Cybersecurity

Subject: Secure Shell

Authentication policy systems are designed to facilitate access control to the operating system by making it independent of the application code and automating the enforcement of security policy regarding user accounts. PAM (Pluggable Authentication Modules) is an example of a modular authentication policy system widely used in the Unix/Linux environment and has a large number of available extensions. The aim of the exercise is to understand how PAM works through applying sample PAM modules in practice.

1. Secure Shell (SSH)

1.1 SSH protocol

The SSH protocol enables secure access to a remote operating system account using cryptographic authentication and encrypted transmission. Additionally, it is also possible to establish encrypted traffic tunneling between arbitrary TCP ports, which enables application level virtual tunnels creation (connection propagation). Transmitted data may be automatically compressed.

1.2 The ssh client

From the very beginning the ssh program was intended to replace the classic *remote operations* programs, such as rlogin:

```
local> ssh -l username remotehost
```

or rsh:

```
local> ssh -l username remotehost cat /etc/HOSTNAME
```

However, compared to those old-fashioned tools, ssh offers significantly increased security: the entire transmission is encrypted (including the login procedure, and thus the password cannot be eavesdropped in transit), and cryptographic mechanisms can be used to verify the user's identity.

Remote account can be specified in a modern-day format `username@remotehost`, e.g.:

```
local> ssh username@remotehost cat /etc/HOSTNAME
```

A remote access session is started only after successful cryptographic authentication of the remote system by a *challenge-response* method (using e.g. RSA or ECDSA algorithms) and after establishing a symmetric session key (Diffie-Hellman method). Remote system authentication is based on the public-key cryptographic schema, and uses a public keystore, placed in the files /usr/local/etc/ssh_known_hosts (system-wide) and ~/.ssh/known_hosts (user-specific). The secrecy of private keys is sufficient to defend the system against name spoofing, IP spoofing or routing spoofing attacks.

1.2.1 User authentication methods

The ssh program authenticates the user in the following order:

1. cryptographic user authentication using *challenge-response* method
2. classic authentication using password (password is not transmitted in plain text thanks to the encrypted commutation).

Cryptographic authentication uses a set of trusted users' public keys stored in ~/.ssh/authorized_keys. Identity verification using the *challenge-response* method (while keeping the private keys of users secret) enables secure authentication. To enable cryptographic authentication for a remote server, the user's public

key must be set up on the target account (of the remote server) in ~/.ssh/authorized_keys, which can be achieved with the `ssh-copy-id` command.

1.2.2 Cryptographic algorithms

SSH performs cryptographic authentication using asymmetric cryptography algorithms, e.g. RSA or ECDSA. Symmetric cryptographic algorithms are used to encrypt communication, e.g. AES, 3DES, Blowfish or Twofish, and to ensure the integrity of communication—incl. ECDSA-SHA2 or Curve25519-SHA2. These algorithms are selected with the appropriate option of the configuration file ~/.ssh/config (`Ciphers` and `MACs` options). See section 1.6 for additional information about configuration files.

1.2.3 Secure copy

The scp (secure copy) program replaces the corresponding rcp command from the old-fashioned *remote operations* group:

```
local> scp username@remotehost:/etc/HOSTNAME  remotename.txt
```

The scp program supports all of the remote access authentication mechanisms mentioned above.

1.3 Key management

The ssh-keygen program is used to create a cryptographic key pair. It saves the user's private key generated for the RSA algorithm (or ECDSA) in the ~/.ssh/id_rsa (~/.ssh/id_ecdsa) file, and the public key in the ~/.ssh/id_rsa.pub (~/.ssh/id_ecdsa.pub). For additional security, the private key may be saved in a file in a encrypted form. In this case, later access to this key will require a `passphrase`.

```
local> ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (~/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in ~/.ssh/id_rsa.
Your public key has been saved in ~/.ssh/id_rsa.pub.
The key fingerprint is:
5c:13:fc:48:75:19:9b:27:ec:5c:4f:d3:b1:a2:6c:da user@host.domain
```

The `ssh-copy-id` command copies the public key to the specified account on the remote system

1.3.1 System keys

The ssh-keygen program is also used to generate private and public system keys stored in the files /usr/local/etc/ssh_hosts_key and /usr/local/etc/ssh_hosts_key.pub, respectively. In addition, it allows you to manage the known-hosts key database:

– displaying `remotehost`'s system key:

```
local> ssh-keygen -F remotehost
```

– removing `remotehost`'s system key:

```
local> ssh-keygen -R remotehost
```

1.4 Application layer virtual tunnels (TCP port forwarding)

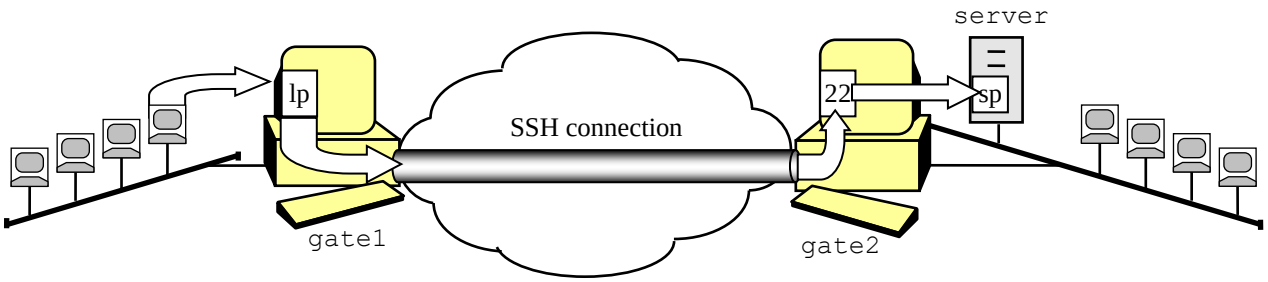
It is possible to build a virtual tunnel between two computers (e.g. BastionHost gateways) passing through an unsecured network (e.g. public). Such tunnel can then be used for secure communication in the application layer between other computers (e.g. a client located in front of the first gateway and the server behind the second gate).

```
gate1> ssh -L [bind_address:]localport:server:serviceport gate2
```

This command creates a tunnel between `gate1` and `gate2` allowing communication over `localport` with the `serviceport` service on the `server` computer (`-L` = Local port forwarding). Communication is encrypted on the `gate1-gate2` section.

`bind_address` – optional address of the local interface to listen (gate `gate1`)
`localport` – local system port (gate `gate1`)
`server` – destination domain name or IP address of the remote-side computer (behind `gate2`) whose port will target the virtual tunnel traffic
`serviceport` – the actual remote `server`'s port

Connection forwarding through the virtual tunnel is presented in the diagram below:

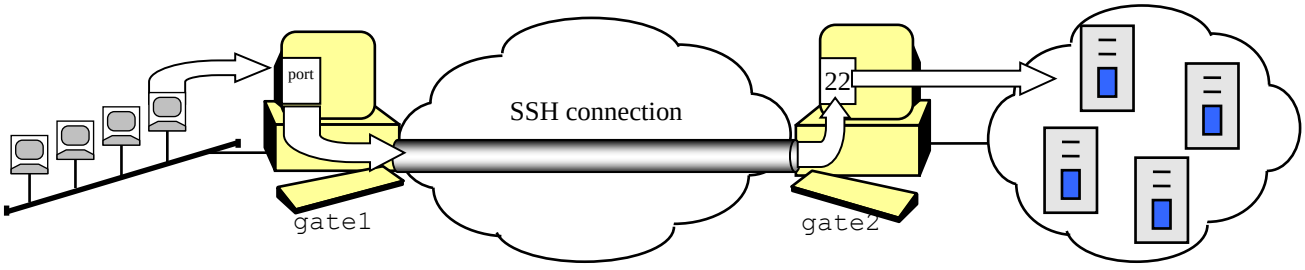


Identical result can be achieved by setting up the tunnel from the opposite (remote) side:

```
gate2> ssh -R [bind_address:]remoteport:server:serviceport gate1
```

`-R` = Remote port forwarding
`bind_address` – optional address of the remote interface to listen (gate `gate1`)
`remoteport` – remote system TCP port (gate `gate1`)
`server` – name (domain) or address (IP) of the computer whose port will be made available via the virtual tunnel (this time in front of `gate2`)
`serviceport` – TCP port on `server`

Application layer virtual tunneling can also use SSH as a full application proxy (SOCKS):



Such effect can be achieved by using command:

```
gate1> ssh -D [bind_address:]localport gate2
```

`-D` – application-level port forwarding
`localport` – local system TCP port (gate `gate1`)

1.5 Network layer virtual tunnels (VPN)

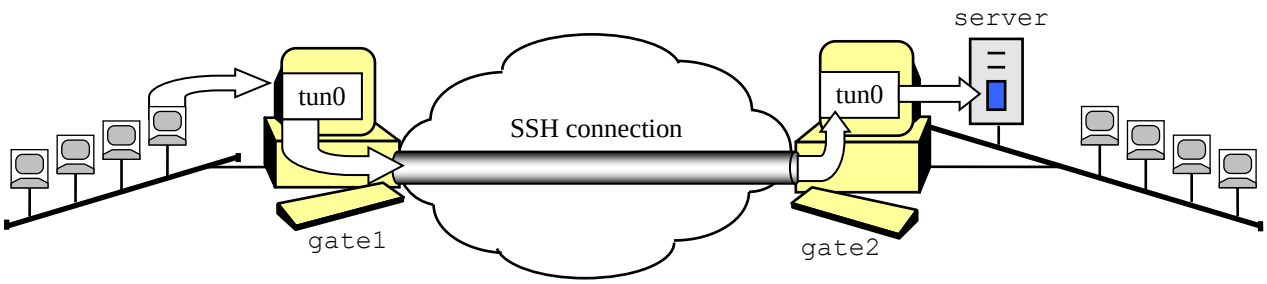
SSH also allows to build a network layer virtual tunnel between two computers (e.g. BastionHost gateways) passing through an unsecured network (e.g. public). This tunnel can then be used for secure communication at the IP protocol layer between other computers (e.g. a client in front of the first gateway and a server behind the second gate).

```
gate1> ssh -w local_tun[:remote_tun] gate2
```

This command will create a VPN tunnel between `gate1` and `gate2` that will allow communication to be tunneled between the virtual `local_tun` and `remote_tun` interfaces. Additionally, you should define IPv4 or IPv6 addresses on these interfaces and appropriate packet routing. Communication is encrypted on the `gate1`-to-`gate2` route. In order to work properly, an SSH connection must first be established between both systems.

- `local_tun` – local-side virtual interface number (or `any` = nearest free virtual interface number)
- `remote_tun` – remote-side virtual interface number

Network layer tunneling is presented in the diagram below:



1.6 Automation through configuration file

It is possible to configure connection profiles. These profiles should be saved in ~/.ssh/config file.

The profile configuration file has the following structure:

```
Host profile_name
  Hostname system_adress
  User user_name
  Port port_number
  LocalForward localport server:serviceport
  RemoteForward remoteport server:serviceport
  DynamicForward localport
Host next_profile_name
…
```

Starting a profile defined in this way is done using command `ssh profile_name`. `Hostname` is redundant if the profile name matches the name of the remote system.

It is possible to use wildcards in the configuration file. For example, you can define a profile for all systems belonging to a specific domain:

```
Host *.cs.put.poznan.pl
  User jbond
  Port 55555
```

The profile above will be automatically applied on any further `ssh unixlab.cs.put.poznan.pl` command, for instance.

1.6.1 Proxy jump

An interesting option is the ability to define proxy commands in the configuration file:

```
Host host3
  ProxyCommand ssh -A -q -e none -W host3:22 host2
```

The `ssh host3` command will establish a tunnel to `host2` and through that tunnel a SSH connection to `host3` will be established.

It is perfectly possible to combine wildcards and the proxy command as follows:

```
Host *.cs.put.poznan.pl
  ProxyCommand ssh -q -e none -A -W %h:22 gate3
```

Here, the execution of `ssh unixlab.cs.put.poznan.pl` command will automatically set up a tunnel to `gate3` and—through that tunnel—another connection to unixlab.cs.put.poznan.pl.

📖 Further reading:

OpenSSH https://www.ssh.com/ssh/openssh/

PuTTY https://www.ssh.com/ssh/putty/

ssh-keygen https://www.ssh.com/ssh/keygen/

🗒 Problems to discuss:

- What encryption algorithm is the user's private key file encrypted with?
- Is it necessary to enter the password (`passphrase`) protecting this file with every access attempt?