SSH Protocol

SSH (Secure Shell Protocol) is used to securely send commands to a computer over an insecure network. SSH uses cryptography for authentication, encryption, and integrity, making it a secure way to transmit data. SSH utilizes tunneling to send data that might be blocked by network restrictions, such as firewalls. Older protocols like Telnet and RSH (Remote Shell) were replaced by SSH because they did not encrypt connections, leaving data vulnerable to interception and modification.

It can be compared to a store owner giving instructions to an employee while traveling, where encryption ensures that no one else can overhear the conversation.

Telnet was an early protocol used to access remote systems, but it transmitted data in plaintext, making it highly insecure. RSH (Remote Shell) was another early protocol used for executing commands on remote machines but lacked encryption, making it vulnerable to attacks. SSH replaced Telnet and RSH due to its ability to provide secure authentication and encrypted communication over an insecure network.

SSH Features

1. Encryption
SSH encrypts data so no one can read it while it travels over the internet.
Example: When you type your password on an SSH session, it is scrambled so hackers can't steal it.

2. Authentication
SSH checks who is logging in to make sure it's a trusted user. There are two main ways:
- Password Login: You enter a password to connect.
- Key Login: You use a special file (SSH key) instead of a password.
Example: Instead of typing your password every time, you can set up an SSH key so your computer logs in automatically.

3. Integrity
SSH ensures that the data you send and receive is not changed by hackers.
Example: If you send a command to a remote server, SSH checks that the response is exactly what was sent, with no tampering.

4. Port Forwarding
SSH can securely tunnel data from one computer to another, bypassing restrictions.
Example: If a website is blocked in your country, you can use SSH tunneling to route your internet through a different server and access it securely. Similarly, a database running on a remote server can be accessed on your local computer without adding that port and protocol in the firewall.

Use Cases of SSH

1. Remote Command Execution: Running commands on remote machines securely.

2. Secure File Transfer: Using SCP (Secure Copy Protocol) or SFTP (SSH File Transfer Protocol) to transfer files securely.

3. SSH Tunneling: Bypassing firewalls or securely forwarding ports.

Remote Command Execution

1.1 Check if SSH is Installed (If Not, Install It)

```
# Before using SSH, ensure it is installed on your system.

# Linux - Check if SSH is installed
ssh -V

# If SSH is missing, install it
sudo apt update && sudo apt install openssh-client -y

# Install SSH server (to allow incoming SSH connections)
sudo apt install openssh-server -y
sudo systemctl enable --now ssh

# macOS - Check SSH version
ssh -V

# If missing (rare case), reinstall via Homebrew
brew install openssh

# Windows - Check if OpenSSH is installed
Get-Service -Name ssh-agent

# Enable OpenSSH if not installed
Add-WindowsFeature -Name OpenSSH-Client, OpenSSH-Server
Start-Service ssh-agent

# Or install OpenSSH using Chocolatey
choco install openssh

# Alternative: Install PuTTY or Git Bash (which includes SSH)
```

1.2 Login via Password

```
# Log in to a remote machine using SSH and a password

# Basic command
ssh savi@remote-ip

# Replace 'remote-ip' with the actual server IP or hostname
# Example:
ssh savi@192.168.1.100

# Enter the password when prompted to log in
```

How SSH Works Internally

When you connect to an SSH server, several security mechanisms are used to establish a secure connection.

Step 1: TCP 3-Way Handshake

Before SSH starts, a basic network connection is established using a 3-way handshake between the client (your computer) and the server.

At this point, a raw connection is established, but the data is not encrypted yet.

Step 2: Protocol Exchange

Now, the client and server exchange SSH protocol details:

- What version of SSH are they using?
- What encryption and authentication methods are supported?

The server responds with:
SSH-2.0-OpenSSH_8.9

The client responds with:
SSH-2.0-ClientVersion

Step 3: Key Exchange using Diffie-Hellman

Now, the Diffie-Hellman algorithm is used to establish a shared secret key 🔑 that both client and server will use for encryption.

1. The client and server each generate random secret values.
2. They exchange public keys and use a mathematical formula to generate the same shared secret without actually sending it over the network.
3. This secret key is now used for encrypting the communication.

- Even if a hacker captures the public key exchange, they cannot derive the private shared key.

Step 4: Authentication

Now, SSH asks for user authentication.

- If using password login, SSH encrypts the password before sending it.
- If using key-based login, the client proves its identity using a private key.

If authentication is successful, the SSH session starts.

Step 5: Secure Communication Begins

Now that authentication is complete, the client and server use AES encryption to send and receive data securely.

- Commands sent via SSH are encrypted before transmission.
- Responses from the server are encrypted before being sent back to the client.

Now you can safely execute remote commands.

SSH Key Login (Public Key Authentication)

Using SSH keys is more secure than password-based login because it eliminates the risk of brute-force attacks and password leaks. It is recommended to use SSH Key login.

Generate SSH Key Pair for User savi

SSH key pairs consist of:

Private Key (`id_rsa`) → Stays on the client (DO NOT share).
Public Key (`id_rsa.pub`) → Placed on the remote server.

Generating an SSH Key Pair

Run the following command on your local machine (client):

```
ssh-keygen -t rsa -b 4096 -C "savi@your-client"
```

- `t rsa` → Use the RSA algorithm.
- `b 4096` → Generate a 4096-bit key (more secure).
- `C "savi@your-client"` → Adds a comment (optional).

This will prompt:

```
Enter file in which to save the key (/home/savi/.ssh/id_rsa):
```

Press Enter to save in the default location (`~/.ssh/id_rsa`) or provide the absolute path of another location.

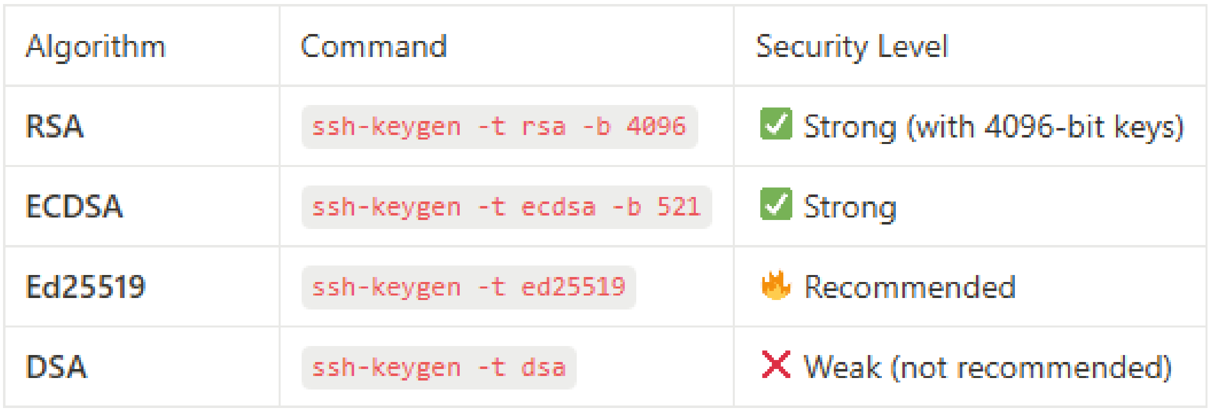
It will then ask for a passphrase (optional but recommended).

- Adding a passphrase provides extra security in case the private key is stolen.
- If left blank, SSH will not ask for a password when using the key.

After this, two files are created:

```
~/.ssh/id_rsa # Private key (KEEP SAFE)
~/.ssh/id_rsa.pub # Public key (SHARE WITH SERVER)
```

SSH supports multiple cryptographic algorithms for key generation:



Algorithm	Command	Security Level
RSA	`ssh-keygen -t rsa -b 4096`	✅ Strong (with 4096-bit keys)
ECDSA	`ssh-keygen -t ecdsa -b 521`	✅ Strong
Ed25519	`ssh-keygen -t ed25519`	🔥 Recommended
DSA	`ssh-keygen -t dsa`	❌ Weak (not recommended)

```
# Store the Keys Properly

# On Client Machine (Local Machine):
# The private key stays on the local machine at:
~/.ssh/id_rsa

# Ensure correct permissions
chmod 600 ~/.ssh/id_rsa

# On Server (Remote Machine):
# Copy the public key to the remote server
ssh-copy-id -i ~/.ssh/id_rsa.pub savi@remote-ip

# Or manually add the public key to the authorized_keys file on
the server
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys

# Ensure correct permissions on the server
chmod 600 ~/.ssh/authorized_keys
chmod 700 ~/.ssh

# Now, log in without a password
ssh -i ~/.ssh/id_rsa savi@remote-ip
```

To enhance security, we will:

1. Disable root login
2. Disable password authentication
3. Change the default SSH port

```
# Edit SSH Configuration

# Open the SSH configuration file on the server
sudo nano /etc/ssh/sshd_config

# Modify these settings:

# Disable root login
PermitRootLogin no

# Disable password authentication (force key-based login)
PasswordAuthentication no

# Change the default SSH port (e.g., from 22 to 2222)
Port 2222

# Apply Changes for SSH Socket and SSH Daemon

# Stop the SSH socket activation mechanism
sudo systemctl stop ssh.socket

# Disable SSH socket activation to prevent auto-start
sudo systemctl disable ssh.socket

# Ensure the main SSH service starts at boot
sudo systemctl enable ssh.service

# Restart the SSH service to apply changes
sudo systemctl restart ssh.service

# Restart SSH to finalize all modifications
sudo systemctl restart ssh
```

```
# Firewall Configuration for SSH (Custom TCP Port 2222)

# If using UFW (Uncomplicated Firewall)

# Allow new SSH port
sudo ufw allow 2222/tcp

# Enable firewall
sudo ufw enable

# Verify rules
sudo ufw status

# If using firewalld (CentOS/RHEL)

# Add custom TCP rule for port 2222
sudo firewall-cmd --permanent --add-port=2222/tcp

# Reload firewall rules
sudo firewall-cmd --reload

# If using cloud security groups (AWS, DigitalOcean, etc.),
manually add a Custom TCP rule allowing port 2222 in the security
group settings.

# Now, connect using SSH with a custom port and key
ssh -i ~/.ssh/id_rsa -p 2222 savi@remote-ip
```

Secure File Transfer

Securely transferring files between a local machine (client) and a remote server or vice-versa can be done using SCP (Secure Copy Protocol) or Rsync.

SCP is a simple and secure way to transfer files between systems over SSH. However, it does not support resuming interrupted transfers and lacks efficient synchronization.

Copy Files from Client to Server (Using Custom Port 2222 & Private Key)

```
scp -P 2222 -i /path/to/private-key /path/to/localfile savi@remote-
ip:/path/to/destination/
```

Copy Files from Server to Client

```
scp -P 2222 -i /path/to/private-key savi@remote-ip:/path/to/remotefile
/path/to/local/destination/
```

Copy an Entire Directory from Client to Server

```
scp -P 2222 -i /path/to/private-key -r /path/to/localdir savi@remote-
ip:/path/to/destination/
```

Copy an Entire Directory from Server to Client

```
scp -P 2222 -i /path/to/private-key -r savi@remote-ip:/path/to/remotedi
/path/to/local/destination/
```

Rsync is more advanced than SCP because it:

- Supports resuming interrupted transfers
- Only transfers changed data (efficient synchronization)
- Compresses data during transfer (faster speeds)
- Preserves file permissions, timestamps, and symbolic links

Copy Files from Client to Server (Using Custom Port 2222 & Private Key)

```
rsync -avz -e "ssh -i /path/to/private-key -p 2222"
/path/to/localfile savi@remote-ip:/path/to/destination/
```

Copy Files from Server to Client

```
rsync -avz -e "ssh -i /path/to/private-key -p 2222" savi@remote-
ip:/path/to/remotefile /path/to/local/destination/
```

Copy an Entire Directory from Client to Server

```
rsync -avz -e "ssh -i /path/to/private-key -p 2222" /path/to/localdir/
savi@remote-ip:/path/to/destination/
```

Copy an Entire Directory from Server to Client

```
rsync -avz -e "ssh -i /path/to/private-key -p 2222" savi@remote-
ip:/path/to/remotedir/ /path/to/local/destination/
```

Advantages of Rsync over SCP

- Resumes Transfers: Rsync can resume interrupted transfers (`--partial`), while SCP cannot.
- Efficient Data Sync: Rsync transfers only changed parts of files, whereas SCP always re-copies the entire file.
- Compression: Rsync supports compression (`-z`), making transfers faster; SCP does not.
- Preserves File Attributes: Rsync keeps timestamps, permissions, and ownership (`-a`), but SCP has limited support.
- Bandwidth Efficiency: Rsync uses less bandwidth by copying only changes, while SCP transfers everything.
- Recursive Directory Copy: Both support it, but Rsync (`-a`) ensures better synchronization.

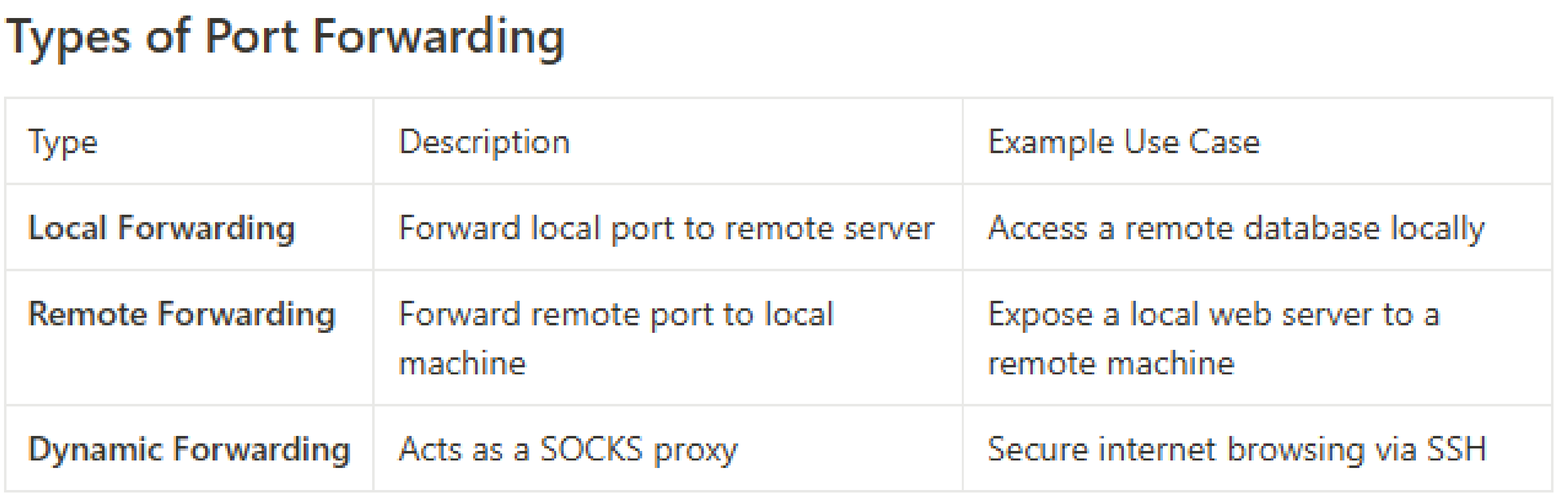
When to Use SCP vs. Rsync?

- Use SCP for quick, one-time transfers (small files).
- Use Rsync for large file transfers and synchronization (efficient and resumable).

SSH Tunneling (Port Forwarding)

SSH tunneling allows secure forwarding of network traffic over an encrypted SSH connection. It is useful for:

- Accessing remote services securely
- Bypassing network restrictions
- Exposing local services remotely



Types of Port Forwarding

Type	Description	Example Use Case
Local Forwarding	Forward local port to remote server	Access a remote database locally
Remote Forwarding	Forward remote port to local machine	Expose a local web server to a remote machine
Dynamic Forwarding	Acts as a SOCKS proxy	Secure internet browsing via SSH

Local Port Forwarding (Access Remote Service Locally)

Example: Forward a remote MySQL database (3306) to your local machine on port 3307

```
ssh -L 3307:127.0.0.1:3306 -N -f -i /path/to/private-key -p 2222
savi@remote-ip
```

```
Now, connect locally:
```

```
mysql -h 127.0.0.1 -P 3307 -u youruser -p
```

Remote Port Forwarding (Expose Local Service to Remote Machine)

Example: Expose a local web server (8080) to a remote machine on port 9090

```
ssh -R 9090:127.0.0.1:8080 -N -f -i /path/to/private-key -p 2222
savi@remote-ip
```

Now, access it from the remote machine:

```
curl http://127.0.0.1:9090
```

Advantages of SSH Tunneling:

- Secure : Encrypts data over SSH
- Bypass Restrictions : Access blocked services
- Remote Access : Connect to remote/local services securely
- No VPN Needed : Simple and lightweight alternative

Use Local Forwarding to access remote services locally.
Use Remote Forwarding to expose local services remotely.