUTH_REQUEST
string    user name in ISO-10646 UTF-8 encoding [RFC3629]
string    service name in US-ASCII
string    "publickey"
boolean   FALSE
string    public key algorithm name
string    public key blob
```

Figura 3

Successivamente, per eseguire l'autenticazione vera e propria, il client invia un nuovo messaggio contenete la propria chiave pubblica ed una signature generata usando la propria chiave privata. Il messaggio ha quindi la seguente struttura:

```

byte      SSH_MSG_USERAUTH_REQUEST
string    user name
string    service name
string    "publickey"
boolean   TRUE
string    public key algorithm name
string    public key to be used for authentication
string    signature

```

Quando il server riceve questo messaggio, deve verificare che la chiave ricevuta sia accettabile per l'autenticazione e, in caso affermativo, deve controllare la validità della firma. Se entrambi i controlli hanno esito positivo, il metodo ha successo.

Autenticazione con password Per l'autenticazione basata su credenziali di accesso il messaggio di autenticazione contiene uno specifico campo utilizzato per trasmettere la password.

```

byte      SSH_MSG_USERAUTH_REQUEST
string    user name
string    service name
string    "password"
boolean   FALSE
string    plaintext password in ISO-10646 UTF-8 encoding [RFC3629]

```

Da notare come, nonostante la password sia trasmessa in chiaro, l'intero pacchetto è crittato dal transport layer. Sia il client che il server dovrebbero controllare se il livello di trasporto sottostante fornisca confidenzialità. In caso negativo (ad esempio se l'algoritmo di crittografia selezionato è none), l'autenticazione con password dovrebbe essere disabilitata.

Il campo di tipo booleano, in figura settato a false, può diventare true nel caso in cui la richiesta di auth venga usata per cambiare la password. In tal caso, il payload avrà il seguente formato:

```

byte      SSH_MSG_USERAUTH_REQUEST
string    user name
string    service name
string    "password"
boolean   TRUE
string    plaintext old password in ISO-10646 UTF-8 encoding
           [RFC3629]
string    plaintext new password in ISO-10646 UTF-8 encoding
           [RFC3629]

```

Autenticazione host-based Nel caso di richiesta di autenticazione host-based da parte dell’utente, il client invia una signature creata con la chiave privata dell’host. Il server, quindi, controllerà la signature con la chiave pubblica di quell’host.

Il messaggio di autenticazione avrà il seguente payload:

```

byte      SSH_MSG_USERAUTH_REQUEST
string    user name
string    service name
string    "hostbased"
string    public key algorithm for host key
string    public host key and certificates for client host
string    client host name expressed as the FQDN in US-ASCII
string    user name on the client host in ISO-10646 UTF-8 encoding
           [RFC3629]
string    signature

```

Oltre ai campi presenti anche in tutti gli altri metodi, si può analizzare come siano presenti il nome dell’algoritmo a chiave pubblica, il nome dell’host e il suo username sulla macchina richiedente.

Ricevuto questo messaggio, il server deve verificare che la host key effettivamente appartiente all’host, che all’utente su quell’host sia permesso di fare il login e che il valore della signature sia valido.

Autenticazione ”none” L’ultimo metodo di autenticazione possibile è il ”none”. Dal nome si può intendere che qui non avviene nessun tipo di autenticazione ed infatti il metodo non è raccomandato. Tuttavia, il client può inviare una richiesta di tal genere e il server deve sempre rifiutarla, a meno che all’utente non venga concessa la possibilità di accedere senza

alcuna autenticazione.

Lo scopo principale di questa richiesta è quello di ottenere dal server l'elenco dei metodi supportati. Il formato di tale richiesta è molto semplice ed è mostrato in figura:

```
byte      SSH_MSG_USERAUTH_REQUEST
string    user name in ISO-10646 UTF-8 encoding [RFC3629]
string    service name in US-ASCII
string    method name in US-ASCII
```

1.3.2 Messaggio di banner

Il server SSH può mandare un messaggio del tipo SSH_MSG_USERAUTH_BANNER in qualunque momento dopo l'avvio del protocollo di autenticazione e prima che l'autenticazione termini con successo. Questo messaggio contiene il testo da mostrare all'utente prima che venga tentata l'autenticazione. Il formato del messaggio è il seguente:

```
byte      SSH_MSG_USERAUTH_BANNER
string    message in ISO-10646 UTF-8 encoding [RFC3629]
string    language tag [RFC3066]
```

Di default, il client dovrebbe mostrare il contenuto del campo "message" sullo schermo dell'utente.

1.3.3 Messaggi aggiuntivi

Per il protocollo di SSH Authentication, viene utilizzato il messaggio SSH_MSG_USERAUTH_FAILURE per il rifiuto di una richiesta da parte del server. Di seguito viene rappresentato il formato del payload:

```
byte      SSH_MSG_USERAUTH_FAILURE
name-list authentications that can continue
boolean   partial success
```

SSH_MSG_USERAUTH_PK_OK è il messaggio che viene inviato dal server in risposta alla prima richiesta di autenticazione del client col metodo di chiave pubblica, nel caso in cui essa venga accettata. Il payload è il seguente:

```
byte      SSH_MSG_USERAUTH_PK_OK
string    public key algorithm name from the request
string    public key blob from the request
```

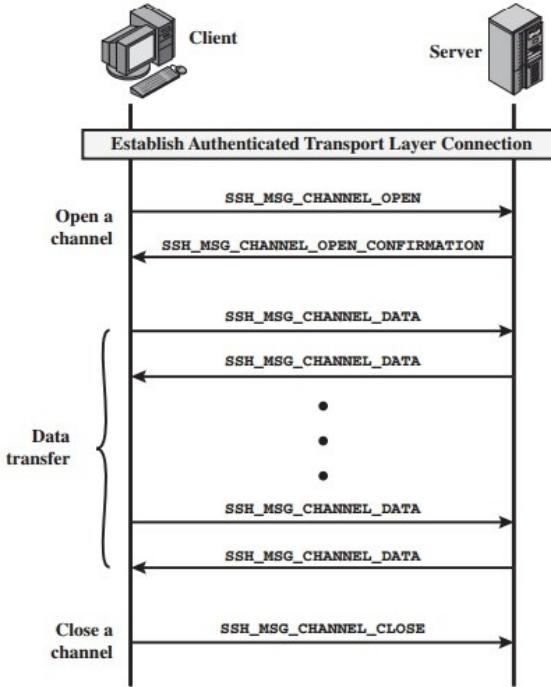
1.4 SSH Connection Protocol

Il protocollo di connessione permette ad una singola connessione SSH, sicura e già autenticata, di essere multiplexata in più canali logici simultaneamente, ciascuno dei quali trasferisce dati in modo bidirezionale [3].

Ogni canale è sottoposto a controllo di flusso: utilizzando un apposito messaggio contenente la dimensione della finestra di ricezione ancora disponibile.

In generale, il protocollo prevede i seguenti passi:

- Quando uno dei due endpoint vuole aprire un nuovo canale, alloca un identificativo numerico per il canale e invia un messaggio SSH MSG CHANNEL OPEN, in cui sarà specificata la tipologia di canale da creare.
- Se l'altra parte è pronta ad aprire il canale, essa risponde al richiedente con un messaggio di SSH MSG CHANNEL OPEN CONFIRMATION, specificando l'identificativo da essa utilizzata (i due identificativi tipicamente non coincidono).
- Su di un canale aperto un endpoint può effettuare all'altro una richiesta di servizio. Questo tipo di richieste avviene tramite messaggi di tipo CHANNEL SERVICE, che specificano il tipo di servizio ed il canale utilizzato.
- i canali possono essere utilizzati per il trasferimento di dati tramite messaggi di tipo SSH MSG CHANNEL DATA, che includono il numero di canale del destinatario e un blocco di dati.
- Quando una delle due parti vuole chiudere il canale, essa manda un messaggio di SSH MSG CHANNEL CLOSE, includendo l'identificativo del canale dell'altro host.



I tipi di canali tipicamente utilizzati sono:

- Session: utilizzati per l'esecuzione remota di comandi e programmi, per le shell remote e le richieste SFTP.
- Direct-tcpip e Forwarded-tcpip: canali utilizzati per i meccanismi di forwarding di SSH, descritti in seguito.
- X11: Canali utilizzati per la trasmissione di informazioni del protocollo X, utilizzato per la gestione remota di finestre grafiche.

1.4.1 Port forwarding

Con il port forwarding, è possibile inviare informazioni utili (che normalmente non sarebbero criptate) attraverso una connessione criptata. Tramite l'SSH Transport Layer, questa funzionalità stabilisce un tunnel sicuro, ottenendo la conversione di una connessione TCP **insicura** in una sessione SSH **sicura** da utilizzare per la comunicazione tra client e server. Un ulteriore vantaggio dell'uso del port forwarding è che, grazie alla crittografia dei tunnel, è possibile aggirare gli sniffer o anche i router mal configurati.

I meccanismi di port forwarding di SSH sono senza dubbio tra le funzionalità più interessanti offerte dal protocollo. Questi si prestano, infatti,

particolarmente bene allo scopo di connettere in sicurezza un host remoto ad una rete locale. Tramite l'utilizzo di queste tecnologie è possibile mettere in sicurezza, sfruttando i meccanismi di autenticazione e confidenzialità di SSH, un servizio nativamente non sicuro.

Questo tipo di interazione avviene a seguito della creazione di uno specifico canale nelle modalità descritte nei paragrafi successivi. A seguito della creazione del canale il traffico viaggerà tra i due endpoint attraverso il tunnel SSH tramite messaggi di tipo CHANNEL_DATA.

1.4.2 Local port forwarding

Questo tecnica di forwarding prevede:

1. Il client SSH è in ascolto del traffico diretto ad una porta TCP e lo dirotta all'interno del tunnel SSH
2. Il server SSH riceve il pacchetto tramite il tunnel crittografato SSH e lo inoltra all'endpoint specificato al momento della creazione del canale.

Una tipica applicazione del direct forwarding SSH è l'utilizzo di questa tecnica per l'interazione con un servizio remoto non sicuro, e di conseguenza configurato in modo che sia raggiungibile esclusivamente dalla rete locale del server. Il servizio potrà essere utilizzato in sicurezza a valle della creazione di una connessione SSH e dell'instaurazione di un meccanismo di forwarding verso la porta d'ascolto di un server SSH appartenente alla rete locale del servizio in questione.

1.4.3 Remote port forwarding

Questo tipo di port forwarding funziona esattamente nel verso opposto. Infatti, si possono considerare i seguenti passi:

1. Il server SSH è in ascolto del traffico diretto ad una porta TCP e lo dirotta all'interno del tunnel SSH
2. Il client SSH riceve il pacchetto e lo ritrasmette alla destinazione specificata al momento della creazione del canale.

1.4.4 Apertura di canale

Questo messaggio viene inviato quando una delle due parti decide di aprire il canale, includendo il numero di canale locale e la dimensione iniziale della finestra nel messaggio.

Sessione Un canale di tipo sessione è iniziato col seguente messaggio:

```
byte      SSH_MSG_CHANNEL_OPEN
string    "session"
uint32   sender channel
uint32   initial window size
uint32   maximum packet size
```

La prima stringa rappresenta il tipo di canale, in questo caso di tipo session. I tre numeri interi successivi rappresentano rispettivamente: un identificatore locale per il canale usato dal mittente di questo messaggio; il numero di byte di dati che possono essere inviati al mittente senza regolare la finestra; la massima dimensione di un pacchetto che può essere inviato al mittente.

In particolare per questo tipo di canale, i client dovrebbero rifiutare qualsiasi richiesta di apertura di tal genere per evitare attacchi da parte di server malevoli.

Canale X11 I canali X11 possono essere creati con una richiesta di apertura di canale apposita, a seguito di una richiesta di forwarding X11 ”req-x11”, inoltrata su di un canale di sessione. Il nuovo canale x11 aperto risulterà indipendente dalla sessione.

```
byte      SSH_MSG_CHANNEL_OPEN
string    "x11"
uint32   sender channel
uint32   initial window size
uint32   maximum packet size
string   originator address (e.g., "192.168.7.38")
uint32   originator port
```

Come si può notare dalla figura, nel payload vengono specificati anche indirizzo e porta del creatore della richiesta. L’RFC impone che tutte le

implementazioni di SSH devono rifiutare qualsiasi messaggio di questo tipo se non hanno richiesto il forwarding X11 in precedenza.

Direct-tcpip Al momento della creazione di una connessione ad una porta TCP/IP per la quale è attivato il forwarding, viene creato un canale di questo tipo attraverso il quale sarà effettuata la ridirezione del traffico. Per la creazione viene inviato il seguente pacchetto all'altro nodo.

```
byte      SSH_MSG_CHANNEL_OPEN
string    "direct-tcpip"
uint32   sender channel
uint32   initial window size
uint32   maximum packet size
string    host to connect
uint32   port to connect
string    originator IP address
uint32   originator port
```

In particolare, l' "host to connect" e la "port to connect" specificano l'host e la porta presso cui il destinatario dovrebbe collegare il canale.

Dall'altro lato, invece, l' "originator IP address" è l'indirizzo IP della macchina da cui la richiesta di connessione viene creata.

Da notare come questi messaggi possono essere anche inviati per delle porte per cui non è stato richiesto esplicitamente alcun forwarding. Il destinatario deve quindi decidere se consentire il forwarding o meno.

Forwarded-tcpip Un endpoint che desidera che le connessioni dell'altro nodo siano reindirizzate verso di lui annuncia la sua volontà tramite un messaggio di tipo GLOBAL_REQUEST, specificando la richiesta "tcpip-forward".

Alla ricezione di questo messaggio l'endpoint può creare un canale di tipo "forwarded-tcpip" tramite il seguente messaggio di tipo CHANNEL_OPEN:

```
byte      SSH_MSG_CHANNEL_OPEN
string    "forwarded-tcpip"
uint32   sender channel
uint32   initial window size
uint32   maximum packet size
string    address that was connected
uint32   port that was connected
string    originator IP address
uint32   originator port
```

Come già detto precedentemente, i client dovrebbero rifiutare questo tipo di richieste di apertura di canale per motivi di sicurezza.

1.4.5 Richieste channel-specific

Molti tipi di canali hanno delle estensioni specifiche per quel tipo di canale. In generale, questi messaggi di richiesta non consumano spazio nella finestra disponibile e quindi possono essere inviati anche se non c'è più spazio nella finestra corrente.

Pseudo-Terminal Uno pseudo-terminal può essere allocato per la sessione mandando il seguente messaggio:

```
byte      SSH_MSG_CHANNEL_REQUEST
uint32    recipient channel
string    "pty-req"
boolean   want_reply
string    TERM environment variable value (e.g., vt100)
uint32    terminal width, characters (e.g., 80)
uint32    terminal height, rows (e.g., 24)
uint32    terminal width, pixels (e.g., 640)
uint32    terminal height, pixels (e.g., 480)
string    encoded terminal modes
```

La seconda stringa rappresenta il tipo di richiesta, in questo caso uno pseudo-terminal. Se il campo "want reply" è false, non verrà inviata nessuna risposta alla richiesta.

Per quanto riguarda questa richiesta in particolare, essa è formata da 4 numeri interi che indicano le dimensioni di larghezza e altezza del terminale, misurate con numero di caratteri, numero di righe e pixel. Infine, c'è una stringa finale che rappresenta le modalità di terminale.

X11 Forwarding La richiesta per l'X11 Forwarding per una sessione è la seguente:

```
byte      SSH_MSG_CHANNEL_REQUEST
uint32    recipient channel
string    "x11-req"
boolean   want reply
boolean   single connection
string    x11 authentication protocol
string    x11 authentication cookie
uint32    x11 screen number
```

E' raccomandato che la stringa "x11 authentication cookie" sia un cookie falso, casuale e che venga controllato e sostituito dal cookie reale quando si riceve una richiesta di connessione. Questa stringa dovrà essere codificata in esadecimale. Se il campo "single connection" è settato a true, solo un'unica

connessione deve essere inoltrata. Nessun'altra connessione sarà inoltrata dopo la prima, o dopo che il canale di sessione è stato chiuso. In ogni caso, forwarding già aperti non dovrebbero essere chiusi in modo automatico alla chiusura del canale di sessione. Infine, il campo "x11 authentication protocol" è il nome del metodo di autenticazione X11 utilizzato.

Variabili d'ambiente Le variabili d'ambiente possono essere passate alla shell o ai comandi eseguiti successivamente:

```
byte      SSH_MSG_CHANNEL_REQUEST
uint32   recipient channel
string    "env"
boolean   want reply
string    variable name
string    variable value
```

E' raccomandato, inoltre, che le implementazioni mantengano una lista di nomi di variabili ammissibili o che impostino le variabili d'ambiente solo dopo che il server abbia concesso i privilegi appositi.

Shell Una volta che una sessione è stata configurata, viene eseguito un programma nel nodo remoto. Nel caso esso sia una shell, si ha il seguente messaggio:

```
byte      SSH_MSG_CHANNEL_REQUEST
uint32   recipient channel
string    "shell"
boolean   want reply
```

In questo caso, verrà eseguita la default shell dell'utente presso il nodo remoto.

Comando remoto In figura, è mostrato un messaggio che richiede che il server inizi l'esecuzione di un comando dato. Il campo "command", infatti, può contenere un path.

```

byte      SSH_MSG_CHANNEL_REQUEST
uint32    recipient channel
string    "exec"
boolean   want reply
string    command

```

Subsystem In questo caso, la richiesta è l'esecuzione di un sottosistema definito nel campo "subsystem name":

```

byte      SSH_MSG_CHANNEL_REQUEST
uint32    recipient channel
string    "subsystem"
boolean   want reply
string    subsystem name

```

Modifica dimensioni della finestra Quando le dimensioni della finestra del terminale cambiano lato client, è possibile che venga inviato un messaggio all'altro endpoint per informarlo riguardo le nuove dimensioni:

```

byte      SSH_MSG_CHANNEL_REQUEST
uint32    recipient channel
string    "window-change"
boolean   FALSE
uint32    terminal width, columns
uint32    terminal height, rows
uint32    terminal width, pixels
uint32    terminal height, pixels

```

Come si può notare dal valore false del campo "want reply", non dovrebbe essere prevista una risposta a tale richiesta.

Controllo di flusso locale Su molti sistemi, è possibile determinare se uno psedu-terminal sta usando il controllo di flusso.

Il messaggio qui mostrato è usato dal server per informare il client su quando può eseguire il controllo di flusso o meno. Se il campo "client can do" è impostato a true, il client ha i permessi per il flow control:

```

byte      SSH_MSG_CHANNEL_REQUEST
uint32   recipient channel
string    "xon-xoff"
boolean   FALSE
boolean   client can do

```

Segnali Un segnale può essere consegnato al processo remoto usando il seguente messaggio. E' da specificare che non tutti i sistemi implementano questo meccanismo, quindi questo messaggio potrebbe essere ignorato.

```

byte      SSH_MSG_CHANNEL_REQUEST
uint32   recipient channel
string    "signal"
boolean   FALSE
string   signal name (without the "SIG" prefix)

```

Exit Status Quando il comando che si stava eseguendo all'altro endpoint termina, il messaggio seguente può essere inviato per mostrare lo stato di uscita del comando. Spesso ritornare lo stato, qui rappresentato come un numero intero su 32 bit, è un'azione consigliata, per motivi legati chiaramente alla sicurezza.

```

byte      SSH_MSG_CHANNEL_REQUEST
uint32   recipient channel
string    "exit-status"
boolean   FALSE
uint32   exit_status

```

Inoltre, il comando potrebbe anche terminare violentemente a causa di un segnale. Tale condizione può essere indicata dal seguente messaggio. In generale, un exit status nullo indica che il comando è terminato con successo.

```

byte      SSH_MSG_CHANNEL_REQUEST
uint32   recipient channel
string    "exit-signal"
boolean   FALSE
string   signal name (without the "SIG" prefix)
boolean   core dumped
string   error message in ISO-10646 UTF-8 encoding
string   language tag [RFC3066]

```

In figura, il campo "error message" contiene una spiegazione testuale aggiuntiva del messaggio d'errore.

2 Il Client SSH

Allo scopo di poter generare traffico SSH e poterlo analizzare nel dettaglio con Wireshark abbiamo optato per la realizzazione di un semplice client SSH in grado di interagire con un server OpenSSH.

Il client fornisce delle funzionalità di autenticazione garantendo:

- Autenticazione anonima
- Autenticazione con username e password
- Autenticazione con public key

A seguito dell'autenticazione, il client mette a disposizione le seguenti funzionalità all'utente:

- Esecuzione di un comando sulla macchina remota
- Creazione di una shell interattiva
- Direct port forwarding

Per decrittografare il traffico generato tra client e server abbiamo optato per la modifica di una libreria SSH in modo da poter avere accesso alle chiavi di sessione.

2.1 Modifica di libssh

Per la realizzazione del client abbiamo utilizzato la libreria libssh. Si tratta di una libreria C che implementa il protocollo SSHv2 e che può essere utilizzata sia per lo sviluppo di client che di server SSH. Allo scopo di poter analizzare il traffico crittografato SSH siamo partiti da una distribuzione di questa libreria disponibile su Github che è stata modificata in modo tale che, una volta instaurata una connessione SSH, stampasse a video le chiavi crittografiche negoziate.

In particolare è stato necessario individuare il momento in cui avviene la negoziazione del materiale crittografico e dunque la struttura in cui tale materiale viene memorizzato. La modifica interessa la funzione *ssh_generate_session_keys* relativa al file *kex.c* presente tra i sorgenti della libreria. La struttura di interesse è invece la *ssh_crypto_struct* allocata all'interno della funzione citata, struttura dati creata durante l'handshake, che memorizza tutte le

informazioni relative ad algoritmi crittografici, chiavi di sessione e di autenticazione utilizzati.

Le informazioni stampate sono rispettivamente L'IV client-server, l'IV server-client , la chiave server-client e la chiave client-server.

A seguito della modifica si è proceduto poi alla ricompilazione del sorgente come libreria statica. La versione modificata è stata impiegata per l'implementazione di un semplice client SSH, al quale è stata linkata durante il processo di compilazione per generare l'eseguibile.

2.2 L'implementazione del client

Il client innanzitutto alloca una sessione SSH utilizzando la funzione di libreria *ssh_new*, e contestualmente ne imposta determinate opzioni tramite la *ssh_option_set* guidata da appositi flag. In particolare:

- **SSH_OPTIONS_HOST**: flag che permette di impostare l' hostname o l'indirizzo IP dell'host sul quale si trova il server SSH. in particolare il nostro server SSH si trova sulla stessa macchina host del client, e quindi è specificato "localhost" come hostname.
- **SSH_OPTIONS_CIPHERS_C_S**: flag che permette di impostare l'algoritmo di crittografia simmetrico da usare nella comunicazione Client-Server, in particolare è stato scelto AES128-cbc (tra gli algoritmi crittografici supportati da SSH).
- **SSH_OPTIONS_CIPHERS_S_C**: flag che permette di impostare l' algoritmo di crittografia simmetrico da usare nella comunicazione Server-Client, in particolare è stato scelto AES128-cbc.

Il client dunque si connette al server SSH utilizzando la funzione di libreria *ssh_connect*. Se c'è un server SSH in ascolto alla porta 22 sull'hostname specificato come opzione, la *ssh_connect* andrà a buon fine e dunque il client procederà alla verifica dell'identità del server chiamando la funzione user-defined *verify_knownhost*.

2.2.1 Autenticazione Server-Client

La funzione *verify_knownhost* legge la chiave pubblica del server (prelevata dalla directory di configurazione del client SSH) dalla sessione utilizzando la funzione di libreria *ssh_get_server_pubkey*. Successivamente è

calcolato l'hash della Puk del server, e memorizzato in un apposito buffer. In particolare è impostato come algoritmo di hashing SHA1 (tramite il flag `SSH_PUBLICKEY_HASH_SHA1`), in quanto openSSH utilizza proprio questo algoritmo per realizzare i digest. Poi con la `ssh_session_is_known_server` il client controlla se la PuK del server per la sessione connessa è conosciuta, verificando l'identità del server a cui si vuole connettere.

2.2.2 Autenticazione Client-Server

L'autenticazione del client presso il server è gestita dalla funzione `authenticate_console`, la quale permette all'utente di selezionare da console quale metodo di autenticazione utilizzare. In particolare nella nostra implementazione l'utente può scegliere tra 3 differenti modalità di autenticazione: none (implementata da `ssh_userauth_none`), Public key (implementata da `ssh_userauth_publickey_auto`), username-password (coppia inserita dall'utente, modalità implementata poi con la `ssh_userauth_password`).

Dunque dopo aver realizzato mutua autenticazione il client SSH permette all'utente di scegliere la funzionalità da utilizzare.

2.2.3 Implementazione Remote Shell

Per la creazione di una remote shell viene anzitutto creato un nuovo canale di connessione tramite le primitive di libreria `ssh_channel_new` e `ssh_channel_open_session`. Su questo canale viene prima effettuata una richiesta di pseudoterminale tramite la funzione `ssh_channel_request_pty`, che è poi configurato tramite il metodo `ssh_channel_change_pty_size`.

Una volta configurato il terminale remoto viene effettuata una richiesta di apertura della remote shell utilizzando `ssh_channel_request_shell`.

Per gestire la scrittura e lettura sul canale sono sfruttate le due funzioni di libreria `ssh_channel_read` e `ssh_channel_write` richiamate fintantochè il canale resta aperto e non è stato inviato un EOF.

Terminata l'interazione il canale è chiuso con le direttive `ssh_channel_send_eof` e `ssh_channel_close`.

2.2.4 Implementazione direct forwarding

All'avvio di questa funzione l'utente inserisce il numero di porto di cui fare il forwarding (`port_source`), il nome simbolico del servizio remoto (`host_destination`)

e il numero di porto su cui è in ascolto tale servizio (`port_destination`). E' successivamente creata una socket TCP e fatto il binding del suo descrittore con il localhost sul porto per il forwarding (ovvero il client SSH è ora in ascolto di connessioni su `localhost:port_source`). Il client SSH si mette quindi in ascolto sulla socket per nuove connessioni (`listen()`) ed estrae la prima connessione pendente nella coda delle richieste (`accept()`), se presente.

Al momento della creazione di una nuova connessione TCP è aperto un nuovo canale di tipo "direct-forwarding" con la primitiva `sshChannelOpenForward` da `localhost:port_source` verso `host_destination:port_destination`.

Finchè il canale verso il server SSH è aperto, ciò che il client SSH riceve sulla socket, tramite la primitiva "recv" è scritto nel canale (`ssh_channel_write()`). Le risposte del servizio sono poi lette dal client SSH sul canale (`sshChannelRead()`) e scritte sulla socket (in modo tale che possono essere ricevute dall'applicativo client effettivo di cui sono state forwardate le connessioni).

2.2.5 Implementazione exec di un comando remoto

Per l'esecuzione di un singolo comando è stata definita la funzione `exec_command()`. Analogamente a quanto visto per la modalità di remote shell, viene inizializzato un canale di tipo sessione. Il comando inserito da tastiera dall'utente è trasmesso sul canale tramite la primitiva `ssh_channel_request_exec()`. La risposta del server relativa all'esecuzione del comando, ottenuta tramite la primitiva `ssh_channel_read` è poi mostrata a video.

3 SSH Wireshark Post Dissector

3.1 Descrizione del plugin

Le informazioni ottenute attraverso la modifica, precedentemente analizzata, di libssh ci permettono di effettuare un’analisi dettagliata del traffico SSH crittografato scambiato tra il nostro client ed il server OpenSSH.

Per lo scopo abbiamo realizzato una estensione di Wireshark in lua che, una volta ricevuto dall’utente i parametri necessari per la decifrazione dei pacchetti, decripta il flusso di messaggi tra i due endpoint ed effettua il parsing dei pacchetti. Il plugin introduce un nuovo protocollo in wireshark denominato ”SSH Payload”. Il pannello di preferenze di questo protocollo consente di fornire al plugin le informazioni ricavate dal client SSH modificato ovvero le due chiavi di crittografia e i due inizializzazione vector utilizzati dagli endpoint, nonchè lo specifico algoritmo crittografico utilizzato.

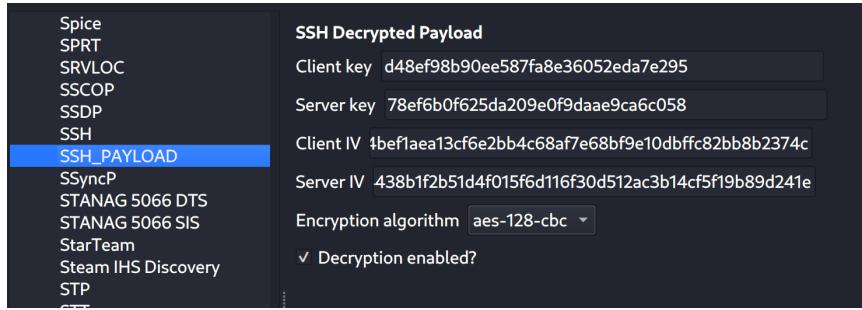


Figura 4: Preferenze del protocollo custom

Il plugin opera a valle della dissezione di default di SSH operata da wireshark. Esso richiede in particolar modo il campo ”ssh.encryptedpacket”, che, se presente all’interno dei campi del dissector tree prodotto dall’analisi di wireshark, identifica un messaggio crittografato di SSH. Nel caso in cui wireshark inserisca erroneamente più messaggi distinti all’interno dello stesso campo, prima dell’analisi i messaggi sono opportunamente distinti utilizzando il campo di lunghezza del messaggio (trasmesso in chiaro nella modalità ETM).

Il plugin identifica la direzione del messaggio tramite il campo ”ssh.direction” in modo da ricostruire i due flussi di messaggi crittografati, memorizzati all’interno di uno stato globale, ed effettua la decodifica utilizzando una libreria di binding che consente al programma lua di utilizzare le funzioni di

OpenSSL. Le informazioni decodificate sono aggiunte allo stato allo scopo di evitare di dover effettuare la decodifica ogni volta che il dissettore è invocato da wireshark. Il messaggio in chiaro viene successivamente passato alla funzione "parse", che lo analizza, ne estraie i campi fondamentali e li aggiunge al dissection tree e modifica il testo mostrato nella colonna info associata ad ogni pacchetto analizzato.

3.2 Catture di esempio

Allo scopo di testare il plugin ed analizzare con esperimenti reali ciò che abbiamo studiato sul protocollo SSH abbiamo realizzato alcuni scenari di esempio in cui il nostro plugin è adoperato per l'analisi.

3.2.1 Autenticazione

Abbiamo inizialmente analizzato la fase di autenticazione realizzando due scenari distinti, con modalità di autenticazione ed esiti differenti.

Tentativi di autenticazione None e Password Nel primo caso il client tenta l'autenticazione prima utilizzando la modalità di autenticazione NONE, ma il server OpenSSHD è configurato per rifiutare questo tipo di autenticazioni. Il client tenta quindi l'accesso con username e password, ma con credenziali errate. Successivamente il client tenta il login correggendo username e password, ma riceve un messaggio di disconnessione dal server poiché questo non consente di modificare l'username precedentemente inserito.

No.	Time	Source/Dest Protocol	Length	Info
4	0.00...	12... 1... SSHv2	98	Client: Protocol (SSH-2.0-libssh_0.10.00)
6	0.02...	12... 1... SSHv2	107	Server: Protocol (SSH-2.0-OpenSSH_8.2p1 Ubuntu-4ubuntu0.5)
8	0.02...	12... 1... SSHv2	722	Client: Key Exchange Init
10	0.02...	12... 1... SSHv2	922	Server: Key Exchange Init
12	0.02...	12... 1... SSHv2	116	Client: New Keys
14	0.02...	12... 1... SSHv2	518	Server: Elliptic Curve Diffie-Hellman Key Exchange Reply, New Keys, Encrypted packet (len=244), SSH_MSG_EXT_INFO (7)
16	0.03...	12... 1... SSHv2	82	Client: New Keys
18	3.04...	12... 1... SSHv2	134	Client: Encrypted packet (len=68), SSH_MSG_SERVICE_REQUEST (5)
20	3.04...	12... 1... SSHv2	134	Server: Encrypted packet (len=68), SSH_MSG_SERVICE_ACCEPT (6)
21	3.04...	12... 1... SSHv2	150	Client: Encrypted packet (len=84), SSH_MSG_USERAUTH_REQUEST (50)
23	3.05...	12... 1... SSHv2	134	Server: Encrypted packet (len=68), SSH_MSG_USERAUTH_FAILURE (51)
25	8.34...	12... 1... SSHv2	166	Client: Encrypted packet (len=100), SSH_MSG_USERAUTH_REQUEST (50)
27	10.7...	12... 1... SSHv2	138	Server: Encrypted packet (len=68), SSH_MSG_USERAUTH_FAILURE (51)
29	26.2...	12... 1... SSHv2	166	Client: Encrypted packet (len=100), SSH_MSG_USERAUTH_REQUEST (50)
31	26.2...	12... 1... SSHv2	214	Server: Encrypted packet (len=148), SSH_MSG_DISCONNECT (1)

Figura 5: Autenticazione, messaggi scambiati

I primi messaggi che i due endpoint si trasmettono sono in chiaro e sono necessari per concordare la suite crittografica da utilizzare, il meccanismo di scambio delle chiavi, lo scambio dei cookie e del materiale crittografico necessario per ricavare le chiavi di sessione.

Nella prima interazione tra i due endpoint essi si scambiano dettagli relativi alla versione e all'implementazione di SSH utilizzata. Seguono due messaggi di KeyExchangeInit in cui sono comunicati tutti i dettagli della suite crittografica utilizzata e i cookie di sessione dei due endpoint. I due endpoint, concordato il meccanismo Elliptic Curve D-H per lo scambio delle chiavi, si scambiano le chiavi pubbliche D-H. Il server, per autenticarsi firma con la sua chiave privata questo valore. A seguito dei messaggi NEW_KEYS, che fanno da strobe terminatori per lo scambio delle chiavi, i due endpoint iniziano a crittare la comunicazione.

Una volta effettuato lo scambio delle chiavi il client richiede ad un server un servizio. Il servizio richiesto è quello di "user-auth", utilizzato per chiedere al server di avviare la fase di autenticazione del client.

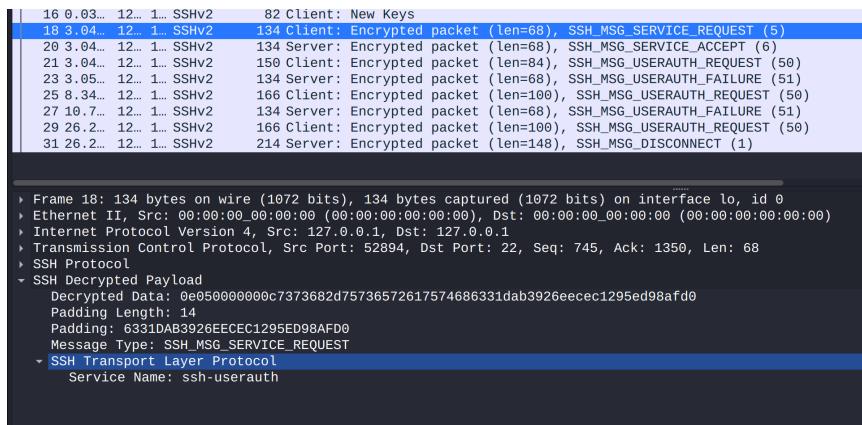


Figura 6: Richiesta del tipo "user-auth"

Il server accetta la richiesta rispondendo con un messaggio di SSH_MSG_SERVICE_ACCEPT. Inizia a questo punto la fase di autenticazione. Il client invia il primo messaggio specifico del protocollo di autenticazione, una richiesta di USER_AUTH. Analizzando nel dettaglio il pacchetto vediamo come la modalità di autenticazione richiesta è none. Gli altri campi presenti all'interno di questa richiesta sono il service name, impostato ad "ssh-connection" per tutti i protocolli di autenticazione.

```

21 3.04... 12... 1... SSHv2      150 Client: Encrypted packet (len=84), SSH_MSG_USERAUTH_REQUEST (50)
23 3.05... 12... 1... SSHv2      134 Server: Encrypted packet (len=68), SSH_MSG_USERAUTH_FAILURE (51)
25 8.34... 12... 1... SSHv2      166 Client: Encrypted packet (len=100), SSH_MSG_USERAUTH_REQUEST (50)
27 10.7... 12... 1... SSHv2      134 Server: Encrypted packet (len=68), SSH_MSG_USERAUTH_FAILURE (51)
29 26.2... 12... 1... SSHv2      166 Client: Encrypted packet (len=100), SSH_MSG_USERAUTH_REQUEST (50)
31 26.2... 12... 1... SSHv2      214 Server: Encrypted packet (len=148), SSH_MSG_DISCONNECT (1)

Frame 21: 150 bytes on wire (1200 bits), 150 bytes captured (1200 bits) on interface lo, id 0
Ethernet II, Src: 00:00:00:00:00:00 (00:00:00:00:00:00), Dst: 00:00:00:00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 52894, Dst Port: 22, Seq: 813, Ack: 1418, Len: 84
SSH Protocol
SSH Decrypted Payload
Decrypted data: 0e3200000002736f000000e7373682d636f6e6e656374696f6e000000046e6f6e653...
Padding Length: 14
Padding: 33EF4225FC3A0C82516FE0D2767F
Message Type: SSH_MSG_USERAUTH_REQUEST
- SSH Authentication
  User Name: so
  Service Name: ssh-connection
  Auth Method: none

```

Figura 7: Autenticazione con modalità none

Il server risponde con un messaggio di tipo USERAUTH_FAILURE, esso è infatti configurato in modo tale da non supportare questa modalità. Nel messaggio di failure il server specifica quali sono i tipi di autenticazione supportati. Questo tipo di autenticazione viene spesso tentata dai client allo scopo di comunicare il proprio username al server ed ottenere da questo le modalità di autenticazione supportate.

```

21 3.04... 12... 1... SSHv2      150 Client: Encrypted packet (len=84), SSH_MSG_USERAUTH_REQUEST (50)
23 3.05... 12... 1... SSHv2      134 Server: Encrypted packet (len=68), SSH_MSG_USERAUTH_FAILURE (51)
25 8.34... 12... 1... SSHv2      166 Client: Encrypted packet (len=100), SSH_MSG_USERAUTH_REQUEST (50)
27 10.7... 12... 1... SSHv2      134 Server: Encrypted packet (len=68), SSH_MSG_USERAUTH_FAILURE (51)
29 26.2... 12... 1... SSHv2      166 Client: Encrypted packet (len=100), SSH_MSG_USERAUTH_REQUEST (50)
31 26.2... 12... 1... SSHv2      214 Server: Encrypted packet (len=148), SSH_MSG_DISCONNECT (1)

Frame 23: 134 bytes on wire (1072 bits), 134 bytes captured (1072 bits) on interface lo, id 0
Ethernet II, Src: 00:00:00:00:00:00 (00:00:00:00:00:00), Dst: 00:00:00:00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 22, Dst Port: 52894, Seq: 1418, Ack: 897, Len: 68
SSH Protocol
SSH Decrypted Payload
Decrypted Data: 073300000127075626c69636b65792c70617373776f72640065a53604a2111d
Padding Length: 7
Padding: 65A53604A2111D
Message Type: SSH_MSG_USERAUTH_FAILURE
- SSH Authentication
  Auth That Can Continue: publickey,password
  Partial Success: false

```

Figura 8: Autenticazione none fallita.

Il messaggio successivo è quello di autenticazione con username e password, presenti all'interno del messaggio in appositi campi. Notiamo inoltre la presenza di un campo che specifica il tipo di autenticazione, in questo caso basata su password.

```

23 3.05.. 12.. 1.. SSHv2      134 Server: Encrypted packet (len=68), SSH_MSG_USERAUTH_FAILURE (51)
25 8.34.. 12.. 1.. SSHv2      166 Client: Encrypted packet (len=100), SSH_MSG_USERAUTH_REQUEST (50)
27 10.7.. 12.. 1.. SSHv2      134 Server: Encrypted packet (len=68), SSH_MSG_USERAUTH_FAILURE (51)
29 26.2.. 12.. 1.. SSHv2      166 Client: Encrypted packet (len=100), SSH_MSG_USERAUTH_REQUEST (50)
31 26.2.. 12.. 1.. SSHv2      214 Server: Encrypted packet (len=148), SSH_MSG_DISCONNECT (1)

.....
> Frame 25: 166 bytes on wire (1328 bits), 166 bytes captured (1328 bits) on interface lo, id 0
> Ethernet II, Src: 00:00:00:00:00:00 (00:00:00:00:00:00), Dst: 00:00:00:00:00:00 (00:00:00:00:00:00)
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> Transmission Control Protocol, Src Port: 52894, Dst Port: 22, Seq: 897, Ack: 1486, Len: 100
> SSH Protocol
> SSH Decrypted Payload
Decrypted data: 113200000002736f0000000e7373682d636f6e6e656374696f6e00000008706173737...
Padding Length: 17
Padding: 1F228BF4FC5FA527C1BC4FD1B0EB8BA82
Message Type: SSH_MSG_USERAUTH_REQUEST
-> SSH Authentication
    User Name: so
    Service Name: ssh-connection
    Auth Method: password
    Password: ciao

```

Figura 9: USER_AUTH_REQUEST con username e password

Dopo un ulteriore tentativo di accesso in cui il client modifica l'username utilizzato per l'autenticazione, il server forza la disconnessione con un messaggio del protocollo trasporto, specificando, tramite il campo reason string, che la causa della disconnessione forzata è proprio la modifica dell'username, che non è consentita.

```

29 20.2.. 12.. 1.. SSHv2      107 Client: Encrypted packet (len=100), SSH_MSG_USERAUTH_REQUEST (50)
31 26.2.. 12.. 1.. SSHv2      214 Server: Encrypted packet (len=148), SSH_MSG_DISCONNECT (1)

.....
> Frame 31: 214 bytes on wire (1712 bits), 214 bytes captured (1712 bits) on interface lo, id 0
> Ethernet II, Src: 00:00:00:00:00:00 (00:00:00:00:00:00), Dst: 00:00:00:00:00:00 (00:00:00:00:00:00)
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> Transmission Control Protocol, Src Port: 22, Dst Port: 52894, Seq: 1554, Ack: 1097, Len: 148
> SSH Protocol
> SSH Decrypted Payload
Decrypted data: 0b0100000002000000574368616e6765206f6620757365726e61d65206f722073657...
Padding Length: 11
Padding: 2D630155673E29E1B290DB
Message Type: SSH_MSG_DISCONNECT
-> SSH Transport Layer Protocol
Reason Code: SSH_DISCONNECT_PROTOCOL_ERROR
Reason String: Change of username or service not allowed: (so,ssh-connection) -> (ciao,ssh-connection)
Language Tag:

```

Figura 10: Messaggio di disconnessione del server.

Autenticazione con public key In questa seconda cattura l'utente effettua tramite il client l'autenticazione utilizzando una chiave pubblica correttamente registrata presso il server.

```

4 0.00.. 12.. 1.. SSHv2      90 Client: Protocol (SSH-2.0-libssh_0.10.90)
6 0.02.. 12.. 1.. SSHv2      107 Server: Protocol (SSH-2.0-OpenSSH_8.2pi Ubuntu-4ubuntu0.5)
8 0.02.. 12.. 1.. SSHv2      722 Client: Key Exchange Init
10 0.02.. 12.. 1.. SSHv2      922 Server: Key Exchange Init
12 0.03.. 12.. 1.. SSHv2      114 Client: Elliptic Curve Diffie-Hellman Key Exchange Init
14 0.04.. 12.. 1.. SSHv2      518 Server: Elliptic Curve Diffie-Hellman Key Exchange Reply, New Keys, Encrypted packet (len=244), SSH_MSG_EXT_INFO (7)
16 0.04.. 12.. 1.. SSHv2      82 Client: New Keys
19 19.3.. 12.. 1.. SSHv2      134 Client: Encrypted packet (len=68), SSH_MSG_SERVICE_REQUEST (5)
21 19.3.. 12.. 1.. SSHv2      134 Server: Encrypted packet (len=68), SSH_MSG_SERVICE_ACCEPT (6)
22 19..3.. 12.. 1.. SSHv2      582 Client: Encrypted packet (len=516), SSH_MSG_USERAUTH_REQUEST (50)
24 19..4.. 12.. 1.. SSHv2      550 Server: Encrypted packet (len=484), SSH_MSG_USERAUTH_PK_OK (60)
26 30..1.. 12.. 1.. SSHv2      982 Client: Encrypted packet (len=916), SSH_MSG_USERAUTH_REQUEST (50)
28 30..1.. 12.. 1.. SSHv2      118 Server: Encrypted packet (len=52), SSH_MSG_USERAUTH_SUCCESS (52)
30 30..1.. 12.. 1.. SSHv2      134 Client: Encrypted packet (len=68), SSH_MSG_CHANNEL_OPEN (90)
32 30..6.. 12.. 1.. SSHv2      874 Server: Encrypted packet (len=888), SSH_MSG_GLOBAL_REQUEST (80), SSH_MSG_DEBUG (4)
34 30..2.. 49.. 1.. SSHv2      220 Client: Encrypted packet (len=144), SSH_MSG_DEBUG (1)

```

Una particolarità che notiamo da questa cattura, specifica di questo tipo di autenticazione, è che il client effettua due richieste distinte di autenticazione. Nella prima il flag *authentication request* è impostato a false ed il client richiede al server la validità del proprio metodo di autenticazione, specificato nel campo public key Alg, ed eventualmente del proprio certificato, presente nel campo public key blob.

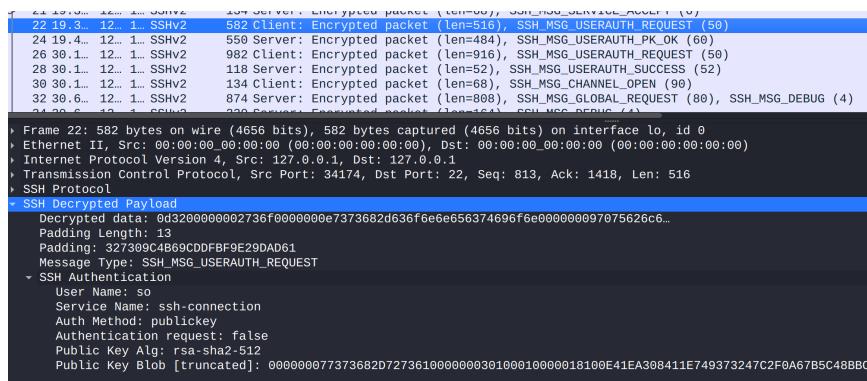


Figura 11: Messaggio di discovery.

A questo messaggio, in caso di validità dei parametri dell'autenticazione, il server autorizza il client ad effettuare la richiesta di autenticazione con il messaggio SSH_USERAUTH_PK_OK, in cui ritrasmette il public key blob che ha ricevuto. Il client effetta quindi la richiesta di autenticazione specificando la propria chiave pubblica ed una firma realizzata crittando questa chiave con la propria chiave privata.

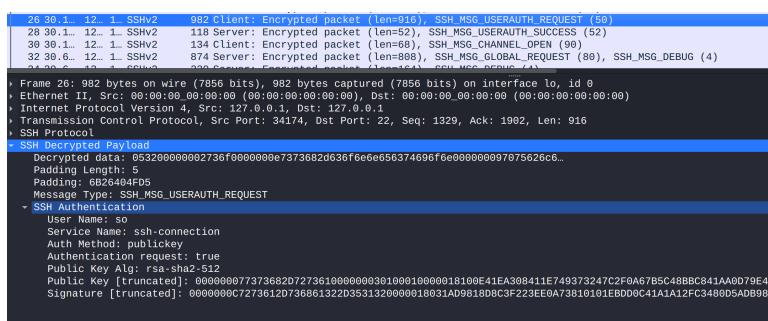


Figura 12: Messaggio di autenticazione.

L'autenticazione ha in questo caso successo, il server risponde infatti con un messaggio di tipo SSH_USERAUTH_SUCCESS.

3.2.2 Remote shell con canale di sessione

Questa cattura, che inizia successivamente alla fase di autenticazione mostrata in precedenza, mostra la creazione di un canale di sessione allo scopo di interagire con il server tramite una Remote Shell.

28 30.1.. 12.. 1. SSHV2	118 Server: Encrypted packet (len=52), SSH_MSG_USERAUTH_SUCCESS (52)
30 30.6.. 12.. 1. SSHV2	134 Client: Encrypted packet (len=68), SSH_MSG_CHANNEL_OPEN (90)
32 30.6.. 12.. 1. SSHV2	874 Server: Encrypted packet (len=808), SSH_MSG_GLOBAL_REQUEST (80), SSH_MSG_DEBUG (4)
34 30.6.. 12.. 1. SSHV2	230 Server: Encrypted packet (len=164), SSH_MSG_DEBUG (4)
36 30.6.. 12.. 1. SSHV2	134 Server: Encrypted packet (len=68), SSH_MSG_CHANNEL_OPEN_CONFIRMATION (91)
38 30.6.. 12.. 1. SSHV2	166 Client: Encrypted packet (len=100), SSH_MSG_CHANNEL_REQUEST (98)
40 30.6.. 12.. 1. SSHV2	118 Server: Encrypted packet (len=52), SSH_MSG_CHANNEL_SUCCESS (99)
42 30.6.. 12.. 1. SSHV2	150 Client: Encrypted packet (len=84), SSH_MSG_CHANNEL_REQUEST (98)
44 30.6.. 12.. 1. SSHV2	134 Client: Encrypted packet (len=68), SSH_MSG_CHANNEL_REQUEST (98)
46 30.6.. 12.. 1. SSHV2	170 Server: Encrypted packet (len=104), SSH_MSG_CHANNEL_WINDOW_ADJUST (93), SSH_MSG_CHANNEL_SUCCESS (99)
47 30.6.. 12.. 1. SSHV2	758 Server: Encrypted packet (len=692), SSH_MSG_CHANNEL_DATA (94)
49 30.6.. 12.. 1. SSHV2	118 Client: Encrypted packet (len=52), SSH_MSG_CHANNEL_WINDOW_ADJUST (93)
51 30.6.. 12.. 1. SSHV2	150 Server: Encrypted packet (len=84), SSH_MSG_CHANNEL_DATA (94)
53 36.4.. 12.. 1. SSHV2	134 Client: Encrypted packet (len=68), SSH_MSG_CHANNEL_DATA (94)
55 36.4.. 12.. 1. SSHV2	374 Server: Encrypted packet (len=308), SSH_MSG_CHANNEL_DATA (94)
57 39.5.. 12.. 1. SSHV2	134 Client: Encrypted packet (len=68), SSH_MSG_CHANNEL_DATA (94)
58 39.5.. 12.. 1. SSHV2	134 Server: Encrypted packet (len=68), SSH_MSG_CHANNEL_DATA (94)
60 39.5.. 12.. 1. SSHV2	166 Server: Encrypted packet (len=100), SSH_MSG_CHANNEL_DATA (94)
62 42.2.. 12.. 1. SSHV2	134 Client: Encrypted packet (len=68), SSH_MSG_CHANNEL_DATA (94)
63 42.2.. 12.. 1. SSHV2	198 Server: Encrypted packet (len=132), SSH_MSG_CHANNEL_DATA (94)
65 42.2.. 12.. 1. SSHV2	166 Server: Encrypted packet (len=100), SSH_MSG_CHANNEL_DATA (94)

Il primo messaggio inoltrato dal client ha lo scopo di creare il canale. In questo messaggio, di tipo CHANNEL_OPEN, viene specificato il tipo del canale ed il numero a questo associato (lato client). Il canale creato è di tipo session.

```
[Direction: client-to-server]
└ SSH Decrypted Payload
  Decrypted Data: 075a00000007736573736961
  Padding Length: 7
  Padding: 91F47CDCADBE70
  Message Type: SSH_MSG_CHANNEL_OPEN
  └ SSH Connection
    Channel Type: session
    Channel Number: 43
    Window Size: 64000
    Max Packet Size: 32768
```

Figura 13: Richiesta di apertura di canale di sessione.

Il server conferma l'apertura del canale con il messaggio di channel open confirmation, specificando il codice del recipient channel, ovvero l'identificativo lato server di questo canale.

```
| 34 30 .. 12 ... SSHV2      239 Server: Encrypted packet (len=164), SSH_MSG_DEBUG (4)
| 35 30 .. 12 ... SSHV2      134 Server: Encrypted packet (len=68), SSH_MSG_CHANNEL_OPEN_CONFIRMATION (91)
| 38 30 .. 12 ... SSHV2      166 Client: Encrypted packet (len=100), SSH_MSG_CHANNEL_REQUEST (98)
| 40 30 .. 12 ... SSHV2      118 Server: Encrypted packet (len=52), SSH_MSG_CHANNEL_SUCCESS (99)

Frame 36: 134 bytes on wire (1072 bits), 134 bytes captured (1072 bits) on interface lo, id 0
Ethernet II, Src: Intel PRO/100 MT [00:00:00:00:00:00], Dst: Intel PRO/100 MT [00:00:00:00:00:00]
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 22, Dst Port: 34174, Seq: 2926, Ack: 2313, Len: 68
SSH Protocol
SSH Decrypted Payload
Decrypted Data: 0e5bb0000002b0900000000000000000000000000d4fee01436d730f046032a1d664
Padding Length: 14
Padding: DCCFEE01436D730f046032A1D664
Message Type: SSH_MSG_CHANNEL_OPEN_CONFIRMATION
- SSH Connection
  Channel Number: 43
  Recipient Channel Number: 0
  Window Size: 0
  Max Packet Size: 32768
```

Il client a questo punto effettua prima una "pty-request", richiedendo un terminale remoto e specificandone alcune caratteristiche di risoluzione, e, dopo la conferma del server, effettua una richiesta di tipo shell.

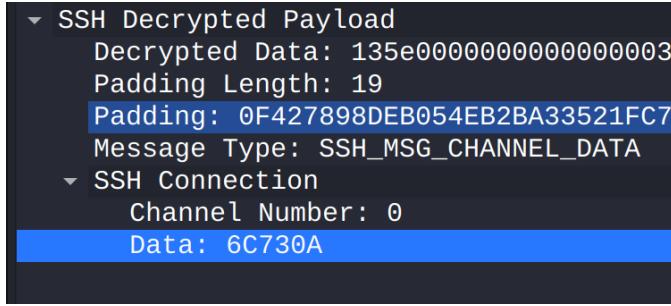
```
36 30.. 12.. 1.. SSHV2      134 Server: Encrypted packet (len=68), SSH_MSG_CHANNEL_OPEN_CONFIRMATION (0)
38 30.. 1.. 1.. SSHV2       166 Client: Encrypted packet (len=100), SSH_MSG_CHANNEL_REQUEST (98)
40 30.. 12.. 1.. SSHV2       118 Server: Encrypted packet (len=52), SSH_MSG_CHANNEL_SUCCESS (99)
42 30.. 12.. 1.. SSHV2       150 Client: Encrypted packet (len=84), SSH_MSG_CHANNEL_REQUEST (98)
44 30.. 12.. 1.. SSHV2       124 Client: Encrypted packet (len=68), SSH_MSG_CHANNEL_REQUEST (98)

> Frame 38: 166 bytes on wire (1328 bits), 166 bytes captured (1328 bits) on interface lo, id 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> Transmission Control Protocol, Src Port: 34174, Dst Port: 22, Seq: 2313, Ack: 2994, Len: 100
> SSH Protocol
> SSH Decrypted Payload
  Decrypted data: 106200000000000000000077074792d726571010000005787465726d00000050000000...
  Padding Length: 16
  Padding: 8E0A2C0D7A39ABC71B59B0F90EA25D23
Message Type: SSH_MSG_CHANNEL_REQUEST
```

Dopo la conferma da parte del server inizia uno scambio interattivo tra i due terminali di dati tramite messaggi di tipo channel_data

51	30..6	12..	1..	SSHv2	150	Server:	Encrypted packet (len=84), SSH_MSG_CHANNEL_DATA (94)
53	36..4..	12..	1..	SSHv2	134	Client:	Encrypted packet (len=68), SSH_MSG_CHANNEL_DATA (94)
55	36..4..	12..	1..	SSHv2	374	Server:	Encrypted packet (len=308), SSH_MSG_CHANNEL_DATA (94)
57	39..5..	12..	1..	SSHv2	134	Client:	Encrypted packet (len=68), SSH_MSG_CHANNEL_DATA (94)
58	39..5..	12..	1..	SSHv2	134	Server:	Encrypted packet (len=68), SSH_MSG_CHANNEL_DATA (94)
60	39..5..	12..	1..	SSHv2	166	Server:	Encrypted packet (len=100), SSH_MSG_CHANNEL_DATA (94)
62	42..2..	12..	1..	SSHv2	134	Client:	Encrypted packet (len=68), SSH_MSG_CHANNEL_DATA (94)
63	42..2..	12..	1..	SSHv2	198	Server:	Encrypted packet (len=132), SSH_MSG_CHANNEL_DATA (94)
65	42..2..	12..	1..	SSHv2	166	Server:	Encrypted packet (len=100), SSH_MSG_CHANNEL_DATA (94)

Un esempio è questo messaggio i cui dati, decodificati in ascii, corrispondono al comando "ls".



3.2.3 Direct forwarding

Abbiamo poi analizzato la funzionalità di direct forwarding di SSH. Come precedentemente descritto, il client da noi implementato, consente un semplice meccanismo di direct forwarding, che, tramite l'utilizzo delle socket e delle funzionalità offerte da libssh e da OpenSSHD consente di reindirizzare il traffico TCP da una specifica porta della macchina client, tramite il tunnel SSH, al server. Per analizzare una semplice interazione di questo tipo abbiamo effettuato una richiesta GET HTTP alla porta soggetta al forwarding ed abbiamo analizzato il risultato.

A valle del key exchange e dell'autenticazione il client richiede la creazione di un canale di tipo "direct-tcpip" tramite un messaggio di tipo SSH_MSG_CHANNEL_OPEN, specificandone, oltre che il numero, porta, indirizzo di origine e porta di destinazione.

```

25 1.30... 12.. 1.. SSHv2      982 Client: Encrypted packet (len=916), SSH_MSG_USERAUTH_REQUEST (56)
26 1.34... 12.. 1.. SSHv2      118 Server: Encrypted packet (len=52), SSH_MSG_USERAUTH_SUCCESS (52)
27 1.34... 12.. 1.. SSHv2      182 Client: Encrypted packet (len=116), SSH_MSG_CHANNEL_OPEN (90)
29 1.56... 12.. 1.. SSHv2      874 Server: Encrypted packet (len=808), SSH_MSG_GLOBAL_REQUEST (80),
35 1.60... 12.. 1.. SSHv2      298 Server: Encrypted packet (len=232), SSH_MSG_DEBUG (4), SSH_MSG_C
44 15.4... 12.. 1.. SSHv2      134 Client: Encrypted packet (len=68), SSH_MSG_CHANNEL_DATA (94)
46 15.5... 12.. 1.. SSHv2      2950 Server: Encrypted packet (len=2884), SSH_MSG_CHANNEL_DATA (94)
48 15.5... 12.. 1.. SSHv2      118 Client: Encrypted packet (len=52), SSH_MSG_CHANNEL_WINDOW_ADJUST

> Frame 27: 182 bytes on wire (1456 bits), 182 bytes captured (1456 bits) on interface lo, id 0
  Ethernet II, Src: 00:00:00:00:00:00 (00:00:00:00:00:00), Dst: 00:00:00:00:00:00 (00:00:00:00:00:00)
  Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
  Transmission Control Protocol, Src Port: 59384, Dst Port: 22, Seq: 2245, Ack: 1954, Len: 116
    SSH Protocol
      - SSH Decrypted Payload
        Decrypted data: 0c5a000000c6469726563742d7463706970000002b0000fa0000080000000000d7...
        Padding Length: 12
        Padding: 243B0E8F249ACCB5E98B8A18
        Message Type: SSH_MSG_CHANNEL_OPEN
      - SSH Connection
        Channel Type: direct-tcpip
        Channel Number: 43
        Recipient Address: www.google.it
        Recipient Port: 80
        Originator Address: localhost
        Originator Port: 5555

```

Dopo la creazione del canale il client inoltra al server la richiesta sotto forma di un messaggio di tipo SSH_MSG_CHANNEL_DATA. I dati contenuti in questa richiesta, dopo aver effettuato una conversione ASCII, corrispondono alla richiesta GET che abbiamo sottoposto alla porta 5555 del client utilizzando Netcat.

```

Client: Encrypted packet (len=116), SSH_MSG_CHANNEL_OPEN (90)
Server: Encrypted packet (len=808), SSH_MSG_GLOBAL_REQUEST (80), SSH_MSG_DEBUG (4)
Server: Encrypted packet (len=232), SSH_MSG_DEBUG (4), SSH_MSG_CHANNEL_OPEN_CONFIRMATION (91)
Client: Encrypted packet (len=68), SSH_MSG_CHANNEL_DATA (94)
Server: Encrypted packet (len=2884), SSH_MSG_CHANNEL_DATA (94)
Client: Encrypted packet (len=52), SSH_MSG_CHANNEL_WINDOW_ADJUST (93)
Client: Encrypted packet (len=52), SSH_MSG_CHANNEL_EOF (66)

> Frame 44: 134 bytes on wire (1072 bits), 134 bytes captured (1072 bits) on interface lo, id 0
  Ethernet II, Src: 00:00:00:00:00:00 (00:00:00:00:00:00), Dst: 127.0.0.1 (00:00:00:00:00:00)
  Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
  Transmission Control Protocol, Src Port: 59384, Dst Port: 22, Seq: 2361, Ack: 2994, Len: 68
    SSH Protocol
      - SSH Decrypted Payload
        Decrypted Data: 055e000000000000000011474554202f20485454502f312e310a0a0a8857981973
        Padding Length: 5
        Padding: 8857981973
        Message Type: SSH_MSG_CHANNEL_DATA
      - SSH Connection
        Channel Number: 0
        Data: 474554202f20485454502f312e310a0a0a

```

A seguito di questo messaggio il client comunica che non utilizzerà ulteriormente il canale di comunicazione, trasmettendo sul canale un messaggio di EOF.

Nel resto della cattura osserviamo vari pacchetti di tipo channel_data che vengono indirizzati dal server al client e che contengono la risposta alla richiesta GET ottenuta dal server SSH. Notiamo inoltre un messaggio di CHANNEL_WINDOW_ADJUST che aumenta la dimensione della finestra di ricezione del client.

```

46 15.5.. 12... 1... SSHV2      2950 Server: Encrypted packet (len=2884), SSH_MSG_CHANNEL_DATA (94)
48 15.5.. 12... 1... SSHV2      118 Client: Encrypted packet (len=52), SSH_MSG_CHANNEL_WINDOW_ADJUST (93)
53 15.5.. 12... 1... SSHV2      118 Client: Encrypted packet (len=52), SSH_MSG_CHANNEL_EOF (96)
56 16.6.. 12... 1... SSHV2      1590 Server: Encrypted packet (len=1524), SSH_MSG_CHANNEL_DATA (94)

Frame 48: 118 bytes on wire (944 bits), 118 bytes captured (944 bits) on interface lo, id 0
Ethernet II, Src: 00:00:00:00:00:00 (00:00:00:00:00:00), Dst: 00:00:00:00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 59384, Dst Port: 22, Seq: 2429, Ack: 5878, Len: 52
SSH Protocol
SSH Decrypted Payload
    Decrypted Data: 065d000000000000129908f078c988f47f
    Padding Length: 6
    Padding: F078C988F47F
    Message Type: SSH_MSG_CHANNEL_WINDOW_ADJUST
    SSH Connection
        Channel Number: 0
        Bytes To Add: 1218824

```

Terminato l'invio della pagina HTML il server trasmette il messaggio di EOF sul canale e i due endpoint procedono alla chiusura del canale.

```

Server: Encrypted packet (len=1812), SSH_MSG_CHANNEL_DATA (94)
Server: Encrypted packet (len=104), SSH_MSG_CHANNEL_EOF (96), SSH_MSG_CHANNEL_CLOSE (97)
Client: Encrypted packet (len=52), SSH_MSG_CHANNEL_CLOSE (97)

> Frame 80: 118 bytes on wire (944 bits), 118 bytes captured (944 bits) on interface lo, id 0
> Ethernet II, Src: 00:00:00:00:00:00 (00:00:00:00:00:00), Dst: 00:00:00:00:00:00 (00:00:00:00:00:00)
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> Transmission Control Protocol, Src Port: 59384, Dst Port: 22, Seq: 2533, Ack: 53610, Len: 52
> SSH Protocol
> SSH Decrypted Payload
    Decrypted Data: 0a6100000000dfe1ee4b11020a2c6671
    Padding Length: 10
    Padding: DFE1EE4B11020A2C6671
    Message Type: SSH_MSG_CHANNEL_CLOSE
    SSH Connection
        Channel Number: 0

```

References

- [1] S Lehtinen and C Lonwick. The secure shell (ssh) protocol assigned numbers. Technical report, 2006.
- [2] Tatu Ylonen and Chris Lonwick. The secure shell (ssh) authentication protocol. Technical report, 2006.
- [3] Tatu Ylonen and Chris Lonwick. The secure shell (ssh) connection protocol. Technical report, 2006.
- [4] Tatu Ylonen and Chris Lonwick. The secure shell (ssh) protocol architecture. Technical report, 2006.
- [5] Tatu Ylonen and Chris Lonwick. The secure shell (ssh) transport layer protocol. Technical report, 2006.