Ruben Boero and Liam Keane
Fall 2024
Advised by Jeff Ondich

Implementing an SSH Client

SSH is a tool that allows secure, remote connection to another machine. It is primarily used to gain remote access to a terminal/shell. The SSH exchange is carried out on top of a TCP connection. The SSH exchange has 5 main sections: protocol version exchange, key exchange initialization, Diffie-Hellman (DH) key exchange, and connection protocol. All sent packets are encoded according to the Binary Packet Protocol.

How to Run Our Client

Instructions to run our SSH client implementation can be found in the README of our GitHub repository. In order for this client to connect to a remote machine, the remote machine must be running an SSH server. Our client was built to connect to the OpenSSH server implementation. The specific server version that we used to test is: OpenSSH_9.7p1 Debian-7, OpenSSL 3.3.2 (the pre-installed Kali Linux SSH server). We have a brief explanation of how to start an SSH server on Kali Linux on our GitHub.

Binary Packet Protocol

Every packet sent during an SSH exchange must be encoded according to the Binary Packet Protocol. Each packet has the following fields (in order):

- Packet length
- Padding length
- Payload
- Random padding
- Message authentication code (MAC) (optional)

These fields are concatenated together to form the SSH packet. For example:

- SSH packet = (4-byte packet length || 1-byte padding length || payload || padding || MAC)

Packet length is an unsigned 4 byte integer encoded in network order (big-endian). The packet length should not include the MAC or the 4 bytes that represent the length of the packet.

Padding length is a single byte.

Payload contains the contents of the message.

Random padding is a sequence of bytes with random values used to pad out the packet to be a multiple of the cipher block size or 8, whichever is larger. The minimum number of padding bytes is 4.

MAC (message authentication code) is a sequence of bytes of a length specified by the MAC algorithm decided upon during the key exchange.

Protocol Version Exchange

The first packet that is sent after establishing a TCP connection is the protocol version exchange packet. This packet contains an identification string for each party. The client sends the first protocol version exchange packet, and the server follows. The form of this message should be as follows:

- SSH-protoversion-softwareversion SP comments CR LF
- Where SP = space, CR = carriage return, and LF = line feed

For example:

- SSH-2.0-OpenSSH_9.7p1 Debian-7

Key Exchange

Following the protocol version exchange, both parties send an SSH_MSG_KEXINIT (key exchange initialize) packet. The client sends the first key exchange packet, and the server follows. This packet contains the lists of algorithms that will be used to establish the SSH connection. Each list is ordered in decreasing preference order. The most preferred algorithm listed by the client that the server also lists is chosen. If no algorithms match between the server and the client, the communication is terminated. Each key exchange payload contains the following fields:

- SSH_MSG_KEXINIT
- Cookie
- Kex algorithms
- Server host key algorithms
- Encryption algorithms client to server (optional)
- Encryption algorithms server to client (optional)
- MAC algorithms client to server (optional)
- MAC algorithms server to client (optional)
- Compression algorithms client to server (optional)

- Compression algorithms server to client (optional)
- Languages client to server (optional)
- Languages server to client (optional)
- First kex packet follows
- 0

These fields are concatenated together, along with some additional information, to form the SSH packet. Each list should be prepended with a 4 byte length value in big endian order. For example:

- Kex init payload = (1-byte SSH_MSG_KEXINIT code || 16-byte random cookie || big endian 4-byte kex algorithm list length || kex algorithm list || …|| 1-byte first kex packet follows (boolean) || 4-byte 0 reserved for future extension)

SSH_MSG_KEXINIT is a single byte containing the number 20 that signals the type of message.

Cookie is a 16 byte long random value.

Kex algorithms (key exchange algorithms) is a comma separated list of algorithms in decreasing preference order. These algorithms determine how the shared key K will be derived. For example, diffie-hellman-group14-sha256, states that the Diffie-Hellman group 14 p and g value should be used, and sha256 will be the hashing algorithm used throughout the SSH connection.

Server host key algorithms is a comma separated list of algorithms in decreasing preference order. These algorithms specify how the server will authorize itself as part of the DH exchange. For example, ssh-ed25519.

Encryption algorithms client to server/server to client is a comma separated list of algorithms in decreasing preference order. These algorithms determine how the SSH messages will be encrypted each way. For example, aes128-ctr. This field is optional. If no encryption algorithm is to be used, “none” should be put in the list.

MAC algorithms client to server/server to client is a comma separated list of algorithms in decreasing preference order. These algorithms determine how the MAC will be computed for SSH messages each way. For example, hmac-sha1. This field is optional. If no MAC algorithm is to be used, “none” should be put in the list.

Compression algorithms client to server/server to client is a comma separated list of algorithms in decreasing preference order. These algorithms determine the compression algorithm that will be used for SSH messages each way. This field is optional. If no compression algorithm is to be used, “none” should be put in the list.

Languages client to server/server to client is a comma separated list of algorithms in decreasing preference order. If no languages are to be specified, send an empty list.

First kex packet follows is a boolean that signals whether a guessed key exchange packet follows or not. (The client is allowed to guess which algorithms the server wants to use.) 0 indicates no packet follows, 1 indicates a packet does follow.

0 is a 4 byte unsigned integer that is reserved for future extension.

In our experience using an OpenSSH server, MAC and encryption algorithms are mandated. It is allowed to have no MAC if the encryption algorithm is an authenticated encryption algorithm where the encryption combines message integrity and authentication of the sender.

Diffie-Hellman Key Exchange

Following the key exchange packets, client and server carry out a Diffie-Hellman (DH) exchange to derive a shared key K and authenticate the server. K will be used for (symmetric) encryption and generating MACs, so it is essential that the DH exchange is secure. Given the pieces of information communicated between the client and server, it is implausible for an eavesdropper to compute K because there exists no known way to efficiently solve the discrete logarithm problem.

RFC 4251 specifies types for the variables used in the DH exchange. These types are described below:

- mpint: **m**ultiple **p**recision **int**eger in two's complement format, stored as a string, 8 bits per byte, big-endian order. Negative numbers have the value 1 as the most significant bit of the first byte of the data partition. If the most significant bit would be set for a positive number, the number MUST be preceded by a 0 byte.
    - The language C does not have data types capable of storing integers of the size required for this exchange, so mpint's are the standard.
- string: an arbitrary length sequence of characters prepended with a 4 byte length value in big-endian order
- byte: a single byte (8 bits)

The following variables are used to describe the DH exchange. The types that are used to compute H are given in square brackets:

- p: A large safe prime. This number is determined during kex init.
- q: The order of the subgroup. In our experience, q is not used because p is a large safe prime.

- g: A generator for the subgroup
- [mpint] e: The client's public key
- [mpint] f: The server's public key
- x: The client's secret key
- y: The server's private key
- [mpint] K: the shared secret
- [string] V_C : The verification string of the client. This is the string sent during the protocol version exchange excluding the CR and LF characters at the end of the string.
- [string] V_S : The verification string of the server. This is the string sent during the protocol version exchange excluding the CR and LF characters at the end of the string.
- [string] I_C : The payload of the client's kex init message. The payload includes all information starting with the SSH_MSG_KEXINIT byte and ending with the reserved 0 prepended with a big endian 4-byte unsigned integer length of the string.
- [string] I_S : The payload of the server's kex init message. The payload includes all information starting with the SSH_MSG_KEXINIT byte and ending with the reserved 0.
- [string] K_S : The (public) host key of the server. This item is encoded strangely. It is a single string made up of 2 smaller strings. For example, the "ssh-ed25519" key format has the following encoding:
    - [string] "ssh-ed25519"
    - [string] key
    - Where, "key" is the 32-octet public key described in RFC 8032 section 5.1.5.
    - These 2 strings are concatenated together, and prepended with 4 bytes in big-endian order containing the length of the 2 strings combined (including the 4 bytes prepending the string data). For example:
        - K_S = (4-byte length || 4-byte length || "ssh=ed25519" || 4-byte length || key)
- H: a hash of V_C , V_S , I_C , I_S , K_S , e, f, and K concatenated together. The hashing algorithm is decided during kex init. H is often referred to as the exchange hash.
    - H = hash(V_C || V_S || I_C || I_S || K_S || e || f || K)
- [string] s: the server's signature of H using its private host key
- [byte] SSH_MSG_KEXDH_INIT: The message code 30
- [byte] SSH_MSG_KEXDH_REPLY: The message code 31

The DH exchange is carried out in 3 steps between the client and the server:

1. The client generates a random number x where $1 < x < q$ or $1 < x < p - 1$ if p is a safe prime. The client then computes, $e = g^x mod\ p$ and sends e to the server with the following payload (wrapped in binary packet protocol):
    - SSH_MSG_KEXDH_INIT || e
2. The server generates a random number y where $1 < y < q$ or $1 < y < p - 1$ if p is a safe prime. The server computes $f = g^y mod\ p$, $K = e^y mod\ p$, H, and a signature s

on H with its private host key. The server sends the following payload to the client (wrapped in binary packet protocol):
    - SSH_MSG_KEXDH_REPLY || K_S || f || s
3. First, the client computes K: $K = f^x mod\ p$. Then, the client verifies that the server is legitimate by computing H, using K_S to "undo" the server's signature on H, and then checks that the H it computed is equal to the H that results from using K_S to "undo" the signature. If both are equal, then the server possesses the private key to the server, so the client can be reasonably sure that the server is authentic since K_S is public.
    - This check does not protect the client from connecting to a malicious server. Since the malicious server would control its own private/public key, the client would just be verifying that the malicious server is legitimate, which doesn't protect it from harm.
    - The client can save the public key of the server, so that on subsequent connections it can check that the server has not been compromised. Note, our implementation does not have this feature yet.

After the completion of these three steps, the client has authenticated the server and both client and server have computed a shared secret K. This shared secret will be used to create encryption keys, integrity keys (to generate MACs), and initialization vectors (to kick off encryption).

The end of the key exchange is marked by server and client sending an SSH_MSG_NEWKEYS message, which indicates that both sides have all the information needed for secure communication (namely encryption and MACs). The SSH_MSG_NEWKEYS message is simply the number 21 encoded as a single byte wrapped in binary packet protocol.

Using K for encryption and MAC

Following the sending of the new keys packet by both parties, every message will be encrypted and be given a MAC (computed with the algorithm decided upon during the key exchange). The shared secret K and exchange hash H are used to generate keys for encryption and MACs as well as initialization vectors (IV). RFC 4253 section 7.2 specifies that keys and IVs are created as follows:

- *Initial* IV (client to server): hash(K || H || "A" || session ID) where "A" is the single character (ASCII 65), and the session ID is equal to the value of H directly after the first DH exchange.
    - Note, if keys are re-exchanged later in the SSH communication, then the value of H will change, but the session ID will not. Our simple implementation of a client did not implement a key re-exchange process.
- Encryption key (client to server): hash(K || H || "C" || session ID
    - This key is to be used as input for an encryption function

- Integrity key (client to server): hash(K || H || “E” || session ID)
    - This key is to be used as input for a MAC function

Note, each of the above 3 items also has a server to client variant. The only item that changes is the single character that is concatenated prior to the hash. The full list of keys is specified in RFC 4253 section 7.2.

Because keys are created using a hash function, the length of the key may not match the key length required by MAC or encryption algorithms. If the key is too long, take the first n bytes of the key required by the algorithm. If the key is too short, the following process can be used until a key of sufficient length is reached:

- K1 = HASH(K || H || X || session ID) (e.g., X = “A”)
- K2 = HASH(K || H || K1)
- K3 = HASH(K || H || K1 || K2)
- …
- key = K1 || K2 || K3 || …

MAC

After the integrity key has been computed, a MAC can be created. The form of the MAC is as follows (specified in RFC 4253 section 6.4):

- mac = MAC(integrity key, sequence number || unencrypted packet)
- The sequence number is an unsigned 4 byte integer that begins at 0, and is incremented after each packet sent following the protocol version exchange packet. The sequence number is never reset, even if keys/algorithms are renegotiated. It wraps around to 0 after every 2^{32} packets.

Because the MAC takes in the integrity key, which is created using the shared secret K, the MAC provides both message integrity and verifies the authenticity of the sender.

A MAC is verified in the following way:

1. Receive an encrypted message (with the MAC appended to the end)
2. Decrypt the message using the appropriate encryption key and IV
3. Compute the MAC using the appropriate integrity key, sequence number, and now decrypted message
4. Verify that the computed MAC matches the MAC appended to the end of the encrypted message

The MAC is always sent as the last n bytes of the packet where n is the length of the MAC:

- Packet = (encrypted packet || MAC)

In our implementation, we use hmac-sha1as our MAC algorithm which outputs twenty byte MAC tags.

Encryption

After the encryption key and IV have been computed, messages can be encrypted. The input into our encryption algorithm (aes128-ctr) is: an encryption key, an IV, and the plaintext (to encrypt). The function then outputs a ciphertext of the same length as the input plaintext.

Note, the *initial* IV is computed as detailed above, but the IV is unique for all calls to the encryption function. To ensure that the IV is different for each encryption, a counter is set equal to the initial IV. Then, for each block encrypted, the counter is incremented by 1. The counter will be used as the IV for subsequent encryption operations.

Since both encryption keys are known by client and server, SSH uses symmetric encryption algorithms. Thus, the same key used to encrypt plaintext into ciphertext is used to decrypt the ciphertext into plaintext.

Service Request

After finalizing the key exchange (deriving encryption and mac key), the client needs to be authenticated. It is the client's responsibility to initiate authentication by sending a SSH_MSG_SERVICE_REQUEST message with the payload "ssh-userauth" to indicate that the client would like to prove they have the credentials to access the server. The server then responds with a SSH_MSG_SERVICE_ACCEPT acknowledging the client's request.

Next, the client sends an SSH_MSG_USERAUTH_REQUEST message, containing their credentials as well as their intention to open channels post-authentication by specifying "ssh-connection" as the service. The server then responds with two messages, the first being SSH_MSG_USERAUTH_SUCCESS to indicate that the given user has been authenticated and they are now willing to open channels, and the second being a SSH_MSG_GLOBAL_REQUEST message, which is an OpenSSH extension for server to send hostkeys after userauth completes; see file PROTOCOL in OpenSSH's source code. We do not maintain a list of known hosts (i.e. a list of public keys of trusted remote hosts) for our client since our goal for this project does not encompass extended/repeated connections, so we ignore this message from the server.

Once authentication is complete, the client is free to open channels with the server. For our purposes, we will open an interactive session channel in order to send an "exec" command to the host's shell.[1] We do so by sending an SSH_MSG_CHANNEL_OPEN message which specifies the type of the channel as "session," channel number, window size, and max packet size. If the server is able to accommodate this channel, it sends a SSH_MSG_CHANNEL_OPEN_CONFIRMATION message signifying that the server is ready to receive channel requests.

---

[1] In further iterations of our project, we would love to explore the capabilities of these different channels—especially, an interactive shell. That is our goal over winter break! :)

The client is now ready to begin sending SSH_MSG_CHANNEL_REQUEST messages. For our example, we send an “exec” message with the payload command “whoami” to our server. If the message is well formed and the server will proceed with the request, the server sends a SSH_MSG_CHANNEL_SUCCESS message. In our case, the server responds with 1 packet that contains 2 messages. The first message is a SSH_MSG_CHANNEL_WINDOW_ADJUST message informing the client that the server is willing to receive more information than previously allowed, and the second message is the SSH_MSG_CHANNEL_SUCCESS. The server then sends a second packet containing a SSH_MSG_CHANNEL_DATA message, which contains the result of running the “whoami” command.

The channel request now completed, the channel is shut down according to the RFC 4254 section 5.3 specification. Namely the server will send the client a SSH_MSG_CHANNEL_EOF message, followed by a SSH_MSG_CHANNEL_CLOSE message.