Network Security (NetSec)

TECHNISCHE UNIVERSITÄT DARMSTADT

Summer 2015
Chapter 04: Transport Level Security
Module 03: TLS and Secure Shell (SSH)

Prof. Dr.-Ing. Matthias Hollick

Technische Universität Darmstadt
Secure Mobile Networking Lab - SEEMOO
Department of Computer Science
Center for Advanced Security Research Darmstadt - CASED

Mornewegstr. 32
D-64293 Darmstadt, Germany
Tel.+49 6151 16-70922, Fax. +49 6151 16-70921
http://seemoo.de or http://www.seemoo.tu-darmstadt.de

Prof. Dr.-Ing. Matthias Hollick
matthias.hollick@seemoo.tu-darmstadt.de

CASED

Learning Objectives

Security objectives, mechanisms and limitations on transport layer (or between network layer and application layer)

- Identify the scope of protection as well as the trade-offs involved in securing networks on transport layer
- Understand the fundamental design principles of transport layer security protocols
- Discuss toy and real-world protocols to secure the transport layer
  - TLS (transport layer security) and SSH (secure shell)

- In early module
  - A toy SSL protocol
  - The Secure Socket Layer protocol

Overview of this Module

(1) TLS – yet another transport layer security protocol?
(2) SSH – Secure Shell
(3) Summary and conclusion on transport layer security
(4) Recommended Readings

Chapter 04, Module 03

TLS (Transport Layer Security)

IETF standard RFC 2246 similar to SSLv3 with minor differences

- in record format version number
- uses HMAC for MAC
  - construction for computing cryptographic hash values should be adopted instead of hashing in prefix and suffix mode
- a pseudo-random function expands secrets
  - based on HMAC using SHA-1 or MD5
- has additional alert codes
- some changes in supported ciphers
- changes in certificate types & negotiations
- changes in crypto computations & padding
- “The IETF, realizing that it was bad for the industry to have three similar but incompatible protocols for the same purpose, introduced a forth similar but incompatible protocol—TLS (Transport Layer Security)” – Radia Perlman

TLS vs. SSL Version Numbers

Just one example: Serial Number Woes

- SSLv2 and v3 have 16 bit version numbers, both
- SSLv2 interprets this as a single 16-bit integer, and the official number is 2, e.g. 0x0002. SSLv3 interprets two-byte version numbers as a one byte "major" number and a one byte "minor" (or fractional) number. So the value 0x0002 is interpret by SSL3 as version 0.2, not 2.0.
  - SSLv2 has higher octet set to 0, lower octet set to 2
  - SSLv3 has higher octet set to 3, lower octet set to 0
  - TLSv1 has higher octet set to 3, lower octet set to 1
- Version number field “got moved”
- Problem: an SSLv2 server is likely to misparse the SSLv3 CLIENT HELLO message … as a SSLv768 message …
- … leads to implementation problems

The Transport Layer Security Protocol

SECURE MOBILE NETWORKING

In order to achieve exportability of TLS compliant products, some cipher-suites specify the use of keys with entropy reduced to 40 bit:

- These cipher-suites contain the word “export” in their name
- As the government of the USA changed its policy concerning the export of cryptographic products, this is of less importance today
- The use of these cipher-suites is strongly discouraged, as they offer virtually no data confidentiality protection

Key exchange algorithms:

- DH exchange without or with DSS / RSA signatures
- DH exchange with certified public DH parameters
- RSA based key exchange
- none

Encryption algorithms: IDEA / DES / 3DES / RC2 in CBC, RC4, AES, null

Hash algorithms: MD5, SHA, null

Concerning its protocol functions, TLS is essentially the same like SSL

Secure Shell (SSH)

Image Source: thelinuxcauldron.wordpress.com/2009/03/18/13

The Secure Shell Protocol

Secure Shell (SSH) Version 1 was originally developed by Tatu Ylönen at the Helsinki University of Finland

- As the author also provided a free implementation with source code, the protocol found widespread use in the Internet
- Later on, the development of SSH was commercialized by the author
- Nevertheless, free versions are still available with the most widely deployed version being OpenSSH
- In 1997 a version 2.0 specification of SSH was submitted to the IETF and has been refined in a series of Internet Drafts since

SSH was originally designed to provide a secure (yet lightweight) replacement for the Unix r-tools (rlogin, rsh, rcp, and rdist), thus it represents an application or session-layer protocol

However, as SSH also includes a generic transport layer security protocol and offers tunneling capabilities, it is discussed in this chapter as a transport layer security protocol

SSH Version 2

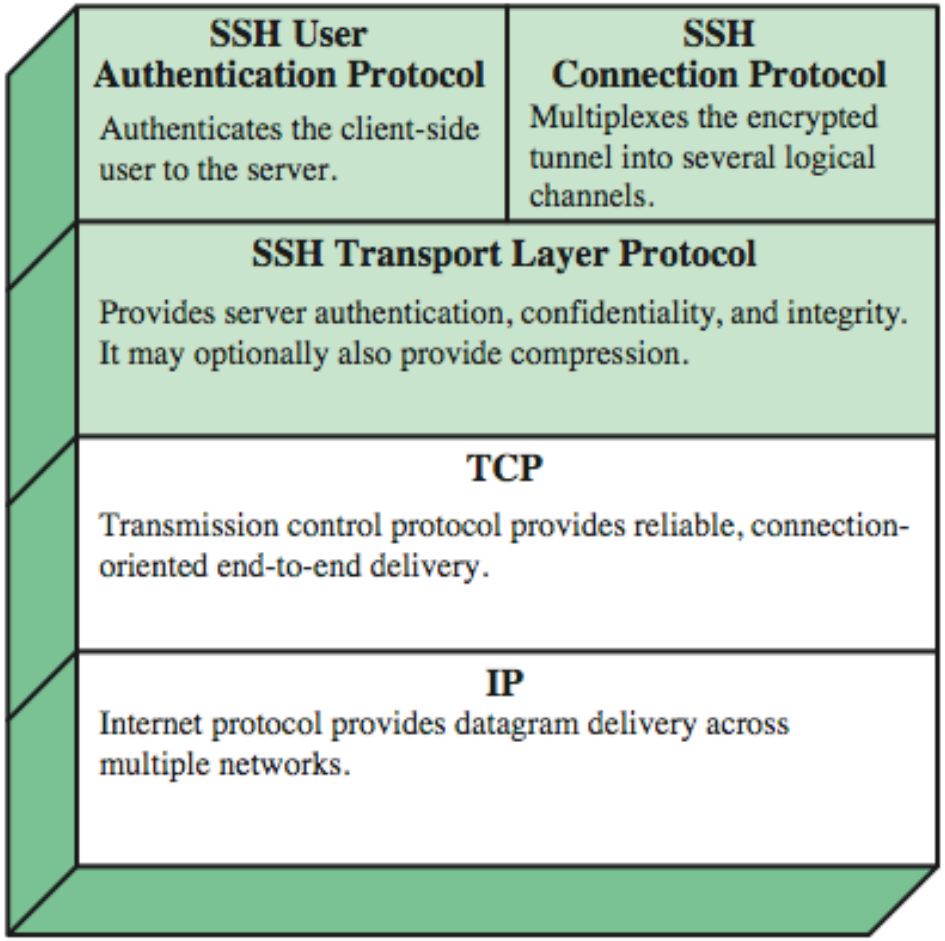
SSH Version 2 is specified in four separate documents (RFCs 4250 through 4254)

- SSH Protocol Architecture [YKS01a]
- SSH Transport Layer Protocol [YKS01b]
- SSH Authentication Protocol [YKS01c]
- SSH Connection Protocol [YKS01d]

SSH Architecture:

- SSH follows a client-server approach
- Every SSH server has at least one host key
- SSH version 2 offers two different trust models:
  - Every client has a local database that associates each host name with the corresponding public host key
  - The hostname to public key association is certified by a CA and every client knows the public key of the CA
- The protocol allows full negotiation of encryption, integrity, key exchange, compression, and public key algorithms and formats

SSH Protocol Stack



SSH Transport Layer Protocol

server authentication occurs at transport layer, based on server/ host key pair(s)

- server authentication requires clients to know host keys in advance

packet exchange

- establish TCP connection
- can then exchange data
  - identification string exchange, algorithm negotiation, key exchange, end of key exchange, service request
- using specified packet format

SSH User Authentication Protocol

SECURE MOBILE NETWORKING

authenticates client to server

three message types:

- SSH_MSG_USERAUTH_REQUEST
- SSH_MSG_USERAUTH_FAILURE
- SSH_MSG_USERAUTH_SUCCESS

authentication methods used

- public-key, password, host-based

SSH Connection Protocol

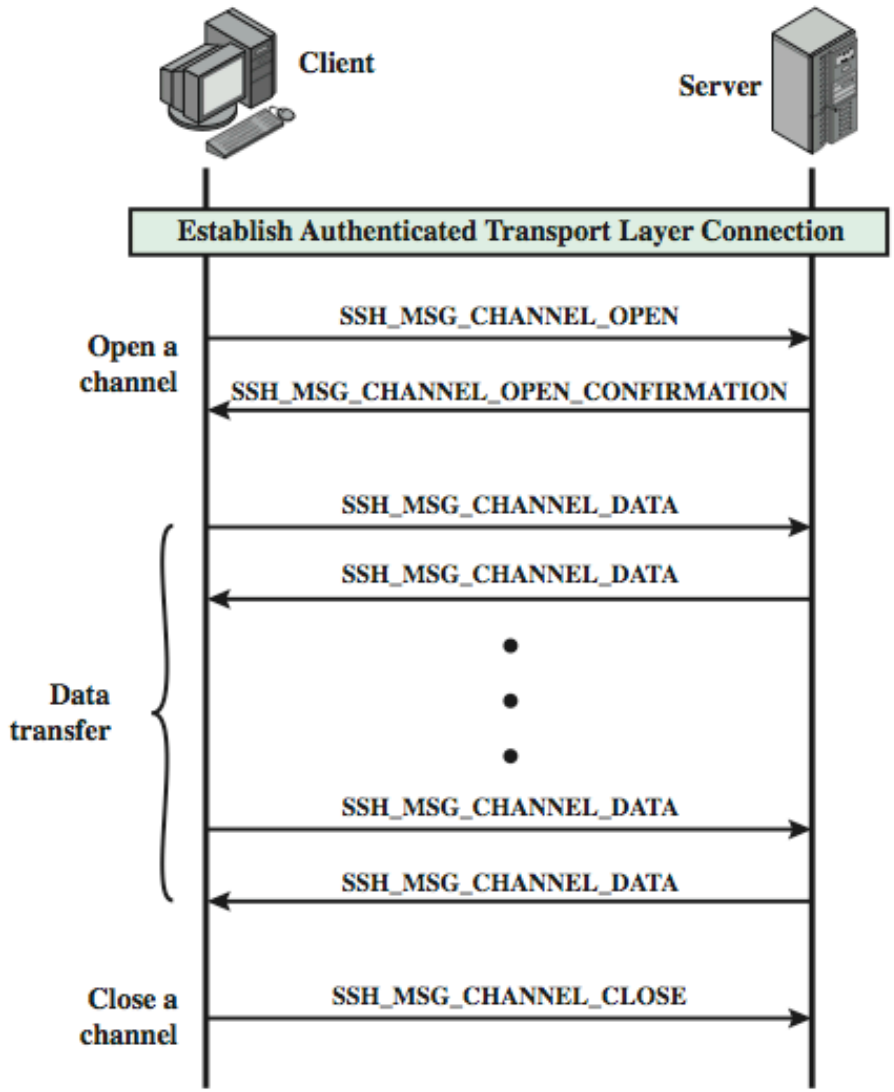
runs on SSH Transport Layer Protocol

assumes secure authentication connection

used for multiple logical channels

- SSH communications use separate channels
- either side can open with unique id number
- flow controlled
- have three stages:
  - opening a channel, data transfer, closing a channel
- four types:
  - session, x11, forwarded-tcpip, direct-tcpip.

SSH Connection Protocol Exchange



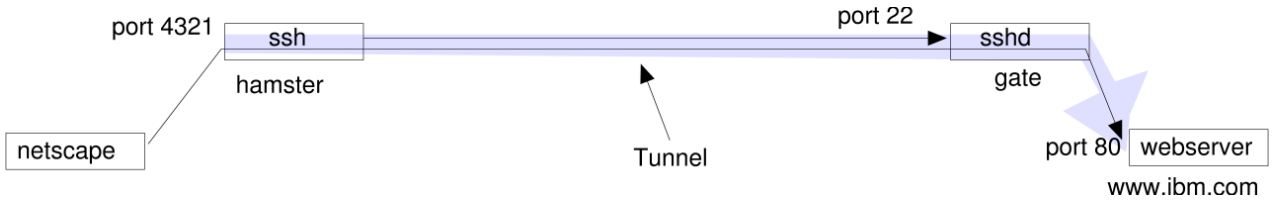
Port Forwarding

convert insecure TCP connection into a secure SSH connection

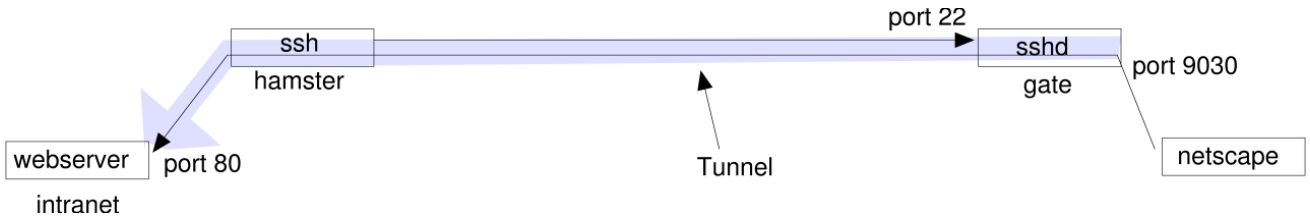
- SSH Transport Layer Protocol establishes a TCP connection between SSH client & server
- client traffic redirected to local SSH, travels via tunnel, then remote SSH delivers to server

supports two types of port forwarding

- local forwarding (outgoing tunnel)
  - "hijacks" selected traffic and sends it to remote port



- remote forwarding (incoming tunnel)
  - client acts for server, i.e. traffic arriving at the server at a certain port is forwarded to some port at client



Summary & Conclusion Transport Layer Security

Both SSL/TLS and SSH are suited to secure Internet communications in (above) the transport layer:

- Both security protocols operate upon and require a reliable transport service, e.g. TCP
- Up to now, securing datagram-oriented transport protocols like UDP is hardly deployed http://tools.ietf.org/html/rfc4347
- Transport layer security protocols offer true end-to-end protection for user data exchanged between application processes
- Furthermore, they may interwork with packet filtering of today's firewalls
- But, protocol header fields of lower layer protocols can not be protected this way, so they offer no countermeasures to threats to the network infrastructure itself

- What is the main difference SSH vs. SSL/TLS?

Continous Testing …

SSL (TLS), SSH

- All had security issues in the original version
- E.g. MITM with SSH downgrade attack or SSLv2 cipher-suite downgrade attack
- Problems due to weak random number generator implementations
- Continous security testing is crucial …

Bricks Test Storm Resistance of Phone Wires

To determine how well telephone wires will carry the extra weight of ice during snow and sleet storms, engineers string bricks along experimental open-wire lines at the Bell Laboratories field station in Chester, N. J. It has been found that an accumulation of ice one inch in radial thickness adds about twenty-two ounces to a foot of wire, or 200 pounds on a 150-foot span.

Vitamin C Locked in Milk by De-aerating

To prevent escape of its vitamin C content, a process of de-aerating pasteurized milk has been developed by three Cornell University professors. Vitamin C in a quart of ordinary pasteurized milk is equal to that contained in the juice of a slice of orange, but by taking air out of the milk it is possible for a quart to retain a vitamin C content equal to the juice of a whole orange. The de-aerating process is not expensive.

Outdoor laboratory crew hanging bricks on telephone wires in test to ascertain amount of ice the lines can endure in storm

Image Source: modernmechanix.com

Some of the Pitfalls in SSL

Cyphersuite: who makes the decision

- SSLv2: Alice suggests ciphers, Bob returns subset that he supports, Alice choses
- What would you change?

Negotiating compression:

- TLS 1.0 originally had exactly one option: NULL (the standard reads: “TLS defines one standard compression method which specifies that data exchanged via the record protocol will not be compressed.”)
- What is the reason for this?

Attacks

- Downgrade attack in SSLv2 (active attacker removes cipher suites)
- Truncation attack in SSLv2 (v2 used the TCP connection close)

More Details ?

If you are interested in details:

- D. Wagner, Analysis of the SSL 3.0 Protocol, USENIX 1996 http://www.usenix.org/publications/library/proceedings/ec96/full_papers/wagner/wagner.pdf

TLS

- [RFC2246] T. Dierks, C. Allen. The TLS Protocol Version 1.0. RFC 2246, 1999.

SSH

- [YKS01a] T. Ylonen, T. Kivinen, M. Saarinen, T. Rinne, S. Lehtinen. SSH Protocol Architecture. Internet Draft (work in progress), draft-ietf-secsh-architecture-09.txt, 2001.
- [YKS01b] T. Ylonen, T. Kivinen, M. Saarinen, T. Rinne, S. Lehtinen. SSH Transport Layer Protocol. Internet Draft (work in progress), draft-ietf-secsh-transport-09.txt, 2001.
- [YKS01c] T. Ylonen, T. Kivinen, M. Saarinen, T. Rinne, S. Lehtinen. SSH Authentication Protocol. Internet Draft (work in progress), draft-ietf-secsh-userauth-11.txt, 2001.
- [YKS01d] T. Ylonen, T. Kivinen, M. Saarinen, T. Rinne, S. Lehtinen. SSH Connection Protocol. Internet Draft (work in progress), draft-ietf-secsh-connect-11.txt, 2001.

SSL Threat Landscape

Never forget: Security is a Process! (schneier.com)

A SSL Threat Model can be found at

- http://blog.ivanristic.com/2009/09/ssl-threat-model.html

SSL Labs: State of Affairs of SSL (plenty of practical pitfalls)

- https://www.ssllabs.com/
- https://www.trustworthyinternet.org/ssl-pulse/

If time permits:

- https://community.qualys.com/servlet/JiveServlet/download/38-9096/SSL_and_Browsers-The_Pillars_of_Broken_Security.pdf
- I would call this mandatory reading! You should definitely take a look!

Acks & Recommended Reading

Selected slides of this chapter courtesy of

- Keith Ross with changes of myself incorporated
- Some other slides courtesy of G. Schäfer (TU Ilmenau) with changes of J. Schmitt (TU Kaiserslautern) and myself incorporated
- Yet some other slides courtesy of R. Perlman, K. Ross, Y. Chen, W. Stallings (L. Brown); changes of myself incorporated

Recommended reading

- [KaPeSp2002] Charlie Kaufman, Radia Perlman, Mike Speciner: Network Security – Private Communication in a Public World, 2nd Edition, Prentice Hall, 2002, ISBN: 978-0-13-046019-6
- [Stallings2014] William Stallings, Network Security Essentials, 4th Edition, Prentice Hall, 2014, ISBN: 978-0-136-10805-4
- [Schäfer2003] G. Schäfer. Netzsicherheit - Algorithmische Grundlagen und Protokolle. dpunkt.verlag, 2003.

Copyright Notice

Contact

Prof. Dr.-Ing. Matthias Hollick
Department of Computer Science

SEEMOO
Mornewegstr. 32
64293 Darmstadt/Germany
matthias.hollick@seemoo.tu-darmstadt.de

Phone +49 6151 16-70920
Fax +49 6151 16-70921
www.seemoo.tu-darmstadt.de

SSH Transport Protocol

The SSH Transport Protocol runs on top of a reliable transport protocol (usually TCP)

It provides the following services:

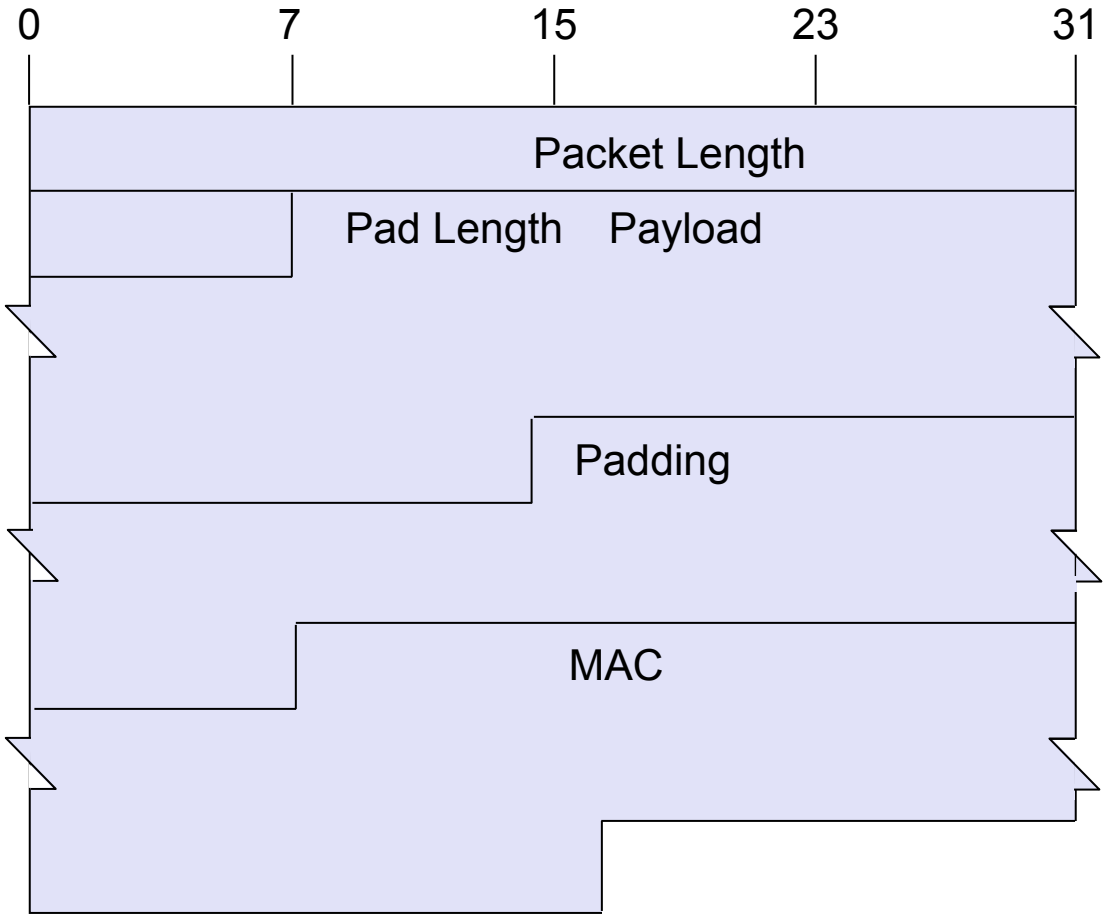
- Encryption of user data
- Data origin authentication (integrity)
- Server authentication (host authentication only)
- Compression of user data prior to encryption

Supported algorithms:

- Encryption:
  - 3DES, Blowfish, Twofish, AES, Serpent, IDEA, CAST in CBC
  - Arcfour (“believed” to be compatible with the “unpublished” RC4)
  - none (not recommended)
- Integrity: HMAC with MD5 or SHA-1, none (not recommended)
- Key exchange: Diffie-Hellman with SHA-1 and one pre-defined group
- Public key: RSA, DSS
- Compression: none, zlib (see RFCs 1950, 1951)

SSH Transport Protocol Packet Format (1)

The packet format is not 32-bit-word aligned



SSH Transport Protocol Packet Format (2)

Packet fields:

- *Packet length:* the length of the packet itself, not including this length field and the MAC
- *Padding length:* length of the padding field, must be between four and 255
- *Payload:* the actual payload of the packet, if compression is negotiated this field is compressed
- *Padding:* this field consists of randomly chosen octets to fill up the payload to an integer multiple of 8 or the block size of the encryption algorithm, whichever is larger
- *MAC:* if message authentication has been negotiated this contains the MAC over the entire packet without the MAC field itself, if the packet is to be encrypted the MAC is computed prior to encryption as follows:
  - *MAC = HMAC(shared_secret, seq_number || unencrypted_packet)* with *seq_number* denoting a 32-bit sequence number for every packet

Encryption: if encryption is negotiated, the entire packet without the MAC is encrypted after MAC computation

SSH Negotiation, Key Exchange & Server Authentication (1)

SECURE MOBILE NETWORKING

Algorithm Negotiation:

- Each entity sends a packet (referred to as *kexinit*) with a specification of methods it support, in the order of preference
- Both entities iterate over the list of the client and chose the first algorithm that is also supported by the server
- This method is used to negotiate: server-host-key algorithm (~ server authentication), as well as encryption, MAC, and compression algorithm
- Additionally, either entity may attach a key exchange packet according to a guess of the preferred key exchange algorithm of the other entity
- If a guess is right, the corresponding key exchange packet is accepted as the first key exchange packet of the other entity
- Wrong guesses are ignored and new key exchange packets are sent after algorithm negotiation

For key exchange [YKS01b] defines only one method:

- Diffie-Hellman with SHA-1 and a predefined group
- $p = 2^{1024} - 2^{960} - 1 + (2^{64} \times \lfloor 2^{894} \times \pi + 129093 \rfloor)$; g = 2; order = (p - 1) / 2

SSH Negotiation, Key Exchange & Server Authentication (2)

If key exchange is realized with the pre-defined DH group:

- The client chooses a random number x, computes $e = g^x \bmod p$ and sends e to the server
- The server chooses a random number y, computes $f = g^y \bmod p$
- Upon reception of e, the server further computes $K = e^y \bmod p$ and a hash value $h = Hash(version_C, version_S, kexinit_C, kexinit_S, +K_S, e, f, K)$ with *version* and *kexinit* denoting the client's and server's version information and initial algorithm negotiation messages
- The server signs h with its private host key $-K_S$ and sends to the client a message containing $(+K_S, f, s)$
- Upon reception the client checks the host key $+K_S$, computes $K = f^x \bmod p$ as well as the hash value h and then checks the signature s over h

After performing these checks, the client can be sure that he has in fact negotiated a secret K with the host that knows $-K_S$

However, the server host can not deduce anything about the client's authenticity, for this purpose the SSH authentication protocol is used

SSH Session Key Derivation

The key exchange method allows to establish a shared secret *K* and the hash value *h* which are used to derived the SSH session keys:

- The hash *h* of the initial key exchange is also taken as the *session_id*
- $IV_{Client2Server}$ = *Hash(K, h, "A", session_id)* // initialization vector
- $IV_{Server2Client}$ = *Hash(K, h, "B", session_id)* // initialization vector
- $EK_{Client2Server}$ = *Hash(K, h, "C", session_id)* // encryption key
- $EK_{Server2Client}$ = *Hash(K, h, "D", session_id)* // encryption key
- $IK_{Client2Server}$ = *Hash(K, h, "E", session_id)* // integrity key
- $IK_{Server2Client}$ = *Hash(K, h, "F", session_id)* // integrity key

Key data is taken from the beginning of the hash output

If more key bits are needed than produced by the hash function:

- *K1* = *Hash(K, h, x, session_id)* // x = "A", "B", etc.
- *K2* = *Hash(K, h, K1)*
- *K2* = *Hash(K, h, K1, K2)*
- *XK* = *K1 || K2 || ...*

SSH Authentication Protocol

The SSH authentication protocol serves to verify the client's identity and it is intended to be run over the SSH transport protocol

The protocol per default supports the following authentication methods:

- *Public key:* the user generates and sends a signature with a per user public key to the server
  Client → Server: E(-K_{User}, (session_id, 50, $Name_{User}$, Service, "publickey", True, PublicKeyAlgorithmName, +K_{User}))
- *Password:* transmission of a per user password in the encrypted SSH session (the password is presented in clear to the server but transmitted with SSH transport protocol encryption)
- *Host-based:* analogous to public key but with with per host public key
- *None:* used to query the server for supported methods and if no authentication is required (server directly responds with success message)

If the client's authentication message is successfully checked, the server responds with a *ssh_msg_userauth_success* message

SSH Connection Protocol (1)

The SSH connection protocol runs on top of the SSH transport protocol and provides the following services:

- Interactive login sessions
- Remote execution of commands
- Forwarded TCP/IP connections
- Forwarded X11 connections

For each of the above services one or more *"channels"* are established, and all channels are multiplexed into a single encrypted and integrity protected SSH transport protocol connection:

- Either side may request to open a channel and channels are identified by numbers at the sender and receiver
- Channels are typed, e.g. *"session"*, *"x11"*, *"forwarded-tcpip"*, *"direct-tcpip"*...
- Channels are flow-controlled by a window mechanism and no data may be sent via a channel before "window space" is available

SSH Connection Protocol (2)

Opening a channel:

- Either side may send the message *ssh_msg_channel_open* signaled with message code 90 and the following parameters:
  - *channel type:* is of data type string, e.g. *“session”, “x11”,* etc.
  - *sender channel:* is a local identifier of type uint32 and chosen by the requestor of this channel
  - *initial window size:* is of type uint32 and specifies how many bytes may be send to the initiator before the window needs to be advanced
  - *maximum packet size:* is of type uint32 and defines the maximum packet size the initiator is willing to accept for this channel
  - further parameters depending on the type of the channel may follow
- If receiver of this message does not want to accept the channel request, it answers with the message *ssh_msg_channel_open_failure* (code 92):
  - *recipient channel:* the id given in the open request by the sender
  - *reason code:* is of type uint32 and signals the reason for the rejection
  - *additional textual information:* is of type string
  - *language tag:* is of type string and according to RFC 1766

SSH Connection Protocol (3)

Opening a channel (cont.):

- If receiver of this message wants to accept the channel request it answers with the message *ssh_msg_channel_open_confirmation* (code 91) and parameters:
  - *recipient channel:* the id given in the open request by the sender
  - *sender channel:* the id given to the channel by the responder
  - *initial window size:* is of type uint32 and specifies how many bytes may be send to the responder before the window needs to be advanced
  - *maximum packet size:* is of type uint32 and defines the maximum packet size the responder is willing to accept for this channel
  - further parameters depending on the channel type may follow

Once a channel is opened, the following actions are possible:

- Data transfer (however, the receiving side should know “what to do with the data” which may require further prior negotiation)
- Channel type specific requests
- Closure of the channel

SSH Connection Protocol (4)

For data transfer the following messages are defined:

- *ssh_msg_channel_data:* with the two parameters *recipient channel, data*
- *ssh_msg_channel_extended_data:* allows to additionally specify a *data type code* and is useful to signal errors, e.g. of interactive shells
- *ssh_msg_channel_window_adjust:* allows to advance the flow control window of the *recipient channel* by the specified number of *bytes to add*

Closing of channels:

- When a peer entity will not longer send data to a channel it should signal this to the other side with the message *ssh_msg_channel_eof*
- When either side wishes to terminate a channel it sends the message *ssh_msg_channel_close* with parameter *recipient channel*
- Upon reception of the message *ssh_msg_channel_close* a peer entity must answer with a similar message unless it has already requested closure of this channel
- After both receiving and sending of the *ssh_msg_channel_close* message for a specific channel, the *id* of that channel may be re-used

SSH Connection Protocol (5)

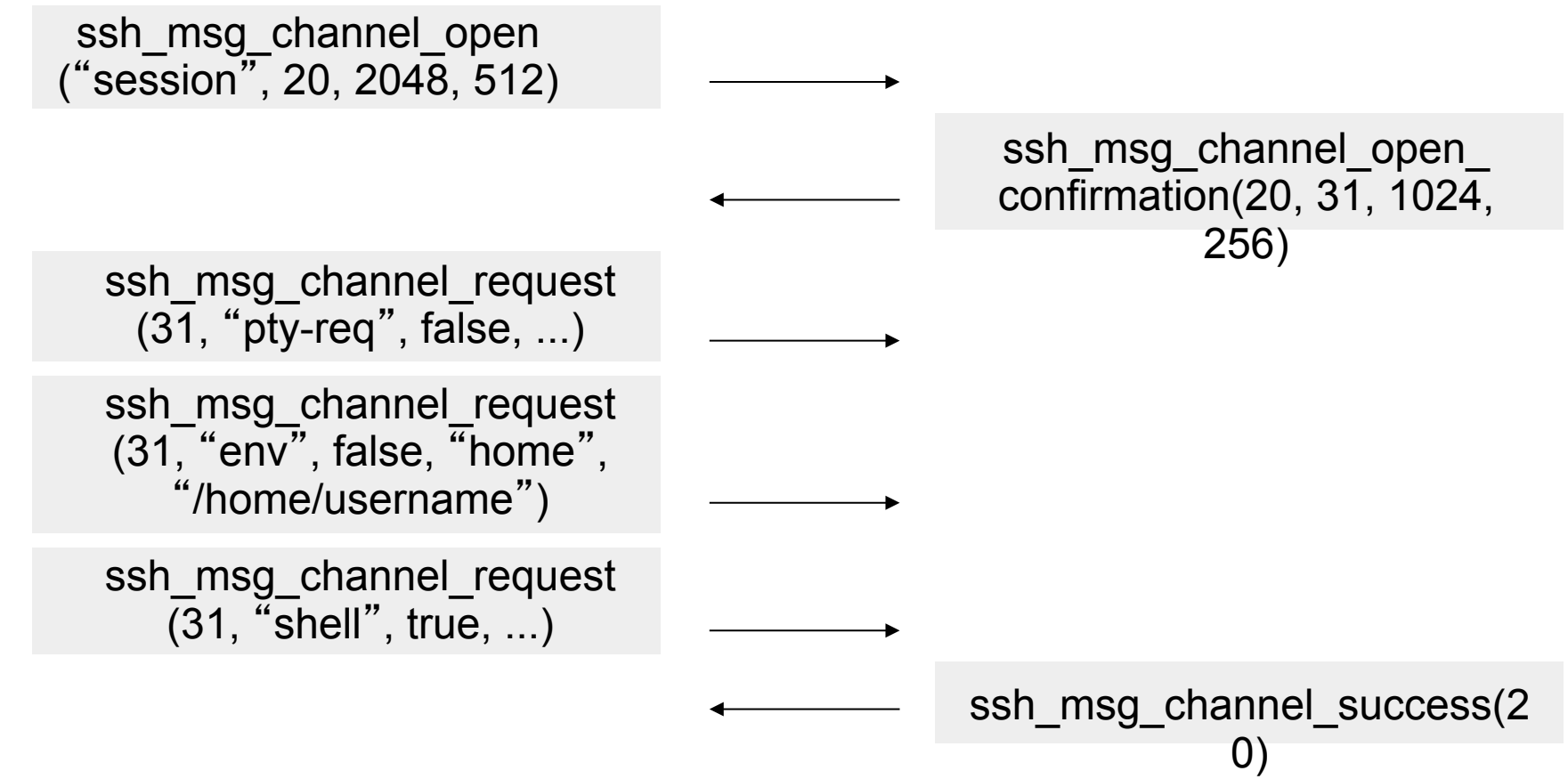
Channel type specific requests allow to demand for specific properties of a channel, e.g. such that the receiving side knows how to process data send via this channel, and are signaled with:

- *ssh_msg_channel_request:* with the parameters *recipient channel, request type* (string), *want reply* (bool) and further request specific parameters
- *ssh_msg_channel_success:* with the parameter *recipient channel*
- *ssh_msg_channel_failure:* with the parameter *recipient channel*

Example 1 – requesting an interactive session and starting a shell in it:

- First, a channel of type *"session"* is opened
- A pseudo-terminal is requested by sending an *ssh_msg_channel_request* message with the request type set to *"pty-req"*
- If needed, environment variables can be set by issuing *ssh_msg_channel_request* messages with request type set to *"env"*
- Then, the start of a shell process is demanded via an *ssh_msg_channel_request* message with the request type set to *"shell"* (usually this results in the start of the default shell for the user as defined in /etc/passwd)

Requesting an Interactive Session and Starting a Shell in it



[Use data exchange takes place from now on...]

SSH Connection Protocol (6)

Example 2 – requesting X11 forwarding:

- First, a channel of type *"session"* is opened
- X11 forwarding is requested by sending an *ssh_msg_channel_request* message with request type set to "x11-req"
- If later on an application is started on the server that needs to access the terminal of the client machine (the X11-server running on the client machine), a new channel is opened via *ssh_msg_channel_open* with the channel type set to *"x11"* and the originator IP address and port number as additional parameters

SSH Connection Protocol (7)

Example 3 – setting up TCP/IP port forwarding:

- A party needs not to explicitly request port forwarding from its own end to the other direction, however, if it wants to have connections to a port on the other side forwarded to its own side, it must explicitly request this via an ssh_msg_global_request message with the parameters *"tcpip-forward", want-reply, address to bind* ("0.0.0.0" for every source address), and *port number to bind* (this request is usually sent by the client)
- When a connection comes for a port for which forwarding has been requested, a new channel is opened via *ssh_msg_channel_open* with the type set to *"forwarded-tcpip"* and the addresses of the port that was connected as well as of the original source port as parameters (this message is usually sent by the server)
- When a connection comes to a (client) port that is locally set to be forwarded, a new channel is requested with the type set to *"direct-tcpip"* and the following address information specified in additional parameters:
  - *host to connect, port to connect:* address to which the recipient should connect this channel
  - *originator IP address, originator port:* source address of the connection