The SSH (Secure Shell) protocol

Prof. Antonio Lioy

< antonio.lioy@polito.it >

Politecnico di Torino

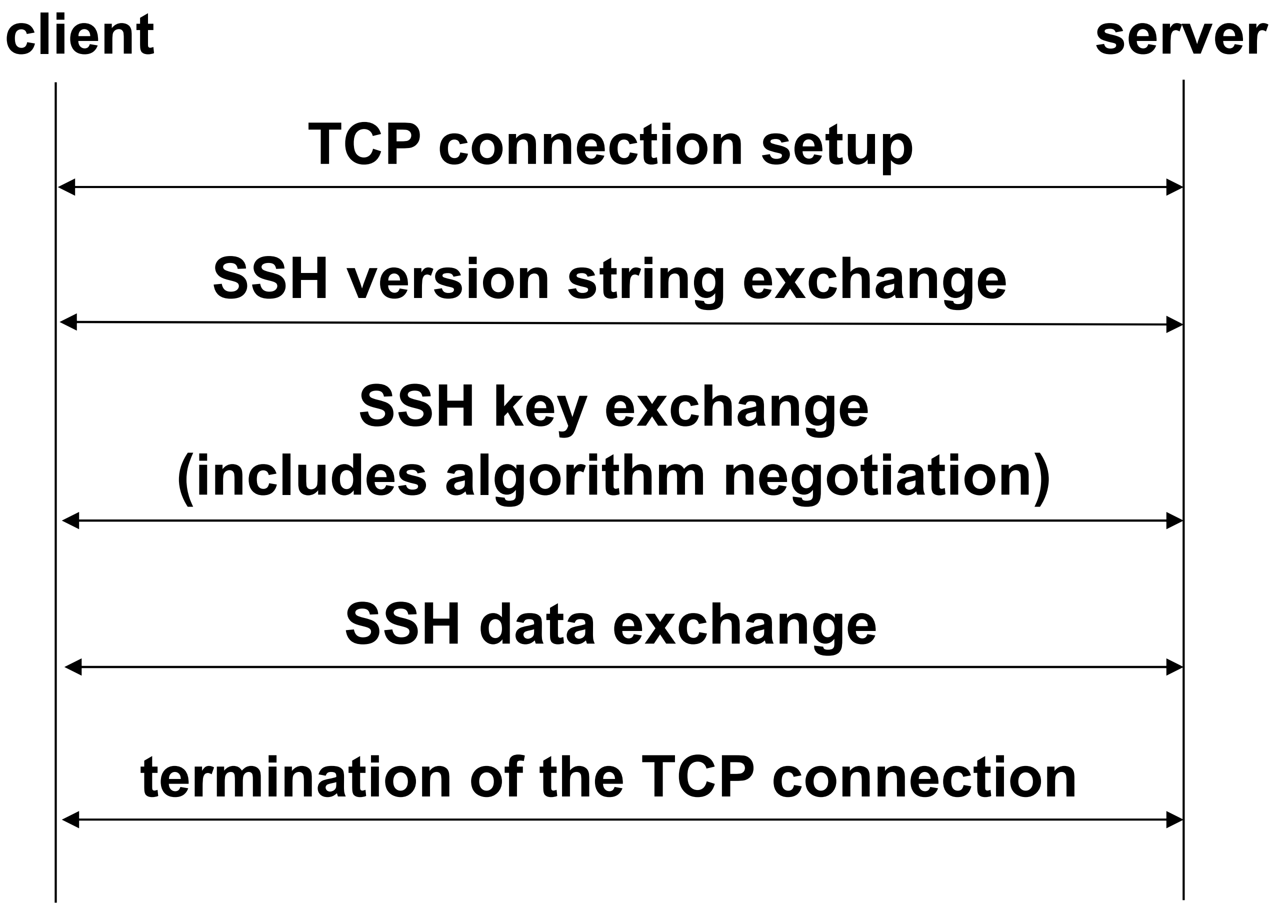
SSH: a bit of history

- **(July 1995) SSH-1 by Tatu Ylönen (HUT)**
  - response to a hacking incident (sniffer on backbone, thousands of user+pwd copied)
- **(December 1995) foundation of SSH Communications Security (commercial)**
- **(1999) OpenSSH fork appears in OpenBSD 2.6**
- **(2006) SSH-2 by IETF**
  - incompatible with SSH-1
  - improvements in security and features
  - frequently is the only version supported nowadays
- **here we will always refer to SSH-2 (unless otherwise noted)**

SSH architecture

- **three-layer architecture**
- **Transport Layer Protocol provides**
  - initial connection
  - server authentication,
  - confidentiality and integrity with perfect forward secrecy
  - key re-exchange (RFC-4253 recommends after 1GB of data transmitted or after 1 hour of transmission)
- **User Authentication Protocol**
  - authenticates the client to the server
- **Connection Protocol**
  - supports multiple connections (channels) over a single secure channel (implemented with the transport layer protocol)

SSH Transport Layer Protocol

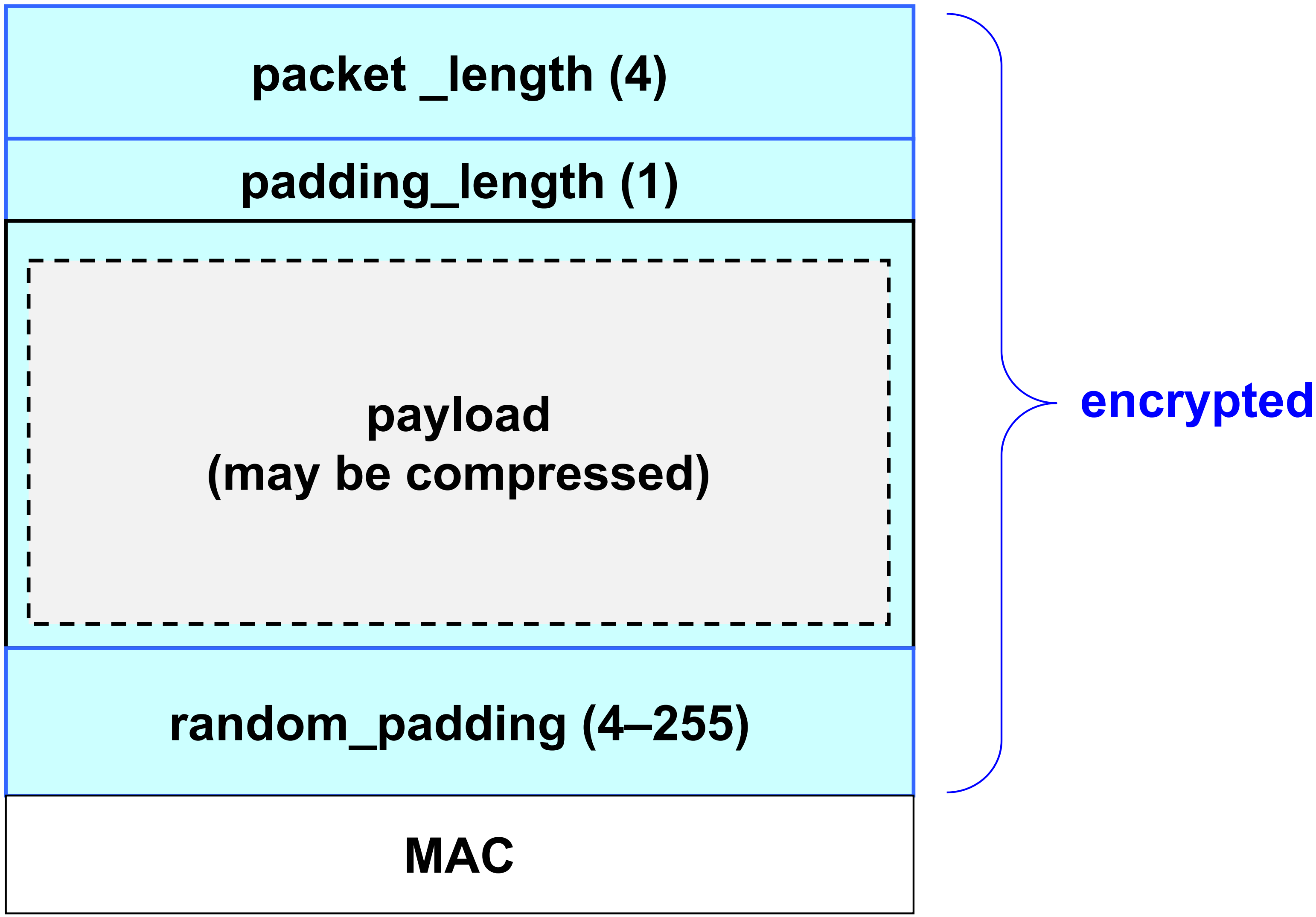


SSH TLP: connection and version exchange

- **TCP connection setup**
  - the server listens on port 22 by default
  - the client initiates the connection
- **SSH version string exchange**
  - both sides must send a version string of the following form:

    SSH-protoversion-softwareversion SP comments CR LF
  - used to indicate the capabilities of an implementation
  - triggers compatibility extensions
- **all packets that follow the version string exchange is sent using the Binary Packet Protocol**

SSH: Binary Packet Protocol



SSH: Binary Packet Protocol

- **packet length:**
  - not including the MAC and the packet length field itself
- **payload (note – might be compressed):**
  - size = packet_length – padding_length – 1
  - max (uncompressed) size is 32768 byte
- **random padding:**
  - 4 – 255 bytes
  - total packet length (MAC excluded) must be multiple of max(8, cipher_block_size) … even if a stream cipher is used (!)
- **MAC (as part of the authenticate-and-encrypt schema)**
  - over the cleartext packet and an implicit sequence number
  - requires decryption before checking integrity (possible DOS!)

SSH TLP: key exchange

- **algorithm negotiation:**
  - SSH_MSG_KEXINIT
  - cookie (16 random bytes)
  - kex_algorithms
  - server_host_key_algorithms
  - encryption_algorithms_client_to_server, …_server_to_client
  - mac_algorithms_client_to_server, …_server_to_client
  - compression_algorithms_client_to_server, …_server_to_client
  - languages_client_to_server, …_server_to_client
  - first_kex_packet_follows (flag)
    - attempt to guess agreed kex_algorithm

SSH TLP: algorithm specification

- **kex_algorithms (note: contains HASH)**
  - e.g. diffie-hellman-group1-sha1, ecdh-sha2-OID_of_curve
- **server_host_key_algorithms**
  - e.g. ssh-rsa
- **encryption_algorithms_X_to_Y**
  - e.g. aes-128-cbc, aes-256-ctr, aead_aes_128_gcm
- **mac_algorithms_X_to_Y**
  - e.g. hmac-sha1, hmac-sha2-256, aead_aes_128_gcm
- **compression_algorithms_X_to_Y**
  - e.g. none, zlib
- **complete list maintained by IANA: https://www.iana.org/assignments/ssh-parameters/ssh-parameters.xhtml**

SSH: DH key agreement (and server authN)

- [C] generates a random number x, computes $e = g^x \bmod p$
- [C > S] e
- [S] generates a random number y, computes $f = g^y \bmod p$
- [S] computes $K = e^y \bmod p = g^{xy} \bmod p$ and the exchange hash H = HASH(c_version_string | s_version_string | c_kex_init_msg | s_kex_init_msg | s_host_PK | e | f | K)
- [S] generates a signature sigH on H using the private key s_host_SK (may involve additional hash computation on H)
- [S > C] s_host_PK | f | sigH
- [C] verifies that s_host_PK is really the server's public key
- [C] computes $K = f^x \bmod p = g^{xy} \bmod p$ and H = HASH(…)
- [C] verifies the signature sigH on H
- [C, S] H becomes the session-id

SSH: key derivation

- **initial IV:**
  - client to server = HASH(K || H || "A" || session_id)
  - server to client = HASH(K || H || "B" || session_id)
- **encryption key:**
  - client to server = HASH(K || H || "C" || session_id)
  - server to client = HASH(K || H || "D" || session_id)
- **integrity key:**
  - client to server = HASH(K || H || "E" || session_id)
  - server to client = HASH(K || H || "F" || session_id)
- **note: session_id is the H value computed for the first key exchange, and remains such even when key re-exchange is performed**

SSH: encryption

- **encryption algorithm negotiated during the key exchange**
- **encryption algorithm can be different (!) in each direction**
- **supported algorithms (basic set):**
  - (required) 3des-cbc (w/ three keys, i.e. 168 bit key)
  - (recommended) aes128-cbc
  - (optional) blowfish-cbc, twofish256-cbc, twofish192-cbc, twofish128-cbc, aes256-cbc, aes192-cbc, serpent256-cbc, serpent192-cbc, serpent128-cbc, arcfour, idea-cbc, cast128-cbc, none
- **key and IV established during the key exchange**
- **all packets sent in one direction is a single data stream**
- **IV is passed from the end of one packet to the beginning of the next one**

SSH: encryption – handling of IV

- **IV for the first packet randomly generated during KEX**
- **if CBC is used, then:**
  - IV for next packet is the last encrypted block of the previous packet
- **if CTR is used, then:**
  - IV for next packet is the IV of the pervious packet incremeted by one

SSH: MAC

- **MAC algorithm and key negotiated during the key exchange**
- **MAC algorithms used in each direction can be different**
- **supported algorithms (basic set):**
  - hmac-sha1 (required) [key length = 160-bit]
  - hmac-sha1-96 (recomm) [key length = 160-bit]
  - hmac-md5 (opt) [key length = 128-bit]
  - hmac-md5-96 (opt) [key length = 128-bit]
- **MAC = mac(key, seq_number | cleartext_packet)**
  - sequence number is implicit, not sent with the packet
  - sequence number is represented on 4 bytes
  - seq_number initially 0 and incremented after each packet
  - seq_number never reset (even if keys/algos are renegotiated)

SSH: peer authentication – server

- **server authentication**
  - asymmetric challenge-response (explicit server signature of the key exchange hash H)
  - client locally stores the public keys of the servers (danger!)
    - typically in ~/.ssh/known_hosts
    - if key absent, then it's offered at first connection
      - TOFU (Trust On First Use) danger!
  - good practice:
    - protect known_hosts for authentication and integrity
    - periodic audit/review all the known_hosts files to quickly detect added/deleted hosts or changed keys

SSH: peer authentication – client

- **client authentication (part of the UAP)**
  - username and password
    - exchanged only after the protected channel is created
    - protects from sniffing but still open to other attacks (e.g. on-line password enumeration)
  - asymmetric challenge-response
    - server locally stores the public keys of the users allowed to connect as a local user
      - typically in ~local_user/.ssh/authorized_keys
    - good practice:
      - protect authorized_keys for authentication and integrity
      - periodic audit/review all the authorized_keys files to quickly detect added/deleted users or changed keys

SSH port forwarding / tunnelling

- **a way to forward TCP traffic through SSH**
  - e.g. securing POP3, SMTP and HTTP connections
    - insecure connections
  - the client-server applications will run their normal authentication over the encrypted tunnel
- **there are two types of port forwarding:**
  - local forwarding (outgoing tunnel)
  - remote forwarding (incoming tunnel)
  - both use the Connection Protocol to encapsulate a TCP channel inside a SSH one

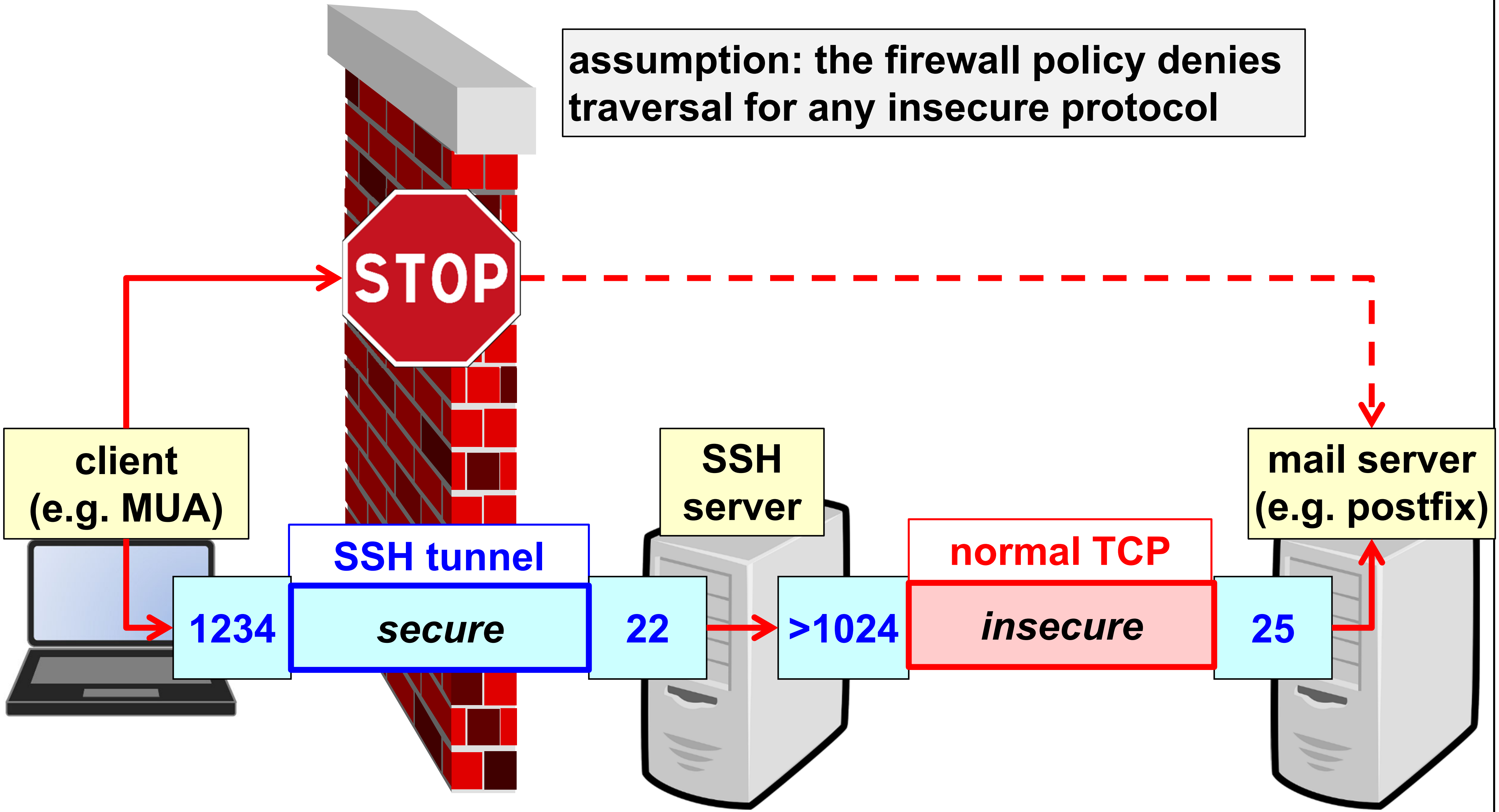
SSH local port forwarding

- **forwards traffic from a local port to a remote port**
- **example (see schema in next slide)**
  - suppose to be behind a firewall that blocks access to an external mail server because only secure traffic is permitted
  - by entering the command

    ```
    ssh -L 1234:mail_server:25 user@ssh_server
    ```

    the traffic to port 1234 on the (internal) client will be forwarded to port 25 on the mail_server by using a tunnel to the (external) ssh_server
  - now configure the MUA to connect to localhost:1234 as outgoing mail server
  - requires a valid user at the SSH server

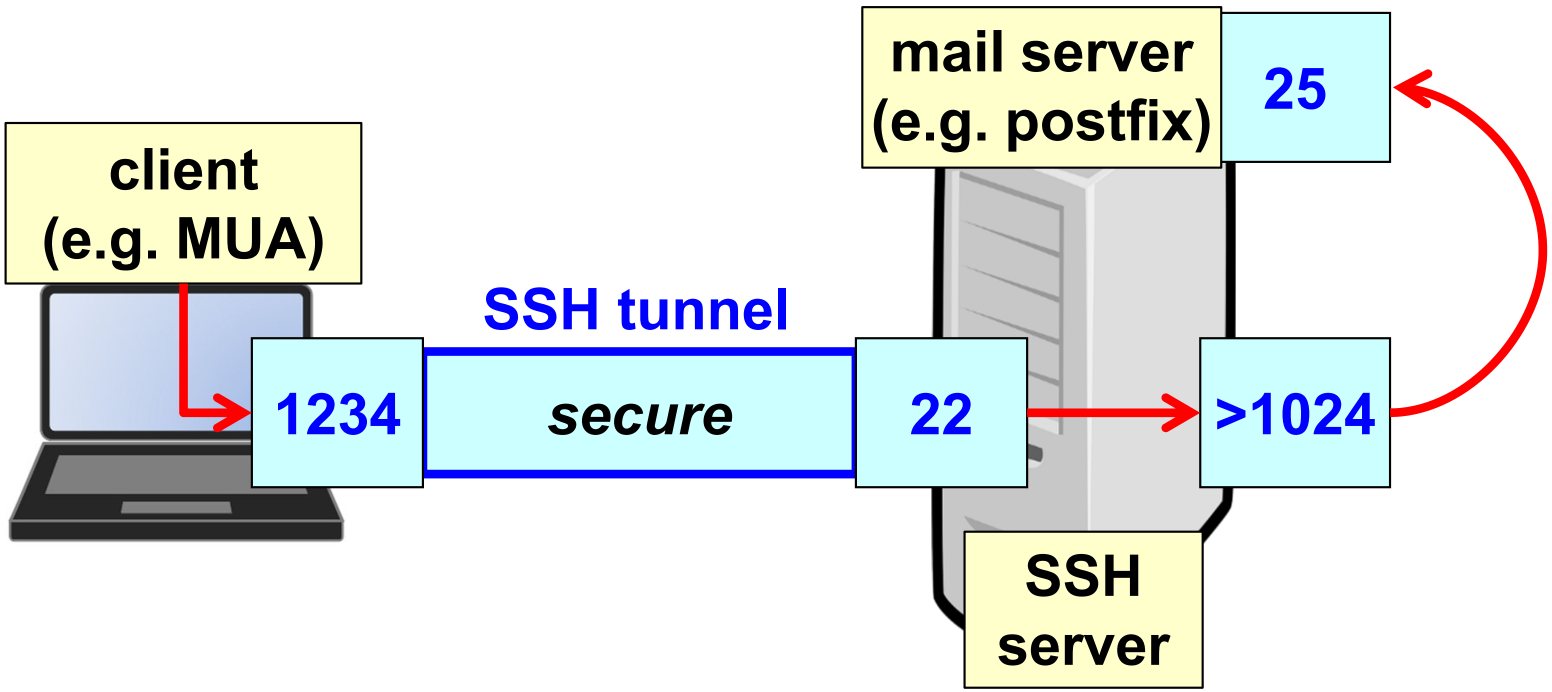
SSH local port forwarding (schema)



SSH local port forwarding (II)

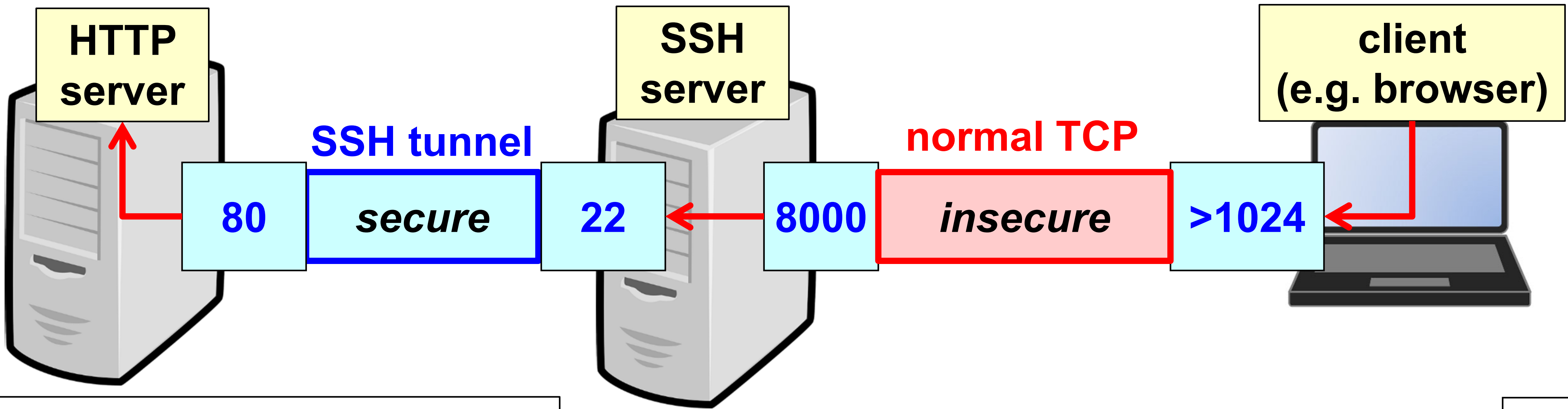
- **to avoid any insecure path, you should have the SSH server hosted on the same node as the application server (or have both hosted inside the same trusted subnet)**

ssh -L 1234:localhost:25 user@ssh_server



SSH remote port forwarding

- **forwards traffic from a remote port at the SSH server to a local port of the SSH client**
- **(example) you want to let an external user connect to your local HTTP server (which is behind a NAT); with the command**
  ssh -R 8000:127.0.0.1:80 user@ssh_server
  the traffic to port 8000 of the ssh_server will be forwarded to port 80 of the SSH client using a tunnel



SSH: causes of insecurity

- **direct trust in public keys (X.509 certs not used, but some commercial versions do and openSSH has SSH certificates)**
- **users ignore warnings and blindly accept new server public key … which leads to MITM attacks ☹**
- **weak server platform security**
  - worms, malicious code, rootkits, …
- **weak client platform security**
  - malware, keylogger, …
- **any connection to a local forwarded port will be tunnelled … even if coming from another node (!)**
  - to avoid this behaviour, specify also the local bind address (e.g. ssh -L **127.0.0.1:**1234:mail_server:25 user@ssh_server) so that only local processes can use the tunnel

Some attacks against SSH / BothanSpy

- **probably a CIA tool (information by Wikileaks on July 6, 2017, as part of their Vault 7 release)**
- **targets Xshell (a Windows SSH client used in USA and KR)**
- **injects a malicious DLL into a Xshell process to steal:**
  - user and password for password authenticated connections
  - user name, private key file name, and passphrase for public-key authenticated connections
- **designed for use with the ShellTerm attack framework:**
  - covert channel with C&C server and DLL injection capabilities
  - direct communication with C&C, no data written to disk (hence difficult to detect for anti-malware)
- **can also be used off-line (write collected data to disk, encrypted with AES, for later collection)**

Some attacks against SSH / Gyrfalcon

- **probably a CIA tool (information by Wikileaks on July 6, 2017, as part of their Vault 7 release)**
- **targets openSSH in enterprise Linux (RedHat, Centos, …)**
- **pre-loads a malicious DLL to intercept plaintext traffic (i.e. before encryption and after decryption)**
  - user+pwd but also actual data
- **encrypted configuration file**
- **encrypted file for captured data**
- **requires root access but:**
  - no integration with C&C
  - freely read/write files on disk (probably relies on rare adoption of anti-malware in Linux)
- **surprisingly not stealthy and unsophisticated**

Brute force attack against SSH

- **false sense of security when using a secure channel**
  - … with insecure authentication (i.e. reusable passwords)
- **typical brute-force attack**
  - trying all passwords from a dictionary of well-known ones
- **example of an attack:**
  - Sep-2016, servers activated to test SSH brute-force attacks
  - one IPv4 and one IPv6, in the cloud
  - IPv4 server immediately attacked and conquered in 12' (!!!)
    - as the password for root was "password" ☺
    - in a few minutes was used for a DDoS
  - after 1 week the IPv6 server was not even attacked (!)

SSH: protection from brute-force attacks

- for Linux
- account lockout after N authentication failures
  - pam_tally2
  - pam_faillock
- use tcpwrappers to permit/deny access to specific hosts or networks
- implement SSH rate control with IPtables (e.g. 5 per minute)
- deny direct access as root via SSH
- change the default port number from 22 to something else
- use Fail2ban to blacklist attackers after N failures in the log
- use 2FA (e.g. by adopting Google Authenticator)
- disable password-based client authentication ☺

Main applications of SSH

- **remote interactive access (with text-based interface)**
- **execution of commands on remote systems**
- **creation of tunnels for various applications**
- **… all with the security properties offered by SSH**

- **notes:**
  - nowadays directly available in Linux, Mac, and Windows,
  - both the client and the server

RFC for SSH (I)

- RFC-4250 "The SSH protocol assigned numbers"
- RFC-4251 "The SSH protocol architecture"
- RFC-4252 "The SSH authentication protocol"
- RFC-4253 "The SSH transport layer protocol"
- RFC-4254 "The SSH connection protocol"
- RFC-4255 "Using DNS to securely publish SSH key fingerprints"
- RFC-4256 "Generic message exchange authentication for the SSH protocol"
- RFC-4335 "The SSH session channel break extension"
- RFC-4344 "The SSH transport layer encryption modes"

RFC for SSH (II)

- RFC-4345 "Improved Arcfour modes for the SSH transport layer protocol"
- RFC-4419 "Diffie-Hellman group exchange for the SSH transport layer protocol"
- RFC-4432 "RSA key exchange for the SSH transport layer protocol"
- RFC-4462 "GSS-API authentication and key exchange for the SSH protocol"
- RFC-4716 "The SSH public key file format"
- RFC-6239 "Suite B cryptographic suites for SSH"
- RFC-6668 "SHA-2 data integrity verification for the SSH transport layer protocol"

RFC for SSH (III)

- RFC-4819 "SSH public key subsystem"
- RFC-5592 "SSH transport model for the SNMP"
- RFC-5647 "AES GCM for the SSH transport layer protocol"
- RFC-5656 "EC algorithm Integration in the SSH transport layer"
- RFC-6187 "X.509v3 certificates for SSH authentication"
- RFC-6242 "Using the NETCONF protocol over SSH"
- RFC-7076 "P6R's SSH public key subsystem"
- RFC-8160 "IUTF8 terminal mode in SSH"
- RFC-8268 "More modular exponentiation (MODP) DH key exchange groups for SSH"
- RFC-8270 "Increase the SSH minimum recommended DH modulus size to 2048 bits"

RFC for SSH (IV)

- RFC-8308 "Extension Negotiation in the Secure Shell (SSH) Protocol"
- RFC-8332 "Use of RSA Keys with SHA-256 and SHA-512 in the Secure Shell (SSH) Protocol"
- RFC-8709 "Ed25519 and Ed448 Public Key Algorithms for the Secure Shell (SSH) Protocol"
- RFC-8731 "Secure Shell (SSH) Key Exchange Method Using Curve25519 and Curve448"
- RFC-8758 "Deprecating RC4 in Secure Shell (SSH)"