FACULTEIT INGENIEURSWETENSCHAPPEN

Network and Computer Security

Chapter 3 – Network and communication security

Prof. dr. ir. Eli De Poorter

Until now

- **Until now: basic concepts**
  - **Symmetric encryption**
  - **Asymmetric encryption**
  - **Hash functions**
  - **Message authentication codes**
- **From now**
  - **Applying these concepts to design security protocols for networked devices**
    - **Secure configuration of devices**
    - **Exchanging keys**
    - **Secure networking protocols**
    - **Firewalls**

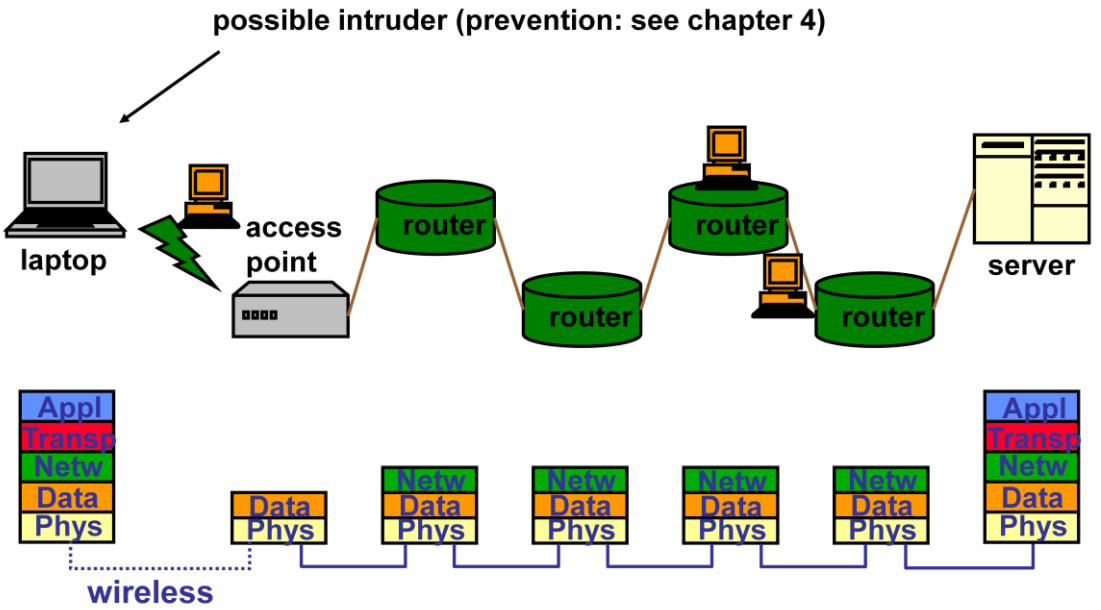
Overview

- Network model
- Secure configuration of devices
  - SSH (Secure Shell)
- Exchanging keys
  - Out of band
  - Diffie-Hellman
  - Asymmetric encryption
  - Trusted third party
    - Key Distribution Centre (KDC)
    - Public Key Infrastructure (PKI)
- Secure networking protocols
  - Transport layer: TLS & SSL
  - Network layer: IPSec & VPN
  - Data link layer: WEB & WPA
- Firewalls
  - Packet filter
  - Circuit-level gateway
  - Application-level gateway (proxy)

Overview

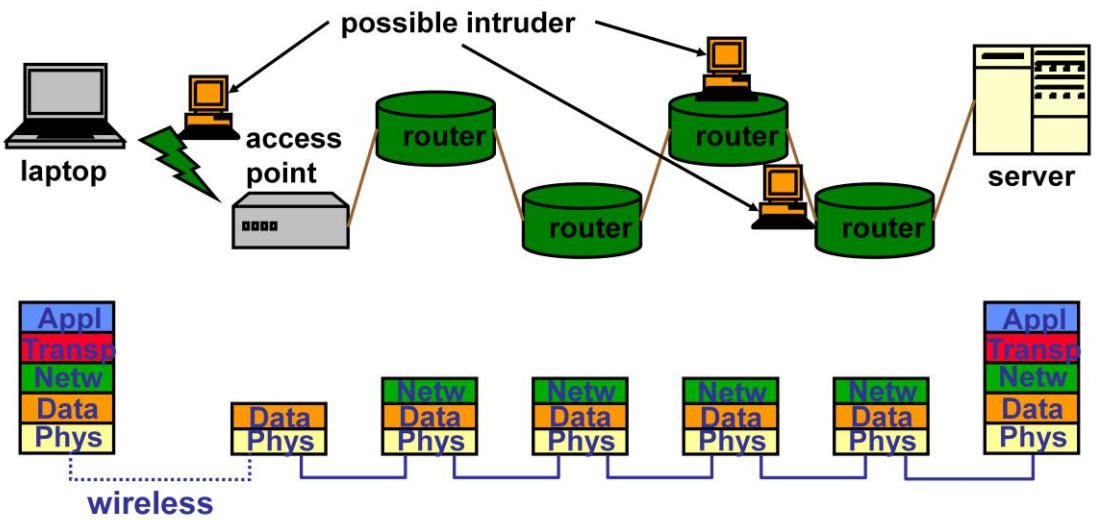
- **Network model**
- Secure configuration of devices
- Exchanging keys
- Secure networking protocols
- Firewalls

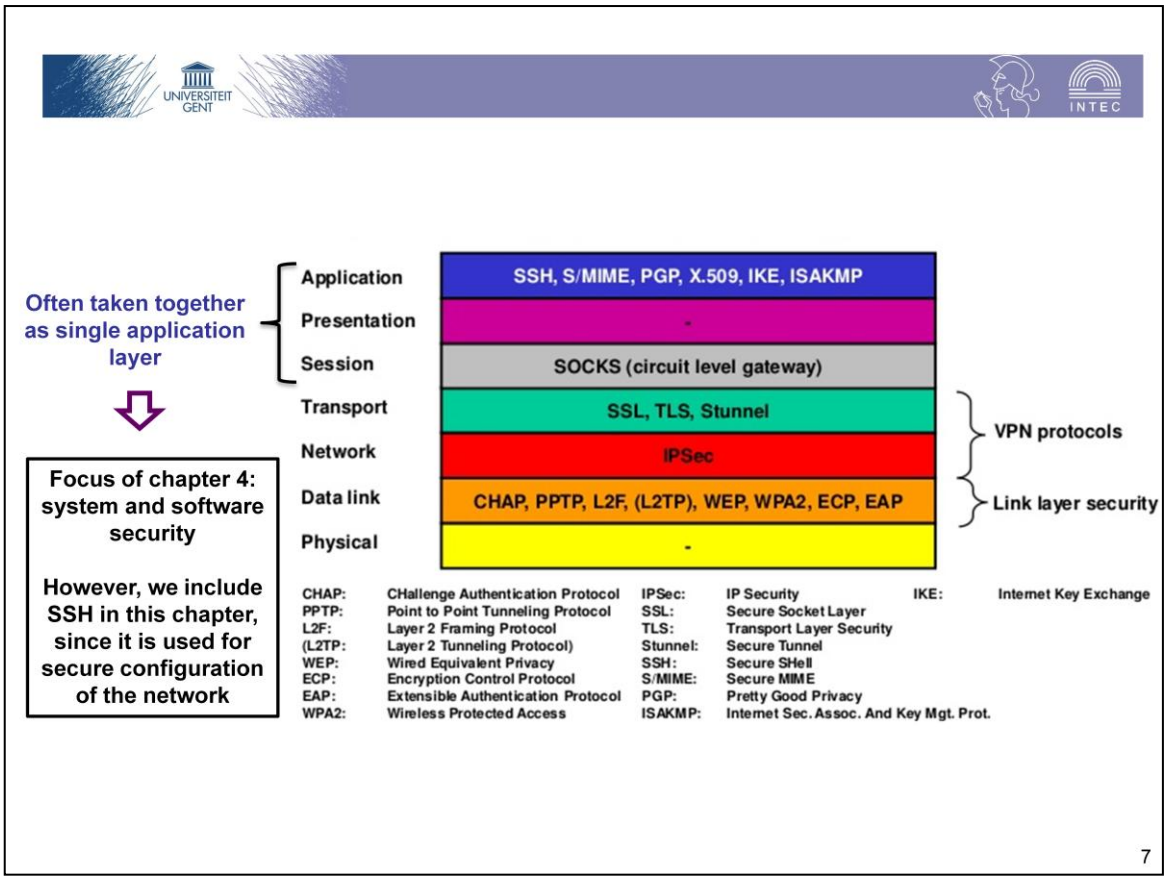
Network architecture



p. 5

Network architecture





SSH is an application layer security protocol. However, it includes a "transport layer protocol" component which runs on top of the OSI transport layer, acting as a secure connection between other OSI application layer protocols (such as HTTP) and the actual OSI transport layer. Due to the confusingly named RFC 4253, SSH is often depicted as a transport layer protocol, which it is not.

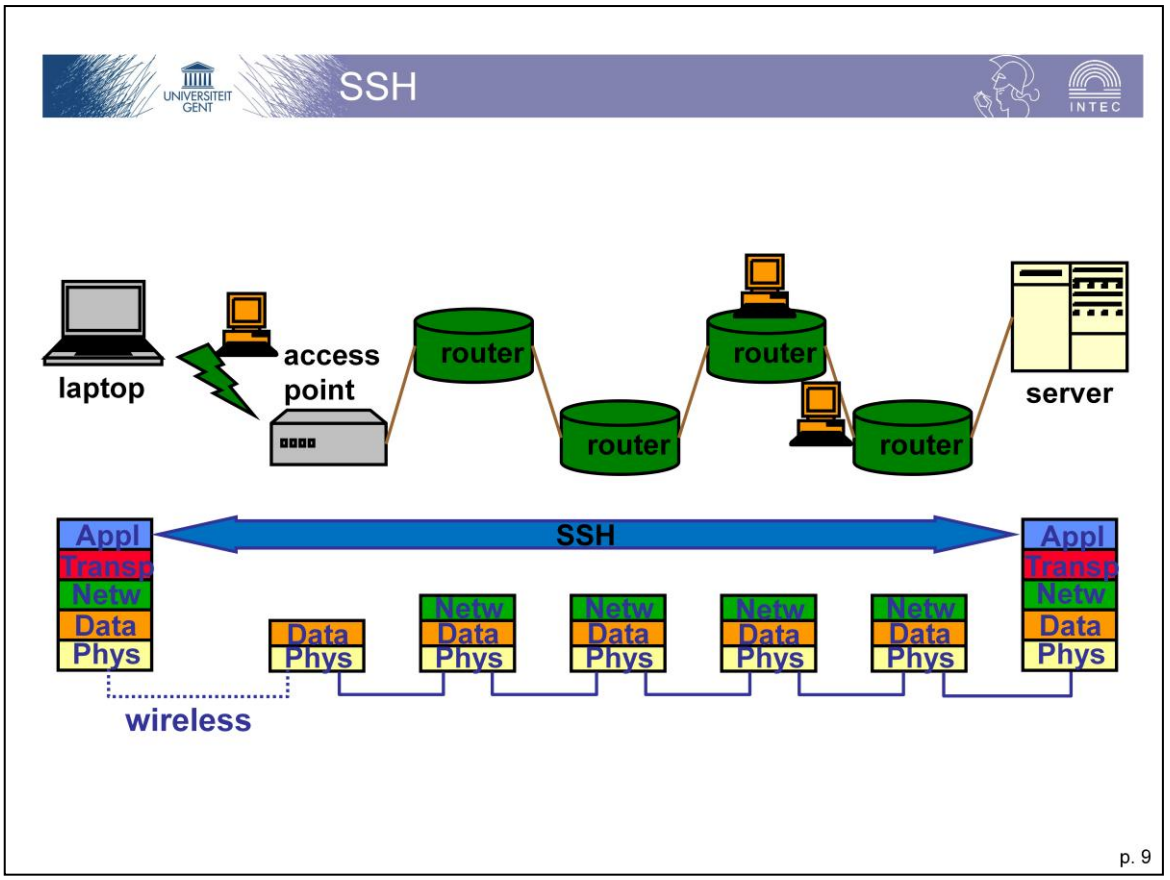
More information can be found in:

RFC 4253 Transport Layer Protocol

RFC 4251 Application layer protocol

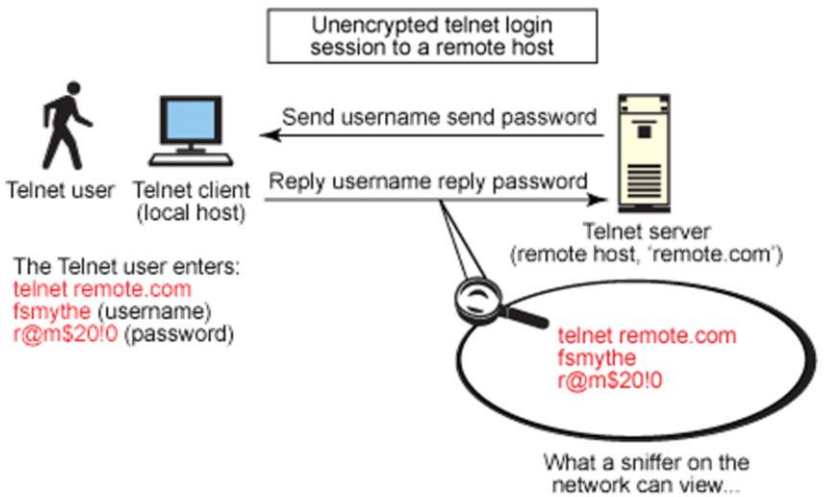
Overview

- Network model
- Secure configuration of devices
  - **SSH (Secure Shell)**
- Exchanging keys
- Secure networking protocols
- Firewalls



Note that many routers and even switches also include SSH management functionality (e.g. they include some application layer functionality).

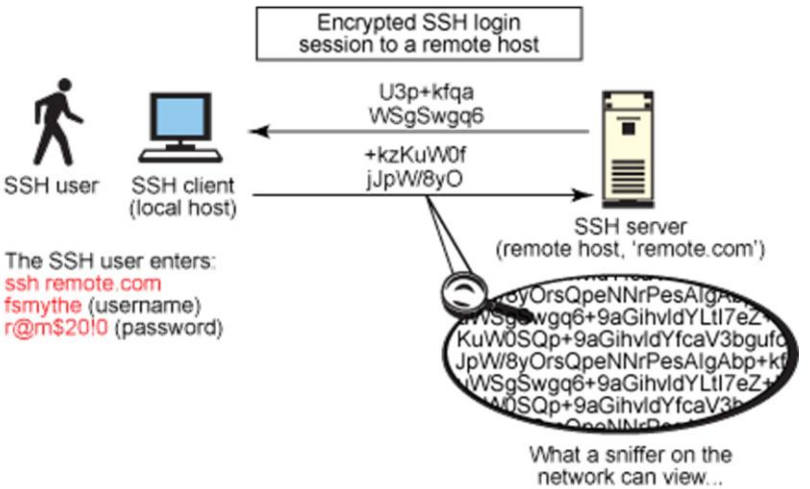
- **Telnet for server management: inherently unsafe**



10

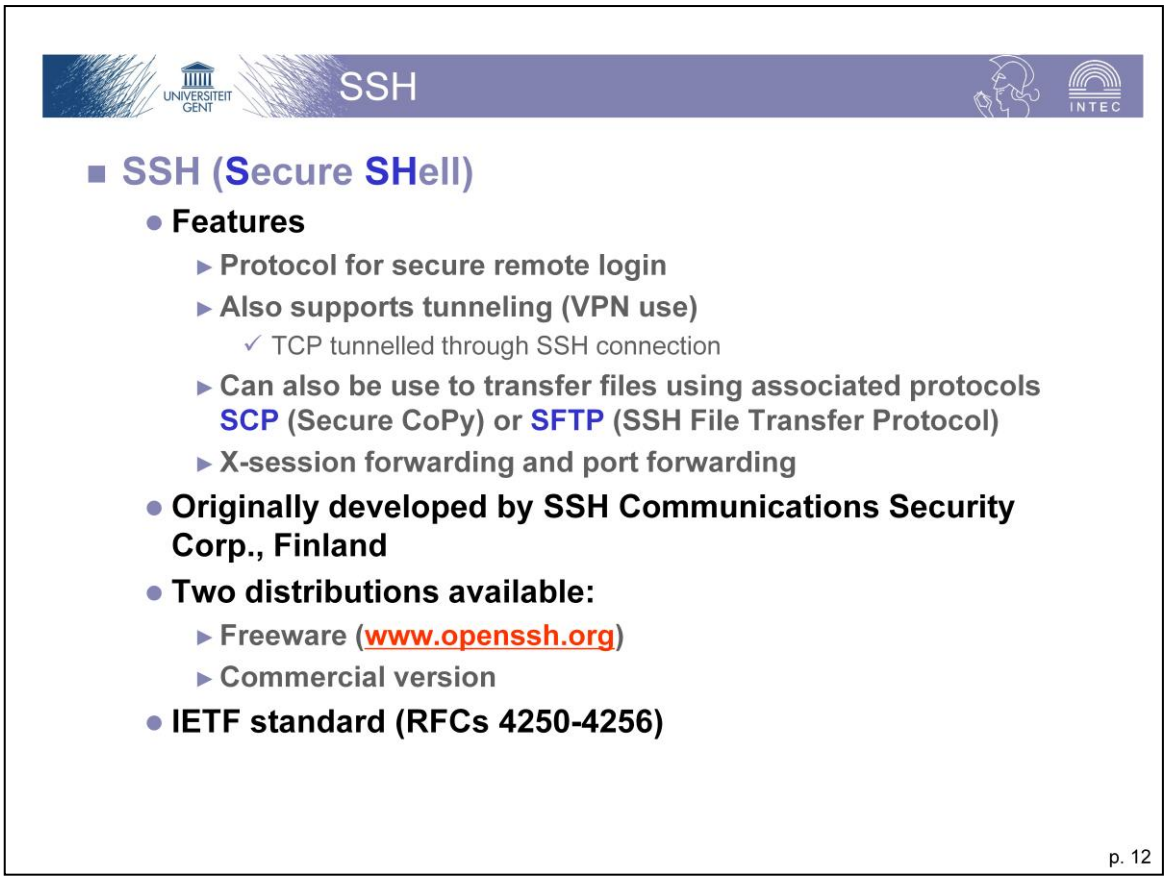
Secure Shell (SSH) was intended and designed for protection when remotely accessing another host over the network. The figures show how easily a telnet session can be casually viewed by anyone on the network using a network-sniffing application such as Wireshark. When using an unsecured, "clear text" protocol such as telnet, anyone on the network can pilfer your passwords and other sensitive information. The figure shows user fsmythe logging in to a remote host through a telnet connection. He enters his user name fsmythe and password r@m$20!0, which are both then viewable by any other user on the same network as our hapless and unsuspecting telnet user.

Server management using SSH



Secure Shell (SSH) was intended and designed to offer protection when remotely accessing another host over the network and was designed to be relatively simple and inexpensive to implement. The initial version, SSH1, focused on providing a secure remote logon facility to replace Telnet and other remote logon schemes that provided no security. A new version, SSH2, provides a standardized definition of SSH and improves on SSH1 in numerous ways.

This slide provides an overview of a typical SSH session and shows how the encrypted protocol cannot be viewed by any other user on the same network segment. Every major Linux and UNIX distribution now comes with a version of the SSH packages installed by default—typically, the open source OpenSSH packages—so there is little need to download and compile from source. If you're not on a Linux or UNIX platform, a plethora of open source and freeware SSH-based tools are available that enjoy a large following for support and practice, such as WinSCP, Putty,FileZilla, TTSSH, and Cygwin (POSIX software installed on top the Windows operating system). These tools offer a UNIX- or Linux-like shell interface on a Windows platform. In addition, Windows 10 will include SSH support by default.

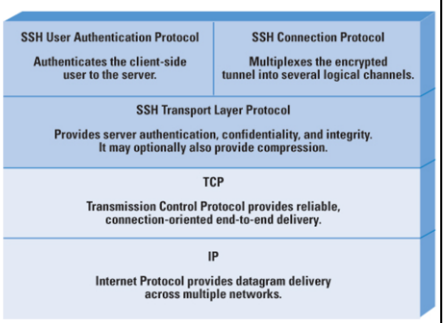


SSH provides a general client-server capability and encrypts the network exchange. In addition, it can be used to secure such network functions as file transfer (Secure Copy (SCP) and Secure File Transfer Protocol (SFTP)), X session forwarding, and port forwarding to increase the security of other insecure protocols. Various types of encryption are available, ranging from 512-bit encryption to as high as 32768 bits, inclusive of ciphers, like Blowfish, Triple DES, CAST-128, Advanced Encryption Scheme (AES), and ARCFOUR. Higher-bit encryption configurations come at a cost of greater network bandwidth use.

SSH2 is documented as a proposed standard in RFCs 4250 through 4256.

SSH: architecture

- **SSH transport layer protocol**
  - Server authentication, confidentiality, and integrity
  - Compression (optional)
  - On top of *reliable* transport layer (e.g. TCP)
- **SSH user authentication protocol**
  - Client authentication
  - On top of SSH transport layer
- **SSH connection protocol**
  - Multiplexes secure tunnel provided by SSH transport layer and user authentication layer into several logical channels
  - Logical channels can be used for various purposes



p. 13

IETF RFCs 4251 through 4256 define SSH as the "Secure Shell Protocol for remote login and other secure network services over an insecure network." The shell consists of three main elements.

Transport Layer Protocol: This protocol accommodates server authentication, data confidentiality, and data integrity with perfect forward privacy (that is, if a key is compromised during one session, the knowledge does not affect the security of earlier sessions). The transport layer is responsible for key exchange and server authentication. It sets up encryption, integrity verification, and (optionally) compression and exposes to the upper layer an API for sending and receiving plain text packets. This layer is typically run over a TCP/IP connection but can also be used on top of any other dependable data stream.

User Authentication Protocol: This protocol authenticates the client to the server and runs over the transport layer. Common authentication methods include password, public key, keyboard-interactive, GSSAPI, SecureID, and PAM.

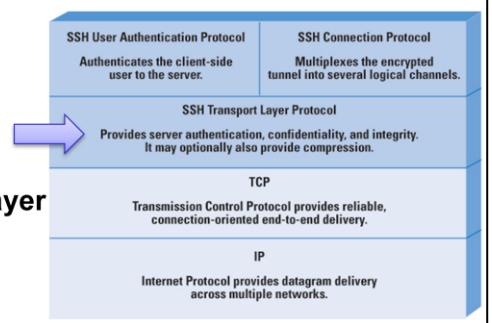
Connection Protocol: This protocol multiplexes the encrypted tunnel to numerous logical channels, running over the User Authentication Protocol. The connection layer defines channels, global requests, and the channel requests through which SSH services are provided. A single SSH connection can host multiple channels concurrently, each transferring data in both directions. Channel requests relay information such as the exit code of a server-side process. The SSH client initiates a request to forward a server-side port.

The "SSH transport layer protocol" is not to be confused with the "OSI transport layer": it is part of the SSH protocol and runs on top of the layer 4 OSI network layer (i.e.: the actual transport layer of the network stack, e.g. TCP or any other reliable transport layer protocol).

- **Security features**
  - **Uses well-established algorithms for encryption, data-integrity, key exchange, and public key management**
  - **Encryption with at least 128 bit keys**
  - **Security algorithm negotiation**
    - **Basic protocol needn't be changed when switching to other, newer algorithms**

SSH: transport layer protocol

- **Features**
  - **Server authentication, confidentiality, and integrity**
  - **Compression (optional)**
  - **On top of *reliable* transport layer (e.g. TCP)**

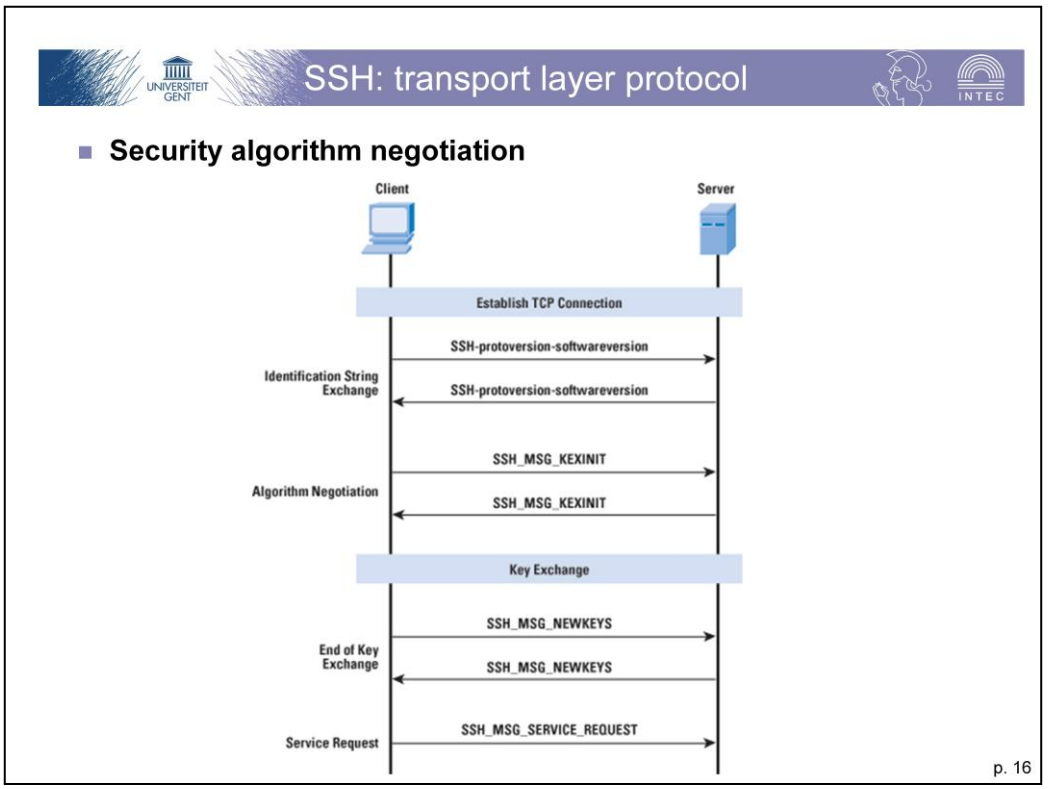


p. 15

Server authentication occurs at the transport layer protocol, based on the server possessing a public-private key pair. A server may have multiple host keys using multiple different asymmetric encryption algorithms. Multiple hosts may share the same host key. In any case, the server host key is used during key exchange to authenticate the identity of the host. For this authentication to be possible, the client must have presumptive knowledge of the server public host key. RFC 4251 dictates two alternative trust models that can be used:

• The client has a local database that associates each host name (as typed by the user) with the corresponding public host key. This method requires no centrally administered infrastructure and no third-party coordination. The downside is that the database of name-to-key associations may become burdensome to maintain.

• The host name-to-key association is certified by a trusted *Certification Authority* (CA) (see later). The client knows only the CA root key and can verify the validity of all host keys certified by accepted CAs. This alternative eases the maintenance problem, because ideally only a single CA key needs to be securely stored on the client. On the other hand, each host key must be appropriately certified by a central authority before authorization is possible.



SSH includes negotiation algorithms to determine suitable encryption algorithms. The SSH Transport Layer packet exchange consists of a sequence of steps, illustrated above. First, the client establishes a TCP connection to the server with the TCP protocol (which is not part of the Transport Layer Protocol). When the connection is established, the client and server exchange the following data packets (using the data field of a TCP segment).

Identification string exchange

The first step, the *identification string exchange,* begins with the client sending a packet with an identification string of the form: *"SSH-protoversion-softwareversion SP comments CR LF",* where SP, CR, and LF are space character, carriage return, and line feed, respectively. An example of a valid string is SSH-2.0-billsSSH_3.6.3q3<CR><LF> (SSH protocol version 2, software version billsSSH_3.6.3q3). The server responds with its own identification string.

Algorithm negotiation

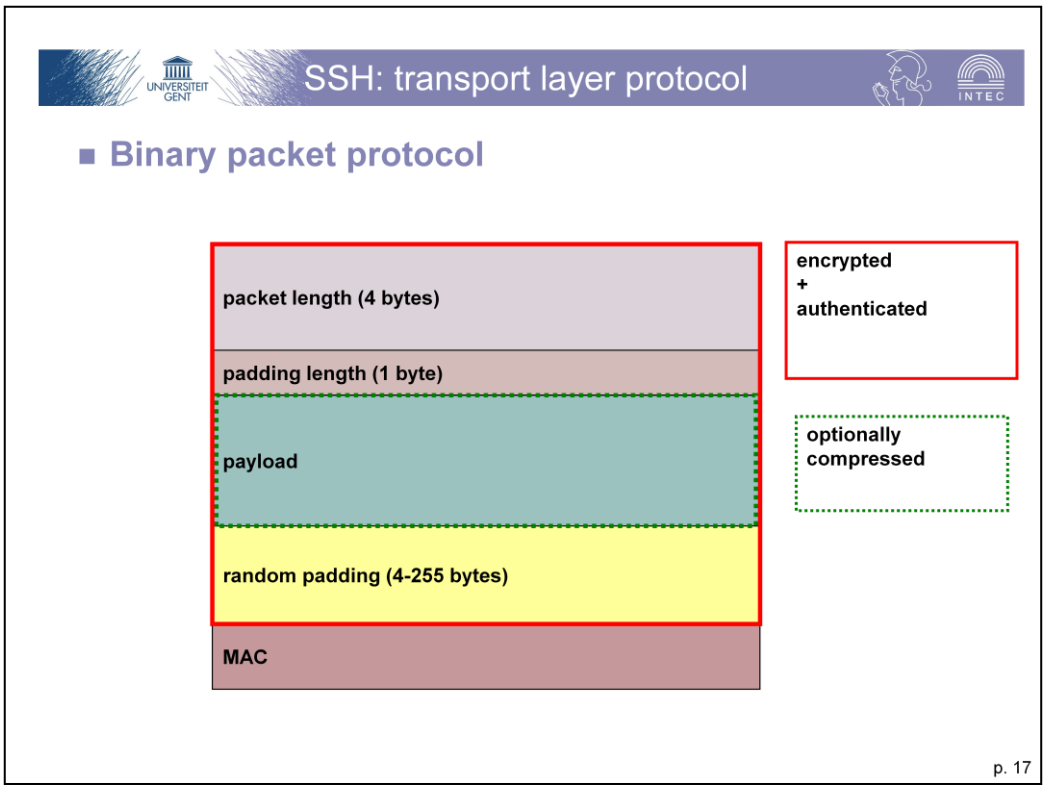
Next comes *algorithm negotiation.* Each side sends an SSH_MSG_KEXINIT containing lists of supported algorithms in the order of preference to the sender. Each type of cryptographic algorithm has one list. The algorithms include key exchange, encryption, MAC algorithm, and compression algorithm. For each category, the algorithm chosen is the first algorithm on the client's list that is also supported by the server.

Key exchange

The next step is *key exchange.* The specification allows for alternative methods of key exchange, but at present only two versions of Diffie–Hellman (assymetrical encryption) key exchange are specified. Both versions are defined in RFC 2409 and require only one packet in each direction. The *end of key exchange* is signaled by the exchange of SSH_MSG_NEWKEYS packets. At this point, both sides may start using the generated keys.

Service request

The final step is *service request.* The client sends an SSH_MSG_SERVICE_REQUEST packet to request either the User Authentication or the Connection Protocol. Subsequent to this request, all data is exchanged as the payload of an SSH Transport Layer packet, protected by encryption and MAC.



Each data packet is in the following format:

Packet length: Packet length is the length of the packet in bytes, not including the packet length and Message Authentication Code (MAC) fields.

Padding length: Padding length is the length of the random padding field.

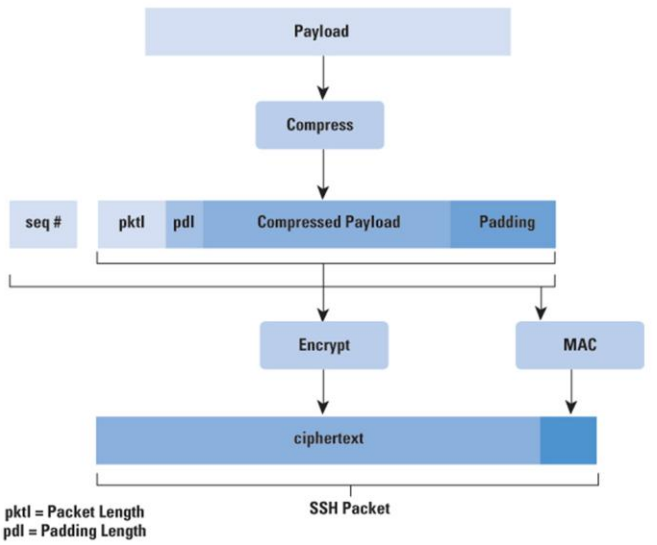
Payload: Payload constitutes the useful contents of the packet. Prior to algorithm negotiation, this field is uncompressed. If compression is negotiated, then in subsequent packets this field is compressed. Maximal (uncompressed) size is (at least) 32768 bytes (longer packets may be supported, depending on the implementation).

Random padding: After an encryption algorithm is negotiated, this field is added. It contains random bytes of padding so that that total length of the packet (excluding the MAC field) is a multiple of the cipher block size, or 8 bytes for a stream cipher. The length of the padding is arbitrary to thwart traffic analysis.

Message Authentication Code (MAC): If message authentication has been negotiated, this field contains the MAC value. The MAC value is computed over the entire packet plus a sequence number, excluding the MAC field. The sequence number is an implicit 32-bit packet sequence that is initialized to zero for the first packet and incremented for every packet. The sequence number is not included in the packet sent over the TCP connection.

SSH: transport layer protocol

- **Binary packet protocol**



p. 18

Process illustrating the calculation of encrypted packets.

SSH: transport layer protocol

- **Encryption algorithm**
  - **Negotiated during key exchange**
  - **Supported algorithms**
    - **REQUIRED: 3des-cbc ("168" bit key)**
    - **RECOMMENDED: aes128-cbc**
    - **OPTIONAL: aes192-cbc, aes256-cbc, arcfour (i.e. RC4), etc.**
  - **May be different for each direction**
  - **Secret key established during key exchange**
  - **All packets sent in 1 direction SHOULD be considered a single data stream**

Supported encryption algorithms	
3des-cbc*	Three-key Triple Digital Encryption Standard (3DES) in Cipher-Block-Chaining (CBC) mode
blowfish-cbc	Blowfish in CBC mode
twofish256-cbc	Twofish in CBC mode with a 256-bit key
twofish256-cbc	Twofish in CBC mode with a 256-bit key
twofish192-cbc	Twofish with a 192-bit key
twofish128-cbc	Twofish with a 128-bit key
aes256-cbc	Advanced Encryption Standard (AES) in CBC mode with a 256-bit key
aes192-cbc	AES with a 192-bit key
aes128-cbc**	AES with a 128-bit key
Serpent256-cbc	Serpent in CBC mode with a 256-bit key
Serpent192-cbc	Serpent with a 192-bit key
Serpent128-cbc	Serpent with a 128-bit key
arcfour	RC4 with a 128-bit key
cast128-cbc	CAST-128 in CBC mode

SSH: transport layer protocol

- **MAC algorithm**
  - **Negotiated during key exchange**
  - **Supported algorithms**
    - **REQUIRED: hmac-sha1**
    - **RECOMMENDED: hmac-sha1-96, hmac-sha2-256**
      - ✓ hmac-sha1-96: first 96 bits of hmac-sha1
    - **OPTIONAL: hmac-md5, hmac-md5-96, hmac-sha2-512, etc.**
  - **May be different for each direction**
  - **MAC = mac(key, seq. nr. | unencrypted packet)**
    - **Implicit seq. nr. (4 bytes), not sent with packet**
    - **Seq. nr. initialised to 0, incremented after each packet**

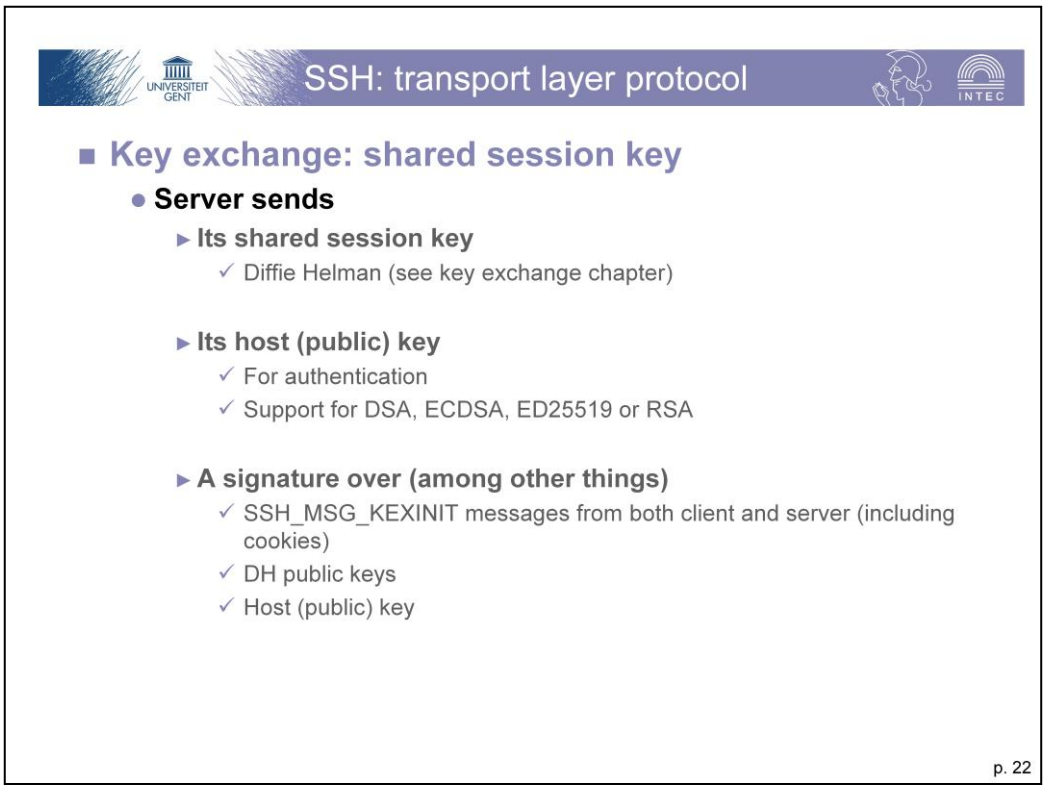
Supported MAC algorithms	
hmac-sha1*	HMAC-SHA1; Digest length = Key length = 20
hmac-sha1-96**	First 96 bits of HMAC-SHA1; Digest length = 12; Key length = 20
hmac-md5	HMAC-SHA1; Digest length = Key length = 16
hmac-md5-96	First 96 bits of HMAC-SHA1; Digest length = 12; Key length = 16

p. 20

The implicit sequence number for the MAC is never reset, not even if the keys and the algorithms for the SSH-connection are renegotiated later on.

SSH: transport layer protocol

- **New symmetric keys are generated and exchanged foe each new session**
  - Different keys for encryption and for MAC algorithms
  - Can even support different keys for each direction
- **How are the symmetric keys exchanged?**
  - In a secure way?



Key exchange happens in two fases: (i) first a shared key is generated using Diffie-Helman, (ii) afterwards the shared key is signed with the public key of the client to provide authentication.

For the creation and exchange of a symmetric key, Diffie Helman (DH) is used (see key exchange chapter). The way SSH uses DH is as an ephemeral algorithm: DH parameters are generated for individual sessions, and are destroyed as soon as they're no longer needed. The only thing the long-lasting key pair is used for is authentication. This gives forward secrecy: stealing the private key doesn't let you decrypt old sessions. This key agreement results in a shared session key.

In addition, after a shared key is established, the client host key is used to sign the Diffie-Hellman parameters. SSH protocol 2 supports signatures based on DSA, ECDSA, ED25519 or RSA signature algorithms. They are only used after key exchange to sign the DH messages and prove the client has the private key (one side of DH parameters being signed is enough to prevent a MITM, because the attacker can't impersonate both client and server; it's easier to make the server always have to have a keypair than make the client always have to have a keypair).

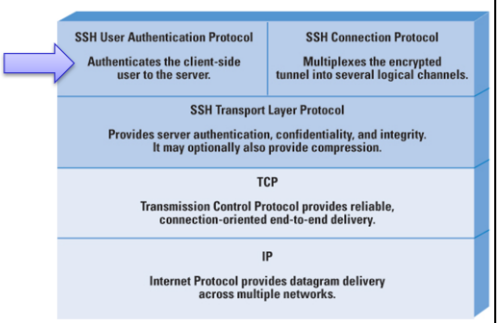
Now that the shared session key has been securely generated and authenticated, the rest of the session is encrypted using a symmetric cipher

- **Key exchange: server authentication**
  - **Based on server host key**
  - **Various possible techniques for key identification**
    - **Using local databases**
    - **Using certification authorities (CA) and certificates**
    - **Using offline channels**
      - ✓ E.g. key “fingerprint”
    - **Using “best effort” approach**
      - ✓ Accept host key without check during first connection to the server
      - ✓ Save host key in local database
      - ✓ Check against saved key for later connections
  - **See alternatives from key exchange chapter**

- **Key re-exchange**
  - Possible at any moment (except during key exchange)
  - Can be initiated by either party
  - Session ID doesn't change
  - Algorithms may be changed
  - Session keys are changed
  - Recommended
    - After some amount of data (e.g. 1 GB)
    - After some amount of time (e.g. 1 h)

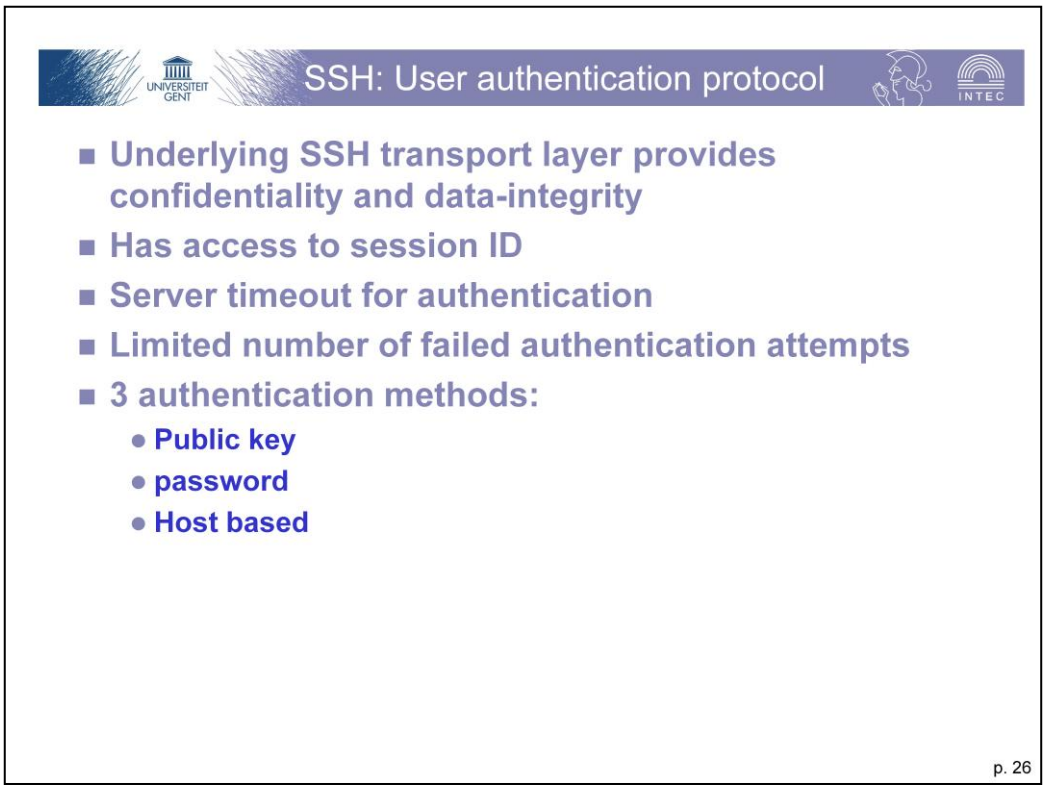
SSH: User authentication protocol

- **SSH user authentication protocol**
  - **Client authentication**
  - **On top of SSH transport layer**



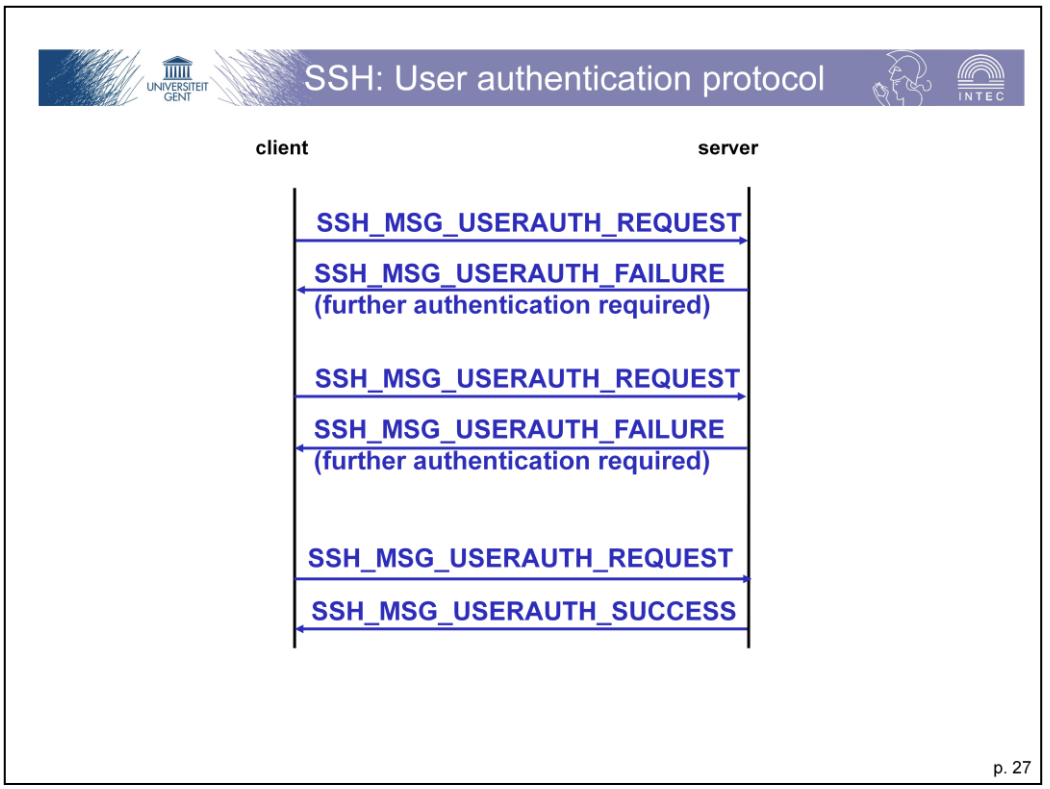
p. 25

The *User Authentication Protocol* provides the means by which the client is authenticated to the server.



The server will disconnect the client if the authentication has not been accepted within the timeout period (recommended value for the timeout: 10 minutes).

The recommended value for the maximal number of failed authentication attempts is 20 within a single session.



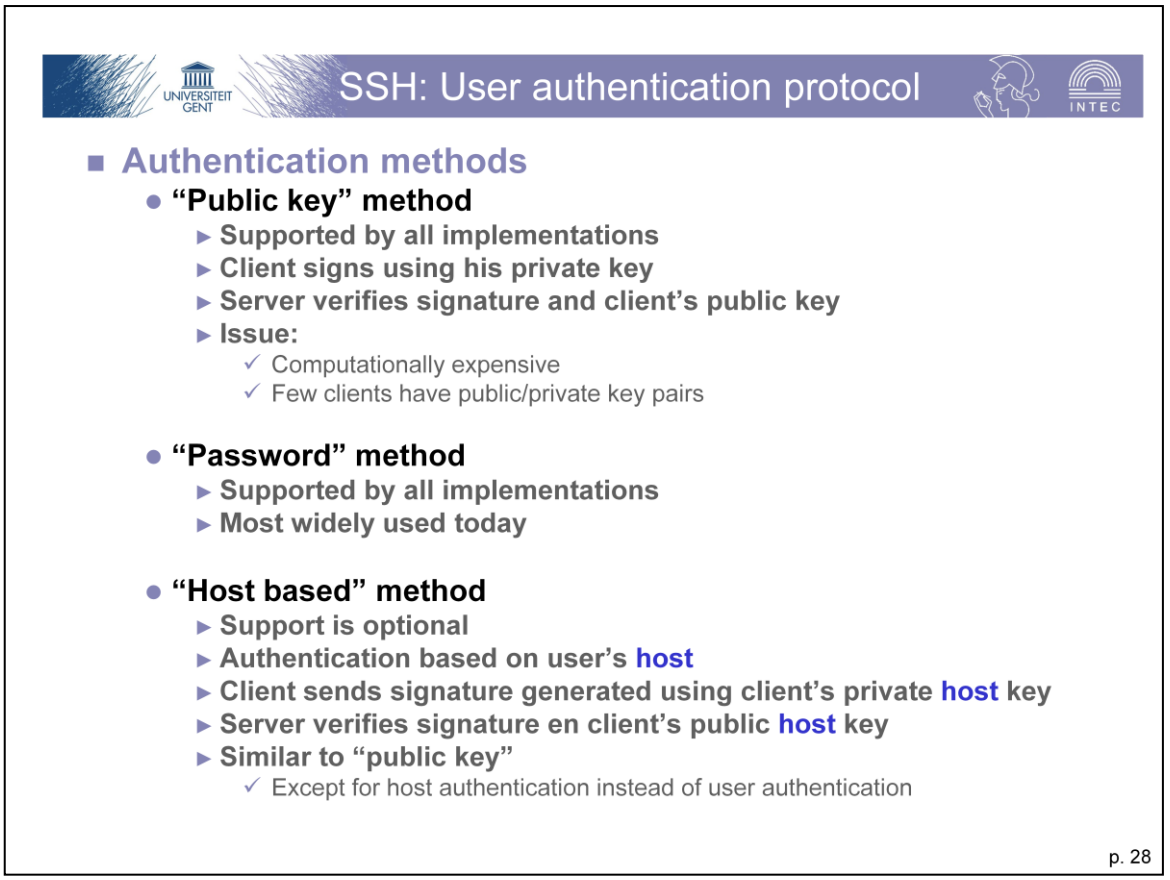
The message exchange involves the following steps:

1. The client sends a SSH_MSG_USERAUTH_REQUEST with a requested method of none.
2. The server checks to determine if the username is valid. If not, the server returns SSH_MSG_USERAUTH_FAILURE with the partial success value of false. If the username is valid, the server proceeds to step 3.
3. The server returns SSH_MSG_USERAUTH_FAILURE with a list of one or more authentication methods to be used.
4. The client selects one of the acceptable authentication methods and sends a SSH_MSG_USERAUTH_REQUEST with that method name and the required method-specific fields. At this point, there may be a sequence of exchanges to perform the method.
5. If the authentication succeeds and more authentication methods are required, the server proceeds to step 3, using a partial success value of true. If the authentication fails, the server proceeds to step 3, using a partial success value of false.
6. When all required authentication methods succeed, the server sends a SSH_MSG_USERAUTH_SUCCESS message, and the Authentication Protocol is over.

The SSH_MSG_USERAUTH_REQUEST message contains the following fields: the user name, the service name, the method name, plus additional fields specific for the authentication method used. *Username* is the authorization identity the client is claiming, *service name* is the facility to which the client is requesting access (typically the SSH Connection Protocol), and *method name* is the authentication method being used in this request ("publickey", "password" or "hostbased").

If the server either rejects the authentication request or accepts the request but requires one or more additional authentication methods, the server sends a SSH_MSG_USERAUTH_FAILURE message. The SSH_MSG_USERAUTH_FAILURE message isn't necessarily an indication of a fatally unsuccessful authentication. It may be merely an indication that the desired authentication isn't yet achieved. This message contains a list of authentication methods that can continue and a partial success flag, which is set to TRUE if the previous request was successful but further authentication is needed, and is set to FALSE if the previous request failed. This implies that the number of SSH_MSG_USERAUTH_REQUEST and SSH_MSG_USERAUTH_FAILURE messages isn't necessarily the same for all authentication processes.

Eventually, when the authentication is complete, the server replies with a SSH_MSG_USERAUTH_SUCCESS message and starts the requested service.



The server may require one or more of the following authentication methods:

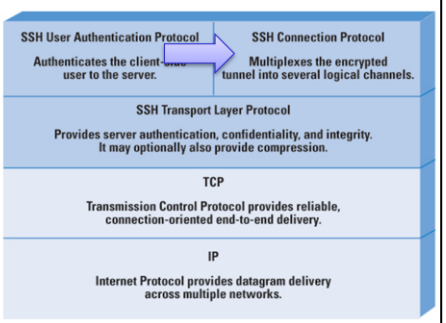
publickey: The details of this method depend on the public-key algorithm chosen. In essence, the client sends a message to the server that contains the client's public key, with the message signed by the client's private key. When the server receives this message, it checks to see whether the supplied key is acceptable for authentication and, if so, it checks to see whether the signature is correct. The SSH_MSG_USERAUTH_REQUEST may also contain a flag set to FALSE (instead of TRUE) and no signature. This allows the client to test whether this (computationally expensive) authentication method is accepted by the server. If it is, the server will respond with a SSH_MSG_USERAUTH_PK_OK message and the authentication can continue.

password: The client sends a message containing a plaintext password, which is protected by encryption by the Transport Layer Protocol. If the server answers with a SSH_MSG_USERAUTH_PASSWD_CHANGEREQ message, the client will change the password by sending a SSH_MSG_USERAUTH_REQUEST for the "password" method containing a TRUE (instead of FALSE) flag, the old password (in plaintext) and the new password (in plaintext).

Host based: Authentication is performed on the client's host rather than the client itself. Thus, a host that supports multiple clients would provide authentication for all its clients. This method works by having the client send a signature created with the private key of the client host. Thus, rather than directly verifying the user's identity, the SSH server verifies the identity of the client host—and then believes the host when it says the user has already authenticated on the client side.

SSH: connection protocol

- **SSH connection protocol**
  - **Multiplexes secure tunnel provided by SSH transport layer and user authentication layer into several logical channels**
  - **Logical channels can be used for various purposes**



p. 29

The SSH Connection Protocol runs on top of the SSH Transport Layer Protocol and assumes that a secure authentication connection is in use. That secure authentication connection, referred to as a *tunnel,* is used by the Connection Protocol to multiplex a number of logical channels.

Confusingly, RFC 4254, "The Secure Shell (SSH) Connection Protocol," states that the Connection Protocol runs on top of the Transport Layer Protocol whilst the User Authentication Protocol. RFC 4251, "SSH Protocol Architecture," states that the Connection Protocol runs over the User Authentication Protocol. In fact, the Connection Protocol runs over the Transport Layer Protocol, but assumes that the User Authentication Protocol has been previously invoked.

SSH: connection protocol

- **Applications implemented as "channels"**
  - **All channels are multiplexed into single encrypted tunnel (provided by SSH transport layer protocol)**
  - **Channels identified by channel numbers at both ends of the connection**
  - **Channel numbers may differ for same channel at client and server sides**
- **Example channels**
  - **Session (shell, file transfer, e-mail, system command, ...)**
  - **X11-connections**
  - **Local port forwarding (direct TCP/IP)**
  - **Remote port forwarding (forwarded TCP/IP)**

p. 30

All types of communication using SSH, such as a terminal session, are supported using separate channels. Either side may open a channel. For each channel, each side associates a unique channel number, which need not be the same on both ends. Channels are flow-controlled using a window mechanism. No data may be sent to a channel until a message is received to indicate that window space is available.

Four channel types are recognized in the SSH Connection Protocol specification:

• *session:* Session refers to the remote execution of a program. The program may be a shell, an application such as file transfer or e-mail, a system command, or some built-in subsystem. When a session channel is opened, subsequent requests are used to start the remote program.

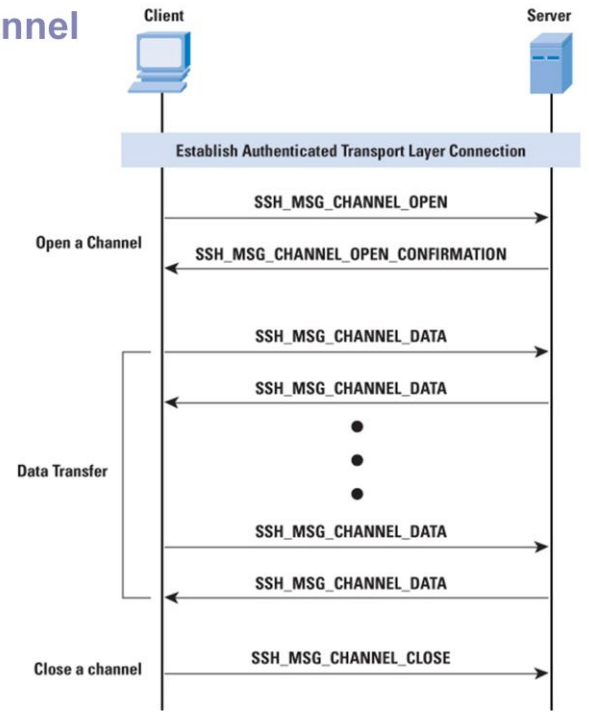
• *x11:* This channel type refers to the X Window System, a computer software system and network protocol that provides a GUI for networked computers. X allows applications to run on a network server but be displayed on a desktop machine.

•*direct-tcpip:* This channel type is local port forwarding, as explained subsequently.

•*forwarded-tcpip:* This channel type is remote port forwarding, as explained subsequently.

SSH: connection protocol

- Opening a channel



p. 31

The life of a channel progresses through three stages: opening a channel, data transfer, and closing a channel.

When either side wishes to open a new channel, it allocates a local number for the channel and then sends a message containing the channel type, sender channel, initial window size and maximum packet size. The *channel type* identifies the application for this channel, as described previously. The *sender channel* is the local channel number. The *initial window size* specifies how many bytes of channel data can be sent to the sender of this message without adjusting the window. The *maximum packet size* specifies the maximum size of an individual data packet that can be sent to the sender. For example, one might want to use smaller packets for interactive connections to get better interactive response on slow links.

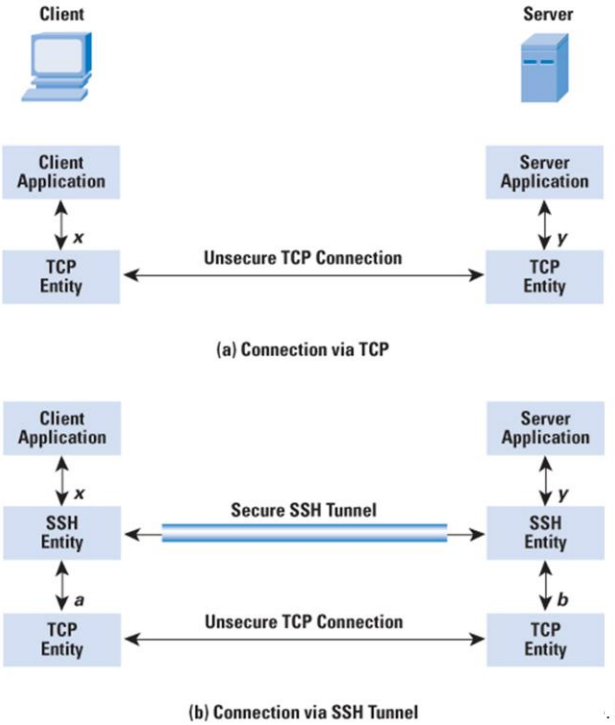
If the remote side is able to open the channel, it returns a SSH_MSG_CHANNEL_OPEN_CONFIRMATION message, which includes the sender channel number, the recipient channel number, and window and packet size values for incoming traffic. Otherwise, the remote side returns a SSH_MSG_CHANNEL_OPEN_FAILURE message with a reason code indicating the reason for failure.

After a channel is open, *data transfer* is performed using a SSH_MSG_CHANNEL_DATA message, which includes the recipient channel number and a block of data. These messages, in both directions, may continue as long as the channel is open.

When either side wishes to close a channel, is sends a SSH_MSG_CHANNEL_CLOSE message, which includes the recipient channel number.

SSH: connection protocol

- **SSH port forwarding**
  - **Goals**
    1. **Transform insecure TCP to secure SSH connections**
       - ✓ "SSH tunnel"
    2. **Bypass network restrictions**
  - **Local and remote port forwarding**



. 32

One of the most useful features of SSH is *port forwarding.* Port forwarding or port mapping is an application of network address translation (NAT) that redirects a communication request from one address and port number combination to another while the packets are traversing a network gateway. SSH port forwarding provides the ability to convert any insecure TCP connection into a secure SSH connection. It is also referred to as SSH tunneling. We need to know what a port is in this context. A port is an identifier of a user of TCP. So, any application that runs on top of TCP has a port number. Incoming TCP traffic is delivered to the appropriate application on the basis of the port number. For example, for the *Simple Mail Transfer Protocol* (SMTP), the server side generally listens on port 25, so that an incoming SMTP request uses TCP and addresses the data to destination port 25. TCP recognizes that this address is the SMTP server address and routes the data to the SMTP server application. The figure illustrates the basic concept behind port forwarding. We have a client application that is identified by port number *x* and a server application identified by port number *y.* At some point, the client application invokes the local TCP entity and requests a connection to the remote server on port *y.* The local TCP entity negotiates a TCP connection with the remote TCP entity, such that the connection links local port x to remote port *y.* To secure this connection, SSH is configured so that the SSH Transport Layer Protocol establishes a TCP connection between the SSH client and server entities with TCP port numbers *a* and *b,* respectively. A secure SSH tunnel is established over this TCP connection. Traffic from the client at port *x* is redirected to the local SSH entity and travels through the tunnel where the remote SSH entity delivers the data to the server application on port *y.* Traffic in the other direction is similarly redirected.

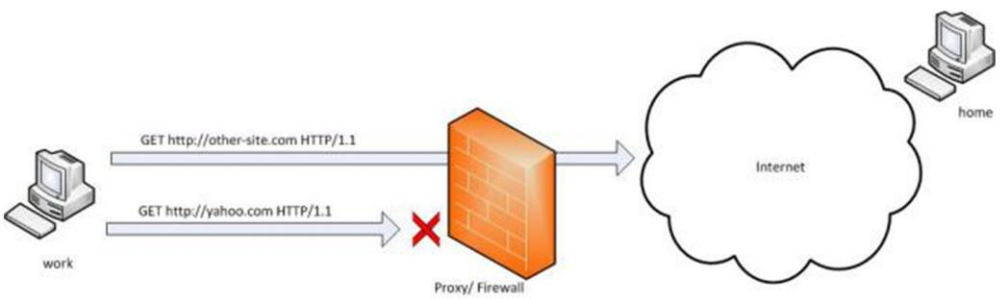
SSH supports two types of port forwarding: local forwarding and remote forwarding.

Local forwarding is used to **forward data securely from another client application running on the same computer** (using different port numbers) as the Secure Shell Client. It is frequently used to (i) transform; an insecure TCP connection by a secure one and/or (ii) to bypass network restrictions. Local port forwarding allows the client to set up a "hijacker" process. This process will intercept selected application-level traffic and redirect it from an unsecured TCP connection to a secure SSH tunnel. SSH is configured to listen on selected ports, grabs all traffic that uses the selected port and sends it through an SSH tunnel. On the other end, the SSH server sends the incoming traffic to the destination port dictated by the client application. Local port forwarding is the most common type of port forwarding.

With ***remote forwarding,*** the user's SSH client acts on the server's behalf. The client receives traffic with a given destination port number, places the traffic on the correct port, and sends it to the destination the user chooses. Remote port forwarding allows **other computers to access applications hosted on remote servers**. This form of port forwarding enables applications on the server side of a Secure Shell (SSH) connection to access services residing on the SSH's client side. With remote forwarding, the user's SSH client acts on the server's behalf. The client receives traffic with a given destination port number, places the traffic on the correct port, and sends it to the destination the user chooses. A typical example of remote forwarding follows: You wish to access a server at work from your home computer. Because the work server is behind a firewall, it will not accept an SSH request from your home computer. However, from work you can set up an SSH tunnel using remote forwarding.

SSH: connection protocol

- **Local Port Forwarding (outgoing tunnel)**
  - **Example: create an outgoing tunnel passing a firewall**

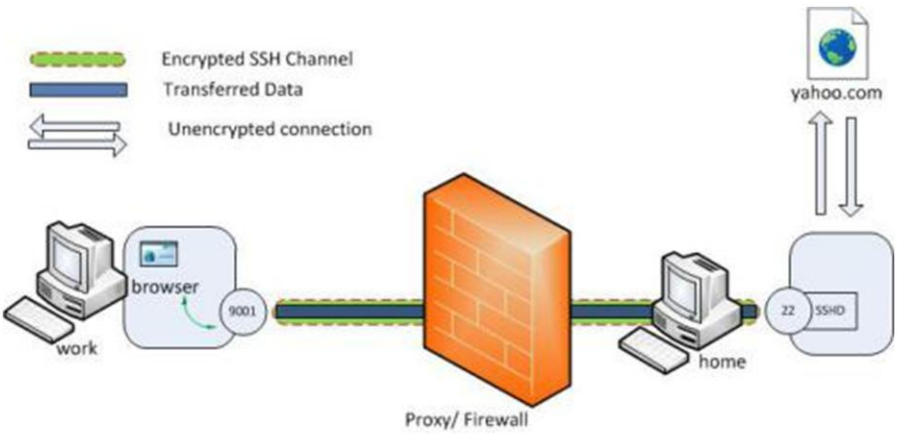


p. 33

Let's say that yahoo.com is being blocked using a proxy filter in the University. A SSH tunnel can be used to bypass this restriction.

SSH: connection protocol

- **Local Port Forwarding (outgoing tunnel)**
  - **The remote host (server) acts as a gateway**
    - **Example: create an outgoing tunnel passing a firewall**
  - **Command:**
    - **ssh -L local_port:remote_host:remote_port login@servername**
    - **ssh -L 9001:yahoo.com:80 home**



p. 34

The syntax of the local port forwarding command is as follows (The 'L' switch indicates that a local port forward is need to be created):

• *ssh -L <local-port-to-listen>:<remote-host>:<remote-port> <gateway>*

It can be interpreted as: I want to securely connect to <remote port> on <remote host> and to this end I set up an SSH tunnel from <local port> at my current host to <gateway> (which will act as a gateway).

Let's name my machine at the university as 'work' and my home machine as 'home' ('home' needs to have a public IP for this to work). I am running an SSH server on my home machine. To create the SSH tunnel execute the following from the 'work' machine.

• *ssh -L 9001:yahoo.com:80 home*

Now the SSH client at 'work' will connect to the SSH server running at 'home' (usually running at port 22) binding port 9001 of 'work' to listen for local requests thereby creating a SSH tunnel between 'home' and 'work'. At the 'home' end it will create a connection to 'yahoo.com' at port 80. So 'work' doesn't need to know how to connect to yahoo.com. Only 'home' needs to worry about that. Now it is possible to browse to yahoo.com by visiting http://localhost:9001 in the web browser at the 'work' computer. The 'home' computer will act as a gateway which would accept requests from 'work' machine and fetch data to tunnel it back. The connection from 'host' to 'yahoo.com' is only made when the browser makes the request (i.e. not already during the tunnel setup). The channel between 'work' and 'home' will be encrypted while the connection between 'home' and 'yahoo.com' will be unencrypted.

It is also possible to specify a port belonging to the 'home' computer itself instead of connecting to an external host. This option is typically not used if gateway functionality is required (i.e.: not for passing a firewall), but if an unsecure connection needs to be replaced by a secure connection. Typical use cases include using SSH to securely retrieve (port 110,POP) or transmit (port 25, SMTP) e-mails. For example, when setting up a VNC session between 'work' and 'home', the command line would be as follows.

• *ssh -L 5900:localhost:5900 home* (executed from 'work')

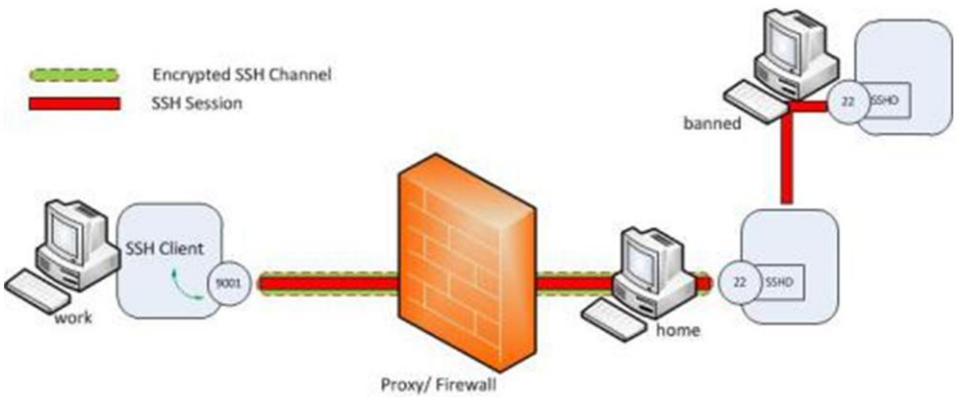
So what does localhost refer to? Is it the 'work' since the command line is executed from 'work'? Turns out that it is not: the localhost is considered relative to the gateway ('home' in this case) , not the machine from where the tunnel is initiated. So this will make a connection to port 5900 of the 'home' computer where the VNC client would be listening in. As such, the command is equivalent to

• *ssh -L 5900:home:5900 home* (executed from 'work')

This last notation makes the fact that in this case the server ('work') does not act as a gateway more explicit.

SSH: connection protocol

- **Local Port Forwarding (outgoing tunnel)**
  - **Example: tunnelling of SSH sessions**
    - ssh -L 9001:banned:22 home
    - ssh -p 9001 localhost



p. 35

The created tunnel can be used to transfer any type of connection and is thus not limited to web browsing sessions. In addition to mail sessions, ftp sessions, etc., we can even tunnel SSH sessions. Let's assume there is another computer ('banned') to which we need to SSH from within University but the SSH access is being blocked. It is possible to tunnel a SSH session to this host using a local port forward.

As can be seen now the transferred data between 'work' and 'banned' are encrypted end to end. For this we need to create a local port forward as follows.

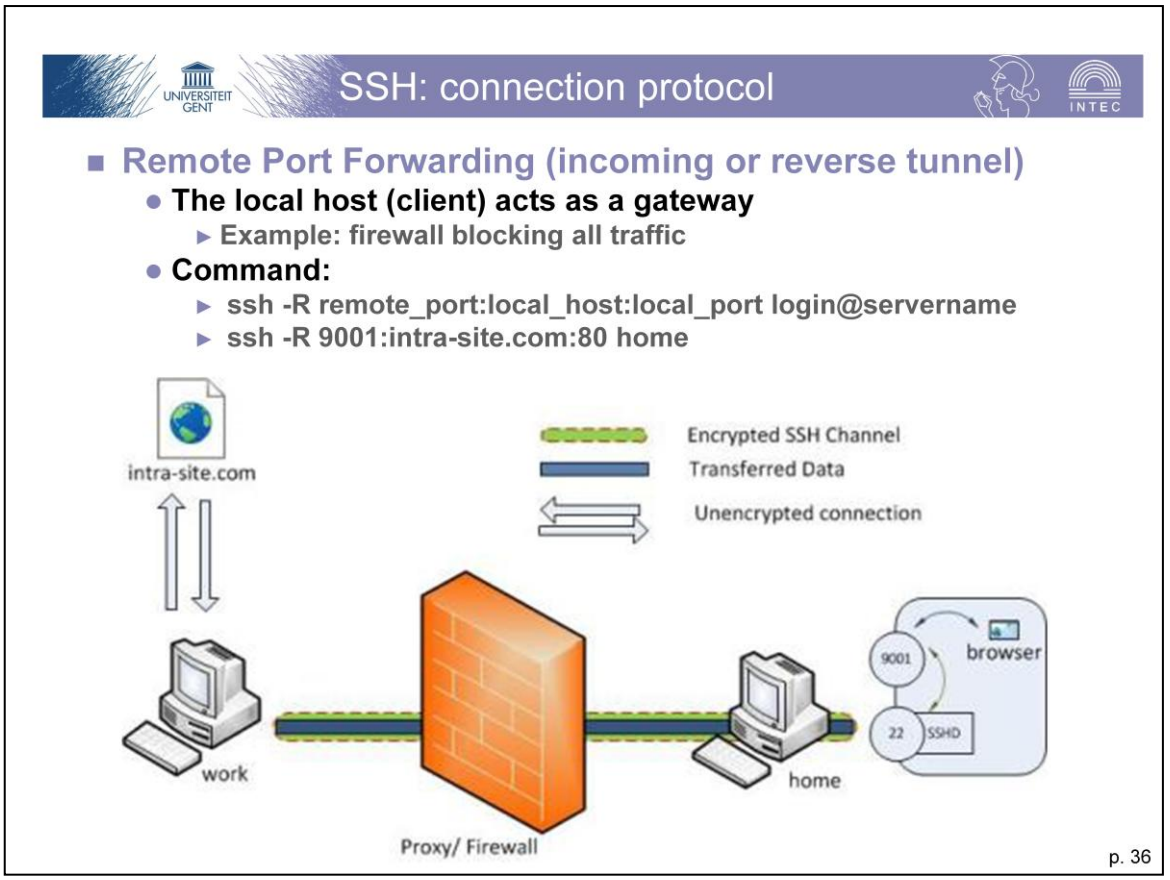
• ssh -L 9001:banned:22 home (executed from 'work')

Now 'home' acts as a gateway for tunneling SSH sessions to 'banned'.

To connect over SSH with 'banned', we create an SSH session to local port 9001.

• ssh -p 9001 localhost

This new SSH session will automatically get tunneled to 'banned' via the 'home' gateway due to the previously configured port forwarding rules.



The syntax of the remote port forwarding command is as follows (The 'R' switch indicates that a remote port forward is created):

• *ssh -R <server-port> :<remote-host>:<remote-port> <server host>*

The command can be interpreted as: I will act as a gateway to forward connections from <server port> on <server host> to <remote-port> on <remote-host> using a secure SSH tunnel.

Let's say we want to connect to an internal university website from home, but the university firewall is blocking all incoming traffic. How can we connect from 'home' to internal network so that we can browse the internal site? (A VPN setup is a good candidate here, but however for this example let's assume we don't have this facility). As discussed, local port forwarding creates an outgoing tunnel and as such is not useful in this case. However, we can use remote port forwarding (also referred to as SSH reverse tunneling) to create an incoming tunnel. As before, we will initiate the tunnel from the 'work' computer behind the firewall. This is possible since only incoming traffic is blocking and outgoing traffic is allowed. However instead of the earlier case the client will now be at the 'home' computer.

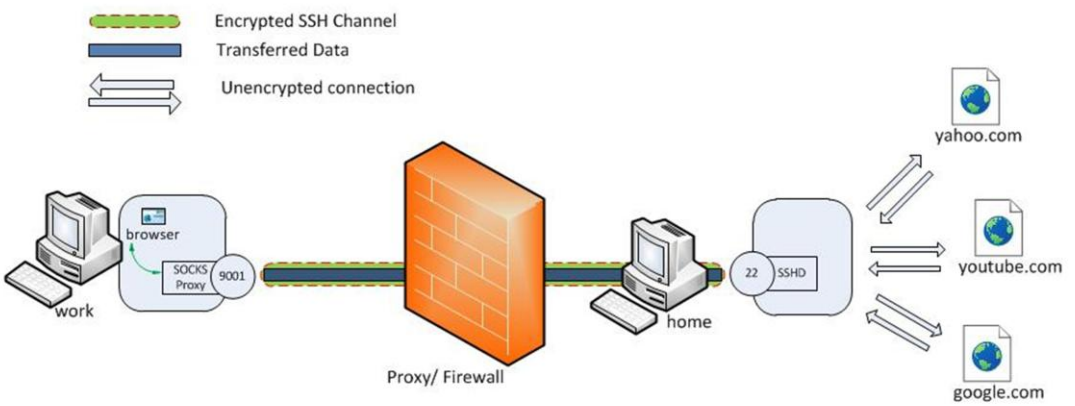
• ssh -R 9001:intra-site.com:80 home (executed from 'work', note the R instead of L)

Once executed the SSH client ('work') will connect to the SSH server (running at home) to create an SSH channel. Then the server will bind port 9001 on the 'home' machine to listen for incoming requests which would subsequently be routed through the created SSH channel between 'home' and 'work'. Now it's possible to browse the internal site by visiting http://localhost:9001 in 'home' web browser. The 'work' SSH server will then create a connection to the intra-site and relay back the response to 'home' via the created SSH channel.

As such, the command is very similar to the local port forwarding command, but the SSH client (work) acts as a gateway for incoming traffic rather than the SSH server (home). The main advantage is that in this case the creation of the tunnel was initiated from the gateway, allowing it to bypass NAS or firewall restrictions.

SSH: connection protocol

- **Dynamic Port Forwarding**
  - **SOCKS proxy**
  - **Command**
    - **ssh -D 9001 home**



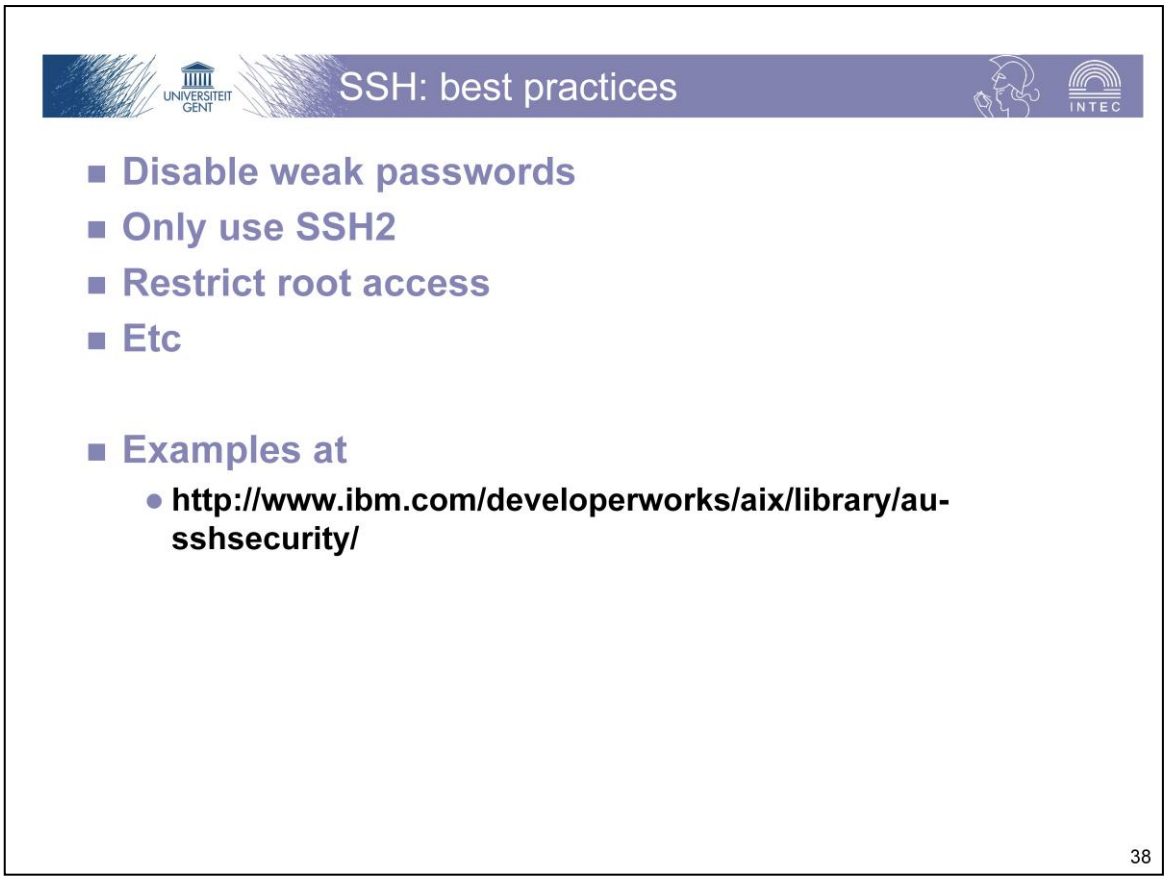
p. 37

Although useful, local and remote port forwarding require the creation of yet another tunnel if you need to connect to another website. Dynamic port forwarding makes it possible to proxy traffic to any site using the SSH created channel.

Dynamic port forwarding allows to configure one local port for tunneling data to all remote destinations. However to utilize this the client application connecting to local port should send their traffic using the SOCKS protocol. At the client side of the tunnel a SOCKS proxy would be created and the application (eg. browser) uses the SOCKS protocol to specify where the traffic should be sent when it leaves the other end of the ssh tunnel.

• *ssh -D 9001 home* (executed from 'work')

Here SSH will create a SOCKS proxy listening in for connections at local port 9001 and upon receiving a request would route the traffic via SSH channel created between ‘work’ and ‘home’. It is required to configure the browser to point to the SOCKS proxy at port 9001 at localhost.



Many good systems administrators are nervous about security. Here is a list of processes and configurations that you can use to tighten and enhance SSH security with regard to remote host access.

Restrict the root account to console access only:

```
# vi /etc/ssh/sshd_config
PermitRootLogin no
```

Create private-public key pairs using a strong passphrase and password protection for the private key (never generate a password-less key pair or a password-less passphrase key-less login):

```
ssh-keygen -t rsa -b 4096  (Use a higher bit rate for the encryption for more security)
```

Only use SSH Protocol 2:

```
# vi /etc/ssh/sshd_config
Protocol 2
```

Restrict the available interfaces that SSH will listen on and bind to:

```
# vi /etc/ssh/sshd_config
ListenAddress 192.168.100.17
ListenAddress 209.64.100.15
```

Set user policy to enforce strong passwords to protect against brute force, social engineering attempts, and dictionary attacks:

```
# < /dev/urandom tr -dc A-Za-z0-9_ | head -c8
oP0FNAUt[
```

Confine SFTP users to their own home directories by using Chroot SSHD:

```
# vi /etc/ssh/sshd_config ChrootDirectory /data01/home/%u
X11Forwarding no
AllowTcpForwarding no
```

Disable empty passwords:

```
# vi /etc/ssh/sshd_config
PermitEmptyPasswords no
```

Always keep the SSH packages and required libraries up to date on patches:

```
# yum update openssh-server openssh openssh-clients –y
```

SSH supports numerous, diverse methods and techniques for authentication that you can enable or disable. Within the /etc/ssh/sshd_config file, you make these configurations changes by entering the keyword listed for the authentication method followed by yes or no. Here are some of the common configuration changes:

```
# RSAAuthentication yes
# PubkeyAuthentication yes
# RhostsRSAAuthentication no
# HostbasedAuthentication no
# RhostsRSAAuthentication and HostbasedAuthentication PasswordAuthentication yes ChallengeResponseAuthentication no
# KerberosAuthentication no GSSAPIAuthentication yes
```

SSH: alternatives

- **SSH shortcomings**
  - Not designed for slow connections
    - lagging SSH console session.
  - Not designed for connection drops
    - SSH state connection is lost
  - SSH works over TCP
    - No IP address roaming supported.
  - Not designed for large network latencies and round trip times

- **Alternative: MOSH (Mobile Shell)**
  - **Designed for mobile devices and unreliable connections**

Overview

- Network model
- Secure configuration of devices
  - **SSH (Secure Shell)**
- Exchanging keys
- Secure networking protocols
- Firewalls

Besides the lecturers' own material, many third party, often copyrighted, material is reused within this lecture (e.g. in the notes) under the 'fair use' approach, for sake of educational purpose only, and very limited edition. As a consequence, the current slide set presentation usage is restricted, and is falling under usual copyrights usage.

At the end of every lecture, appropriate references to used materials are included.

References

- This work contains content adapted from, amongst other, the following sources (in no particular order)
  - William Stallings, "Cryptography and Network Security, principles and practices", 6th (international) edition, Prentice Hall, 2010;
  - Matt Bishop, "Computer Security: Art and Science", Addison Wesley, Pearson Education, 2003, ISBN-13: 978-0-201-44099-7
  - Lecture slides: "Informatiebeveiliging", Universiteit Gent, Eric Laermans & Thom Dhaene
  - Wikipedia (additional note page descriptions)
  - https://crypto.stackexchange.com
  - http://www.ibm.com/developerworks/aix/library/au-sshsecurity/
  - https://chamibuddhika.wordpress.com/2012/03/21/ssh-tunnelling-explained/