

System Administration (2)

IT401

Prepared by
Dr. Ahmed El-Douh

Lecture 6



System Administration 2

Lecture 6

IT401

Controlling Daemons and Securing OpenSSH

Prepared by
Dr. Ahmed El-Douh

Objectives

Upon completion of this lecture, you will be able to:

- Manage and control system services and daemons using command-line tools.
- Configure and secure the OpenSSH service for remote access.
- Troubleshoot, enable, disable, and protect services to ensure secure system operation

Outlines

- Controlling Services and Daemons
- Configuring and Securing OpenSSH Service



Controlling Services and Daemons



Identifying Automatically Started System Processes

Objective

- List **System Daemons** and **Network Services**.
- Identify services managed by **systemd** and **Socket Units**.

Introduction to systemd

- The primary **System and Service Manager** in modern Linux.
- Manages system **startup** and **server processes**.
- Activates **system resources** at **boot time** and during runtime.

Key Concepts

- **Daemons:** Background processes that perform automatic tasks.
 - Start automatically at boot (Names often end in 'd').
- **Sockets:** Primary communication channels used by Daemons to listen for connections.
- **Socket Units:** A **systemd** mechanism to manage **Sockets** separately from **Daemons**.
 - Improves **system startup** speed by only starting the daemon when a connection is received.

A Bit of History: From init to systemd

The Traditional Init System:

- For many years, **Process ID 1** was the **init process**.
- Used **System V** and **LSB init scripts** to start frequently used daemons at boot.
- These scripts are shell scripts that varied across distributions.
- Less frequent daemons started **on-demand** using tools like **initd** or **xinetd**.
- Traditional init systems had several limitations.

The systemd Era:

- In modern Linux e.g. RHEL 7, **Process ID 1** is **systemd**, the new init system.
- **systemd provides key new features:**
 - **Parallelization capabilities** which increase the boot speed of a system.
 - **On-demand starting** of daemons without requiring a separate service.
 - **Automatic service dependency management** to prevent long timeouts.
 - Tracking related processes using **Linux control groups**.

Systemctl and systemd Units

❖ Managing systemd

- The **systemctl** command is used to manage **systemd objects**.
- These objects are called **Units**.
- Use `systemctl -t help` to see a list of available unit types.

❖ Common systemd Unit Types

- **Service Units** (`.service` extension):
 - Represent system services like a web server.
 - Used to start frequently accessed daemons.
- **Socket Units** (`.socket` extension):
 - Represent Inter-Process Communication IPC sockets.
 - Used to delay service start at boot time.
 - Starts services on demand, similar to `xinetd`.
- **Path Units** (`.path` extension):
 - Delay service activation until a specific file system change occurs.
 - Common for services using spool directories, such as a printing system.

Systemctl and systemd Units

❏ Viewing Service Status

- Use the command: `systemctl status name.type`.
- If you omit the **unit type**, `systemctl` will assume it is a **service unit**.
- The command output provides the complete status and log information.

❖ Key Status Fields

- **Loaded:** Shows if the unit file is loaded and enabled/disabled.
- **Active:** Shows the current state of the service.
 - Example: **active (running)** indicates the service is running.
- **Main PID:** The Process ID of the service's main process.
- **CGroup:** The control group path for all related processes.
- **Log Output:** Recent log entries generated by the service.

➤ Example Command

- `systemctl status sshd.service`

Systemctl and systemd Units

```
[root@serverX ~]# systemctl status sshd.service
```

```
sshd.service - OpenSSH server daemon
```

```
Loaded: loaded (/usr/lib/systemd/system/sshd.service; enabled)
```

```
Active: active (running) since Thu 2014-02-27 11:51:39 EST; 7h ago
```

```
Main PID: 1073 (sshd)
```

```
CGroup: /system.slice/sshd.service
```

```
└─1073 /usr/sbin/sshd -D
```

```
Feb 27 11:51:39 server0.example.com systemd[1]: Started OpenSSH server daemon.
```

```
Feb 27 11:51:39 server0.example.com sshd[1073]: Could not load host key: /et...y
```

```
Feb 27 11:51:39 server0.example.com sshd[1073]: Server listening on 0.0.0.0 ....
```

```
Feb 27 11:51:39 server0.example.com sshd[1073]: Server listening on :: port 22.
```

```
Feb 27 11:53:21 server0.example.com sshd[1270]: error: Could not load host k...y
```

```
Feb 27 11:53:22 server0.example.com sshd[1270]: Accepted password for root f...2
```

```
Hint: Some lines were ellipsized, use -l to show in full.
```

Systemctl and systemd Units

Several keywords indicating the state of the service can be found in the status output:

KEYWORD:	DESCRIPTION:
loaded	Unit configuration file has been processed.
active (running)	Running with one or more continuing processes.
active (exited)	Successfully completed a one-time configuration.
active (waiting)	Running but waiting for an event.
inactive	Not running.
enabled	Will be started at boot time.
disabled	Will not be started at boot time.
static	Can not be enabled, but may be started by an enabled unit automatically.

Listing Unit Files with systemctl

- The **systemctl** command automatically **paginates** the output using the **less** utility.

❖ Command Examples:

1. Query the state of all units:

- Command: **systemctl**
- Purpose: Verify the overall system startup and status.

2. Query the state of only the service units:

- Command: **systemctl --type=service**
- Purpose: Filter the output to show only **Service Units**
e.g. **.service** files.

Query the state of all units to verify a system startup.

```
[root@serverX ~]#  
systemctl
```

Query the state of only the service units.

```
[root@serverX ~]#  
systemctl --type=service
```

Listing Unit Files with systemctl

3. Detailed Status Investigation

- Investigate failed or maintenance units:
 - Command: `systemctl status rngd.service -l`
 - The `-l` option shows the full log output.

```
[root@serverX ~]#  
systemctl status rngd.service -l
```

4. Checking Specific States

- The `Status` argument shows the active and enabled states.
- Alternate Commands for Specific States:
 - Check Active State: `systemctl is-active sshd`
 - Check Enabled State: `systemctl is-enabled sshd`

```
[root@serverX ~]#  
systemctl is-active sshd  
[root@serverX ~]#  
systemctl is-enabled sshd
```

5. Listing and Filtering Units

- List the active state of all loaded units:
 - Command: `systemctl list-units --type=service`
 - the `--all` option to show inactive units as well.
 - Command: `systemctl list-units --type=service --all`

```
[root@serverX ~]#  
systemctl list-units --type=service  
[root@serverX ~]#  
systemctl list-units --type=service --all
```

Listing Unit Files with systemctl

6. View enabled and disabled settings for all units:

- Command: `systemctl list-unit-files --type=service`
- This output shows the **enabled** or **disabled** status for each unit.

```
[root@serverX ~]#  
systemctl list-unit-files --type=service
```

7. View only failed services:

- Command: `systemctl --failed --type=service`
- The `--failed` option filters the unit list to show only units that are in a **failed** state.

```
[root@serverX ~]#  
systemctl --failed --type=service
```



Controlling System Services



Controlling System Services

❑ Service Management Actions

- **Restarting:** Necessary after configuration file changes or updates.
- **Stopping:** Used before removing software or when a service is no longer needed.
- **Manual Starting:** An administrator may start a service on-demand if it is infrequently used.

Example Steps:

1. View the status of a service.

```
[root@serverX ~]#  
systemctl status sshd.service
```

2. Verify that the process is running.

```
[root@serverX ~]#  
ps -up PID
```

Controlling System Services

3. Stop the service and verify the status.

```
[root@serverX ~]#  
systemctl stop sshd.service  
[root@serverX ~]#  
systemctl status sshd.service
```

4. Start the service and view the status. The process ID has changed.

```
[root@serverX ~]#  
systemctl start sshd.service  
[root@serverX ~]#  
systemctl status sshd.service
```

Controlling System Services

5. Stop, then start, the service in a single command.

```
[root@serverX ~]#  
systemctl restart sshd.service  
[root@serverX ~]#  
systemctl status sshd.service
```

6. Issue instructions for a service to read and reload its configuration file without a complete stop and start. The process ID will not change.

```
[root@serverX ~]#  
systemctl reload sshd.service  
[root@serverX ~]#  
systemctl status sshd.service
```

Unit Dependencies



Unit Dependencies

❖ Dependency Activation

- Services may be started as **dependencies** of other units.
- **Socket Units** can automatically start a service when a network request is made.
- **Path Units** can start a service when a specific file system condition is met.

Example: A file in the **print spool directory** triggers the **cups.service**.

Unit Dependencies

❖ Stopping Dependent Services

- To completely stop a service, all its dependent units **must be stopped**.
- Disabling a service will automatically disable its dependencies.

```
[root@serverX ~]# systemctl stop cups.service
Warning: Stopping cups, but it can still be activated by:
  cups.path
  cups.socket
```

❖ Listing Dependencies (systemctl list-dependencies)

- Command: `systemctl list-dependencies UNIT`
- Shows a tree of other units that must be started.

❖ Reverse Dependency:

- Use the `--reverse` option to show which units **depend on** the specified unit.

Masking Services

❏ Purpose of Masking

- Used to prevent an **administrator** from accidentally starting a conflicting service.
- Relevant when multiple methods exist to manage a resource e.g., networks, firewalls.

How Masking Works

- Masking creates a symbolic link in the configuration directories.
- This link points the service file to `/dev/null`.
- If anyone tries to start the masked service, **nothing will happen**.

Masking Prevent start:

- Command: `systemctl mask network`

Unmasking Allow start:

- Command: `systemctl unmask network`

```
[root@serverX ~]# systemctl mask network
ln -s '/dev/null' '/etc/systemd/system/network.service'
[root@serverX ~]# systemctl unmask network
rm '/etc/systemd/system/network.service'
```

Enabling Services at Boot

✓ Key Concept

- Starting/Stopping a service manually does **not** guarantee its state after a **reboot**.
- Services are started at **boot time** via symbolic **links** managed by **systemctl**.

❏ Service Management Actions:

1. View the status of a service:

Command: `systemctl status sshd.service`

```
[root@serverX ~]#
systemctl status sshd.service
```

2. Disable the service at boot:

Command: `systemctl disable sshd.service`

Note: Disabling a service does **not** stop it immediately.

```
[root@serverX ~]#
systemctl disable sshd.service
[root@serverX ~]#
systemctl status sshd.service
```

3. Enable the service at boot:

Command: `systemctl enable sshd.service`

Verification: `systemctl is-enabled sshd.service`

```
[root@serverX ~]#
systemctl enable sshd.service
[root@serverX ~]#
systemctl is-enabled sshd.service
```

SUMMARY OF `systemctl` COMMANDS

Services can be started and stopped on a running system and enabled or disabled for automatic start at boot time.

TASK:	COMMAND:
View detailed information about a unit state.	<code>systemctl status UNIT</code>
Stop a service on a running system.	<code>systemctl stop UNIT</code>
Start a service on a running system.	<code>systemctl start UNIT</code>
Restart a service on a running system.	<code>systemctl restart UNIT</code>
Reload configuration file of a running service.	<code>systemctl reload UNIT</code>
Completely disable a service from being started, both manually and at boot.	<code>systemctl mask UNIT</code>
Make a masked service available.	<code>systemctl unmask UNIT</code>
Configure a service to start at boot time.	<code>systemctl enable UNIT</code>
Disable a service from starting at boot time.	<code>systemctl disable UNIT</code>
List units which are required and wanted by the specified unit.	<code>systemctl list-dependencies UNIT</code>



Configuring and Securing OpenSSH Service



Configuring and Securing OpenSSH Service

Goal

- To configure secure command-line access on remote systems using **OpenSSH**.

Objectives

- **Log in** to a remote system using **ssh** to run commands from a shell prompt.
- **Set up ssh** for secure **password-free logins** using a private authentication key file.
- **Customize sshd configuration** to restrict direct logins as **root** or to disable **password-based authentication**.

Sections

- **Accessing the Remote Command Line with SSH.**
- **Configuring SSH Key-based Authentication.**
- **Customizing SSH Service Configuration.**



Accessing the Remote Command Line with SSH



Accessing the Remote Command Line with SSH

What is OpenSSH Secure Shell (SSH)?

- **OpenSSH** is the software implementation of the **Secure Shell** protocol.
- **SSH** is used to securely run a **shell** on a **remote system**.
- The **ssh command** is the primary tool used to **remotely log into** a system.
- **SSH** can also be used to run an **individual command** on a remote system.

Accessing the Remote Command Line with SSH

❑ Secure Shell SSH Examples:

Here are some examples of **ssh** command syntax for remote login and remote execution:

- Create a remote interactive shell as the current user, then return to your previous shell when done with the **exit** command.

```
[student@host ~]$ssh remotehost
student@remotehost's password:
[student@remotehost ~]$ exit
Connection to remotehost closed.
[student@host ~]$
```

- Connect to a remote shell as a different user (**remoteuser**) on a selected host (**remotehost**):

```
[student@host ~]$ssh remoteuser@remotehost
remoteuser@remotehost's password:
[remoteuser@remotehost ~]$
```

- Execute a single command (**hostname**) on a remote host (**remotehost**) and as a remote user (**remoteuser**) in a way that returns the output to the local display:

```
[student@host ~]$ssh remoteuser@remotehost hostname
remoteuser@remotehost's password:
remotehost.example.com
[student@host ~]$
```

Accessing the Remote Command Line with SSH

❖ Listing Logged-In Users (w command)

▪ Purpose of the 'w' command

- The **w command** displays a list of all users **currently logged into the computer**.
- It is useful for showing which users logged in using **ssh** from **remote locations** and what they are currently doing.

```
[student@host ~]$w -f
```

USER	TTY	FROM	LOGIN@	IDLE	JCPU	PCPU	WHAT
student	tty1	:0	Wed08	2days	1:52m	0.07s	pam: gdm-passwo
root	tty6	-	12:33	4:14m	16.27s	15.74s	-bash
student	pts/0	:0.0	Wed08	5:11	1.63s	1.60s	/usr/bin/gnome-
student	pts/1	:0.0	Wed08	43:44	14.48s	13.81s	vim hello.c

Command Syntax and Output:

▪ Command: `w -f`

- The `-f` option shows the **FROM** column, which indicates the **remote host** or location of the user's connection.

▪ Output Information:

- **USER:** The username.
- **TTY:** The terminal type (e.g., `tty1` for local, `pts/0` for remote/pseudo-terminal).
- **FROM:** The remote location or IP address (especially for SSH logins).
- **WHAT:** The current command being run by the user.

SSH HOST KEYS

Security Mechanism:

- **SSH** secures communication via **public-key encryption**.
- The server sends its **public key** to the client before the login process.
- **Purpose:** Sets up secure encryption and **authenticates the server to the client**.

The known_hosts File:

- The first time a user connects, the **ssh command** stores the server's public key in the user's `$\sim/.ssh/known_hosts` file.
- **Subsequent Connections:** The client compares the stored key with the key sent by the server.
- **Key Mismatch:** If keys do not match, the client **breaks the connection**, assuming the server is compromised or the traffic is being hijacked.

Key Change Management:

- If the server's public key changes (e.g., due to loss or replacement), users must **remove the old entry** from their `$\sim/.ssh/known_hosts` file to log in successfully.

SSH Key Locations

1. Client System Key Storage

- **Host IDs** (Public Keys of remote hosts) are stored on the **local client system**.
- **File Location:** `$\sim/.ssh/known_hosts`
- **Content Example:** Shows the remote host name/IP and its public key hash.

- Host IDs are stored in `~/.ssh/known_hosts` on your local client system:

```
$ cat ~/.ssh/known_hosts
remotehost,192.168.0.101 ssh-rsa AAAAB3Nzac...
```

- Host keys are stored in `/etc/ssh/ssh_host_key*` on the SSH server.

```
$ ls /etc/ssh/*key*
ssh_host_dsa_key      ssh_host_key          ssh_host_rsa_key
ssh_host_dsa_key.pub  ssh_host_key.pub      ssh_host_rsa_key.pub
```

2. SSH Server Key Storage

- **Host keys** (The server's actual Public and Private keys) are stored on the **SSH server**.
- **Directory Location:** `/etc/ssh/`
- **Key Files:** These include the **private** key and the corresponding **.pub** (public) key for different encryption types (e.g., `ssh_host_rsa_key`, `ssh_host_dsa_key`).



CONFIGURING SSH KEY-BASED AUTHENTICATION



SSH Key-Based Authentication

○ Authentication Scheme:

- **Key Generation:** SSH uses a **private-public key scheme** where two keys are generated (a private key and a public key).
- **Private Key:** Used as the **authentication credential** and must be kept **secret and secure** (like a password).
- **Public Key:** Copied to the **remote systems** the user wants to log into and is used to **verify the private key**. It does not need to be secret.

✓ **Benefit:** Allows access to systems **without typing a password every time**, while remaining secure.

Key generation is done using the **ssh-keygen** command. This generates the private key `~/.ssh/id_rsa` and the public key `~/.ssh/id_rsa.pub`.

SSH Key-Based Authentication

- Use **ssh-keygen** to create a public-private key pair.

```
[student@desktopX ~]$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/student/.ssh/id_rsa): Enter
Created directory '/home/student/.ssh'.
Enter passphrase (empty for no passphrase): redhat
Enter same passphrase again: redhat
Your identification has been saved in /home/student/.ssh/id_rsa.
Your public key has been saved in /home/student/.ssh/id_rsa.pub.
The key fingerprint is:
a4:49:cf:fb:ac:ab:c8:ce:45:33:f2:ad:69:7b:d2:5a student@desktopX.example.com
The key's randomart image is:
+--[ RSA 2048 ]-----+
|
|      . .
|     . *
|    . * S
|   + + .
|  o . E
| o oo+oo
| .=. **ooo
+-----+
```

- Use **ssh-copy-id** to copy the public key to the correct location on a remote system. For example:

```
[student@desktopX ~]$
ssh-copy-id -i ~/.ssh/id_rsa.pub root@serverX.example.com
```



Customizing SSH Service Configuration



Customizing SSH Service Configuration

❖ The OpenSSH Server Configuration File:

- The **OpenSSH server configuration** typically does not require modification, but **additional security measures** are available.
- Various aspects of the OpenSSH server can be modified in the configuration file:
`/etc/ssh/sshd_config`.

Prohibiting Root SSH Login

Security Rationale:

- It is advisable to **prohibit direct root login with ssh** from a security standpoint.
- **Reasons:**
 - The `root` username exists by default, requiring an attacker only to guess the **password**.
 - The `root` user has **unrestricted privileges** (highest risk if compromised).

Configuration Steps:

The OpenSSH server has an internal configuration file setting to prohibit a system login as user `root`, which is commented out by default in the `/etc/ssh/sshd_config` file:

```
#PermitRootLogin yes
```

By enabling the previous option in the `/etc/ssh/sshd_config` configuration file as follows, the `root` user will be unable to log into the system using the `ssh` command after the `sshd` service has been restarted:

```
PermitRootLogin no
```

Applying SSH Changes

1. Restart Requirement:

- The **sshd service** must be **restarted** after any changes to the `/etc/ssh/sshd_config` file to put the changes into effect.
- **Restart Command (Implicit):** `systemctl restart sshd` (though the slide shows only `systemctl sshd`).

2. Alternative Root Login Setting:

- To allow the **root user** to log in **only with key-based authentication** (for security), use the following configuration:
 - `PermitRootLogin without-password`
- **Note:** This setting prevents root login via **passwords** but still allows secure key-based access

Prohibiting Password Authentication

Advantages of Key-based Login

- **Security:** SSH keys are **longer** than typical passwords, which enhances security.
- **Convenience:** Less effort is needed to initiate **remote shell access** after the initial setup.

Configuration Steps

```
PasswordAuthentication yes
```

- **Default Setting:** Password authentication is **on** by default: `PasswordAuthentication yes`.
- **Secure Setting:** To **prevent password authentication** and rely only on key-based logins, set the option to:

```
PasswordAuthentication no. PasswordAuthentication no
```

- **Activation:** Whenever the `/etc/ssh/sshd_config` file is changed, the **sshd service** must be **restarted** to apply the changes.

```
[root@serverX ~]#  
systemctl reload sshd
```



Thank You