



Secure Remote Access: SSH



What is SSH?

- **SSH – Secure Shell**
- **SSH is a protocol for secure remote login and other secure network services over an insecure network**
- **developed by SSH Communications Security Corp., Finland**
- **two distributions are available:**
 - commercial version
 - freeware (www.openssh.com)
- **specified in a set of Internet drafts**

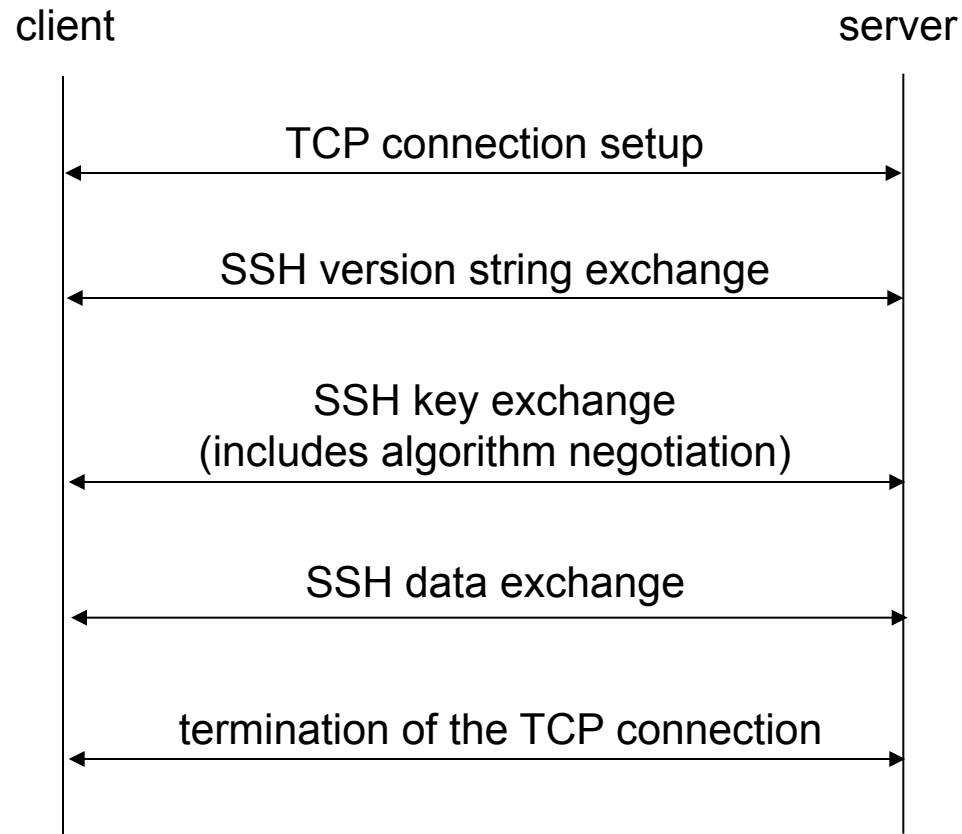
Major SSH components

- **SSH Transport Layer Protocol**
 - provides server authentication, confidentiality, and integrity services
 - it may provide compression too
 - runs on top of any reliable transport layer (e.g., TCP)
- **SSH User Authentication Protocol**
 - provides client-side user authentication
 - runs on top of the SSH Transport Layer Protocol
- **SSH Connection Protocol**
 - multiplexes the secure tunnel provided by the SSH Transport Layer and User Authentication Protocols into several logical channels
 - these logical channels can be used for a wide range of purposes
 - secure interactive shell sessions
 - TCP port forwarding
 - carrying X11 connections

SSH security features

- **strong algorithms**
 - uses well established strong algorithms for encryption, integrity, key exchange, and public key management
- **large key size**
 - requires encryption to be used with at least 128 bit keys
 - supports larger keys too
- **algorithm negotiation**
 - encryption, integrity, key exchange, and public key algorithms are negotiated
 - it is easy to switch to some other algorithm without modifying the base protocol

SSH TLP – Overview

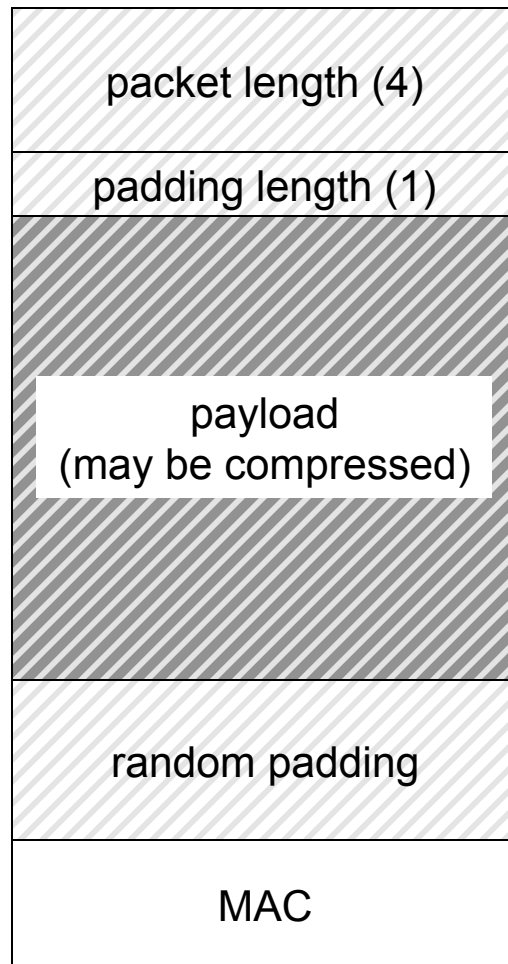


Connection setup and version string exchange

- **TCP connection setup**
 - the server listens on port 22
 - the client initiates the connection

- **SSH version string exchange**
 - both side must send a version string of the following form:
 “SSH-protoversion-softwareversion comments” \CR \LF
 - used to indicate the capabilities of an implementation
 - triggers compatibility extensions
 - current protocol version is 2.0
 - all packets that follow the version string exchange is sent using the Binary Packet Protocol

Binary Packet Protocol



- **packet length**:
 - length of the packet not including the MAC and the packet length field
- **padding length**:
 - length of padding
- **payload**:
 - useful contents
 - might be compressed
 - max payload size is 32768
- **random padding**:
 - 4 – 255 bytes
 - total length of packet not including the MAC must be multiple of $\max(8, \text{cipher block size})$
 - even if a stream cipher is used
- **MAC**:
 - message authentication code
 - computed over the clear packet and an implicit sequence number



encryption



compression

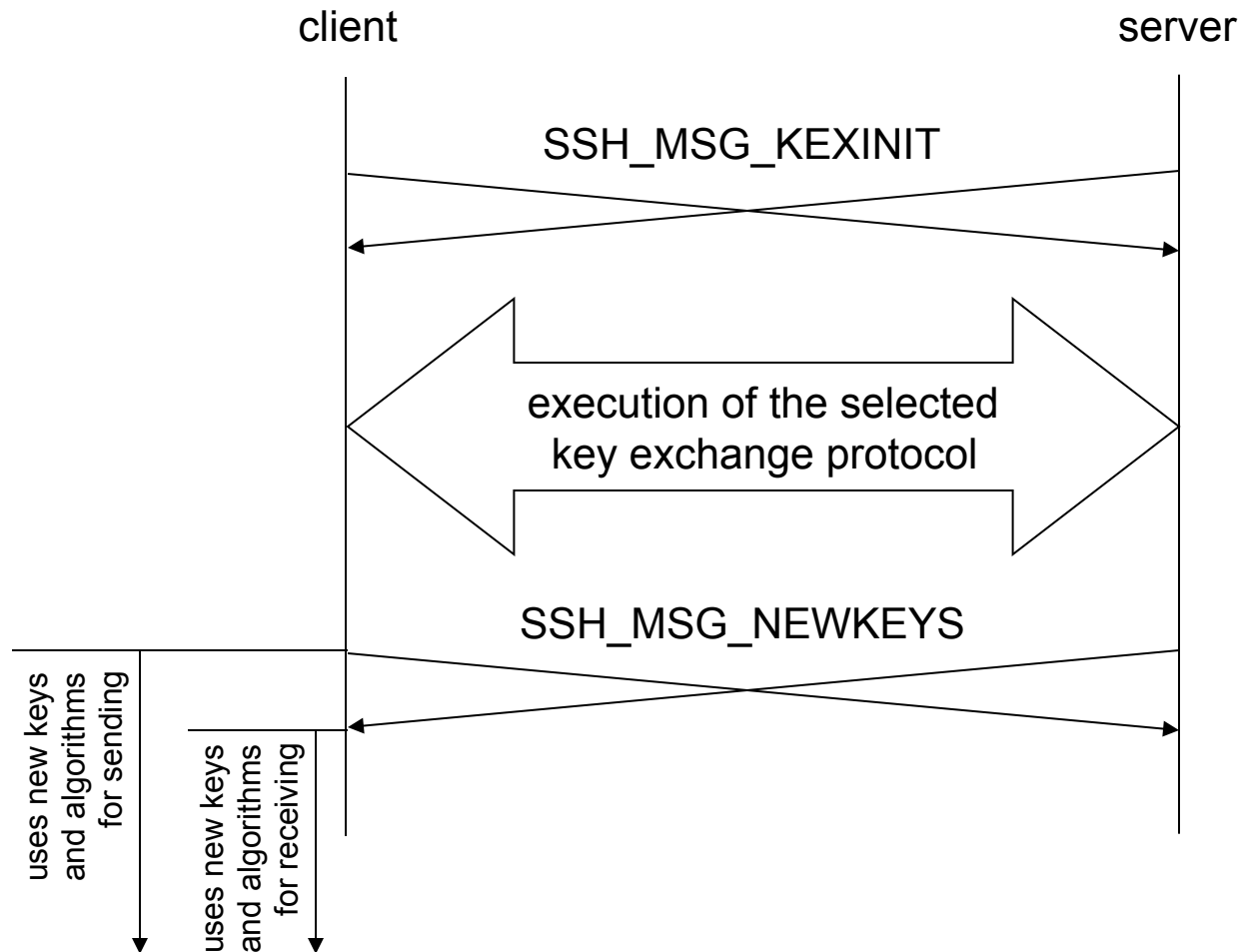
Encryption

- the encryption algorithm is negotiated during the key exchange
- supported algorithms
 - 3des-cbc (required) (168 bit key)
 - blowfish-cbc (recommended)
 - twofish256-cbc (opt) / twofish192-cbc (opt) / twofish128-cbc (recomm)
 - aes256-cbc (opt) / aes192-cbc (opt) / aes128-cbc (recomm)
 - serpent256-cbc (opt) / serpent192-cbc (opt) / serpent128-cbc (opt)
 - arcfour (opt) (RC4)
 - idea-cbc (opt) / cast128-cbc (opt)
- key and IV are also established during the key exchange
- all packets sent in one direction is considered a single data stream
 - IV is passed from the end of one packet to the beginning of the next one
- encryption algorithm can be different in each direction

MAC

- **MAC algorithm and key are negotiated during the key exchange**
- **supported algorithms**
 - hmac-sha1 (required) [MAC length = key length = 160 bits]
 - hmac-sha1-96 (recomm) [MAC length = 96, key length = 160 bits]
 - hmac-md5 (opt) [MAC length = key length = 128 bits]
 - hmac-md5-96 (opt) [MAC length = 96, key length = 128 bits]
- **MAC algorithms used in each direction can be different**
- **MAC = mac(key, seq. number | clear packet)**
 - sequence number is implicit, not sent with the packet
 - sequence number is represented on 4 bytes
 - sequence number initialized to 0 and incremented after each packet
 - it is never reset (even if keys and algs are renegotiated later)

Key exchange - Overview



Algorithm negotiation

- **SSH_MSG_KEXINIT**
 - kex_algorithms (comma separated list of names)
 - server_host_key_algorithms
 - encryption_algorithms_client_to_server
 - encryption_algorithms_server_to_client
 - mac_algorithms_client_to_server
 - mac_algorithms_server_to_client
 - compression_algorithms_client_to_server
 - compression_algorithms_server_to_client
 - first_kex_packet_follows (boolean)
 - random cookie (16 bytes)
- **algorithm lists**
 - the server list the algorithms it supports
 - the client lists the algorithms that it is willing to accept
 - algorithms are listed in order of preference
 - selection: first algorithm on the client's list that is also on the server's list

Deriving keys and IVs

- any key exchange algorithm produces two values
 - a shared secret K
 - an exchange hash H
- H from the first key exchange is used as the session ID
- keys and IVs are derived from K and H as follows:
 - IV client to server = $\text{HASH}(K | H | \text{"A"} | \text{session ID})$
 - IV server to client = $\text{HASH}(K | H | \text{"B"} | \text{session ID})$
 - encryption key client to server = $\text{HASH}(K | H | \text{"C"} | \text{session ID})$
 - encryption key server to client = $\text{HASH}(K | H | \text{"D"} | \text{session ID})$
 - MAC key client to server = $\text{HASH}(K | H | \text{"E"} | \text{session ID})$
 - MAC key server to client = $\text{HASH}(K | H | \text{"F"} | \text{session ID})$
- where HASH is the hash function specified by the key exchange method (e.g., diffie-hellman-group1-sha1)
- if the key length is longer than the output of HASH...
 - $K1 = \text{HASH}(K | H | X | \text{session ID})$
 - $K2 = \text{HASH}(K | H | K1)$
 - $K3 = \text{HASH}(K | H | K1 | K2)$
 - ...
 - $\text{key} = K1 | K2 | K3 | \dots$

Diffie-Hellman key exchange

1.

- the client generates a random number x and computes $e = g^x \bmod p$
- the client sends e to the server

2.

- the server generates a random number y and computes $f = g^y \bmod p$
- the server receives e from the client
- it computes $K = e^y \bmod p = g^{xy} \bmod p$ and $H = \text{HASH}(\text{client version string} \mid \text{server version string} \mid \text{client kex init msg} \mid \text{server kex init msg} \mid \text{server host key } K_{\text{srv}} \mid e \mid f \mid K)$
- it generates a signature s on H using the private part of the server host key (may involve additional hash computation on H)
- it sends $(K_{\text{srv}} \mid f \mid s)$ to the client

3.

- the client verifies that K_{srv} is really the host key of the server
- the client computes $K = f^x \bmod p = g^{xy} \bmod p$ and the exchange hash H
- the client verifies the signature s on H

Server authentication

- based on the server's host key K_{srv}
- the client must check that K_{srv} is really the host key of the server
- models
 - the client has a local database that associates each host name with the corresponding public host key
 - the host name – to – key association is certified by a trusted CA and the server provides the necessary certificates or the client obtains them from elsewhere
 - check fingerprint of the key over an external channel (e.g., phone)
 - best effort:
 - accept host key without check when connecting the first time to the server
 - save the host key in the local database, and
 - check against the saved key on all future connections to the same server

Key re-exchange

- **either party may initiate a key re-exchange**
 - sending an `SSH_MSG_KEXINIT` packet when not already doing a key exchange
- **key re-exchange is processed identically to the initial key exchange**
 - except for the session ID, which will remain unchanged
- **algorithms may be changed**
- **keys and IVs are recomputed**
- **encryption contexts are reset**
- **it is recommended to change keys after each gigabyte of transmitted data or after each hour of connection time**

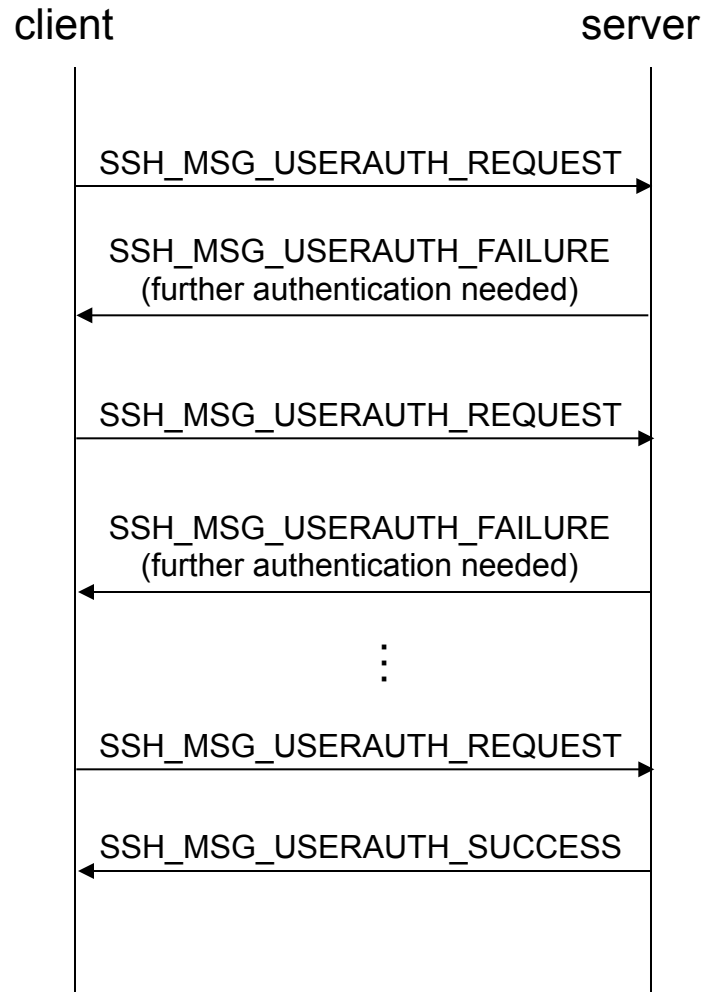
Service request

- after key exchange the client requests a service
- services
 - ssh-userauth
 - ssh-connection
- when the service starts, it has access to the session ID established during the first key exchange

SSH – User Authentication Protocol

- the protocol assumes that the underlying transport protocol provides integrity and confidentiality (e.g., SSH Transport Layer Protocol)
- the protocol has access to the session ID
- the server should have a timeout for authentication and disconnect if the authentication has not been accepted within the timeout period
 - recommended value is 10 minutes
- the server should limit the number of failed authentication attempts a client may perform in a single session
 - recommended value is 20 attempts
- three authentication methods are supported
 - publickey
 - password
 - hostbased

User authentication overview



■ USERAUTH_REQUEST

- user name
- service name
- method name
- ... method specific fields ...

■ USERAUTH_FAILURE

- list of authentication methods that can continue
- partial success flag
 - TRUE: previous request was successful, but further authentication is needed
 - FALSE: previous request was not successful

■ USERAUTH_SUCCESS

(authentication is complete, the server starts the requested service)

The “publickey” method

- all implementations must support this method
- however, most local policies will not require authentication with this method in the near future, as users don't have public keys
- authentication is based on demonstration of the knowledge of the private key (the client signs with the private key)
- the server verifies that
 - the public key really belongs to the user specified in the authentication request
 - the signature is correct

The “publickey” method cont’ d

- **SSH_MSG_USERAUTH_REQUEST**
 - user name
 - service name
 - “publickey”
 - TRUE (a flag set to TRUE)
 - public key algorithm name (e.g., ssh-dss)
 - public key
 - signature (computed over the session ID and the data in the request)
- the server responds with **SSH_MSG_USERAUTH_FAILURE** if the request failed or more authentication is needed, or **SSH_MSG_USERAUTH_SUCCESS** otherwise

The “publickey” method cont’ d

- **using the private key**
 - involves expensive computations
 - may require the user to type a password if the private key is stored in encrypted form on the client machine
- **in order to avoid unnecessary processing, the client may check whether authentication using the public key would be acceptable**
 - **SSH_MSG_USERAUTH_REQUEST**
 - user name
 - service name
 - “publickey”
 - FALSE
 - public key algorithm name
 - public key
 - if OK then the server responds with **SSH_MSG_USERAUTH_PK_OK**

The “password” method

- all implementations should support this method
- this method is likely the most widely used
- **SSH_MSG_USERAUTH_REQUEST**
 - user name
 - service name
 - “password”
 - FALSE (a flag set to FALSE)
 - password (plaintext)
- the server may respond with **SSH_MSG_USERAUTH_FAILURE**, **SSH_MSG_USERAUTH_SUCCESS**, or **SSH_MSG_USERAUTH_PASSWD_CHANGEREQ**

The “password” method cont’ d

- changing the password
 - SSH_MSG_USERAUTH_REQUEST
 - user name
 - service name
 - “password”
 - TRUE
 - old password (plaintext)
 - new password (plaintext)

The “hostbased” method

- authentication is based on the host where the user is coming from
- this method is optional
- the client sends a signature that has been generated with the private host key of the client
- the server verifies that
 - the public key really belongs to the host specified in the authentication request
 - the signature is correct

The “hostbased” method cont’ d

- **SSH_MSG_USERAUTH_REQUEST**
 - user name
 - service name
 - “hostbased”
 - public key algorithm name
 - public key and certificates for client host
 - client host name
 - user name on client host
 - signature (computed over the session ID and the data in the request)

SSH – Connection Protocol

- **provides**
 - interactive login sessions
 - remote execution of commands
 - forwarded TCP/IP connections
 - forwarded X11 connections
- **all these applications are implemented as “channels”**
- **all channels are multiplexed into the single encrypted tunnel provided by the SSH Transport Layer Protocol**
- **channels are identified by channel numbers at both ends of the connection**
- **channel numbers for the same channel at the client and server sides may differ**

Channel mechanisms

- opening a channel
 - **SSH_MSG_CHANNEL_OPEN**
 - channel type
 - sender channel number
 - initial window size
 - maximum packet size
 - ... channel type specific data ...
 - **SSH_MSG_CHANNEL_OPEN_CONFIRMATION**
 - recipient channel number (sender channel number from the open request)
 - sender channel number
 - initial window size
 - maximum packet size
 - ... channel type specific data ...
 - **SSH_MSG_CHANNEL_OPEN_FAILURE**
 - recipient channel number (sender channel number from the open request)
 - reason code and additional textual information

Channel mechanisms cont' d

- data transfer over a channel
 - SSH_MSG_CHANNEL_DATA
 - recipient channel number
 - data
 - SSH_MSG_CHANNEL_WINDOW_ADJUST
 - recipient channel number
 - bytes to add to the window size
- closing a channel
 - SSH_MSG_CHANNEL_EOF
 - recipient channel number(sent if the party doesn't want to send more data)
 - SSH_MSG_CHANNEL_CLOSE
 - recipient channel(receiving party must respond with an SSH_MSG_CHANNEL_CLOSE, the channel is closed if the party has sent and received the closing msg)

Channel mechanisms cont' d

- **channel type specific requests**
 - **SSH_MSG_CHANNEL_REQUEST**
 - recipient channel number
 - request type
 - want reply flag (TRUE if reply is needed)
 - ... request type specific data ...
 - **SSH_MSG_CHANNEL_SUCCESS**
 - recipient channel
 - **SSH_MSG_CHANNEL_FAILURE**
 - recipient channel

Example: Starting a remote shell

C → S: SSH_MSG_CHANNEL_OPEN

- channel type = “session”
- sender channel number = 5
- initial window size
- maximum packet size

C ← S: SSH_MSG_CHANNEL_OPEN_CONFIRMATION

- recipient channel number = 5
- sender channel number = 21
- initial window size
- maximum packet size

Example: Starting a remote shell cont' d

C → S: SSH_MSG_CHANNEL_REQUEST

- recipient channel number = 21
- request type = “pty-req” (pseudo terminal request)
- want reply flag = TRUE
- TERM environment variable value (e.g., vt100)
- terminal width in characters (e.g., 80)
- terminal height in rows (e.g., 24)
- ...

C ← S: SSH_MSG_CHANNEL_SUCCESS

- recipient channel number = 5

Example: Starting a remote shell cont' d

C → S: SSH_MSG_CHANNEL_REQUEST

- recipient channel number = 21
- request type = “shell”
- want reply flag = TRUE

C ← S: SSH_MSG_CHANNEL_SUCCESS

- recipient channel number = 5

C ↔ S: SSH_MSG_CHANNEL_DATA,
SSH_MSG_CHANNEL_WINDOW_ADJUST

...