4. Web Server Administration

Miquel Àngel París i Peñaranda

Web Application Deployment

2nd C-VET Web Application Development

CC BY NC SA

Index of contents

1. Goals....3
Introduction....4
Installing OpenSSH....4
Remote Connection....4
SSH Server Configuration....5
User Authentication....6
Authentication with Encryption Keys....7
X11 Forwarding....8
Secure File Transfer....9
TCP Port Forwarding / SSH Tunneling....10
Virtual Private Network (VPN)....13
Web Links....17

1. Goals.

1. Install and configure web servers on virtual machines and/or the cloud.
2. Perform functional tests on web and application servers.
3. Document the installation and configuration processes performed on web and application servers.

Introduction.

The definition of SSH (Secure Shell), according to Wikipedia, is a protocol and program whose main function is to provide remote access to a server using a secure communication channel where the transmitted information is encrypted. In other words, SSH replaces the Telnet service, which transmits information in plain text. Additionally, this service allows secure file exchange using encrypted FTP, key management, tunnel creation, and X-Windows session connections to execute graphical programs on remote systems.

The SSH service is a standard protocol available in different implementations and for various operating systems. Among these, the most popular is the OpenSSH project.

Installing OpenSSH

This service facilitates communication between a client and a server. To securely access systems remotely, we must install the server application with the following command:

```
# apt install openssh-server
```

During this process, the applications for remote access and secure file exchange are installed, and the necessary public and private keys for establishing secure connections with service clients are generated. Specifically, the following components are installed: `openssh-server`, `openssh-sftp-server`, and `ssh-import-id`.

The SSH client application is usually installed by default.

The service can be managed with the following commands to check its status, start, stop, or restart it:

```
# systemctl start [ stop | restart | status ] ssh
# service ssh start [ stop | restart | status ]
```

The process associated with this service is called `sshd`.

Remote Connection

As mentioned earlier, the `ssh` application replaces `telnet`. Both services have a client-server architecture, but while `telnet` uses port 21, `ssh` uses port 22.

When a connection is established, you access a terminal on the remote system. The difference between the two is that `ssh` ensures secure communication through encryption. If communication is intercepted, its content cannot be recovered.

To establish a remote connection, execute the following on the client:

```
$ ssh [options] IP_address_or_remote_system_name
```

The first time you connect to a remote system, the system will display its fingerprint and ask whether you want to proceed. If you agree, this information is saved in the `~/.ssh/known_hosts` file and won't prompt you again unless the fingerprint changes or the entry is removed.

One commonly used option is to specify the user account for logging into the remote system, for which you need to know the password:

```
$ ssh [options] remote_user@IP_or_remote_system_name
$ ssh -l remote_user IP_or_remote_system_name
```

Now you have access to a terminal on the remote system, and commands will be executed on it.

Both of the above commands are equivalent.

To terminate the connection, type `exit`, and you will return to the local terminal where the remote session was started.

SSH Server Configuration

The basic configuration of an OpenSSH server is simple and is located in the file `/etc/ssh/sshd_config`. Below are a few key parameters:

1. **Port:** The default port for the service is 22, but you can change it to another free port that suits your organization's needs.
2. **ListenAddress:** This specifies the IP address the service listens on. By default, it allows access from all network interfaces (`0.0.0.0`). You can restrict it to specific addresses.
3. **Logging**

- **SysLogFacility**: Determines the log code (default is `AUTH`), which logs events in `/var/log/auth.log`. You can change this to other values like `DAEMON`, `USER`, or `LOCAL0` to `LOCAL7`.
- **LogLevel**: Sets the importance level of events to log. The default is `INFO`, which logs many service-related events. You can change it to higher levels like `NOTICE` or `WARN` to reduce the logged information.

4. **PermitRootLogin:** Controls whether the root user can log in remotely. By default, this is set to `prohibit-password`, which prevents root access with passwords. It is safer to log in as a normal user and use `sudo` for administrative tasks. Setting this to `yes` allows root login, which can be risky. If needed, use certificate-based access for root.
5. **MaxSessions:** Limits the number of simultaneous sessions.
6. **AllowUsers/DenyUsers:** Specifies lists of users allowed or denied remote access. Similarly, **AllowGroups/DenyGroups** can be used for groups.
7. **PubKeyAuthentication:** Enables or disables public key authentication (default is `yes`).
8. **X11Forwarding:** Setting this to `yes` allows remote access to graphical applications. (This is discussed later in the document.)
9. **PermitTunnel:** Configures the creation of communication tunnels (options: `yes`, `no`, or `point-to-point`).
10. **Secure FTP:** The `Subsystem sftp /usr/lib/openssh/sftp-server` option enables secure file transfer.

You can view the complete list of configuration options using the command:

```
$ man sshd_config
```

User Authentication

As mentioned earlier, you can establish a connection to a remote server or secure FTP session (`sftp`) by specifying the username of the system you want to connect to and its password. This is the simplest way to authenticate using `ssh`. However, authentication can also be achieved using

encryption keys or by defining a list of allowed computers for remote connections to your SSH server.

Authentication with Encryption Keys

To use this type of authentication, encryption keys must first be generated. These keys are a pair: a public key and a private key. The public key can be shared without any problem with anyone who needs to encrypt information to send to you. The private key, on the other hand, must be kept in a secure location and is used to decrypt the information encrypted with your public key. It's impossible to decrypt data with the public key that was encrypted with the same public key.

The authentication process follows these steps:

1. **Securely transfer the public key to the remote server.** This is done only once.
2. **Initiate the connection:** Use the `ssh` command with the username and IP or server name.
3. **Challenge-response mechanism:**
    - The server generates a random number, encrypts it with your public key, and sends it to you.
    - Your system uses the private key to decrypt the number and sends it back to the server.
    - If your response matches the server's generated number, access is granted because only the correct private key could decrypt the number.

This mechanism is known as a "challenge" or "challenge-response" validation. The steps described above are handled automatically by the `ssh` and `sshd` processes.

To generate the key pair, use the following command:

```
$ ssh-keygen -t rsa
```

This creates both public and private keys. You can optionally protect the private key with a password, which will be required whenever the key is used. However, in some cases (e.g., automated processes), it's not recommended to set a password. In such cases, the file containing the private key should be secured with appropriate access permissions.

The command generates two files stored in the `.ssh` directory within your home directory:

- **Public key:** `id_rsa.pub`
- **Private key:** `id_rsa`

You can change the names when creating the keys if needed.

Next, transfer the public key to the server you wish to access remotely. This key must be copied into the `.ssh` directory of the target user's home directory on the server, specifically into the `authorized_keys` file.

There are several ways to copy the public key to the server. The simplest method is to use the `ssh-copy-id` command, which automates the process in one step:

```
$ ssh-copy-id -i .ssh/id_rsa.pub remote_user@IP_or_server_name
```

This command will prompt for the remote user’s password to perform the operation.

Once the key is transferred, you no longer need to provide the remote user’s password (nor the private key password, if one was not set in this example).

If you’ve set a password on the private key, you can use the `ssh-agent` process to manage it. This allows the system to use the private key password automatically when needed. Use the following commands:

1. Start `ssh-agent` and add the private key:

   ```
   $ ssh-add [file]
   ```

   If no file is specified, it defaults to `.ssh/id_rsa`.

2. When connecting to a remote system, you’ll initially be prompted for the passphrase. Afterward, `ssh-agent` will manage it automatically.

X11 Forwarding

This feature allows you to remotely execute graphical applications. Normally, when a remote connection is established, you gain access to a terminal where console applications can be executed on the remote system. However, by enabling X11 forwarding, graphical applications can also be run remotely.

Configuration:

1. **On the SSH server:** In `/etc/ssh/sshd_config`, enable:

```
X11Forwarding yes
```

2. **On the SSH client:** In `/etc/ssh/ssh_config`, enable:

```
ForwardX11 yes
```

Usage:

To use X11 forwarding, execute `ssh` with the `-X` option:

```
$ ssh -X user@SSH_server
```

In the remote session, run the graphical application. For example:

```
$ firefox &
```

Requirements:

For this to work, the client system must have X-Windows and a window manager installed to display the graphical application. If the client does not have these installed, you can set them up with the following command:

```
# apt-get install xorg fvwm
```

To start the X-Windows session on the client when needed, execute:

```
$ startx
```

Secure File Transfer

OpenSSH provides functionality for secure file transfer between local and remote systems. Two commands are available for this purpose:

1. **`scp` (Secure Copy):** Works like the Linux `cp` command but allows copying files between local and remote systems securely.

   Syntax:

```
$ scp path/to/file user@SSH_server:path/to/destination
```

```
$ scp user@SSH_server:path/to/file path/to/local/destination
```

The security level applied is the same as for `ssh`. It will prompt for a password or passphrase if needed.

2. **`sftp` (Secure FTP):** Functions like traditional `ftp` but within a secure environment.

   Usage:

   ```
   $ sftp [user@]SSH_server
   ```

   This opens an interactive session for file exchange in both directions. The functionality is similar to the standard FTP protocol but uses encryption for secure communication.

TCP Port Forwarding / SSH Tunneling

This functionality enables the redirection (forwarding) of TCP ports associated with applications from the client system. It can be used to add encryption for older applications (e.g., POP3), bypass firewalls, or allow system administrators to create backdoors for accessing internal networks remotely (e.g., from home).

However, port forwarding should be used cautiously, as it can also be exploited to launch attacks on the network from outside.

Port forwarding can be performed in both directions of the secure SSH connection.

Local Port Forwarding

In this case, a port on the client system is forwarded to an external destination (either the SSH server itself or another system accessible from the server).

The client listens for requests on the specified local port, forwards them securely to the SSH server, and then routes them to the target system's specified port. If the SSH server is not the final destination, it acts as a **jump server**.

Command syntax:

```
$ ssh -L [local_IP]:local_port:destination_system:remote_port SSH_server
```

- **`-L`**: Indicates local port forwarding (from the client system).

- **`local_IP`** (optional): The name or IP of the client. If set to `127.0.0.1`, only the client system can use the forwarded port. Omitting this allows any system to connect to the client's forwarded port. To avoid unauthorized access, it is recommended to specify `127.0.0.1`.
- **`local_port`**: The port number on the client that will be forwarded.
- **`destination_system`**: The name or IP of the target system where the forwarded traffic will be sent.
- **`remote_port`**: The port number on the destination system to forward the traffic to.
- **`SSH_server`**: The name or IP of the SSH server that creates the tunnel between the client and the destination system.

Example: In the following scenario, a local port (8001) is forwarded:

- **Destination system:** `172.22.21.16` on port `8080`
- **SSH server:** `10.20.30.136`

Run this command:

```
$ ssh -L 127.0.0.1:8001:172.22.21.16:8080 10.20.30.136
```

Now, accessing `127.0.0.1:8001` on the client opens the application running on the remote system (`172.22.21.16`) at port `8080`. The connection appears as though the application is local.

Connection Diagram:

```
Client (127.0.0.1:8001) → SSH Server (10.20.30.136) → Destination System
(172.22.21.16:8080)
```

Remote Port Forwarding

In this case, a port on the SSH server is forwarded to a port on the client system.

Command syntax:

```
$ ssh -R remote_port:local_system:local_port SSH_server
```

- **`-R`**: Indicates remote port forwarding (from the SSH server).
- **`remote_port`**: The port number on the SSH server where connections are accepted and forwarded to the client system.

- **`local_system`**: The name or IP of the client system where the forwarded traffic will be sent. This can also be `localhost` or `127.0.0.1`.
- **`local_port`**: The port number on the client system to forward the traffic to.
- **`SSH_server`**: The name or IP of the SSH server.

Dynamic Port Allocation:

- If `remote_port` is set to `0`, the server dynamically selects an available port and informs the client.
- Use the `-O forward` option to display the assigned port.

Example: Forward traffic from a remote port (8002) on the SSH server to:

- **Client system:** `172.22.21.16` on port `80`
- **SSH server:** `10.20.30.136`

Run this command:

```
$ ssh -R 8002:172.22.21.16:80 10.20.30.136
```

Now, accessing `10.20.30.136:8002` opens the application running on the client system (`172.22.21.16`) at port `80`.

Connection Diagram:

```
Remote System (10.20.30.136:8002) → SSH Server (10.20.30.136) → Client
System (172.22.21.16:80)
```

Configuration in `sshd_config`

To enable port forwarding on the SSH server, the following options can be configured in `/etc/ssh/sshd_config`:

- **AllowTcpForwarding**:
    - `yes` or `all`: Allows port forwarding (default).
    - `no`: Disables port forwarding.
    - `local`: Only allows local port forwarding.
    - `remote`: Only allows remote port forwarding.

- **GatewayPorts**:
    - `no`: Blocks access to forwarded ports from other systems (default).
    - `yes`: Allows any system (even on the internet) to access forwarded ports.
    - `clientspecified`: The client specifies which IPs can connect to the forwarded port. Example:

```
$ ssh -R allowed_IP:remote_port:local_system:local_port
SSH_server
```

Only the specified IP (`allowed_IP`) will be able to connect to the remote port through the SSH server.

Virtual Private Network (VPN)

A Virtual Private Network (VPN) is a technology that connects two remote networks using an insecure intermediate network (e.g., the internet) while ensuring that the transmitted data is secure. From the user's perspective, it appears as though the two networks are directly connected as a point-to-point link.

VPNs can be implemented using a variety of solutions, including both hardware- and software-based options. These implementations can operate at different levels of the OSI model, mainly at Levels 2 and 3, but also at Levels 4 and 7. Because of this, VPNs are fundamental to modern networking, and the range of available solutions is vast to suit different needs.

VPNs with OpenSSH

OpenSSH enables the creation of a Level 3 (Network Layer) VPN using a tunneling method. This involves encapsulating data packets within other network packets to securely transmit them between the VPN's endpoints. Here's how it works:

1. Data packets generated at the network layer are encrypted.
2. These encrypted packets are encapsulated in new network layer packets.
3. The encapsulated packets are transmitted across the insecure intermediate network.
4. At the destination, the VPN interface extracts and decrypts the original packets and forwards them to their intended target.

Considerations for OpenSSH VPNs

OpenSSH uses the **TCP protocol** for communication. This means the VPN involves encapsulating **TCP packets within TCP packets**, which can lead to certain problems:

- When network issues arise, TCP's reliable delivery mechanisms (retransmissions and acknowledgments) can create conflicts between the inner and outer TCP layers, potentially causing significant performance degradation or even collapse.

For these reasons, while OpenSSH can create simple and occasional VPNs, it is not recommended for permanent or high-traffic VPN implementations. Alternatives like **OpenVPN** (based on TCP over UDP) are more suitable for such cases. You can learn more about OpenVPN at OpenVPN.net.

Configuring a VPN with OpenSSH

Here's how to configure a basic VPN using OpenSSH:

1. **On the SSH Server:**
    - Enable the root user, as this user is not typically activated. For this:

      ```
      # passwd root  # Set a password for the root user
      # mkdir /root/.ssh
      ```

    - Edit the SSH server configuration file (`/etc/ssh/sshd_config`) and set the following:

      ```
      PermitTunnel yes
      PermitRootLogin yes
      ```

2. **On the Client:**
    - Generate a key pair and transfer the public key to the server:

      ```
      # ssh-keygen -t rsa  # Leave the passphrase blank
      # cd .ssh
      # ssh-copy-id -i id_rsa.pub user@SSH_server
      ```

3. **Creating the VPN Tunnel:**
    - OpenSSH uses a virtual network interface called `tun`. Ensure the `tun` module is enabled in the kernel. If necessary, load the module:

```
# modprobe tun
```

- Establish the VPN tunnel from the client:

```
# ssh -f [-C] -w 0:0 SSH_server_IP -N
```

  - `-f`: Puts the SSH session in the background.
  - `-C`: Enables compression (optional).
  - `-w 0:0`: Specifies the tunnel interface pair (e.g., `tun0` on both ends).
  - `-N`: Prevents commands from being executed, keeping the connection open only for tunneling.

- Configure the client-side `tun0` interface:

```
# ifconfig tun0 10.10.10.2 pointopoint 10.10.10.1 netmask
255.255.255.252
```

4. **On the Server:**

   - Load the `tun` module if needed:

```
# modprobe tun
```

   - Configure the server-side `tun0` interface:

```
# ifconfig tun0 10.10.10.1 pointopoint 10.10.10.2 netmask
255.255.255.252
```

How It Works

The above steps create a network between two hosts using the `tun0` interfaces. The traffic between these hosts is encrypted and encapsulated within standard TCP packets sent over the intermediate network. If the packets are intercepted, the internal encrypted data remains secure.

Example Usage:

- On the client, traffic sent through the `tun0` interface with destination `10.10.10.1` (the server's tunnel IP) is encrypted and transmitted to the SSH server.
- On the server, this traffic is decrypted and sent through the `tun0` interface to its final destination.

Capture Example:

- On the client, you'll see traffic being encrypted and sent through the tunnel interface.
- On the server, decrypted traffic is delivered to the appropriate network or application.

Web Links

In this section, you will find the relevant links of interest necessary to expand and explore the contents of the unit.

- Secure Shell – Wikipedia
- OpenSSH
- Business VPN For Secure Networking | OpenVPN