Engineer

SSH

an encrypted shell on another machine

What it is

SSH, short for Secure Shell, is a network protocol for signing in to another computer and running commands on it across a network nobody trusts, with the traffic encrypted and both ends proving who they are first. The name covers three things: the protocol, published by the IETF in January 2006 as a set of RFCs led by RFC 4251; OpenSSH, the implementation maintained by the OpenBSD project and the one this article's examples use; and the `ssh` command, which talks to a server daemon called `sshd`.

RFC 4251 describes three layers over one connection, normally TCP port 22, the number IANA assigned to it. The transport layer runs the key exchange, proves the server's identity with its host key, and then encrypts and integrity-checks everything that crosses. The user authentication layer runs inside that tunnel and settles who the client is: by public key, which every implementation must support, by password, which is optional, or by host-based trust. The connection layer divides the tunnel into channels, so one login can carry an interactive shell, a single command, an SFTP file transfer and several forwarded ports at once.

SSH replaced telnet and rlogin, which sent sessions and passwords across the network as plain text. The protocol in use today is version 2.

Use the concept in a decision

Put "What is SSH?" to work in a real decision. We can help relate the concept to the system you are building or buying. Accounts, passwords and permissions arranged so people hold exactly the access their job needs, and leavers lose it that day.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Wrote 1,000 RBAC rules to secure Linux servers — 1,000 access rules written to close a security gap

Wrote 1,000 RBAC rules to secure Linux servers Get in touch →

Why it matters

On a server, SSH is the front door. A machine in a data centre or a cloud rarely has a keyboard within anyone's reach, so administration, deployments, file copies and automated configuration all arrive through SSH, and whoever logs in with enough privilege owns the machine.

Public key authentication is what lets SSH be convenient and strong at once. The user keeps a private key on their own machine and puts the matching public key on the server, by default in `~/.ssh/authorized_keys`. At login the client signs data tied to that session, so the key itself never crosses the wire. A stolen password can be replayed from anywhere, and a server that accepts passwords can be guessed at by anyone who reaches its port; automated tools do exactly that. OpenSSH accepts passwords out of the box, so turning them off is a decision each server has to make.

Trust also runs the other way, because the client needs to know it reached the right server. Each server holds a host key, and RFC 4251 offers two ways to trust it: a local list on the client that maps each host name to its key, or a certification authority that signs host keys so the client holds only the authority's key. Most people use the first in its loosest form, which the RFC accepts as a concession to usability: take the key unchecked on first contact, save it in `~/.ssh/known_hosts`, and compare against it on every later connection. This is usually called trust on first use. The first login is only as safe as the network it crosses, which is why OpenSSH asks before saving a new key and refuses outright when a known host presents a different one.

SSH is also a transport that other tools build on. Git pushes and pulls over it, `scp` and `sftp` copy files with it, Ansible reaches its hosts through it, and port forwarding lets a laptop reach a database that listens only on a server's loopback interface without opening it to the internet.

What it looks like in practice

The company's own host accepts interactive logins over SSH and nothing else, and its hardening is checked in three stages each time the automation role that owns the configuration runs. The checks date from September 2026, when an access report compared the settings the daemon had resolved with the ones the role wrote, and two differed: the distribution's defaults were live lines read before the role's drop-in, and sshd kept the earlier value. Renaming the drop-in so it sorts before the vendor's file closed the gap. Now an SSH configuration has to pass three gates: the candidate parses as valid syntax before it is put in place, the assembled set of files passes a check after installation, and the daemon's resolved settings are compared with 24 expected directives, so a value that loses to an earlier line stops the run. Two directives are left out deliberately: the current daemon has dropped both.

A month earlier, a review in August 2026 found three brute-force protections that had never worked since the host was built. The ban jail was watching port 22 while the daemon listened on 1986. The firewall's rate limit came after a broader rule that allowed the same traffic, and since the first matching rule wins, the limit never saw a packet, though the status output kept listing it. The ban action relied on a packet-filter tool that the package manager only recommends, so the jail spotted attackers and then errored each time it tried to block one. The review left two standing rules: every security control is backed by a check that proves it acts, and a firewall rule is confirmed by its position as well as its presence.

The company's own git forge uses SSH as a transport rather than a way to log in. It was picked because it is driven over SSH and has no web admin panel to defend. Nobody gets in anonymously or without a key, and every repository is private. Its SSH listener accepts connections on loopback only, at port 23231, and people outside reach it by hopping through a jump host, so the firewall opens no extra port for it.

Where it goes wrong

A hardening line sits in the file and the server ignores it. For most keywords `sshd` keeps the first value it reads, and an `Include` expands its files in lexical order, so when the main file pulls in a directory of drop-ins first, a vendor file that sorts early beats yours and both files still look right. Searching the configuration proves only that a line was written. Ask the daemon instead: `sshd -t` checks the syntax, and `sshd -T` prints the settings it will actually use.

```
/etc/ssh/sshd_config.d/10-vendor.conf:     PasswordAuthentication yes
/etc/ssh/sshd_config.d/50-hardening.conf:  PasswordAuthentication no

$ sudo sshd -T | grep -i '^passwordauthentication'
passwordauthentication yes
```

A changed host key gets deleted and accepted. When a known server presents a different host key, `ssh` prints a warning and refuses to connect. The reflex is `ssh-keygen -R` on the host name, a reconnect and a typed yes, which is everything a man-in-the-middle needs. Confirm the new fingerprint through another channel first, for example with `ssh-keygen -lf` on the server's public host key from its console. In scripts, `StrictHostKeyChecking no` is worse than it looks: besides saving new keys silently, it lets connections to hosts whose keys have changed proceed, within some limits. The `accept-new` setting saves unknown hosts and still refuses changed keys.

Password logins stay open to the whole internet. `PasswordAuthentication` defaults to yes, and so does keyboard-interactive authentication, which can ask for the same password by another route. Once every user has a key, set both to no. `PermitRootLogin` already defaults to `prohibit-password`, which refuses passwords for root but still lets a root key in. Moving the daemon off port 22 changes where it listens, not who can guess at it, and every control that names the port has to move with it, as the ban jail above shows.

Agent forwarding lends your keys to the server. With `ForwardAgent` on, or `ssh -A`, anyone who can bypass file permissions on the remote host, root included, can use your agent to authenticate elsewhere as you while you stay connected. They cannot copy the key, but they can sign with it. To reach a machine behind a bastion, use a jump host instead: `ssh -J bastion.example.com app.internal` forwards a TCP connection through the bastion and negotiates the session with the target end to end, so the bastion relays bytes it cannot read and the key stays on your laptop.

Further reading

- RFC 4251: The Secure Shell (SSH) Protocol Architecture — the three layers, the two host key trust models and the security considerations.
- RFC 4253: The Secure Shell (SSH) Transport Layer Protocol — key exchange, host key verification and the version string each side sends first.

- RFC 4252: The Secure Shell (SSH) Authentication Protocol — how public key, password and host-based logins work on the wire.
- sshd_config(5) — every server keyword with its default, and the rule that the first value obtained is used.
- ssh_config(5) — the client side, including StrictHostKeyChecking, ProxyJump and ForwardAgent.
- sshd(8) — the `-t` and `-T` test modes and the authorized_keys and known_hosts formats.

Glossary

Choose for your actual constraints

Bring the deployment path, the operational risks and the people on call. We can scope a change with checks, a recovery route and clear ownership.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is SSH?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/ssh/