



Red Hat System Administration I

Dr. Eng. Gamal Selim

Lecture 05



Agenda

- **Chapter 9: Controlling Services and Daemons**
- **Chapter 10: Configuring and Securing OpenSSH Service**

Chapter 9: Controlling Services and Daemons

CONTROLLING SERVICES AND DAEMONS

GOAL

To control and monitor network services and system daemons using systemd.

OBJECTIVES

- List system daemons and network services started by the systemd service and socket units.
- Control system daemons and network services using **systemctl**.

SECTIONS

- Identifying Automatically Started System Processes (and Practice)
- Controlling System Services (and Practice)

IDENTIFYING AUTOMATICALLY STARTED SYSTEM PROCESSES

OBJECTIVES

After completing this section, students should be able to list system daemons and network services started by the **systemd** service and socket units.

INTRODUCTION TO **systemd**

System startup and server processes are managed by the *systemd System and Service Manager*. This program provides a method for activating system resources, server daemons, and other processes, both at boot time and on a running system.

Daemons are processes that wait or run in the background performing various tasks. Generally, daemons start automatically at boot time and continue to run until shutdown or until they are manually stopped. By convention, the names of many daemon programs end in the letter "d".

To listen for connections, a daemon uses a *socket*. This is the primary communication channel with local or remote clients. Sockets may be created by daemons or may be separated from the daemon and be created by another process, such as systemd. The socket is passed to the daemon when a connection is established by the client.

A bit of history

For many years, process ID 1 of Linux and UNIX systems has been the **init** process. This process was responsible for activating other services on the system and is the origin of the term "init system." Frequently used daemons were started on systems at boot time with *System V* and *LSB* init scripts. These are shell scripts, and may vary from one distribution to another. Less frequently used daemons were started on demand by another service, such as **initd** or **xinetd**, which listens for client connections. These systems have several limitations, which are addressed with **systemd**.

In Red Hat Enterprise Linux 7, process ID 1 is **systemd**, the new init system. A few of the new features provided by **systemd** include:

- Parallelization capabilities, which increase the boot speed of a system.
- On-demand starting of daemons without requiring a separate service.
- Automatic service dependency management, which can prevent long timeouts, such as by not starting a network service when the network is not available.
- A method of tracking related processes together by using Linux control groups.

systemctl and systemd units

The **systemctl** command is used to manage different types of systemd objects, called *units*. A list of available unit types can be displayed with **systemctl -t help**.

Some common unit types are listed below:

- Service units have a .service extension and represent system services. This type of unit is used to start frequently accessed daemons, such as a web server.
- Socket units have a .socket extension and represent inter-process communication (IPC) sockets. Control of the socket will be passed to a daemon or newly started service when a client connection is made. Socket units are used to delay the start of a service at boot time and to start less frequently used services on demand. These are similar in principle to services which use the **xinetd** superserver to start on demand.
- Path units have a .path extension and are used to delay the activation of a service until a specific file system change occurs. This is commonly used for services which use spool directories, such as a printing system.

Service states

The status of a service can be viewed with **systemctl status *name.type***. If the unit type is not provided, **systemctl** will show the status of a service unit, if one exists.

```
[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.
```

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`

In this example, please follow along with the next steps while your instructor demonstrates obtaining status information of services.



NOTE

Notice that the **`systemctl`** command will automatically paginate the output with **`less`**.

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

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

2. Query the state of only the service units.

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

- Investigate any units which are in a failed or maintenance state. Optionally, add the **-l** option to show the full output.

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

- The **status** argument may also be used to determine if a particular unit is active and show if the unit is enabled to start at boot time. Alternate commands can also easily show the active and enabled states:

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

- List the active state of all loaded units. Optionally, limit the type of unit. The **--all** option will add inactive units.

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

6. View the enabled and disabled settings for all units. Optionally, limit the type of unit.

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

7. View only failed services.

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

CONTROLLING SYSTEM SERVICES

OBJECTIVES

After completing this section, students should be able to control system daemons and network services using **systemctl**.

STARTING AND STOPPING SYSTEM DAEMONS ON A RUNNING SYSTEM

Changes to a configuration file or other updates to a service may require that the service be restarted. A service that is no longer used may be stopped before removing the software. A service that is not frequently used may be manually started by an administrator only when it is needed.

In this example, please follow along with the next steps while your instructor demonstrates managing services on a running system.

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
```

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
```

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

Services may be started as dependencies of other services. If a socket unit is enabled and the service unit with the same name is not, the service will automatically be started when a request is made on the network socket. Services may also be triggered by path units when a file system condition is met. For example, a file placed into the print spool directory will cause the **cups** print service to be started if it is not running.

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

To completely stop printing services on a system, stop all three units. Disabling the service will disable the dependencies.

The **systemctl list-dependencies UNIT** command can be used to print out a tree of what other units must be started if the specified unit is started. Depending on the exact dependency, the other unit may need to be running before or after the specified unit starts. The **--reverse** option to this command will show what units need to have the specified unit started in order to run.

Masking services

At times, a system may have conflicting services installed. For example, there are multiple methods to manage networks (network and NetworkManager) and firewalls (iptables and firewalld). To prevent an administrator from accidentally starting a service, that service may be *masked*. Masking will create a link in the configuration directories so that if the service is started, nothing will happen.

```
[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 SYSTEM DAEMONS TO START OR STOP AT BOOT

Starting a service on a running system does not guarantee that the service will be started when the system reboots. Similarly, stopping a service on a running system will not keep it from starting again when the system reboots. Services are started at boot time when links are created in the appropriate **systemd** configuration directories. These links are created and removed with **systemctl** commands.

In this example, please follow along with the next steps while your instructor demonstrates enabling and disabling services.

1. View the status of a service.

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

2. Disable the service and verify the status. Note that disabling a service does not stop the service.

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

3. Enable the service and verify the status.

```
[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>

End of Chapter 9

Chapter 10: Configuring and Securing OpenSSH Service

CONFIGURING AND SECURING OPENSSH SERVICE

GOAL

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

OBJECTIVES

- Log into a remote system using ssh to run commands from a shell prompt.
- Set up ssh to allow secure password-free logins by 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 (and Practice)
- Configuring SSH Key-based Authentication (and Practice)
- Customizing SSH Service Configuration (and Practice)

ACCESSING THE REMOTE COMMAND LINE WITH SSH

OBJECTIVE

After completing this section, students should be able to log into a remote system using `ssh` to run commands from a shell prompt.

WHAT IS THE OPENSSH SECURE SHELL (SSH)?

The term OpenSSH refers to the software implementation of the Secure Shell software used in the system. The OpenSSH Secure Shell, **ssh**, is used to securely run a shell on a remote system. If you have a user account on a remote Linux system providing SSH services, **ssh** is the command normally used to remotely log into that system. The **ssh** command can also be used to run an individual command on a remote system.

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

The `w` command displays a list of users currently logged into the computer. This is especially useful to show which users are logged in using `ssh` from which remote locations, and what they are 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

SSH HOST KEYS

SSH secures communication through public-key encryption. When an **ssh** client connects to an SSH server, before the client logs in, the server sends it a copy of its *public key*. This is used to set up the secure encryption for the communication channel and to authenticate the server to the client.

The first time a user uses **ssh** to connect to a particular server, the **ssh** command stores the server's public key in the user's `~/.ssh/known_hosts` file. Every time the user connects after that, the client makes sure it gets the same public key from the server by comparing the server's entry in the `~/.ssh/known_hosts` file to the public key the server sent. If the keys do *not* match, the client assumes that the network traffic is being hijacked or that the server has been compromised, and breaks the connection.

This means that if a server's public key is changed (because the key was lost due to hard drive failure, or replaced for some legitimate reason), users will need to update their `~/.ssh/known_hosts` files and remove the old entry in order to log in.

- 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
```

CONFIGURING SSH KEY-BASED AUTHENTICATION

OBJECTIVE

After completing this section, students should be able to set up SSH to allow secure logins without passwords by using a private authentication key file.

SSH KEY-BASED AUTHENTICATION

Users can authenticate **ssh** logins without a password by using *public key authentication*. **ssh** allows users to authenticate using a private-public key scheme. This means that two keys are generated, a private key and a public key. The private key file is used as the authentication credential, and like a password, must be kept secret and secure. The public key is copied to systems the user wants to log into, and is used to verify the private key. The public key does not need to be secret. An SSH server that has the public key can issue a challenge that can only be answered by a system holding your private key. As a result, you can authenticate using the presence of your key. This allows you to access systems in a way that doesn't require typing a password every time, but is still 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`.

- 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

OBJECTIVE

After completing this section, students should be able to customize sshd configuration to restrict direct logins as root or to disable password-based authentication.

THE OPENSSH SERVER CONFIGURATION FILE

While OpenSSH server configuration usually does not require modification, additional security measures are available.

Various aspects of the OpenSSH server can be modified in the configuration file `/etc/ssh/sshd_config`.

PROHIBIT THE ROOT USER FROM LOGGING IN USING SSH

From a security standpoint, it is advisable to prohibit the root user from directly logging into the system with **ssh**.

- The username root exists on every Linux system by default, so a potential attacker only has to guess the password, instead of a valid username and password combination.
- The root user has unrestricted privileges.

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
```

The sshd service has to be restarted to put the changes into effect:

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

```
sshd
```

Another option is to only allow key-based ssh login as root with:

```
PermitRootLogin without-password
```

PROHIBIT PASSWORD AUTHENTICATION USING SSH

Only allowing key-based logins to the remote command line has various advantages:

- SSH keys are longer than an average password, which adds security.
- Less effort to initiate remote shell access after the initial setup.

There is an option in the `/etc/ssh/sshd_config` configuration file which turns on password authentication by default:

```
PasswordAuthentication yes
```

To prevent password authentication, the **PasswordAuthentication** option has to be set to **no** and the sshd service needs to be restarted:

```
PasswordAuthentication no
```

Keep in mind that whenever you change the `/etc/ssh/sshd_config` file, the sshd service has to be restarted:

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

End of Chapter 10

*Thank
you*

