Liberté • Égalité • Fraternité
RÉPUBLIQUE FRANÇAISE

PREMIER MINISTRE

Secrétariat général
de la défense
et de la sécurité nationale

*Agence nationale de la sécurité
des systèmes d'information*

Paris, August 17, 2015

N° DAT-NT-007-EN/ANSSI/SDE/NP

Number of pages
(including this page): 21

Technical report

(Open)SSH secure use recommendations

Targeted audience

Developers	
Administrators	✓
IT security managers	✓
IT managers	
Users	

Document Information

Disclaimer

This document, written by the ANSSI, presents the **"(Open)SSH secure use recommendations"**. It is freely available at www.ssi.gouv.fr/nt-ssh. It is an original creation from the ANSSI and it is placed under the "Open Licence" published by the Etalab mission (www.etalab.gouv.fr). Consequently, its diffusion is unlimited and unrestricted.

This document is a courtesy translation of the initial French document **"Recommandations pour un usage sécurisé d'(Open)SSH"**, available at www.ssi.gouv.fr/nt-ssh. In case of conflicts between these two documents, the latter is considered as the only reference.

These recommendations are provided as is and are related to threats known at the publication time. Considering the information systems diversity, the ANSSI cannot guarantee direct application of these recommendations on targeted information systems. Applying the following recommendations shall be, at first, validated by IT administrators and/or IT security managers.

Document contributors

Contributors	Written by	Approved by	Date
Cisco[1], DAT	DAT	SDE	August 17, 2015

Document changelog

Version	Date	Changelog
based on 1.3 – french	August 17, 2015	Translation

Contact information

Contact	Address	Email	Phone
Bureau Communication de l'ANSSI	51 bd de La Tour-Maubourg 75700 Paris Cedex 07 SP	communication@ssi.gouv.fr	01 71 75 84 04

1. This document is based on a courtesy translation provided by Cisco Systems Inc.

Contents

1 Introduction

1.1 Some historical reminders

The emergence of the first Unix and communicating information systems also involved the emergence of protocol stacks oriented towards data exchange between machines, such as FTP, TELNET, or RSH.

Although still widely used, these protocols were not designed for security; they have poor features when it comes to authenticating the source or sender, or ensuring the flow integrity and confidentiality.

Their use has even become a problem from a filtering point of view. For instance, FTP requires a dynamic port opening on a NAT gateway.

For these reasons, the need for a secure application protocol aiming at replacing these software components has rapidly given birth to SSH.

1.2 Goals of this document

Commonly used for remote administration, transferring files, forwarding and encapsulating sensitive flows, OpenSSH has become a key element of a large number of IT systems.

Therefore, it is critical to control its configuration, harden its installation and apply strict hygiene rules for its operation. The recommendations presented in this document cover all these different aspects.

1.3 Targeted audience

This technical document is intended for network and system administrators and integrators who care about properly installing and administrating an infrastructure with the help of the SSH protocol and its reference implementation: OpenSSH.

2 Presentation of SSH

SSH, or Secure SHell, is an application layer protocol (layer 7 of the OSI model), which aims at correcting known deficiencies in FTP, RSH, RCP and TELNET protocols, through 3 sub-protocols:

- SSH-USERAUTH, a client-side authentication protocol – RFC 4252;
- SSH-TRANS, a secure transfer protocol that allows server authentication and establishment of a secure communication channel (confidentiality and integrity) – RFC 4253;
- SSH-CONNECT, a connection protocol that allows communication channels multiplexing (data and commands) – RFC 4254.

2.1 SSH Protocol

SSH is standardized by a set of RFC (4250-4254) specifying its communication protocols and the cryptographic mechanisms the protocol must support.

SSH now exists in 2 versions: SSHv1 and SSHv2. SSHv1 has structural vulnerabilities that have been corrected in the next version. Version 1 of the protocol is now obsolete.

R1

Only version 2 of the SSH protocol shall be authorized.

SSH is mainly a communication protocol. It is neither a Unix shell, nor a terminal, nor a command interpreter.

OpenSSH is the most commonly encountered implementation of the SSH protocol. More than a protocol, it is a toolbox offering a large number of features.

Under OpenSSH, mandatory use of version 2 of the protocol is enforced by the following directive in `sshd_config`:

```
Protocol 2
```

2.2 Presentation of OpenSSH

OpenSSH is developed and maintained by the OpenBSD project. It is, to date, the reference implementation of the SSH protocol found on a large number of systems, including servers (Unix, GNU/Linux, Windows), client machines and network devices.

This software suite includes many tools:

- the server, `sshd`.
- several clients, depending of the purpose:
  - remote shell connections: `ssh`;
  - file transfers and downloads: `scp`, `sftp`;
- a key generation tool, `ssh-keygen`;
- a keyring service, `ssh-agent` and `ssh-add`;
- a utility for gathering the public SSH host keys of a number of hosts, `ssh-keyscan`.

3 Use cases

3.1 Remote shell administration

Remote shell administration is the most frequent use of SSH. It consists in connecting to a remote computer and opening a remote shell session after successful authentication.

The obvious advantage of SSH is its security. Whereas telnet provides neither server authentication nor an encrypted and authenticated channel, SSH does so, provided that some simple hygiene rules are followed.

R2

SSH shall be used instead of historical protocols (TELNET, RSH, RLOGIN) for remote shell access.

R3

TELNET, RSH and RLOGIN remote access servers shall be uninstalled from the system.

3.2 File transfers and downloads

The second most common use of SSH is file transfer, both for uploading (client to server) and downloading (server to client).

SSH provides two mechanisms to this end: SCP and SFTP. SFTP is more elaborate than SCP as it allows navigation in a tree of files, whereas SCP merely allows data transfer. In both cases, the security is mainly based on SSH, which provides the communication channel.

From a network point of view, the use of SCP/SFTP simplifies filtering rules setup throughout the network. SSH allows to multiplex the data channel and the control channel in the same SSH connection, so there is no need for a dynamic port opening rule, as required with FTP.

R4

SCP or SFTP shall be used instead of historical protocols (RCP, FTP) for file transfers.

3.3 Flow forwarding

SSH Flow forwarding is also commonly used. It consists in encapsulating TCP/IP flows directly into the SSH tunnel, to allow (among others things) the secure transport of a non-secured protocol, or to provide access to services protected by a gateway.

These forwardings can be set up on the SSH client (Figure 1) or server (Figure 2).

Figure 1: Diagram of a flow forwarding on the SSH client

Figure 2: Diagram of a flow forwarding on the SSH server

R5

The implementation of SSH tunnels shall only be applied to protocols that do not provide robust security mechanisms and that can benefit from it (for example: X11, VNC).
This recommendation does not exempt from using additional low level security protocols, such as IPsec [2].

4 Best practices for using OpenSSH

The following recommendations apply to OpenSSH tools and services.

4.1 Cryptography

4.1.1 Authentication

SSH heavily relies on asymmetric cryptography for authentication.
Failure to ensure the server authentication may result in numerous security impacts:

- risk of data theft through server impersonation (impossibility to verify the identity of the server);
- exposure to "man in the middle" attacks allowing to retrieve all data exchanged (keystrokes, displayed elements, logins, passwords, viewed or edited files...).

R6

The server authenticity shall always be checked prior to access. This is achieved through preliminary machine authentication by checking the server public key fingerprint, or by verifying the server certificate.

With OpenSSH, this control can be achieved by the client in different ways:

- ensure the SSH public key fingerprint of the server is the correct one (**previously** obtained by using `ssh-keygen -l`);
- add the SSH public key of the server in the `known_hosts` file manually;
- verify the signature the SSH certificate presented by the server with a trusted Certification Authority (see section 4.5).

Explicit validation of the host key by the user can be specified using the `StrictHostKeyChecking` attribute in `ssh_config`:

```
StrictHostKeyChecking  ask
```

2. Refer to "IPsec Security Recommendations" available at `www.ssi.gouv.fr` for additional information.

OpenSSH applies, by default, a *Trust On First Use* (TOFU) security model: during the first connection, if the client cannot authenticate the remote host, `ssh` requests the user to verify the server key fingerprint. If the user validates the fingerprint, `ssh` registers the key into the `known_hosts` file to allow automatic validation for the following connections.

4.1.2 Key Generation - sizes and algorithms

SSH can handle several cryptographic algorithms and key sizes. Some of them do not meet the security requirements detailed in the RGS [3] security criteria.

In practice, the SSH "host" keys (used to authenticate an SSH server) are rarely renewed. It is, therefore, important to choose a key long enough from the beginning.

The DSA implementation in OpenSSH supports key up to 1024 bits long. Such keys are now considered insecure and their use is strongly discouraged in the RGS.

R7

The use of DSA keys is not recommended.

In order to forbid the use of DSA keys:
- on the `ssh` client: delete `~/.ssh/id_dsa` and `~/.ssh/id_dsa.pub` files;
- on the `sshd` server: comment out `HostKey` lines pointing to a DSA key (like `/etc/ssh/ssh_host_dsa_key`).

The following recommendations apply for a 3 years lifetime use [4].

R8

The minimum key size shall be 2048 bits for RSA.

R9

The minimum key size shall be 256 bits for ECDSA.

These sizes follow the directives given in chapter 2.2 of the RGS, Appendix B1, *Asymmetric Cryptography*.

Keys meeting these requirements may be generated using the following commands:

```
ssh-keygen -t rsa   -b 2048 -f <RSA key file>
ssh-keygen -t ecdsa -b 256  -f <ECDSA key file>
```

`ssh-keygen` will generate two files: the private key and the public key (with a `.pub` extension).

These files can then be used for host authentication (`HostKey` attribute of `sshd`), or user authentication key (`IdentityFile` attribute of `ssh`).

3. The RGS is a security baseline from the French government which defines security criteria for online government services, local authorities and public establishments

4. The RGS limits the lifetime of application services digital certificates to 3 years in its Appendix A.

R10

ECDSA keys should be favoured over RSA keys when supported by SSH clients and servers.

4.1.3 Key generation and Random Number Generator (RNG) quality

Key quality is an important robustness factor, directly linked to the RNG used during the generation process. Providing a proper RNG is thus essential. Several equipments and computers, including basic embedded components or virtual machines lack such a proper generator.

R11

Keys should be generated in a context where the RNG is reliable, or at least in an environment where enough entropy has been accumulated.

This recommendation is all the more important since it is common to generate the `sshd` host keys by script upon first service startup. A mechanism aimed at changing the generated keys shall be proposed and initial keys shall be replaced.

R12

Some rules can ensure that the entropy pool is properly filled:
- keys must be generated on a physical equipment;
- system must have several **independent** sources of entropy;
- key generation shall occur only after a long period of activity (several minutes or even hours)

As virtual machines mainly rely on virtual devices, their interrupts are also virtualized. Since they can be used as a source of entropy, virtualizing them leads to a certain determinism in their triggering, therefore decreasing the resulting entropy.

Using independent sources ensures that the bias of one of the sources can not affect the others.

For obvious reasons, newly started systems cannot have a sufficient entropy accumulation. It is therefore recommended to generate sensitive material once a sufficient period of activity has elapsed.

When a PKI is available in the IT system, OpenSSH compatible keys can be generated as long as the private key format is handled by OpenSSL.

`ssh-keygen` can handle private keys generated by openSSL and produce the corresponding public keys:

```
# Example: Generation of a password protected private key
openssl genrsa -aes128 -passout stdin -rand /dev/urandom 2048 > <private key>

# Importing the private key in ssh-keygen in order to obtain
# an OpenSSH compatible public key
ssh-keygen -y -f <private key> > <public key>
```

Since version 6.1, OpenSSH can handle various types of public key encoding. Public keys can now directly be imported from X.509 certificates without knowledge of the corresponding private keys and PKCS#12 files can even be used as an authentication factor.

Public keys are imported using `ssh-keygen`:

```
# Converting a PEM encoded public key into an OpenSSH public key
ssh-keygen -i -m PKCS8 -f <public key>

# Retrieving the public key of an X.509 certificate and importing it into OpenSSH
openssl x509 -pubkey -noout -in <certificate> | ssh-keygen -i \
    -m PKCS8 -f /dev/stdin

# Retrieving the private key in order to obtain the corresponding
# OpenSSH public key from a PKCS#12 file
openssl pkcs12 -in <PKCS#12 file> -nocerts -aes128 > <private key>
openssl pkcs12 -in <PKCS#12 file> -nocerts -nodes | ssh-keygen -y \
    -f /dev/stdin > <public key>
```

4.1.4 Access control and keys disclosure

SSH authentication keys can be grouped following two different roles:
- those used for user authentication;
- those used for host/server authentication.

The authentication service relies on asymmetric cryptography. As such, the following guidelines apply:

R13

The private key should only be known by the entity who needs to prove its identity to a third party and possibly to a trusted authority. This private key should be properly protected in order to avoid its disclosure to any unauthorized person.

The protective measures will slightly vary whether the key authenticates a host/server or a user.

An host authentication private key shall only be readable by the `sshd` service. Under Unix/Linux, only `root` is granted such a privilege :

```
-rw------- root:root /etc/ssh/ssh_host_rsa_key
-rw------- root:root /etc/ssh/ssh_host_ecdsa_key
```

A user authentication private key should only be readable by the corresponding user. Under Unix/Linux, this means that:
- the private key is only readable by the owner;
- the private key is password protected. Such a password is only known by the user (see `-p` option of `ssh-keygen`).

The risk of fraudulent use of a user authentication private key is non-negligible, the text file used to store this private key shall therefore be encrypted in order to ensure its protection. In case of loss

or theft, this security measure grants administration teams enough time to revoke the key in the IT system.

R14

Private keys shall be password protected using AES128-CBC mode.

Prior to OpenSSH version 5.3, private keys are password-protected using 3DES-CBC. Since version 5.3, AES128-CBC is used by default.

Old keys protected by 3DES-CBC can be protected with AES128-CBC by requesting a password change in recent versions of OpenSSH (`-p` option of `ssh-keygen`). Once the password is modified, the old 3DES encrypted file shall be properly erased.

On the servers, access permissions to the `~/.ssh/` directory and its content shall be correctly set. Only the concerned account user shall have write permissions on it.

The `sshd` server should verify the correctness of file modes and file permissions of the user prior to any session opening. This behaviour can be configured via the `StrictModes` directive of the `sshd_config` file.

```
StrictModes yes
```

4.1.5 Choosing Symmetric algorithms

Once the peers are mutually authenticated, the communication channel is protected in confidentiality and in integrity.

R15

The encryption algorithm shall either be AES128-CTR, AES192-CTR or AES256-CTR. The integrity mechanism shall rely on HMAC-SHA1, HMAC-SHA256 or HMAC-SHA512.

The cryptographic algorithms are negociated between the client and the server, so the list of available algorithms must be set on both sides. Weak algorithms shall be removed from the negotiated cryptographic suites.

CBC mode implemented in OpenSSH has known vulnerabilities, CTR mode is thus recommended [5].

Integrity algorithms HMAC-SHA256 and HMAC-SHA512, recommended by the RGS, are supported since OpenSSH version 5.9. Operating systems using earlier versions should then rely on HMAC-SHA1.

5. "Plaintext Recovery Attacks Against SSH", Martin R. Albrecht, Kenneth G. Paterson and Gaven J. Watson.

Under OpenSSH, the following directives shall be added in `sshd_config` and `ssh_config` configuration files:

```
Ciphers aes256-ctr,aes192-ctr,aes128-ctr
# For 6.3+ versions, OpenSSH supports ETM (encrypt-then-mac),
# safer than older implementations (mac-then-encrypt)
MACs    hmac-sha2-512-etm,hmac-sha2-256-etm,hmac-sha1-etm

# on older versions, use only "hmac-sha1"
MACs    hmac-sha2-512,hmac-sha2-256,hmac-sha1
```

4.2 System hardening

4.2.1 Hardening at compilation time

The `sshd` service often runs as root because it requires superuser privileges (to switch to the connected user after authentication).

It is, therefore, mandatory to harden its code in order to delay or prevent potential compromises.

R16

A preliminary step in hardening the `sshd` service is to use proper compilation flags. Refer to "*Security recommendation for systems using GNU/Linux*" as an example.

4.2.2 Privilege separation

Privilege separation limits the potential impacts of a flaw, thanks to the principle of least privilege.

Since version 5.9, OpenSSH implements more advanced containment mechanisms, such as SECCOMP (Linux) or systrace (OpenBSD). Their use is strongly recommended if they are available.

Privilege separation shall be enabled in the `sshd_config` configuration file. The use of sandboxing should be preferred, when available:

```
# If "sandbox" cannot be used, set "yes"
UsePrivilegeSeparation sandbox
```

4.2.3 SFTP et Chroot

As previously indicated, `sshd` also provides an SFTP server used for downloading or uploading files. This service does not require any shell access.

Users or groups dedicated to the SFTP service should be chrooted in a dedicated partition of the system through the `ChrootDirectory` option:

```
# For a user: "Match user <login/user>", or
# for a group: "Match group <group>"
Match group sftp-users
    ForceCommand internal-sftp
    ChrootDirectory /sftp-home/\%u
    ...
```

If the `group` option is used, users must be members of the according system group (in this example: `sftp-users`)

The use of a dedicated file hierarchy ensures that:

- SFTP service users are isolated in a file tree and have neither knowledge nor access to other file hierarchies;
- the volume is not shared with other partitions, thus limiting the impacts of a saturation on the system.

Depending on the disk partitioning layout and the underlying file system, the dedicated partition should be mounted with the `nodev`, `noexec` and `nosuid` options. Moreover, the chroot mitigation method (detailed in section 4.2.3) is only effective when privilege separation (see 4.2.2) is enabled.

4.3 Authentication and user access control

4.3.1 User authentication

Several mechanisms can be used to authenticate the different users.

R17

User authentication should be performed with one of the following mechanisms, given by order of preference:

- ECDSA asymmetric cryptography;
- RSA asymmetric cryptography;
- symmetric cryptography (Kerberos tickets from the GSSAPI);
- authentication modules that expose neither the user password nor its hash (third-party PAM or BSD Auth modules);
- password check against a database (such as `passwd/shadow`) or a directory.

Note: in all cases, password authentication must not be used for highly privileged users.

Asymmetric cryptography mechanisms (ECDSA, RSA) are directly provided by the OpenSSH toolbox. Owners of public keys listed in the `authorized_keys` file of a user account will be granted access to the system under this account.

User accounts always have rights on the system, no matter how minimal they are. Their access should thus be authenticated and brute force attacks should be mitigated.

R18

Users rights shall follow the least privilege principle. Restrictions can be applied on several parameters: available commands, source IP, redirection of forwarding permissions...

When restrictions cannot be directly applied on the server (via `sshd_config`), they can be defined on a per-key basis in the `authorized_keys` file:

```
# The key only allows access if the client connects from the 192.168.15/24 network
from="192.168.15.*" ssh-ecdsa AAAAE2Vj...
# The key allows access to a customer coming from the ssi.gouv.fr domain except
# from the public.ssi.gouv.fr domain
from="!*.public.ssi.gouv.fr, *.ssi.gouv.fr" ssh-ecdsa AAAAE2Vj...
# Denies the use of the forwarding agent for this key
no-agent-forwarding ssh-ecdsa AAAAE2Vj...
```

Some PAM modules, BSD Auth... require the user password to be sent to the server in plain text. The SSH tunnel created only protects the password on the network. If the server is compromised, the password can be reused to access other services.

Although the GSSAPI and the `pam_krb5` module rely on the same mechanisms, the `pam_krb5` module does not take advantage of Kerberos benefits (user password protection against third party services): GSSAPI directly handles Kerberos cryptographic tickets, while the PAM module retrieves the user password in cleartext and submits it to the KDC. The `pam_krb5` module should, therefore, not be used.

Access to empty password accounts shall be banned in `sshd_config`. Moreover, the authentication timeframe shall be relatively short and the number of authentication attempts per connection limited to one.

```
PermitEmptyPasswords no
# Warning, MaxAuthTries must be strictly greater than 1.
# In addition, a low number can cause problems.
MaxAuthTries 2
LoginGraceTime 30
```

4.3.2 Authentication agent

Users frequently bounce from one SSH host to another.

After the first connection, the user encounters an authentication problem for the subsequent bounces since the relay host has no knowledge of the user's secret. Therefore the user must provide its secret (e.g. by typing in the password) to authenticate to the successive hosts.

The `ssh-agent` utility provided by OpenSSH can retain a user's private keys when authentication is based en ECDSA or RSA keys. A relay host using agent forwarding proxies authentication requests from the connected host to the user's workstation, where the authentication occurs.

R19

When SSH bouncing is necessary through a relay host, Agent Forwarding (`-A` option of `ssh`) should be used.

This mechanism keeps the user private key away from the relay server while providing the benefits

of public key cryptography protection.

R20

The relay host server shall be a trusted host.

Although the private key is never exposed to the relay host server, the latter can directly communicate with the Forwarding Agent in order to initiate authentication operations. Therefore, the compromise of the host relay server exposes the Forwarding Agent to authentication requests not initiated by the user.

To minimize their exposure to non-legitimate authentication requests, the keys can be loaded in the ssh-agent keyring with the following options (via `ssh-add`):

`-c` prompts the user for confirmation through an external graphical program (such as `ssh-askpass`) when using a key of the keyring;

`-t <s>` automatically removes the key from the keyring after `<s>` seconds.

4.3.3 Access accountability

Servers Access Accountability is essential for traceability.

R21

Every user must have his own, unique, non-transferable account.

`root` is an example of a generic account existing on all Unix/Linux systems, and used as an administrative account. The `root` account shall not be remotely accessible:

```
PermitRootLogin no
```

Having a dedicated account for each user allows finer access management and traceability. Moreover, a user knowing that his/her account is an exclusively dedicated one may be more cautious when using it.

The `PrintLastLog` directive of `sshd_config` displays last connection details upon every user connection.

```
PrintLastLog yes
```

4.3.4 AllowUsers, AllowGroups

R22

Access to a service shall be restricted to users having a legitimate need. This restriction shall apply on a white-list basis: only explicitly allowed users shall connect to a host via SSH and possibly from specified source IP addresses.

Under OpenSSH, the `AllowUsers` directive of `sshd_config` specifies a list of SSH authorized users and groups.

Some features offered by OpenSSH can cause security problems when maliciously exploited: shell access, forwarding, sending/downloading binaries, etc. Restricting access to legitimate accounts is thus mandatory.

`sshd` may be exposed to IP addresses not belonging to the management network. The following directives can restrain user access according to their source IP addresses:

```
# Restrict the 'admin' account to the administration network IP (192.168.17/24)
# and the 'user' account to the Intranet network IP (10.127/16)
AllowUsers admin@192.168.17/24 user@10.127/16
# Same restrictions but with 'wheel' and 'users' groups instead
AllowGroups wheel@192.168.17/24 users@10.127/16
```

4.3.5 Restrictions of the user environment

Environment variables can significantly alter a program behaviour. Some of them may have security implications, in particular when they affect their configuration or execution (`LD_PRELOAD`, `LD_LIBRARY_PATH`...).

R23

The ability for a user to tamper with the environment shall be denied by default. User-supplied environment variables shall be selected on a case-by-case basis.

Environment modification through the `sshd` service is denied with the `PermitUserEnvironment` directive of `sshd_config`:

```
PermitUserEnvironment no
```

The `AcceptEnv` directive provides a list of user alterable environment variables.

The `ForceCommand` directive, which forces the execution of a command upon connection of a user, is another effective measure to restrict a user environment.

R24

Users shall only execute strictly necessary commands. This restriction can be achieved in the following ways:
- using the `ForceCommand` directive on a per user basis in the `sshd_config` file;
- specifying some options in the `authorized_keys` file (See 4.3.1);
- using secure binaries such as `sudo` or `su`

The `ForceCommand` directive is especially suited to prevent easy shell access from a compromised account when SSH is used to launch executables remotely. Its use should be restricted to monitoring activities, simple administration commands (services restart or status) or harvesting scripts output.

Let's say a *myproc* user is dedicated to a monitoring service and is only allowed to read the list of currently running processes on a given host. His `sshd_config` file should contain:

```
Match user myproc
ForceCommand /bin/ps -elf
```

`SSH` connections of the myproc user automatically trigger the `/bin/ps -elf` command and exits.

4.4 Protocol and network access

4.4.1 ListenAddress and Ports

Administrative operations are commonly performed through SSH. As such, the SSH server should only be reachable from the administration network, which should ideally be separated physically from the operational network.

R25

The SSH server shall only listen on the administration network.

When the SSH server is exposed to uncontrolled networks due to operational constraints, brute force attacks are common on the default port (22). Those attacks will flood the logs in addition to present a threat to weak password accounts.

R26

When the SSH server is exposed to an uncontrolled network, one should change its listening port (22). Preference should be given to privileged ports (below 1024) to prevent spoofing attempts by unprivileged services on the remote machine.
On a controlled network, the SSH server shall only listen on a management network interface, separated from the operational network.

Under OpenSSH, the `ListenAddress` directive of `sshd_config` specifies the listening address and port of the sshd server. Multiple `ListenAddress` directives may be used.

```
# Listening only on IP address A.B.C.D and port number 26
ListenAddress A.B.C.D:26
```

4.4.2 AllowTCPForwarding

As previously mentioned, OpenSSH allows a user to forward flows through the server, both from the local network and the remote network.

This feature can be used in rebound attacks to contact protected services (behind filtering gateways, listening on a different network...). In addition, forwarding local flows may result in exposure of the

internal network to uncontrolled flows.

R27

Except for duly justified needs, any flow forwarding feature shall be turned off:
- in the SSH server configuration;
- in the local firewall by blocking connections.

Under OpenSSH, server side forwarding can be disabled using the `AllowTcpForwarding` directive of `sshd_config`:

```
AllowTcpForwarding no
```

4.4.3 X11Forwarding

X11 forwarding (Figure 3) is very similar to remote flow forwarding, detailed in section 4.4.2. This feature shall be deactivated by default. Exceptions to this recommendation should be carefully considered and security implications deeply analysed.

Figure 3: Diagram of X11 forwarding through SSH

As a general rule, installing an X server on a console-only host should be avoided, due to several security issues in the X Window System.

R28

X11 forwarding shall be disabled on the server.

Disabling X11 forwarding can be done using the `X11Forwarding` directive of `sshd_config`:

```
X11Forwarding no
```

When X11 forwarding is active (remote execution of an X11 client), compromise of the remote host can lead to uncontrolled information leakage:
- user keystrokes capture;
- display forwarding...

When X11 redirection is required due to operational constraints, remote X11 clients should be treated with suspicion and their privileges shall be strictly limited.

An `ssh` client should use the `-X` option and disable the `ForwardX11Trusted` option from its `ssh_config` configuration file:

```
ForwardX11Trusted no
```

4.5 OpenSSH PKI

4.5.1 Certification Authority

Since version 5.4, OpenSSH provides a PKI in order to facilitate key management by simplifying the number of relationships to maintain between entities (hosts and users).

However, this PKI implementation is specific to OpenSSH and does not rely on standards such as X.509. Its use consists in creating one (or more) Certificate Authority (CA), and in using its private key to sign certificates and its associated public key to check their signature.

For example, a CA can be used to authenticate hosts on the network: once the public key of the CA is known to the client, it can be used to check the signature validity of host certificates. The obvious advantage is the absence of additional work required on all clients to support each newly installed host (updating the `known_hosts` file becomes optional).

Note that the OpenSSH design choices do not provide a PKI that complies with the RGS (certificates are not using the X.509 format, there is no certification chain...). It shall therefore be dedicated to operational and security management of "OpenSSH" machines.

R29

It is recommended to create distinct CAs when their roles differ. There will be, for example:

- one CA for the "hosts" CA role;
- one CA for the "users" CA role.

Each CA private key shall be protected by a unique and robust password.

The OpenSSH PKI does not support certification chains, where a root CA can sign several delegated CAs. Each CA must then be confined to a particular role.

An OpenSSH CA is created like a "user" entity: key generation is performed via `ssh-keygen`. See 4.1.2.

Once the CA is created, it can be used to generate and sign certificates with its private key. Its public key must be distributed to entities (hosts or users) expected to trust this CA.

On an `sshd` host, the "users" CA public key is declared with the `TrustedUserCAKeys` directive of the `sshd_config` file.

On an `ssh` client, the "hosts" CA public key is specified with the `@cert-authority` marker in the `known_hosts` file. The line would look like the following:

```
# Declare a CA key acceptable for any host
@cert-authority * AAAAB4Nz1...1P4==
```

Please note that the CA signs the public key obtained by various means of communication. As this key will be associated with a given identity possibly attached with some options, the link between all these must be sound. Providing a public key alone is generally not sufficient to verify such a property. For example, a face to face meeting, a signed cryptographically message or a fingerprint obtained through a secure communication channel are measures that allow the preservation of the relationship between a key and the owner's identity for certification.

4.5.2 Certificates

A certificate consists in various elements whose integrity is guaranteed by the CA signature. The `ssh-keygen` documentation gives a comprehensive description of this mechanism.

The settings of a **"user" certificate** will be:

- **principals:** the user login used for connecting to the server;
- **key ID:** an arbitrary string, unique, for storing data related to the certificate (version number, user's full name...);
- **public key:** the user's public key;
- **lifetime:** 3 years.

"Hosts" certificates will present the following settings:

- **principals:** hostname(s) used for the connection (coma separated);
- **key ID:** an arbitrary string, unique;
- **public key:** the host's public key;
- **lifetime:** 3 years.

The certificate lifetime is compliant with Appendix A of the RGS.

"User" certificate generation and signature step is performed via `ssh-keygen`:

```
ssh-keygen -s <CA private key> -I <key ID> \
     -n <principal1,principal2,...> -V +156w <public key file>
```

"Host" certificate generation and signature requires the `-h` option.

Once generated and signed, the resulting certificate is stored in the `<public key file>-cert.pub` file. It shall be transfered:

- in the path specified by the `HostCertificate` attribute of `sshd_config` for a host;
- in the `~/.ssh/` repository for a user.

4.5.3 Revocation

Revocation process can happen either on hosts, users or CA keys. Once a key is revoked, it cannot be used to authenticate entities anymore.

R30

If a key cannot be considered safe anymore, it shall be quickly revoked at the SSH level.

The revocation mechanism chosen by OpenSSH is based on flat text files. The software suite does not provide any tool nor protocol for revocation lists synchronization; operators are required to define adequate procedures in order to maintain these files.

On an `sshd` host, revoked user keys are contained in a file specified by the `RevokedKeys` attribute of `sshd_config`.
On an `ssh` client, revoked host keys are declared with the `@revoked` marker in the `known_hosts` file.

4.6 DNS records

DNS zones can store SSH hosts fingerprints via `SSHFP` records.

Recording SSH key fingerprints in the DNS allows to rely on a third party system to validate the fingerprint: the `ssh` client will try to obtain the host's key fingerprint by a request to the name server.

As DNS is not a secure protocol (except when DNSSEC is deployed), the obtained fingerprint record is only an indicator. This verification method is weaker than any method based on reliable cryptographic mechanisms, but has the advantage of being easier to deploy.

It ensures that two distinct entities provide the same host credentials: a mismatch between the key fingerprint provided by the `sshd` host and the one recovered from the DNS should be considered as a critical security incident.

OpenSSH implements RFC 4255 specifying SSH fingerprints publication through DNS records.
The fingerprint record can be obtained via `ssh-keygen`:

```
ssh-keygen -r <hostname> -f <public key file>
```

DNS fingerprint validation should be activated on the `ssh` client using the `VerifyHostKeyDNS` attribute of `ssh_config`:

```
VerifyHostKeyDNS  ask
```

R31

SSH host key fingerprints obtained through DNS records should not be trusted without complimentary verifications.