SSH

Network Infrastructures Lab Sessions

SAPIENZA
UNIVERSITÀ DI ROMA

Instructors:
- Pietro Spadaccino

Today

- Crypto 101 - Symmetric and Asymmetric encryption
- SSH
- SSH Port Forwarding

Further readings:

- SSH: The Definitive Guide; D.J. Barret et al.; O'Reilly

Remote managing

In real life, physical access to network nodes is not always an option. Often, we need to reconfigure network nodes **remotely**.

- Remote access to a network node, usually with a terminal

Historically, the first protocol that allowed such things was telnet.

Telnet has been now deprecated since it does not use any encryption on the sent data!

Connecting to a SSH server

Every Linux distro has an ssh client installed called OpenSSH client.

On the other hand, the server daemon, sshd, needs to be manually installed. On kathara, they are both already installed.

The configuration file of the server is `/etc/ssh/sshd_config` while for the client is `/etc/ssh/ssh_config`. To start a connection with a server:

```
$ ssh username@host_ip_addr
# starts ssh session with the host specified by
# host_ip_addr logging in as username
```

You logout with CTRL+D

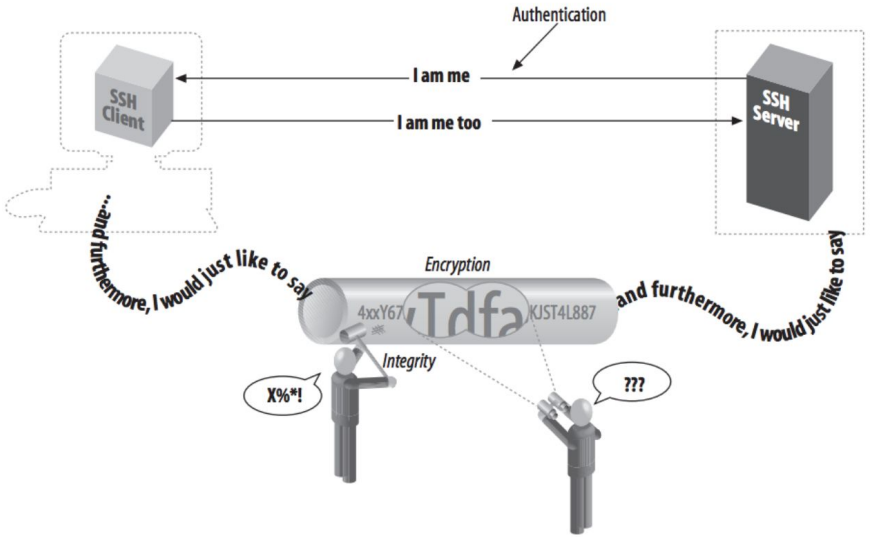
By default SSH runs on TCP port 22. If another port (e.g. 12345) is used it can be specified by:

```
$ ssh username@host_ip_addr -p 12345
```

Secure SHell

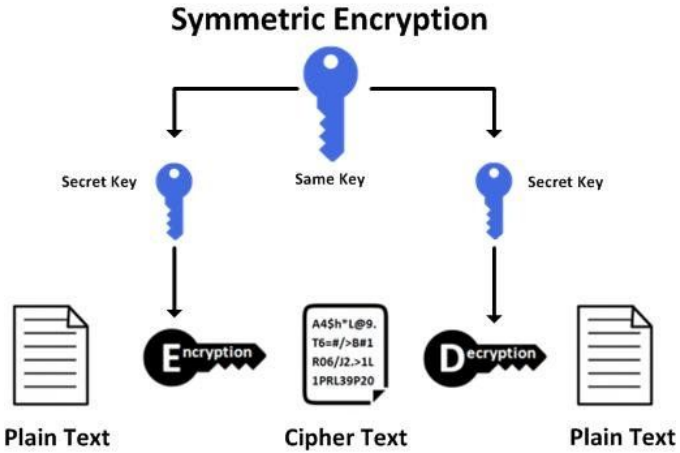
SSH (SSH-2) is a protocol that allows remote management over a secured channel. Is a client-server protocol. The main features that SSH provides are:

- Authentication
- Encryption
- Integrity



Symmetric-key cryptography

In Symmetric cryptography, the same key is used for encryption and decryption

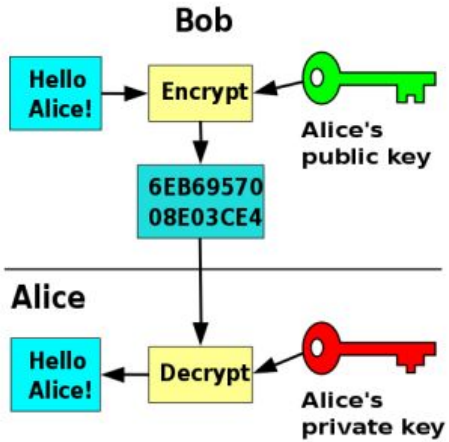


Most widely used symmetric key algorithms include Twofish, AES, Blowfish

Asymmetric-key cryptography

In Asymmetric cryptography we have two keys (often called a key-pair): *data is encrypted with one key and it is decrypted with the other key*!

Idea: we can share with everyone the key needed for encryption (**public key**) while keeping the decryption key as a secret (**private key**)

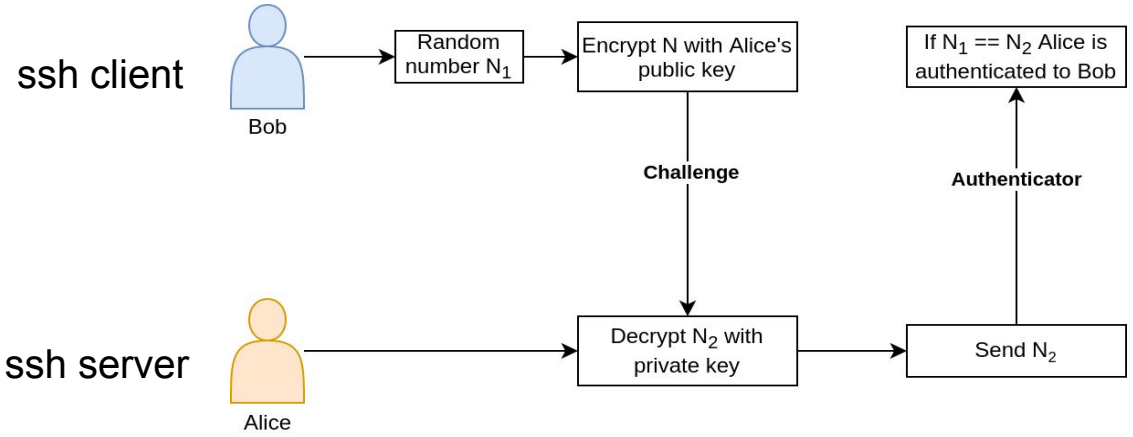


Asymmetric-key as authentication

In practice, asymmetric-key ciphers are not used to encrypt large quantities of data, since asymmetric-key algorithms tend to be slower.

Asymmetric-key is used in practice as a mean of **authentication** i.e. to prove your identity to someone else.

Example:



This is called **challenge-response authentication**. More complex methods may be involved but the concept is the same:

Bob sends to Alice a challenge that can be solved only by whom has the private key. At the end Alice is authenticated to Bob

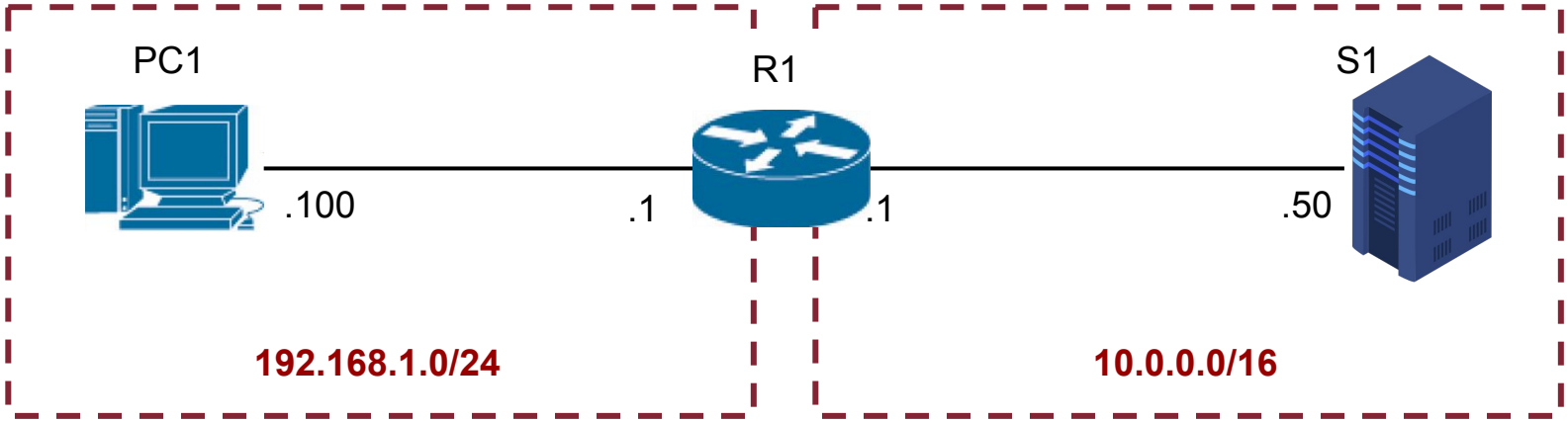
Creating a user on the server

In order to connect via SSH to a user, the user should be present on the **server** and should have a password set. The following .startup file:

1. Restarts the networking and the ssh daemon
2. Creates the user ssh_user with its home folder
3. Assigns the password “ilovessh” to ssh_user. (Without a password we can’t perform password authentication)

```
/etc/init.d/networking restart
/etc/init.d/ssh restart
useradd ssh_user -m
echo -e 'ilovessh\nilovessh\n' | passwd ssh_user
```

lab_ssh



Tip: mnemonic assignment of addresses

Linux allows us to specify string literals to ip addresses inside the `/etc/hosts` file.
The syntax is ip addr - tab - string, one entry per line.
For example, we assign the 10.0.1.100 to the string pc2 at startup on pc1:

```
/etc/init.d/networking restart
echo "10.0.1.100        pc2" >> /etc/hosts
```

After that we can type on pc1:

```
$ ssh ssh_user@pc2
```

`/etc/hosts` is not used necessarily for SSH: it “translates” addresses for everything (e.g. ping, netcat, …)

SSH protocol at a glance

Encrypted connection setup

1. The client opens a TCP on (by default) port 22. They both disclose the SSH protocol versions they support
2. They both provide a list of accepted cipher methods
3. The client selects a cipher method and negotiate a *session key*
   a. From now on the rest of the communication is encrypted symmetrically using the *session key*. It provides encryption.

Encrypted connection after

Server authentication

4. The server identifies itself to the client: it sends to the client its *host key*
5. The client sends a challenge to the server
6. The server replies with an authenticator
7. The client checks the authenticator. If all went well the server is authenticated to the client.

Client authentication

8. The client now proceeds to authenticate to the server, either with *password* authentication or with *public key* authentication

Encrypted connection setup

(inspect with wireshark the provided ssh_conn.pcap)

First you can observe the three-way handshake of a TCP connection initiation on port 22

Then both the client and the server send to each other their SSH version and implementation.

(Notice that each packet sent by a node generates a TCP ACK (acknowledgment packet) sent by the other node. You can filter them out by specifying “ssh” in the Wireshark filter)

Then both the client and the server send to each other a list of supported encryption algorithms

The client selects a cipher and initiate the key exchange

Once the key exchange is terminated, both parties have a secret, shared, symmetric key. This key will be used from now on to encrypt the connection

Server authentication

The server send his public key (named *host key*) to the client.

The client is now asked to trust or not trust the host key fingerprint

```
iMac-di-Marco:~ marcospaziani$ ssh marcos@stud.netgroup.uniroma2.it
The authenticity of host 'stud.netgroup.uniroma2.it (160.80.221.14)' can't be established.
ECDSA key fingerprint is SHA256:UKRm/pfFevBVndWv6nAjdOpueHhFacxD5XsiswY3Q20.
Are you sure you want to continue connecting (yes/no)?
```

If yes, the client sends challenge based on the host key. The server replies with authenticator. If the authentication process is successful the server is authenticated to the client

Client authentication - password

For our purposes, we consider **two** methods:

- **password** authentication
- **public-key** authentication

With **password** authentication the client sends the password *as-is* to the server.

Remember that we established an encrypted connection before, therefore a man-in-the-middle cannot spoof the password.

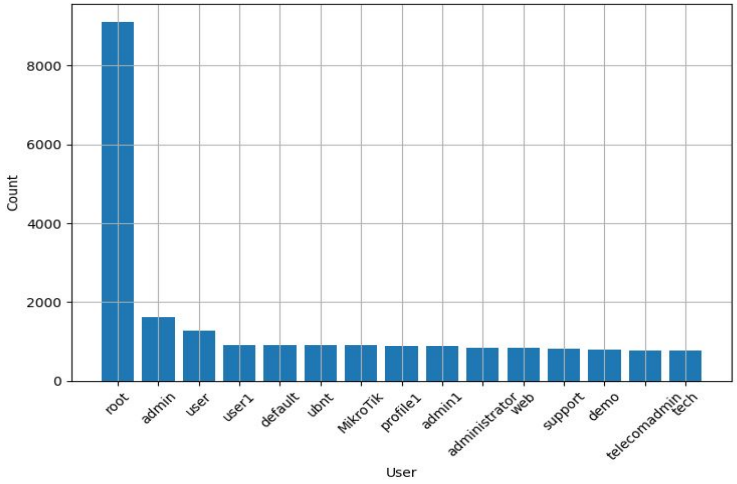
However what happens if the server itself is malicious?

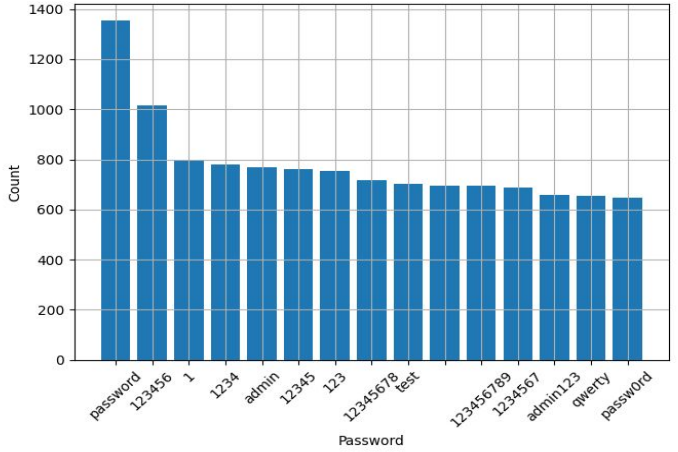
- With performing password authentication you just gifted your password to the attacker

Client authentication - password

On a side note, the internet is full of bots trying to access SSH servers.

Below I reported the most common username and password tentatives on a SSH server with password authentication enabled in a time window of ~72 hours





Client authentication - public-key

If using **public-key** authentication method, the client must posses a public-private key pair.

A classic challenge-response authentication takes place:

1. The server sends a challenge to the client
2. The client sends back to the server the authenticator
3. If all goes well the client is authenticated

Enabling public-key client authentication - 1/3

Okay, so password authentication is **bad,** how can we enable public-key authentication? Three **steps**:

1. **Generate** a new public-private key pair on the client
2. **Transfer** the public key from the client to the server
3. **Authorize** the new key on the server
4. (Optional) **Remove** the possibility of password authentication

Step 1: Generate a key pair on the client

We want to make a step forward and use asymmetric auth.

In the following example pc1 will be the client and pc2 the server.

First of all we have to generate a key pair on the client:

```
$ ssh-keygen
```

A hidden folder inside the home folder of the user, a directory called .ssh is created. Inside that there are 3 files:

```
$ known_host    #Database of the trusted fingerprints
$ id_rsa        #private key generated using RSA
$ id_rsa.pub    #public key generated using RSA
```

We now have to transfer the public key of the client to the list of trusted client keys of the server

Enabling public-key client authentication - 2/3

Step 2: Copy the client public key to the server

We can transfer files between remote hosts using “scp” (Secure CoPy).

```
$ scp local_path user@host_ip:/remote_path
# transfer file at local_path to remote_path over SSH

$ scp user@host_ip:/remote_path local_path
# transfer file at remote_path to local_path over SSH
```

scp is based itself on ssh: it connects to `user@host_ip` and copies the files.

On the client (pc1) we can use the following command to copy its public key to the server (pc2):

```
$ scp .ssh/id_rsa.pub ssh_user@s1:/home/ssh_user/client_key.pub
```

Enabling public-key client authentication - 3/3

Step 3: Authorize the generated key <u>on the server</u>

We have to put the public key of the client into the trusted clients “database” in the server

The “database” is the file `/home/ssh_user/.ssh/authorized_keys` which we have to create

```
$ mkdir .ssh                          #create .ssh folder
$ touch .ssh/authorized_keys          #create authorized_keys file
# Set correct permissions
$ chown ssh_user:ssh_user /home/ssh_user/.ssh
$ chmod 700 .ssh
$ chown ssh_user:ssh_user /home/ssh_user/.ssh/authorized_keys
$ chmod 600 .ssh/authorized_keys
```

After done all of this we can copy the public key of the client to the list of trusted keys of the server, by appending the public key to the authorized_keys file:

```
$ cat client_key.pub >> .ssh/authorized_keys
```

Enabling public-key client authentication

Tip: the previous steps 2 and 3 (copying the key and adding the key to authorized_keys) can be done by:

```
$ ssh-copy-id ssh_user@pc2
```

`ssh-copyid` works by creating a ssh connection to `ssh_user@pc2` and performing the previous steps 2 and 3.

→ Therefore, this method only works if you can already access `ssh_user@pc2` via ssh!

Enabling public-key client authentication

Once public key authentication has been set up, one should remove the option to perform password authentication (must be done on server side).

In the server, there is a file `/etc/ssh/sshd_config` containing configuration options of the ssh server:

```
# To disable tunneled clear text passwords, change to no here!
PasswordAuthentication no
```

And after that restart the ssh daemon via

```
$ /etc/init.d/ssh restart
```

lab_ssh

Consider lab_ssh:

1. Create the lab topology
2. Create a user on S1 and try to login with SSH using password authentication
3. Set up public-key authentication. If set correctly, it should not prompt you for the password anymore
4. Remove the possibility of password authentication in the server

Now you are free to experiment with commands on the interactive terminal, but remember that when delivering the homework all must be configured at startup!

lab_ssh_live

This will work only during the current lecture

On my personal server[1] I've given to each one of the enrolled in the course a user:

Username: your surname (in lowercase, no spaces or ')

Password: your matricola

Task: connect to the server and **setup a public-key authentication for your user**.

Connect to the server from your machine if you have a SSH client on your PC (Linux and MacOS should have an SSH client by default). Or you can use the provided kathara laboratory lab_ssh_live that can connect to the outside.

The server address is given during the lecture.

[1] in reality you are not directly on my server but in a docker container. Anyway, behave well

Port Forwarding

Port Forwarding

- The SSH suite gives you the possibility of performing **port forwarding**
- In the context of SSH, Port Forwarding redirects packets inside an existing SSH connection
- Typical use cases include:
  - Running a public web service inside a LAN
  - Bypassing firewalls filtering specific ports and/or IP ranges
  - Permitting SSH access to a host on the private LAN from the Internet
- SSH gives you the possibility to perform **local** or **remote** port forwarding

Local Port Forwarding - scenario 1

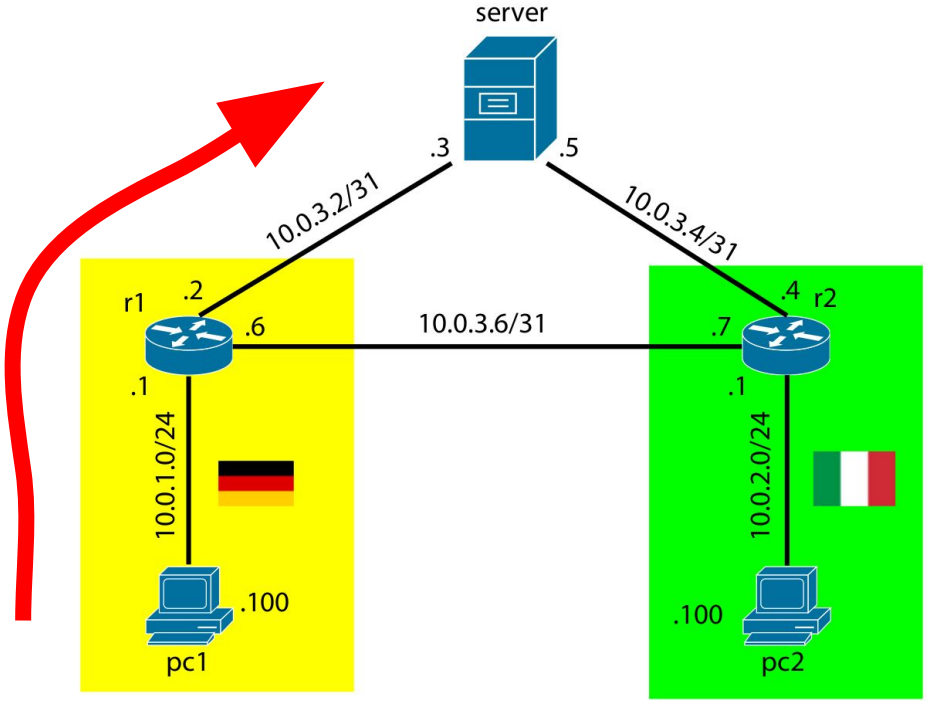
Suppose you are abroad (e.g. Germany) and the server you wish to connect has some regional restrictions (e.g. Netflix) that does not allow you to visit a very important page that you could actually access in Italy.

How to solve that? → SSH **Local Port forwarding**

The only thing we need is that our home PC has an SSH server running and that we can access with a user (not necessarily with asymmetric auth)

Local Port Forwarding - scenario 1

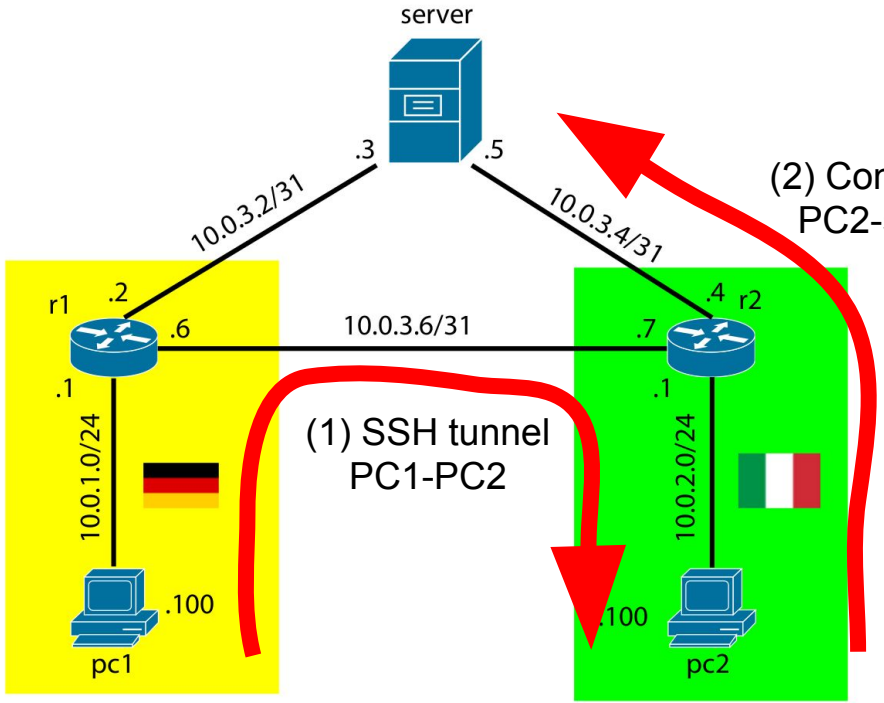
Server **rejects** connections from **Germany**



user: `myuser`
password: `password`

Local Port Forwarding - scenario 1

Server **accepts** connections from **Italy**



user: `myuser`
password: `password`

Local Port Forwarding - scenario 1

On server runs an echo server on TCP port 8080

The server has a firewall blocking all requests coming from Germany subnet: if we launch netcat to connect to the server from pc1 we obtain no response.

We now want to redirect all the traffic directed to the 8080 port of server through an SSH tunnel over 9000 from pc1 to pc2:

```
pc1: $ ssh -NL -v 9000:10.0.3.5:8080 user@10.0.2.100
```

Now on pc1 (*on another terminal*) we can connect to the server with:

```
pc1: $ nc localhost 9000
```

Every packet sent **to** localhost:9000 is redirected **through** the SHH connection and then sent from the SSH host **to** 10.0.3.5:8080

In general, we have:

```
ssh -NL local_p:dest_ip_addr:dest_p user@tunnel_ip
```

Where `-N` no shell and `-L` local port forwarding

(if you see “bind: Cannot assign requested address” it’s not an error, but just a warning. Can be disabled by passing “-4” parameter to ssh command)

Local Port Forwarding - scenario 2

I want to play some multiplayer games instead of following the Network Infrastructures lessons.

The university firewall is blocking the port on which the game runs :(

Can we solve it with local port forwarding?

- Yes.
- Most of the applications that use internet messaging **rely** on TCP/UDP. This includes HTTP(s), emails, and games.
- We can create a **tunnel** using ssh port forwarding

Demo only during lecture, no kathara lab available

Remote Port Forwarding

The concept of **remote port forwarding** is the same of local port forwarding: tunnel a connection inside an SSH connection.

Remote port forwarding opens a listening port on the **remote** host, instead of a local port of local port forwarding

Useful especially to bypass NAT and/or private addresses restrictions

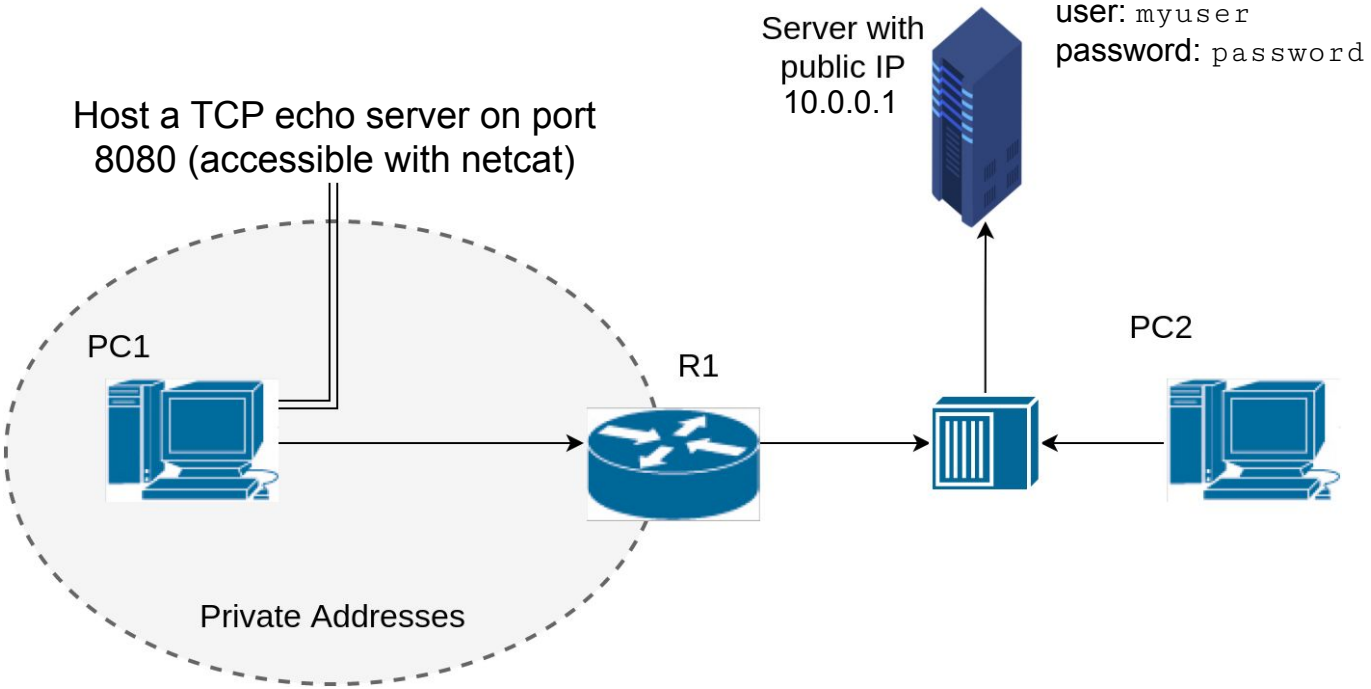
- A host `h1` cannot initiate a connection to `h2` if `h2` has not a public address
- Private addresses are used everywhere (e.g. your home wifi, Sapienza LAN, etc.)

Remote Port Forwarding - scenario

Suppose I am on a LAN without a public IP address (e.g. at home):

- On my PC it's running some web service I am developing (e.g. hosting a web page)
- I want people from outside the LAN to connect to this service
- I have a SSH-enabled server with a public IP

Remote Port Forwarding - scenario



How can PC2 connect to the web service hosted by PC1?

Remote Port Forwarding - scenario

From PC1 we can:

- Initiate a SSH connection
- Tell the server to open a listening port, e.g. 9000
- Tell the server to tunnel every packet destined to its port 9000 to PC1:8000
  - Notice that the server can send packets to PC1 since it uses the SSH connection already in place

This is done by SSH remote port forwarding:

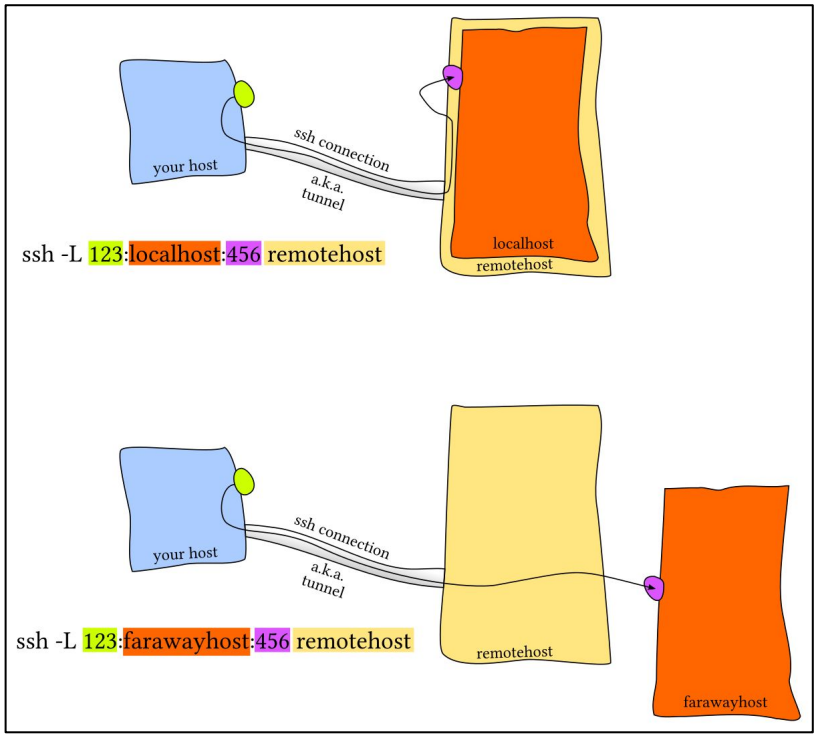
```
pc1: $ ssh -NR 9000:localhost:8080 myuser@10.0.0.1
```

Forwarding

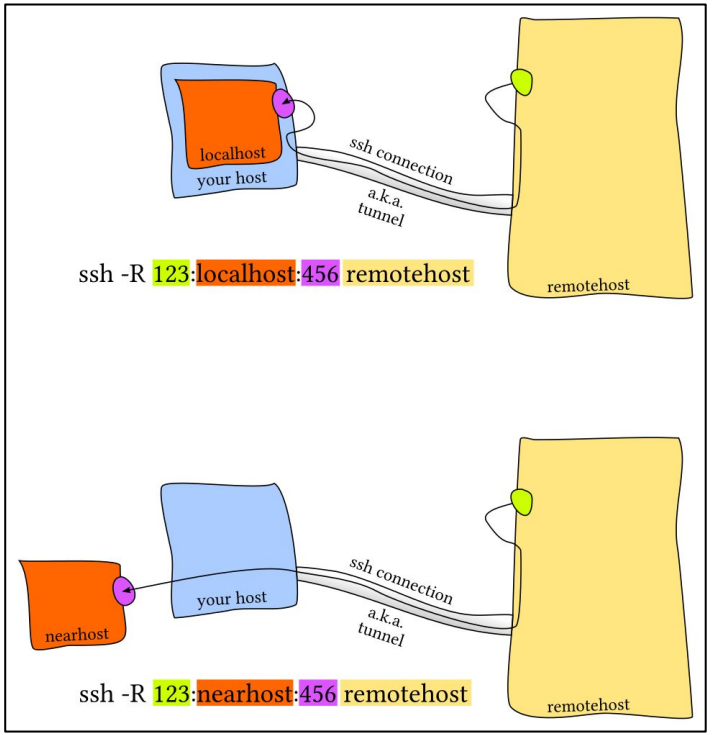
```
ssh -NR remote_p:dest_ip_addr:dest_p user@tunnel_ip
```

By default, the forwarded port on the server listens only on localhost. If we want it to listen on all the available interfaces in the file sshd_config on the server there should be a line: `GatewayPorts yes`

Local port forwarding



Remote port forwarding

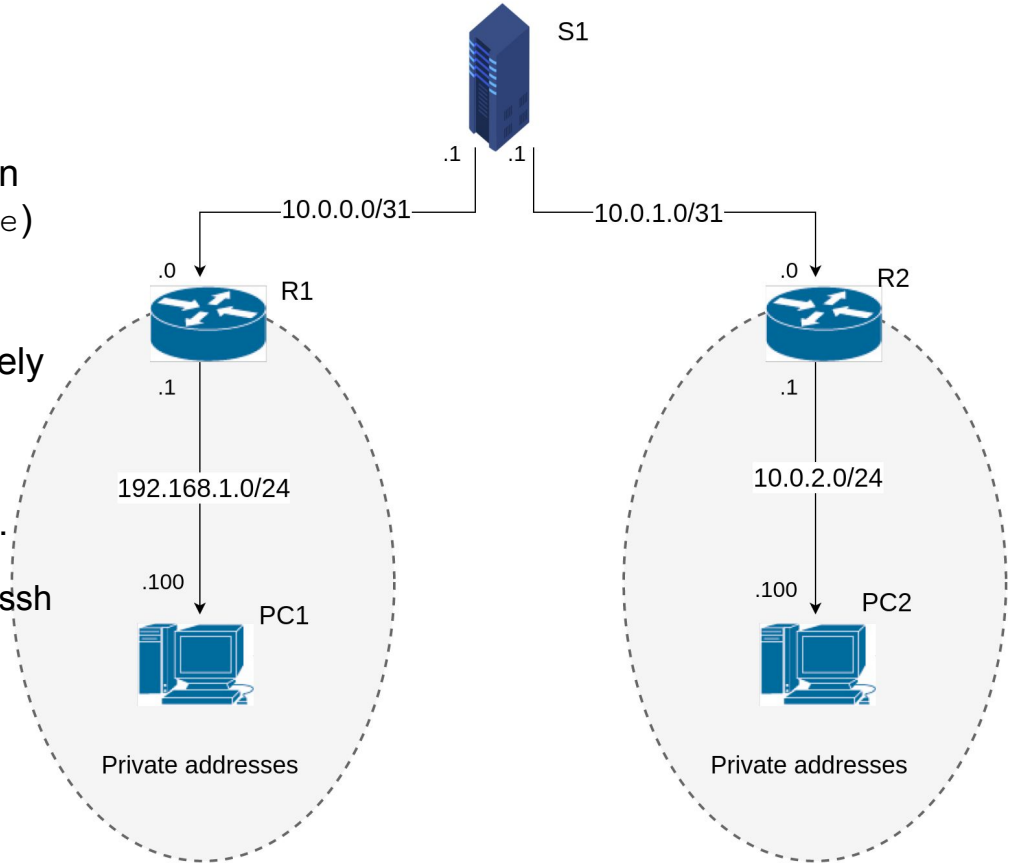


Sketches taken from accepted answer at:
https://unix.stackexchange.com/questions/115897/whats-ssh-port-forwarding-and-whats-the-difference-between-ssh-local-and-remot

Exercise

- The lab is already configured (uploaded in the drive `lab_ssh_forward_exercise`)
- PC1 and PC2 have private IP addresses
- PC1, PC2 and S1 have a user (respectively `pc1user`, `pc2user` and `s1user`) with password: `password`
- **Goal:** connect with ssh from PC1 to PC2.

  i.e. from PC1 you should be able to type an ssh command which opens a terminal on pc2



Exercise

Using the provided laboratory `lab_ssh_forward_exercise`

Goal 1: Set up port forwarding in order to connect with ssh from PC1 to PC2

i.e. from PC1 you should be able to type an ssh command which opens a terminal on pc2

Constraint: DON'T touch r1 and r2, they are not under your control

Info:

- s1 has a public address i.e. everyone can reach it.
- PC1 and PC2 have NAT-ed private addresses i.e. PC1 and PC2 can reach s1 if they start the connection. But if a connection is started from the outside their LAN and directed to PC1 or PC2, it gets discarded.
  - In other words, PC1 and PC2 can start a connection to s1, but s1 cannot start a connection to PC1 or PC2.

Goal 2: Did the previous point work? Ok, now go to r1.startup and r2.startup and in both files de-comment line 11 (remove the hashtag #). This makes r1 and r2 to drop any packet which is not using the default SSH port (port 22). Restart the lab and try again to connect PC1 and PC2 via ssh.