SSH Background

SSH Protocol Description

1. Transport Layer Protocol establishes secure transport layer communications.
2. Authentication Protocol authenticates a client; it runs on top of the secure transport layer.
3. Connection Protocol is used in conjunction with the Authentication Protocol to establish secure sessions, e.g. interactive login, remote execution of commands, TCP/IP forwarding, and X11 forwarding.

To understand attacks on SSH, we need to first understand aspects of the Transport Layer and Authentication Protocols, so we will look at these in detail. We will not spend any time on the Connection Protocol.

Transport Layer Protocol

Reference: RFC 4253

The server host key is used to authenticate the server – the server's public key may be already known to the client and stored in a local database, or it may be signed by a Trusted Authority to establish its validity. More commonly, it is simply presented to the client, and the user must decide whether or not to accept the key; if it is accepted, it is stored in a local database (`.ssh/known_hosts` on Linux) for use with future connections to the server.

In addition to the server host key, there are two ephemeral Diffie-Hellman keys that are used to derive keys for transport layer encryption. The following table shows a simplified version of the Transport Layer Protocol: it is assumed that both client and server are using SSH 2.0 and that a Diffie-Hellman key exchange will be used. The ephemeral keys, and other cryptographic elements, are hilighted.

Client (send)	**Server (send)**
Initiate TCP connection on server port 22	
SSH-clientprotover-clientsoftwarever SP comments CR LF clientprotover = 2.0 clientsoftwarever e.g. BillsSSH comments are optional Ex: SSH-2.0-BillsSSH_3.6.3q3<CR><LF>	SSH-serverprotover-protosoftwarever SP comments CR LF serverprotover = 2.0 serversoftwarever e.g. BillsSSH

SSH_MSG_KEXINIT *Client provides the following values:* 16-byte random cookie list of key exchange algorithms list of accepted server_host_key_algorithms list of encryption_algorithms_c_to_s list of encryption_algorithms_s_to_c list of mac_algorithms_c_to_s list of mac_algorithms_s_to_c list of compression_algorithms_c_to_s list of compression_algorithms_s_to_c list of languages_c_to_s (*optional*) list of languages_s_to_c (*optional*) boolean first_kex_packet_follows	SSH_MSG_KEXINIT *Server provides the following values, possibly different from those provided by client:* 16-byte random cookie list of key exchange algorithms list of accepted server_host_key_algorithms list of encryption_algorithms_c_to_s list of encryption_algorithms_s_to_c list of mac_algorithms_c_to_s list of mac_algorithms_s_to_c list of compression_algorithms_c_to_s list of compression_algorithms_s_to_c list of languages_c_to_s (*optional*) list of languages_s_to_c (*optional*) boolean first_kex_packet_follows
Client and server determine a common set of algorithms: key exchange, encryption, etc. DH Key Exchange: p is a large, safe prime; g is a generator of a subgroup of GF(p); q is the order of g.	
Generate random x ($1 < x < q$). Compute $e = g^x \bmod p$ Send e to server	
	Receive e from client Generate random y ($1 < y < q$) Compute $f = g^y \bmod p$ Compute $K = e^y \bmod p$ Compute H = hash of (client version string || server version string || client SSH_MSG_KEXINIT || server SSH_MSG_KEXINIT || server public host key || e || f || K) Compute signature s on H using server private host key. Send (server host key || f || s) to client

Verify server public host key (e.g. using certificates or a local database) Compute $K = f^x \bmod p$ Compute hash H Verify signature s	
Client and server have shared secret K and exchange hash H. The hash H from the *first* key exchange is used as the session_id.	
Compute initial_iv_c_to_s = hash of (K || H || "A" || session_id) Compute initial_iv_s_to_c = hash of (K || H || "B" || session_id) Compute encryption_key_c_to_s = hash of (K || H || "C" || session_id) Compute encryption_key_s_to_c = hash of (K || H || "D" || session_id) Compute integrity_key_c_to_s = hash of (K || H || "E" || session_id) Compute integrity_key_s_to_c = hash of (K || H || "F" || session_id)	
Key material must be taken from the beginning of the hash output. If the selected algorithm requires more key material than a single hash value can provide, additional key material can be generated as follows: *K1* = hash of (K || H || X || session_id) where X is "A", "B", "C", "D", "E", or "F". *K2* = hash of (K || H || *K1*) *K3* = hash of (K || H || *K1* || *K2*) etc. and *key material*= *K1* || *K2* || *K3* || ...	
SSH_MSG_NEWKEYS Ends key exchange and takes new keys into use.	SSH_MSG_NEWKEYS Ends key exchange and takes new keys into use.

Authentication Protocol

Reference: RFC 4252

Although the RFC supports at least three authentication methods, we will only discuss the two most common: “password” and “public key.” Password-based authentications is straight-forward. The client sends an authentication request with the authentication method set to “password.” The server may accept or reject the request, but if it is accepted, the client follows with a message that includes the plain-text password. Note that the entire exchange is protected by transport layer encryption.

Public key authentication is only slightly more complicated. The client must have associated public and private keys, and the public key must be known to the server. Again, the process begins with the client sending an authentication message, but with authentication method of “publickey”, and including a preferred public key algorithm and public key. The server must respond with either a user authorization failure or a message indicating acceptance of the public key algorithm and key. At this point, the client generates a message containing the public key algorithm identifier, public key, and several other values; it then computes a signature over all these values, appends it to the message, and sends the signed message to the server. The server must verify that the public key identifies the client *and* that the signature is valid (and so the client must be in possession of the private key).

Implementations of ssh provide multiple ways to associate public keys with users. In a Linux environment, the simplest method is for each user to place a public key on the server in `~/.ssh/authorized_keys`. The server must be configured to allow this option. A more secure method would be for the administrator to generate keys for each user (or receive a public key from each user) and place all the public keys in a protected location in the file system. Again, it is a matter of how the ssh software is configured. In either case, the user can login to the server from any machine on which their public and private keys are stored.

References:

RFC 4253, http://tools.ietf.org/html/rfc4253

RFC 4252, http://tools.ietf.org/html/rfc4252