# Chapter 9: Controlling Services and Daemons

[**Identifying Automatically Started System Processes**](https://rha.ole.redhat.com/rha/app/courses/rh124-8.2/pages/ch09/9a2ef70f-4e72-42df-a498-b694b274af27)

[**Guided Exercise: Identifying Automatically Started System Processes**](https://rha.ole.redhat.com/rha/app/courses/rh124-8.2/pages/ch09s02/9a2ef70f-4e72-42df-a498-b694b274af27)

[**Controlling System Services**](https://rha.ole.redhat.com/rha/app/courses/rh124-8.2/pages/ch09s03/9a2ef70f-4e72-42df-a498-b694b274af27)

[**Guided Exercise: Controlling System Services**](https://rha.ole.redhat.com/rha/app/courses/rh124-8.2/pages/ch09s04/9a2ef70f-4e72-42df-a498-b694b274af27)

[**Lab: Controlling Services and Daemons**](https://rha.ole.redhat.com/rha/app/courses/rh124-8.2/pages/ch09s05/9a2ef70f-4e72-42df-a498-b694b274af27)

[**Summary**](https://rha.ole.redhat.com/rha/app/courses/rh124-8.2/pages/ch09s06/9a2ef70f-4e72-42df-a498-b694b274af27)

# Abstract

|  |  |
| --- | --- |
| **Goal** | * 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 Guided Exercise) * Controlling System Services (and Guided Exercise) |
| **Lab** | * Controlling Services and Daemons |

# Identifying Automatically Started System Processes Objectives

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

## Introduction to systemd

The *systemd* daemon manages startup for Linux, including service startup and service management in general. It activates system resources, server daemons, and other processes both at boot time and on a running system.

Daemons are processes that either 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. It is a convention for names of many daemon programs to end in the letter d.

A *service* in the systemd sense often refers to one or more daemons, but starting or stopping a service may instead make a one-time change to the state of the system, which does not involve leaving a daemon process running afterward (called oneshot).

In Red Hat Enterprise Linux, the first process that starts (PID 1) is systemd. A few of the features provided by systemd include:

* Parallelization capabilities (starting multiple services simultaneously), 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. For example, a network-dependent service will not attempt to start up until the network is available.
* A method of tracking related processes together by using Linux control groups.

## Describing Service Units

systemd uses *units* to manage different types of objects. 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 that systemd should monitor. If a client connects to the socket, systemd will start a daemon and pass the connection to it. Socket units are used to delay the start of a service at boot time and to start less frequently used services 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.

The **systemctl** command is used to manage units. For example, display available unit types with the **systemctl -t help** command.

## Important

When using **systemctl**, you can abbreviate unit names, process tree entries, and unit descriptions.

## Listing Service Units

You use the **systemctl** command to explore the current state of the system. For example, the following command lists all currently loaded service units, paginating the output using **less**.

**[root@host ~]# systemctl list-units --type=service**

UNIT LOAD ACTIVE SUB DESCRIPTION

atd.service loaded active running Job spooling tools

auditd.service loaded active running Security Auditing Service

chronyd.service loaded active running NTP client/server

crond.service loaded active running Command Scheduler

dbus.service loaded active running D-Bus System Message Bus

*...output omitted...*

The above output limits the type of unit listed to service units with the --type=service option. The output has the following columns:

## Columns in the systemctl list-units Command Output

## UNIT

The service unit name.

## LOAD

Whether systemd properly parsed the unit's configuration and loaded the unit into memory.

## ACTIVE

The high-level activation state of the unit. This information indicates whether the unit has started successfully or not.

## SUB

The low-level activation state of the unit. This information indicates more detailed information about the unit. The information varies based on unit type, state, and how the unit is executed.

## DESCRIPTION

The short description of the unit.

By default, the systemctl list-units --type=service command lists only the service units with active activation states. The --all option lists all service units regardless of the activation states. Use the --state= option to filter by the values in the LOAD, ACTIVE, or SUB fields.

**[root@host ~]# systemctl list-units --type=service --all**

UNIT LOAD ACTIVE SUB DESCRIPTION

atd.service loaded active running Job spooling tools

auditd.service loaded active running Security Auditing ...

auth-rpcgss-module.service loaded inactive dead Kernel Module ...

chronyd.service loaded active running NTP client/server

cpupower.service loaded inactive dead Configure CPU power ...

crond.service loaded active running Command Scheduler

dbus.service loaded active running D-Bus System Message Bus

● display-manager.service not-found inactive dead display-manager.service

*...output omitted...*

The systemctl command without any arguments lists units that are both loaded and active.

**[root@host ~]# systemctl**

UNIT LOAD ACTIVE SUB DESCRIPTION

proc-sys-fs-binfmt\_misc.automount loaded active waiting Arbitrary...

sys-devices-....device loaded active plugged Virtio network...

sys-subsystem-net-devices-ens3.deviceloaded active plugged Virtio network...

...

-.mount loaded active mounted Root Mount

boot.mount loaded active mounted /boot

...

systemd-ask-password-plymouth.path loaded active waiting Forward Password...

systemd-ask-password-wall.path loaded active waiting Forward Password...

init.scope loaded active running System and Servi...

session-1.scope loaded active running Session 1 of...

atd.service loaded active running Job spooling tools

auditd.service loaded active running Security Auditing...

chronyd.service loaded active running NTP client/server

crond.service loaded active running Command Scheduler

*...output omitted...*

The **systemctl list-units** command displays units that the systemd service attempts to parse and load into memory; it does not display installed, but not enabled, services. To see the state of all unit files installed, use the **systemctl list-unit-files** command. For example:

**[root@host ~]# systemctl list-unit-files --type=service**

UNIT FILE STATE

arp-ethers.service disabled

atd.service enabled

auditd.service enabled

auth-rpcgss-module.service static

autovt@.service enabled

blk-availability.service disabled

*...output omitted...*

In the output of the **systemctl list-units-files** command, valid entries for the STATE field are enabled, disabled, static, and masked.

# Viewing Service States

View the status of a specific unit 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@host ~]# systemctl status sshd.service**

● sshd.service - OpenSSH server daemon

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

Active: active (running) since Thu 2019-02-14 12:07:45 IST; 7h ago

Main PID: 1073 (sshd)

CGroup: /system.slice/sshd.service

└─1073 /usr/sbin/sshd -D ...

Feb 14 11:51:39 host.example.com systemd[1]: Started OpenSSH server daemon.

Feb 14 11:51:39 host.example.com sshd[1073]: Could not load host key: /et...y

Feb 14 11:51:39 host.example.com sshd[1073]: Server listening on 0.0.0.0 ....

Feb 14 11:51:39 host.example.com sshd[1073]: Server listening on :: port 22.

Feb 14 11:53:21 host.example.com sshd[1270]: error: Could not load host k...y

Feb 14 11:53:22 host.example.com sshd[1270]: Accepted password for root f...2

*...output omitted...*

This command displays the current status of the service. The meaning of the fields are:

## Table 9.1. Service Unit Information

| **Field** | **Description** |
| --- | --- |
| Loaded | Whether the service unit is loaded into memory. |
| Active | Whether the service unit is running and if so, how long it has been running. |
| Main PID | The main process ID of the service, including the command name. |
| Status | Additional information about the service. |

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

## Table 9.2. Service States in the Output of systemctl

| **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 | Is started at boot time. |
| disabled | Is not set to be started at boot time. |
| static | Cannot be enabled, but may be started by an enabled unit automatically. |

## Note

The systemctl status NAME command replaces the service NAME status command used in Red Hat Enterprise Linux 6 and earlier.

# Verifying the Status of a Service

The **systemctl** command provides methods for verifying the specific states of a service. For example, use the following command to verify that the a service unit is currently active (running):

**[root@host ~]# systemctl is-active sshd.service**

active

The command returns state of the service unit, which is usually active or inactive.

Run the following command to verify whether a service unit is enabled to start automatically during system boot:

**[root@host ~]# systemctl is-enabled sshd.service**

enabled

The command returns whether the service unit is enabled to start at boot time, which is usually enabled or disabled.

To verify whether the unit failed during startup, run the following command:

**[root@host ~]# systemctl is-failed sshd.service**

active

The command either returns active if it is properly running or failed if an error has occurred during startup. In case the unit was stopped, it returns unknown or inactive.

To list all the failed units, run the systemctl --failed --type=service command.

## References

systemd(1), systemd.unit(5), systemd.service(5), systemd.socket(5), and systemctl(1) man pages

For more information, refer to the *Managing services with systemd* chapter in the *Red Hat Enterprise Linux 8 Configuring basic system settings Guide* at <https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html-single/configuring_basic_system_settings/managing-services-with-systemd_configuring-basic-system-settings#managing-services-with-systemd_configuring-basic-system-settings>

# Guided Exercise: Identifying Automatically Started System Processes

In this exercise, you will list installed service units and identify which services are currently enabled and active on a server.

## Outcomes

You should be able to list installed service units and identify active and enabled services on the system.

Log in as the student user on workstation using student as the password.

From workstation, run the lab services-identify start command. The command runs a start script that determines if the host, servera, is reachable on the network.

**[student@workstation ~]$ lab services-identify start**

### Use the ssh command to log in to servera as the student user. The systems are configured to use SSH keys for authentication, therefore a password is not required to log in to servera.

**[student@workstation ~]$ ssh student@servera**

*...output omitted...*

**[student@servera ~]$**

### List all service units installed on servera.

**[student@servera ~]$ systemctl list-units --type=service**

UNIT LOAD ACTIVE SUB DESCRIPTION

atd.service loaded active running Job spooling tools

auditd.service loaded active running Security Auditing Service

chronyd.service loaded active running NTP client/server

crond.service loaded active running Command Scheduler

dbus.service loaded active running D-Bus System Message Bus

*...output omitted...*

Press **q** to exit the command.

### List all socket units, active and inactive, on servera.

**[student@servera ~]$ systemctl list-units --type=socket --all**

UNIT LOAD ACTIVE SUB DESCRIPTION

dbus.socket loaded active running D-Bus System Message Bus Socket

dm-event.socket loaded active listening Device-mapper event daemon FIFOs

lvm2-lvmpolld.socket loaded active listening LVM2 poll daemon socket

*...output omitted...*

systemd-udevd-control.socket loaded active running udev Control Socket

systemd-udevd-kernel.socket loaded active running udev Kernel Socket

LOAD = Reflects whether the unit definition was properly loaded.

ACTIVE = The high-level unit activation state, i.e. generalization of SUB.

SUB = The low-level unit activation state, values depend on unit type.

12 loaded units listed.

To show all installed unit files use 'systemctl list-unit-files'.

### Explore the status of the chronyd service. This service is used for network time synchronization (NTP).

* 1. Display the status of the chronyd service. Note the process ID of any active daemon.

**[student@servera ~]$ systemctl status chronyd**

● chronyd.service - NTP client/server

Loaded: loaded (/usr/lib/systemd/system/chronyd.service; enabled; vendor preset: enabled)

Active: active (running) since Wed 2019-02-06 12:46:57 IST; 4h 7min ago

Docs: man:chronyd(8)

man:chrony.conf(5)

Process: 684 ExecStartPost=/usr/libexec/chrony-helper update-daemon (code=exited, status=0/SUCCESS)

Process: 673 ExecStart=/usr/sbin/chronyd $OPTIONS (code=exited, status=0/SUCCESS)

**Main PID: 680 (chronyd)**

Tasks: 1 (limit: 11406)

Memory: 1.5M

CGroup: /system.slice/chronyd.service

└─680 /usr/sbin/chronyd

... servera.lab.example.com systemd[1]: Starting NTP client/server...

*...output omitted...*

... servera.lab.example.com systemd[1]: Started NTP client/server.

... servera.lab.example.com chronyd[680]: Source 172.25.254.254 offline

... servera.lab.example.com chronyd[680]: Source 172.25.254.254 online

... servera.lab.example.com chronyd[680]: Selected source 172.25.254.254

Press **q** to exit the command.

* 1. Confirm that the listed daemon is running. In the preceding command, the output of the process ID associated with the chronyd service is 680. The process ID might differ on your system.

**[student@servera ~]$ ps -p *680***

PID TTY TIME CMD

680 ? 00:00:00 chronyd

## Explore the status of the sshd service. This service is used for secure encrypted communication between systems.

* 1. **Determine whether the sshd service is enabled to start at system boot.**

**[student@servera ~]$ systemctl is-enabled sshd**

enabled

* 1. **Determine if the sshd service is active without displaying all of the status information.**

**[student@servera ~]$ systemctl is-active sshd**

active

* 1. **Display the status of the sshd service.**

**[student@servera ~]$ systemctl status sshd**

● sshd.service - OpenSSH server daemon

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

Active: active (running) since Wed 2019-02-06 12:46:58 IST; 4h 21min ago

Docs: man:sshd(8)

man:sshd\_config(5)

Main PID: 720 (sshd)

Tasks: 1 (limit: 11406)

Memory: 5.8M

CGroup: /system.slice/sshd.service

└─720 /usr/sbin/sshd -D -oCiphers=aes256-gcm@openssh.com,

chacha20-poly1305@openssh.com,aes256-ctr,

aes256-cbc,aes128-gcm@openssh.com,aes128-ctr,

aes128-cbc -oMACs=hmac-sha2-256-etm@openssh.com,hmac-sha>

... servera.lab.example.com systemd[1]: Starting OpenSSH server daemon...

... servera.lab.example.com sshd[720]: Server listening on 0.0.0.0 port 22.

... servera.lab.example.com systemd[1]: Started OpenSSH server daemon.

... servera.lab.example.com sshd[720]: Server listening on :: port 22.

*...output omitted...*

... servera.lab.example.com sshd[1380]: pam\_unix(sshd:session): session opened for user student by (uid=0)

Press **q** to exit the command.

### List the enabled or disabled states of all service units.

**[student@servera ~]$ systemctl list-unit-files --type=service**

UNIT FILE STATE

arp-ethers.service disabled

atd.service enabled

auditd.service enabled

auth-rpcgss-module.service static

autovt@.service enabled

blk-availability.service disabled

chrony-dnssrv@.service static

chrony-wait.service disabled

chronyd.service enabled

*...output omitted...*

Press **q** to exit the command.

### Exit from servera.

**[student@servera ~]$ exit**

logout

Connection to servera closed.

**[student@workstation]$**

## Finish

On workstation, run the **lab services-identify finish** script to complete this exercise.

**[student@workstation ~]$ lab services-identify finish**

This concludes the guided exercise.

# Controlling System Services Objectives

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

## Starting and Stopping Services

Services need to be stopped or started manually for a number of reasons: perhaps the service needs to be updated; the configuration file may need to be changed; or a service may need to be uninstalled; or an administrator may manually start an infrequently used service.

To start a service, first verify that it is not running with **systemctl status**. Then, use the **systemctl start** command as the root user (using **sudo** if necessary). The example below shows how to start the sshd.service service:

**[root@host ~]# systemctl start sshd.service**

The systemd service looks for .service files for service management in commands in the absence of the service type with the service name. Thus the above command can be executed as:

**[root@host ~]# systemctl start sshd**

To stop a currently running service, use the stop argument with the **systemctl** command. The example below shows how to stop the sshd.service service:

**[root@host ~]# systemctl stop sshd.service**

# Restarting and Reloading Services

During a restart of a running service, the service is stopped and then started. On the restart of service, the process ID changes and a new process ID gets associated during the startup. To restart a running service, use the restart argument with the **systemctl** command. The example below shows how to restart the sshd.service service:

**[root@host ~]# systemctl restart sshd.service**

Some services have the ability to reload their configuration files without requiring a restart. This process is called a *service reload*. Reloading a service does not change the process ID associated with various service processes. To reload a running service, use the reload argument with the **systemctl** command. The example below shows how to reload the sshd.service service after configuration changes:

**[root@host ~]# systemctl reload sshd.service**

In case you are not sure whether the service has the functionality to reload the configuration file changes, use the reload-or-restart argument with the **systemctl** command. The command reloads the configuration changes if the reloading functionality is available. Otherwise, the command restarts the service to implements the new configuration changes:

**[root@host ~]# systemctl reload-or-restart sshd.service**

# Listing Unit Dependencies

Some services require that other services be running first, creating dependencies on the other services. Other services are not started at boot time but rather only on demand. In both cases, systemd and **systemctl** start services as needed whether to resolve the dependency or to start an infrequently used service. For example, if the CUPS print service is not running and a file is placed into the print spool directory, then the system will start CUPS-related daemons or commands to satisfy the print request.

**[root@host ~]# 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 disables the dependencies.

The **systemctl list-dependencies *UNIT*** command displays a hierarchy mapping of dependencies to start the service unit. To list reverse dependencies (units that depend on the specified unit), use the --reverse option with the command.

**[root@host ~]# systemctl list-dependencies sshd.service**

sshd.service

● ├─system.slice

● ├─sshd-keygen.target

● │ ├─sshd-keygen@ecdsa.service

● │ ├─sshd-keygen@ed25519.service

● │ └─sshd-keygen@rsa.service

● └─sysinit.target

*...output omitted...*

# Masking and Unmasking Services

At times, a system may have different services installed that are conflicting with each other. For example, there are multiple methods to manage mail servers (postfix and sendmail, for example). Masking a service prevents an administrator from accidentally starting a service that conflicts with others. Masking creates a link in the configuration directories to the /dev/null file which prevents the service from starting.

**[root@host ~]# systemctl mask sendmail.service**

Created symlink /etc/systemd/system/sendmail.service → /dev/null.

**[root@host ~]# systemctl list-unit-files --type=service**

UNIT FILE STATE

*...output omitted...*

sendmail.service masked

*...output omitted...*

Attempting to start a masked service unit fails with the following output:

**[root@host ~]# systemctl start sendmail.service**

Failed to start sendmail.service: Unit sendmail.service is masked.

Use the **systemctl unmask** command to unmask the service unit.

**[root@host ~]# systemctl unmask sendmail**

Removed /etc/systemd/system/sendmail.service.

# Important

A disabled service can be started manually or by other unit files but it does not start automatically at boot. A masked service does not start manually or automatically.

# Enabling Services to Start or Stop at Boot

Starting a service on a running system does not guarantee that the service automatically starts when the system reboots. Similarly, stopping a service on a running system does not keep it from starting again when the system reboots. Creating links in the systemd configuration directories enables the service to start at boot. The **systemctl** commands creates and removes these links.

To start a service at boot, use the **systemctl enable** command.

**[root@root ~]# systemctl enable sshd.service**

Created symlink /etc/systemd/system/multi-user.target.wants/sshd.service → /usr/lib/systemd/system/sshd.service.

The above command creates a symbolic link from the service unit file, usually in the /usr/lib/systemd/system directory, to the location on disk where systemd looks for files, which is in the /etc/systemd/system/*TARGETNAME*.target.wants directory. Enabling a service does not start the service in the current session. To start the service and enable it to start automatically during boot, execute both the **systemctl start** and **systemctl enable** commands.

To disable the service from starting automatically, use the following command, which removes the symbolic link created while enabling a service. Note that disabling a service does not stop the service.

**[root@host ~]# systemctl disable sshd.service**

Removed /etc/systemd/system/multi-user.target.wants/sshd.service.

To verify whether the service is enabled or disable, use the **systemctl is-enabled** command.

# Summary of systemctl Commands

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

## Table 9.3. Useful Service Management Commands

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

# References

systemd(1), systemd.unit(5), systemd.service(5), systemd.socket(5), and systemctl(1) man pages

For more information, refer to the *Managing system services* chapter in the *Red Hat Enterprise Linux 8 Configuring basic system settings Guide* at <https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html-single/configuring_basic_system_settings/managing-services-with-systemd_configuring-basic-system-settings#managing-system-services_managing-services-with-systemd>

# Guided Exercise: Controlling System Services

In this exercise, you will use **systemctl** to stop, start, restart, reload, enable, and disable a systemd-managed service.

## Outcomes

You should be able to use the systemctl command to control systemd-managed services.

Log in as the student user on workstation using student as the password.

**From** workstation, run the lab services-control start command. The command runs a start script that determines whether the

**host**, servera, is reachable on the network. The script also ensures that the sshd and chronyd services are running on servera.

**[student@workstation ~]$ lab services-control start**

### Use the ssh command to log in to servera as the student user. The systems are configured to use SSH keys for authentication, and therefore a password is not required.

**[student@workstation ~]$ ssh student@servera**

**...output omitted...**

**[student@servera ~]$**

### Execute the systemctl restart and systemctl reload commands on the sshd service. Observe the different results of executing these commands.

* 1. Display the status of the sshd service. Note the process ID of the sshd daemon.

**[student@servera ~]$ systemctl status sshd**

**● sshd.service - OpenSSH server daemon**

**Loaded: loaded (/usr/lib/systemd/system/sshd.service; enabled; vendor preset: enabled)**

**Active: active (running) since Wed 2019-02-06 23:50:42 EST; 9min ago**

**Docs: man:sshd(8)**

**man:sshd\_config(5)**

**Main PID: 759 (sshd)**

**Tasks: 1 (limit: 11407)**

**Memory: 5.9M**

**...output omitted...**

Press **q** to exit the command.

* 1. Restart the sshd service and view the status. The process ID of the daemon must change.

**[student@servera ~]$ sudo systemctl restart sshd**

**[sudo] password for student: student**

**[student@servera ~]$ systemctl status sshd**

**● sshd.service - OpenSSH server daemon**

**Loaded: loaded (/usr/lib/systemd/system/sshd.service; enabled; vendor preset: enabled)**

**Active: active (running) since Wed 2019-02-06 23:50:42 EST; 9min ago**

**Docs: man:sshd(8)**

**man:sshd\_config(5)**

**Main PID: 1132 (sshd)**

**Tasks: 1 (limit: 11407)**

**Memory: 5.9M**

**...output omitted...**

In the preceding output, notice that the process ID changed from 759 to 1132 (on your system, the numbers likely will be different). Press **q** to exit the command.

* 1. Reload the sshd service and view the status. The process ID of the daemon must not change and connections are not interrupted.

**[student@servera ~]$ sudo systemctl reload sshd**

**[student@servera ~]$ systemctl status sshd**

**● sshd.service - OpenSSH server daemon**

**Loaded: loaded (/usr/lib/systemd/system/sshd.service; enabled; vendor preset: enabled)**

**Active: active (running) since Wed 2019-02-06 23:50:42 EST; 9min ago**

**Docs: man:sshd(8)**

**man:sshd\_config(5)**

**Main PID: 1132 (sshd)**

**Tasks: 1 (limit: 11407)**

**Memory: 5.9M**

**...output omitted...**

Press **q** to exit the command.

### Verify that the chronyd service is running.

**[student@servera ~]$ systemctl status chronyd**

**● chronyd.service - NTP client/server**

**Loaded: loaded (/usr/lib/systemd/system/chronyd.service; enabled; vendor preset: enabled)**

**Active: active (running) since Wed 2019-02-06 23:50:38 EST; 1h 25min ago**

**...output omitted...**

Press **q** to exit the command.

### Stop the chronyd service and view the status.

**[student@servera ~]$ sudo systemctl stop chronyd**

**[student@servera ~]$ systemctl status chronyd**

**● chronyd.service - NTP client/server**

**Loaded: loaded (/usr/lib/systemd/system/chronyd.service; enabled; vendor preset: enabled)**

**Active: inactive (dead) since Thu 2019-02-07 01:20:34 EST; 44s ago**

**...output omitted...**

**... servera.lab.example.com chronyd[710]: System clock wrong by 1.349113 seconds, adjustment started**

**... servera.lab.example.com systemd[1]: Stopping NTP client/server...**

**... servera.lab.example.com systemd[1]: Stopped NTP client/server.**

Press **q** to exit the command.

### Determine if the chronyd service is enabled to start at system boot.

**[student@server ~]$ systemctl is-enabled chronyd**

**enabled**

### Reboot servera, then view the status of the chronyd service.

**[student@servera ~]$ sudo systemctl reboot**

**Connection to servera closed by remote host.**

**Connection to servera closed.**

**[student@workstation ~]$**

Log in as the student user on servera and view the status of the chronyd service.

**[student@workstation ~]$ ssh student@servera**

**...output omitted...**

**[student@servera ~]$ systemctl status chronyd**

**● chronyd.service - NTP client/server**

**Loaded: loaded (/usr/lib/systemd/system/chronyd.service; enabled; vendor preset: enabled)**

**Active: active (running) since Thu 2019-02-07 01:48:26 EST; 5min ago**

**...output omitted...**

Press **q** to exit the command.

### Disable the chronyd service so that it does not start at system boot, then view the status of the service.

**[student@servera ~]$ sudo systemctl disable chronyd**

[sudo] password for student: **student**

Removed /etc/systemd/system/multi-user.target.wants/chronyd.service.

**[student@servera ~]$ systemctl status chronyd**

● chronyd.service - NTP client/server

Loaded: loaded (/usr/lib/systemd/system/chronyd.service; **disabled**; vendor preset: enabled)

Active: active (running) since Thu 2019-02-07 01:48:26 EST; 5min ago

*...output omitted...*

Press **q** to exit the command.

### Reboot servera, then view the status of the chronyd service.

**[student@servera ~]$ sudo systemctl reboot**

**Connection to servera closed by remote host.**

**Connection to servera closed.**

**[student@workstation ~]$**

Log in as the student user on servera and view the status of the chronyd service.

**[student@workstation ~]$ ssh student@servera**

**...output omitted...**

**[student@servera ~]$ systemctl status chronyd**

**● chronyd.service - NTP client/server**

**Loaded: loaded (/usr/lib/systemd/system/chronyd.service; disabled; vendor preset: enabled)**

**Active: inactive (dead)**

**Docs: man:chronyd(8)**

**man:chrony.conf(5)**

### Exit from servera.

**[student@servera ~]$ exit**

**logout**

**Connection to servera closed.**

**[student@workstation]$**

# Finish

On workstation, run the **lab services-control finish** script to complete this exercise.

**[student@workstation ~]$ lab services-control finish**

This concludes the guided exercise.

# Summary

## In this chapter, you learned:

* systemd provides a method for activating system resources, server daemons, and other processes, both at boot time and on a running system.
* Use the systemctl to start, stop, reload, enable, and disable services.
* Use the systemctl status command to determine the status of system daemons and network services started by systemd.
* The systemctl list-dependencies command lists all service units upon which a specific service unit depends.
* systemd can mask a service unit so that it does not run even to satisfy dependencies.

# Chapter 10: Configuring and Securing SSH

[**Accessing the Remote Command Line with SSH**](https://rha.ole.redhat.com/rha/app/courses/rh124-8.2/pages/ch10/9a2ef70f-4e72-42df-a498-b694b274af27)

[**Guided Exercise: Accessing the Remote Command Line**](https://rha.ole.redhat.com/rha/app/courses/rh124-8.2/pages/ch10s02/9a2ef70f-4e72-42df-a498-b694b274af27)

[**Configuring SSH Key-based Authentication**](https://rha.ole.redhat.com/rha/app/courses/rh124-8.2/pages/ch10s03/9a2ef70f-4e72-42df-a498-b694b274af27)

[**Guided Exercise: Configuring SSH Key-based Authentication**](https://rha.ole.redhat.com/rha/app/courses/rh124-8.2/pages/ch10s04/9a2ef70f-4e72-42df-a498-b694b274af27)

[**Customizing OpenSSH Service Configuration**](https://rha.ole.redhat.com/rha/app/courses/rh124-8.2/pages/ch10s05/9a2ef70f-4e72-42df-a498-b694b274af27)

[**Guided Exercise: Customizing OpenSSH Service Configuration**](https://rha.ole.redhat.com/rha/app/courses/rh124-8.2/pages/ch10s06/9a2ef70f-4e72-42df-a498-b694b274af27)

[**Lab: Configuring and Securing SSH**](https://rha.ole.redhat.com/rha/app/courses/rh124-8.2/pages/ch10s07/9a2ef70f-4e72-42df-a498-b694b274af27)

[**Summary**](https://rha.ole.redhat.com/rha/app/courses/rh124-8.2/pages/ch10s08/9a2ef70f-4e72-42df-a498-b694b274af27)

# ****Abstract****

|  |  |
| --- | --- |
| **Goal** | * Configure secure command-line service on remote systems, using OpenSSH. |
| **Objectives** | * Log in to a remote system and run commands using ssh. * Configure key-based authentication for a user account to log in to remote systems securely without a password. * Restrict direct logins as root and disable password-based authentication for the OpenSSH service. |
| **Sections** | * Accessing the Remote Command Line with SSH (and Guided Exercise) * Configuring SSH Key-Based Authentication (and Guided Exercise) * Customizing OpenSSH Service Configuration (and Guided Exercise) |
| **Lab** | * Configuring and Securing SSH |

# Accessing the Remote Command Line with SSH

## Objectives

After completing this section, you should be able log into a remote system and run commands using **ssh**.

# What is OpenSSH?

OpenSSH implements the Secure Shell or SSH protocol in the Red Hat Enterprise Linux systems. The SSH protocol enables systems to communicate in an encrypted and secure fashion over an insecure network.

You can use the ssh command to create a secure connection to a remote system, authenticate as a specific user, and get an interactive shell session on the remote system as that user. You may also use the ssh command to run an individual command on the remote system without running an interactive shell.

# Secure Shell Examples

The following ssh command would log you in on the remote server remotehost using the same user name as the current local user. In this example, the remote system prompts you to authenticate with that user's password.

**[user01@host ~]$ ssh remotehost**

**user01@remotehost's password: *redhat***

*...output omitted...*

**[user01@remotehost ~]$**

You can the **exit** command to log out of the remote system.

**[user01@remotehost ~]$ exit**

logout

Connection to remotehost closed.

**[user01@host ~]$**

The next ssh command would log you in on the remote server remotehost using the user name user02. Again, you are prompted by the remote system to authenticate with that user's password.

**[user01@host ~]$ ssh user02@remotehost**

**user02@remotehost's password: *shadowman***

*...output omitted...*

**[user02@remotehost ~]$**

This ssh command would run the hostname command on the remotehost remote system as the user02 user without accessing the remote interactive shell.

**[user01@host ~]$ ssh user02@remotehost hostname**

**user02@remotehost's password: *shadowman***

remotehost.lab.example.com

**[user01@host ~]$**

Notice that the preceding command displayed the output in the local system's terminal.

# Identifying Remote Users

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.

**[user01@host ~]$ ssh user01@remotehost**

**user01@remotehost's password: *redhat***

**[user01@remotehost ~]$ w**

16:13:38 up 36 min, 1 user, load average: 0.00, 0.00, 0.00

USER TTY FROM LOGIN@ IDLE JCPU PCPU WHAT

user02 pts/0 172.25.250.10 16:13 7:30 0.01s 0.01s -bash

user01 pts/1 172.25.250.10 16:24 3.00s 0.01s 0.00s w

**[user02@remotehost ~]$**

The preceding output shows that the user02 user has logged in to the system on the pseudo-terminal 0 at 16:13 today from the host with the 172.25.250.10 IP address, and has been idle at a shell prompt for seven minutes and thirty seconds. The preceding output also shows that the user01 user has logged in to the system on the pseudo-terminal 1 and has been idle since since last three seconds after executing the w command.

# SSH host keys

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

When a user uses the ssh command to connect to an SSH server, the command checks to see if it has a copy of the public key for that server in its local known hosts files. The system administrator may have pre-configured it in /etc/ssh/ssh\_known\_hosts, or the user may have a ~/.ssh/known\_hosts file in their home directory that contains the key.

If the client has a copy of the key, ssh will compare the key from the known hosts files for that server to the one it received. If the keys do not match, the client assumes that the network traffic to the server could be hijacked or that the server has been compromised, and seeks the user's confirmation on whether or not to continue with the connection.

## Note

Set the StrictHostKeyChecking parameter to yes in the user-specific ~/.ssh/config file or the system-wide /etc/ssh/ssh\_config to cause the ssh command to always abort the SSH connection if the public keys do not match.

If the client does not have a copy of the public key in its known hosts files, the ssh command will ask you if you want to log in anyway. If you do, a copy of the public key will be saved in your ~/.ssh/known\_hosts file so that the server's identity can be automatically confirmed in the future.

**[user01@host ~]$ ssh newhost**

The authenticity of host 'remotehost (172.25.250.12)' can't be established.

ECDSA key fingerprint is SHA256:qaS0PToLrqlCO2XGklA0iY7CaP7aPKimerDoaUkv720.

**Are you sure you want to continue connecting (yes/no)? yes**

Warning: Permanently added 'newhost,172.25.250.12' (ECDSA) to the list of known hosts.

**user01@newhost's password: *redhat***

*...output omitted...*

**[user01@newhost ~]$**

# SSH Known Hosts Key Management

If a server's public key is changed because the key was lost due to hard drive failure, or replaced for some legitimate reason, you will need to edit the known hosts files to make sure the entry for the old public key is replaced with an entry with the new public key in order to log in without errors.

Public keys are stored in the /etc/ssh/ssh\_known\_hosts and each users' ~/.ssh/known\_hosts file on the SSH client. Each key is on one line. The first field is a list of hostnames and IP addresses that share that public key. The second field is the encryption algorithm for the key. The last field is the key itself.

**[user01@host ~]$ cat ~/.ssh/known\_hosts**

remotehost,172.25.250.11 ecdsa-sha2-nistp256 AAAAE2VjZHNhLXNoYTItbmlzdHAyNTYAAAAIbmlzdHAyNTYAAABBBOsEi0e+FlaNT6jul8Ag5Nj+RViZl0yE2w6iYUr+1fPtOIF0EaOgFZ1LXM37VFTxdgFxHS3D5WhnIfb+68zf8+w=

Each remote SSH server that you conect to stores its public key in the /etc/ssh directory in files with the extension .pub.

**[user01@remotehost ~]$ ls /etc/ssh/\*key.pub**

/etc/ssh/ssh\_host\_ecdsa\_key.pub /etc/ssh/ssh\_host\_ed25519\_key.pub /etc/ssh/ssh\_host\_rsa\_key.pub

## Note

It is a good practice to add entries matching a server's ssh\_host\_\*key.pub files to your ~/.ssh/known\_hosts file or the system-wide /etc/ssh/ssh\_known\_hosts file.

# References

ssh(1), w(1), and hostname(1) man pages

For more information refer to the Using secure communications between two systems with OpenSSH chapter in the Red Hat Enterprise Linux 8 Securing networks Guide at:

<https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html-single/securing_networks/index#using-secure-communications-between-two-systems-with-openssh_securing-networks>

# Guided Exercise: Accessing the Remote Command Line

### In this exercise, you will log into a remote system as different users and execute commands.

## Outcomes

### You should be able to:

### Log in to a remote system.

### Execute commands with the OpenSSH secure shell.

### Log in to workstation as student using student as the password.

### On workstation, run lab ssh-access start to start the exercise. This script ensures that the environment is setup correctly.

**[student@workstation ~]$ lab ssh-access start**

### From workstation, open an SSH session to servera as student.

**[student@workstation ~]$ ssh student@servera**

**...output omitted...**

**[student@servera ~]$**

### Open an SSH session to serverb as student. Accept the host key. Use student as the password when prompted for the password of the student user on serverb.

**[student@servera ~]$ ssh student@serverb**

**The authenticity of host 'serverb (172.25.250.11)' can't be established.**

**ECDSA key fingerprint is SHA256:ERTdjooOIrIwVSZQnqD5or+JbXfidg0udb3DXBuHWzA.**

**Are you sure you want to continue connecting (yes/no)? yes**

**Warning: Permanently added 'serverb,172.25.250.11' (ECDSA) to the list of known hosts.**

**student@serverb's password: student**

**...output omitted...**

**[student@serverb ~]$**

The host key is recorded in the /home/student/.ssh/known\_hosts file on servera to identify serverb because the student user has initiated the SSH connection from servera. If the /home/student/.ssh/known\_hosts file does not already exist, it comes into existence as a new file along with the new entry in it. The **ssh** command fails to execute properly if the remote host appears to have a different key than the recorded key.

### Run the w command to display the users that are currently logged in to serverb.

**[student@serverb ~]$ w**

**18:49:29 up 2:55, 1 user, load average: 0.00, 0.00, 0.00**

**USER TTY FROM LOGIN@ IDLE JCPU PCPU WHAT**

**student** pts/0 **172.25.250.10** 18:33 0.00s 0.01s 0.00s w

The preceding output indicates that the student user has logged in to the system from the host with an IP address of 172.25.250.10 which is servera in the classroom network.

Note:

The IP address of a system identifies the system on a network. You are going to learn about IP addresses in the later chapter.

### Exit the student user's shell on serverb.

**[student@serverb ~]$ exit**

**logout**

**Connection to serverb closed.**

**[student@servera ~]$**

### Open an SSH session to serverb as root. Use redhat as the password of the root user.

**[student@servera ~]$ ssh root@serverb**

**root@serverb's password: redhat**

**...output omitted...**

**[root@serverb ~]#**

Notice that the preceding ssh command did not ask you to accept the host key because it was found among the known hosts. Should the identity of serverb change at any time, OpenSSH prompts you to verify and accept the new host key.

### Run the w command to display the users that are currently logged in to serverb.

**[root@serverb ~]# w**

**19:10:28 up 3:16, 1 user, load average: 0.00, 0.00, 0.00**

**USER TTY FROM LOGIN@ IDLE JCPU PCPU WHAT**

**root** pts/0 **172.25.250.10** 19:09 1.00s 0.01s 0.00s w

The preceding output indicates that the root user has logged in to the system from the host with an IP address of 172.25.250.10 which is servera in the classroom network.

### Exit the root user's shell on serverb.

**[root@serverb ~]# exit**

**logout**

**Connection to serverb closed.**

**[student@servera ~]$**

### Remove the /home/student/.ssh/known\_hosts file from servera. This causes ssh to lose the recorded identities of the remote systems.

**[student@servera ~]$ rm /home/student/.ssh/known\_hosts**

Host keys can change for legitimate reasons: perhaps the remote machine was replaced because of a hardware failure, or perhaps the remote machine was reinstalled. Usually, it is advisable only to remove the key entry for the particular host in the known\_hosts file. Since this particular known\_hosts file has only one entry, you can remove the entire file.

### Open an SSH session to serverb as student. Accept the host key if asked. Use student as the password when prompted for the password of the student user on serverb.

**[student@servera ~]$ ssh student@serverb**

**The authenticity of host 'serverb (172.25.250.11)' can't be established.**

**ECDSA key fingerprint is SHA256:ERTdjooOIrIwVSZQnqD5or+JbXfidg0udb3DXBuHWzA.**

**Are you sure you want to continue connecting (yes/no)? yes**

**Warning: Permanently added 'serverb,172.25.250.11' (ECDSA) to the list of known hosts.**

**student@serverb's password: student**

**...output omitted...**

**[student@serverb ~]$**

Notice that the ssh command asked for your confirmation to accept or reject the host key because it could not find one for the remote host.

### Exit the student user's shell on serverb and confirm that a new instance of known\_hosts exists on servera.

**[student@serverb ~]$ exit**

**logout**

**Connection to serverb closed.**

**[student@servera ~]$ ls -l /home/student/.ssh/known\_hosts**

-rw-r--r--. 1 student student 183 Feb 1 20:26 /home/student/.ssh/known\_hosts

### Confirm that the new instance of known\_hosts file has the host key of serverb.

**[student@servera ~]$ cat /home/student/.ssh/known\_hosts**

**serverb,172.25.250.11 ecdsa-sha2-nistp256 AAAAE2VjZHNhLXNoYTItbmlzdHAyNTYAAAAIbmlzdHAyNTYAAABBBI9LEYEhwmU1rNqnbBPukH2Ba0/QBAu9WbS4m03B3MIhhXWKFFNa/UlNjY8NDpEM+hkJe/GmnkcEYMLbCfd9nMA=**

Actual output will vary.

### Run hostname remotely on serverb without accessing the interactive shell.

**[student@servera ~]$ ssh student@serverb hostname**

**student@serverb's password: student**

**serverb.lab.example.com**

The preceding command displayed the full hostname of the remote system serverb.

### Exit the student user's shell on servera.

**[student@servera ~]$ exit**

**logout**

**Connection to servera closed.**

# Finish

On workstation, run lab ssh-access finish to complete this exercise.

**[student@workstation ~]$ lab ssh-access finish**

This concludes the guided exercise.

# Configuring SSH Key-based Authentication

## Objectives

After completing this section, you should be able to configure a user account to use key-based authentication to log in to remote systems securely without a password.

# SSH Key-based Authentication

You can configure an SSH server to allow you to authenticate without a password by using key-based authentication. This is based on a private-public key scheme.

To do this, you generate a matched pair of cryptographic key files. One is a private key, the other a matching 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 connect to, and is used to verify the private key. The public key does not need to be secret.

You put a copy of the public key in your account on the server. When you try to log in, the SSH server can use the public key to issue a challenge that can only be correctly answered by using the private key. As a result, your ssh client can automatically authenticate your login to the server with your unique copy of the private key. This allows you to securely access systems in a way that doesn't require you to enter a password interactively every time.

# Generating SSH Keys

To create a private key and matching public key for authentication, use the ssh-keygen command. By default, your private and public keys are saved in your ~/.ssh/id\_rsa and ~/.ssh/id\_rsa.pub files, respectively.

**[user@host ~]$ ssh-keygen**

**Generating public/private rsa key pair.**

**Enter file in which to save the key (/home/user/.ssh/id\_rsa): Enter**

**Created directory '/home/user/.ssh'.**

**Enter passphrase (empty for no passphrase): Enter**

**Enter same passphrase again: Enter**

**Your identification has been saved in /home/user/.ssh/id\_rsa.**

**Your public key has been saved in /home/user/.ssh/id\_rsa.pub.**

**The key fingerprint is:**

**SHA256:vxutUNPio3QDCyvkYm1oIx35hmMrHpPKWFdIYu3HV+w user@host.lab.example.com**

**The key's randomart image is:**

**+---[RSA 2048]----+**

**| |**

**| . . |**

**| o o o |**

**| . = o o . |**

**| o + = S E . |**

**| ..O o + \* + |**

**|.+% O . + B . |**

**|=\*oO . . + \* |**

**|++. . +. |**

**+----[SHA256]-----+**

If you do not specify a passphrase when ssh-keygen prompts you, the generated private key is not protected. In this case, anyone with your private key file could use it for authentication. If you set a passphrase, then you will need to enter that passphrase when you use the private key for authentication. (Therefore, you would be using the private key's passphrase rather than your password on the remote host to authenticate.)

You can run a helper program called ssh-agent which can temporarily cache your private key passphrase in memory at the start of your session to get true passwordless authentication. This will be discussed later in this section.

The following example of the ssh-keygen command shows the creation of the passphrase-protected private key alongside the public key.

**[user@host ~]$ ssh-keygen -f .ssh/key-with-pass**

**Generating public/private rsa key pair.**

**Enter passphrase (empty for no passphrase):**

**Enter same passphrase again:**

**Your identification has been saved in .ssh/key-with-pass.**

**Your public key has been saved in .ssh/key-with-pass.pub.**

**The key fingerprint is:**

**SHA256:w3GGB7EyHUry4aOcNPKmhNKS7dl1YsMVLvFZJ77VxAo user@host.lab.example.com**

**The key's randomart image is:**

**+---[RSA 2048]----+**

**| . + =.o ... |**

**| = B XEo o. |**

**| . o O X =.... |**

**| = = = B = o. |**

**|= + \* \* S . |**

**|.+ = o + . |**

**| + . |**

**| |**

**| |**

**+----[SHA256]-----+**

The -f option with the ssh-keygen command determines the files where the keys are saved. In the preceding example, the private and public keys are saved in the /home/user/.ssh/key-with-pass /home/user/.ssh/key-with-pass.pub files, respectively.

## Warning

During further SSH keypair generation, unless you specify a unique file name, you are prompted for permission to overwrite the existing id\_rsa and id\_rsa.pub files. If you overwrite the existing id\_rsa and id\_rsa.pub files, then you must replace the old public key with the new one on all the SSH servers that have your old public key.

Once the SSH keys have been generated, they are stored by default in the .ssh/ directory of the user's home directory. The permission modes must be 600 on the private key and 644 on the public key.

# Sharing the Public Key

Before key-based authentication can be used, the public key needs to be copied to the destination system. The ssh-copy-id command copies the public key of the SSH keypair to the destination system.

If you omit the path to the public key file while running ssh-copy-id, it uses the default /home/user/.ssh/id\_rsa.pub file.

**[user@host ~]$ ssh-copy-id -i .ssh/key-with-pass.pub user@remotehost**

**/usr/bin/ssh-copy-id: INFO: Source of key(s) to be installed: "/home/user/.ssh/id\_rsa.pub"**

**/usr/bin/ssh-copy-id: INFO: attempting to log in with the new key(s), to filter out any that are already installed**

**/usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- if you are prompted now it is to install the new keys**

**user@remotehost's password: redhat**

**Number of key(s) added: 1**

**Now try logging into the machine, with: "ssh 'user@remotehost'"**

**and check to make sure that only the key(s) you wanted were added.**

After the public key is successfully transferred to a remote system, you can authenticate to the remote system using the corresponding private key while logging in to the remote system over SSH. If you omit the path to the private key file while running the ssh command, it uses the default /home/user/.ssh/id\_rsa file.

**[user@host ~]$ ssh -i .ssh/key-with-pass user@remotehost**

**Enter passphrase for key '.ssh/key-with-pass': redhatpass**

**...output omitted...**

**[user@remotehost ~]$ exit**

**logout**

**Connection to remotehost closed.**

**[user@host ~]$**

# Using ssh-agent for Non-interactive Authentication

If your SSH private key is protected with a passphrase, you normally have to enter the passphrase to use the private key for authentication. However, you can use a program called ssh-agent to temporarily cache the passphrase in memory. Then any time that you use SSH to log in to another system with the private key, ssh-agent will automatically provide the passphrase for you. This is convenient, and can improve security by providing fewer opportunities for someone "shoulder surfing" to see you type the passphrase in.

Depending on your local system's configuration, if you initially log in to the GNOME graphical desktop environment, the ssh-agent program might automatically be started and configured for you.

If you log in on a text console, log in using ssh, or use sudo or su, you will probably need to start ssh-agent manually for that session. You can do this with the following command:

**[user@host ~]$ eval $(ssh-agent)**

**Agent pid 10155**

**[user@host ~]$**

## Note

When you run ssh-agent, it prints out some shell commands. You need to run these commands to set environment variables used by programs like ssh-add to communicate with it. The eval $(ssh-agent) command starts ssh-agent and runs those commands to automatically set those environment variables for that shell session. It also displays the PID of the ssh-agent process.

Once ssh-agent is running, you need to tell it the passphrase for your private key or keys. You can do this with the ssh-add command.

The following ssh-add commands add the private keys from /home/user/.ssh/id\_rsa (the default) and /home/user/.ssh/key-with-pass files, respectively.

**[user@host ~]$ ssh-add**

**Identity added: /home/user/.ssh/id\_rsa (user@host.lab.example.com)**

**[user@host ~]$ ssh-add .ssh/key-with-pass**

**Enter passphrase for .ssh/key-with-pass: redhatpass**

**Identity added: .ssh/key-with-pass (user@host.lab.example.com)**

After successfully adding the private keys to the ssh-agent process, you can invoke an SSH connection using the ssh command. If you are using any private key file other than the default /home/user/.ssh/id\_rsa file, then you must use the -i option with the ssh command to specify the path to the private key file.

The following example of the ssh command uses the default private key file to authenticate to an SSH server.

**[user@host ~]$ ssh user@remotehost**

**Last login: Fri Apr 5 10:53:50 2019 from host.example.com**

**[user@remotehost ~]$**

The following example of the ssh command uses the /home/user/.ssh/key-with-pass (non-default) private key file to authenticate to an SSH server. The private key in the following example has already been decrypted and added to its parent ssh-agent process, so the ssh command does not prompt you to decrypt the private key by interactively entering its passphrase.

**[user@host ~]$ ssh -i .ssh/key-with-pass user@remotehost**

**Last login: Mon Apr 8 09:44:20 2019 from host.example.com**

**[user@remotehost ~]$**

When you log out of the session that started ssh-agent, the process will exit and your the passphrases for your private keys will be cleared from memory.

# References

ssh-keygen(1), ssh-copy-id(1), ssh-agent(1),ssh-add(1) man pages

# Guided Exercise: Configuring SSH Key-based Authentication

In this exercise, you will configure a user to use key-based authentication for SSH.

## Outcomes

You should be able to:

* Generate an SSH key pair without passphrase protection.
* Generate an SSH key pair with passphrase protection.
* Authenticate using both passphrase-less and passphrase-protected SSH keys.

## Log in to workstation as student using student as the password.

On workstation, run lab ssh-configure start to start the exercise. This script creates the necessary user accounts.

**[student@workstation ~]$ lab ssh-configure start**

### From workstation, open an SSH session to serverb as student.

**[student@workstation ~]$ ssh student@serverb**

**...output omitted...**

**[student@serverb ~]$**

### Use the su command to switch to the operator1 user on serverb. Use redhat as the password of operator1.

**[student@serverb ~]$ su - operator1**

**Password: redhat**

**[operator1@serverb ~]$**

### Use the ssh-keygen command to generate SSH keys. Do not enter a passphrase.

**[operator1@serverb ~]$ ssh-keygen**

**Generating public/private rsa key pair.**

**Enter file in which to save the key (/home/operator1/.ssh/id\_rsa): Enter**

**Created directory '/home/operator1/.ssh'.**

**Enter passphrase (empty for no passphrase): Enter**

**Enter same passphrase again: Enter**

**Your identification has been saved in /home/operator1/.ssh/id\_rsa.**

**Your public key has been saved in /home/operator1/.ssh/id\_rsa.pub.**

**The key fingerprint is:**

**SHA256:JainiQdnRosC+xXhOqsJQQLzBNUldb+jJbyrCZQBERI operator1@serverb.lab.example.com**

**The key's randomart image is:**

**+---[RSA 2048]----+**

**|E+\*+ooo . |**

**|.= o.o o . |**

**|o.. = . . o |**

**|+. + \* . o . |**

**|+ = X . S + |**

**| + @ + = . |**

**|. + = o |**

**|.o . . . . |**

**|o o.. |**

**+----[SHA256]-----+**

### Use the ssh-copy-id command to send the public key of the SSH key pair to operator1 on servera. Use redhat as the password of operator1 on servera.

**[operator1@serverb ~]$ ssh-copy-id operator1@servera**

**/usr/bin/ssh-copy-id: INFO: Source of key(s) to be installed: "/home/operator1/.ssh/id\_rsa.pub"**

**The authenticity of host 'servera (172.25.250.10)' can't be established.**

**ECDSA key fingerprint is SHA256:ERTdjooOIrIwVSZQnqD5or+JbXfidg0udb3DXBuHWzA.**

**Are you sure you want to continue connecting (yes/no)? yes**

**/usr/bin/ssh-copy-id: INFO: attempting to log in with the new key(s), to filter out any that are already installed**

**/usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- if you are prompted now it is to install the new keys**

**operator1@servera's password: redhat**

**Number of key(s) added: 1**

**Now try logging into the machine, with: "ssh 'operator1@servera'"**

**and check to make sure that only the key(s) you wanted were added.**

### Execute the hostname command on servera remotely using SSH without accessing the remote interactive shell.

**[operator1@serverb ~]$ ssh operator1@servera hostname**

**servera.lab.example.com**

Notice that the preceding ssh command did not prompt you for a password because it used the passphrase-less private key against the exported public key to authenticate as operator1 on servera. This approach is not secure, because anyone who has access to the private key file can log in to servera as operator1. The secure alternative is to protect the private key with a passphrase, which is the next step.

### Use the ssh-keygen command to generate another set of SSH keys with passphrase-protection. Save the key as /home/operator1/.ssh/key2. Use redhatpass as the passphrase of the private key.

## Warning

If you do not specify the file where the key gets saved, the default file (/home/user/.ssh/id\_rsa) is used. You have already used the default file name when generating SSH keys in the preceding step, so it is vital that you specify a non-default file, otherwise the existing SSH keys will be overwritten.

**[operator1@serverb ~]$ ssh-keygen -f .ssh/key2**

**Generating public/private rsa key pair.**

**Enter passphrase (empty for no passphrase): redhatpass**

**Enter same passphrase again: redhatpass**

**Your identification has been saved in .ssh/key2.**

**Your public key has been saved in .ssh/key2.pub.**

**The key fingerprint is:**

**SHA256:OCtCjfPm5QrbPBgqbEIWCcw5AI4oSlMEbgLrBQ1HWKI operator1@serverb.lab.example.com**

**The key's randomart image is:**

**+---[RSA 2048]----+**

**|O=X\* |**

**|OB=. |**

**|E\*o. |**

**|Booo . |**

**|..= . o S |**

**| +.o o |**

**|+.oo+ o |**

**|+o.O.+ |**

**|+ . =o. |**

**+----[SHA256]-----+**

### Use the ssh-copy-id command to send the public key of the passphrase-protected key pair to operator1 on servera.

**[operator1@serverb ~]$ ssh-copy-id -i .ssh/key2.pub operator1@servera**

**/usr/bin/ssh-copy-id: INFO: Source of key(s) to be installed: ".ssh/key2.pub"**

**/usr/bin/ssh-copy-id: INFO: attempting to log in with the new key(s), to filter out any that are already installed**

**/usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- if you are prompted now it is to install the new keys**

**Number of key(s) added: 1**

**Now try logging into the machine, with: "ssh 'operator1@servera'"**

**and check to make sure that only the key(s) you wanted were added.**

Notice that the preceding ssh-copy-id command did not prompt you for a password because it used the public key of the passphrase-less private key that you exported to servera in the preceding step.

### Execute the hostname command on servera remotely with SSH without accessing the remote interactive shell. Use /home/operator1/.ssh/key2 as the identity file. Specify redhatpass as the passphrase, which you set for the private key in the preceding step.

**[operator1@serverb ~]$ ssh -i .ssh/key2 operator1@servera hostname**

**Enter passphrase for key '.ssh/key2': redhatpass**

**servera.lab.example.com**

Notice that the preceding ssh command prompted you for the passphrase you used to protect the private key of the SSH key pair. This passphrase protects the private key. Should an attacker gain access to the private key, the attacker cannot use it to access other systems because the private key itself is protected with a passphrase. The ssh command uses a different passphrase than the one for operator1 on servera, requiring users to know both.

You can use ssh-agent, as in the following step, to avoid interactively typing in the passphrase while logging in with SSH. Using ssh-agent is both more convenient and more secure in situations where the administrators log in to remote systems regularly.

### Run ssh-agent in your Bash shell and add the passphrase-protected private key (/home/operator1/.ssh/key2) of the SSH key pair to the shell session.

**[operator1@serverb ~]$ eval $(ssh-agent)**

**Agent pid 21032**

**[operator1@serverb ~]$ ssh-add .ssh/key2**

**Enter passphrase for .ssh/key2: redhatpass**

**Identity added: .ssh/key2 (operator1@serverb.lab.example.com)**

The preceding eval command started ssh-agent and configured this shell session to use it. You then used ssh-add to provide the unlocked private key to ssh-agent.

### Execute the hostname command on servera remotely without accessing a remote interactive shell. Use /home/operator1/.ssh/key2 as the identity file.

**[operator1@serverb ~]$ ssh -i .ssh/key2 operator1@servera hostname**

**servera.lab.example.com**

Notice that the preceding **ssh** command did not prompt you to enter the passphrase interactively.

### Open another terminal on workstation and open an SSH session to serverb as student.

**[student@workstation ~]$ ssh student@serverb**

**...output omitted...**

**[student@serverb ~]$**

### On serverb, use the su command to switch to operator1 and invoke an SSH connection to servera. Use /home/operator1/.ssh/key2 as the identity file to authenticate using the SSH keys.

Use the su command to switch to operator1. Use redhat as the password of operator1.

**[student@serverb ~]$ su - operator1**

**Password: redhat**

**[operator1@serverb ~]$**

Open an SSH session to servera as operator1.

**[operator1@serverb ~]$ ssh -i .ssh/key2 operator1@servera**

**Enter passphrase for key '.ssh/key2': redhatpass**

**...output omitted...**

**[operator1@servera ~]$**

Notice that the preceding ssh command prompted you to enter the passphrase interactively because you did not invoke the SSH connection from the shell that you used to start ssh-agent.

### Exit all the shells you are using in the second terminal.

Log out of servera.

**[operator1@servera ~]$ exit**

**logout**

**Connection to servera closed.**

**[operator1@serverb ~]$**

Exit the operator1 and student shells on serverb to return to the student user's shell on workstation.

**[operator1@serverb ~]$ exit**

**logout**

**[student@serverb ~]$ exit**

**logout**

**Connection to serverb closed.**

**[student@workstation ~]$**

Close the second terminal on workstation.

**[student@workstation ~]$ exit**

### Log out of serverb on the first terminal and conclude this exercise.

From the first terminal, exit the operator1 user's shell on serverb.

**[operator1@serverb ~]$ exit**

**logout**

**[student@serverb ~]$**

The exit command caused you to exit the operator1 user's shell, terminating the shell session where ssh-agent was active, and return to the student user's shell on serverb.

Exit the student user's shell on serverb to return to the student user's shell on workstation.

**[student@serverb ~]$ exit**

**logout**

**Connection to serverb closed.**

**[student@workstation ~]$**

# Finish

On workstation, run lab ssh-configure finish to complete this exercise.

**[student@workstation ~]$ lab ssh-configure finish**

This concludes the guided exercise.

# Customizing OpenSSH Service Configuration

## Objectives

After completing this section, you should be able to restrict direct logins as root and disable password-based authentication for the OpenSSH service.

# Configuring the OpenSSH Server

OpenSSH service is provided by a daemon called sshd. Its main configuration file is /etc/ssh/sshd\_config.

The default configuration of the OpenSSH server works well. However, you might want to make some changes to strengthen the security of your system. There are two common changes you might want to make. You might want to prohibit direct remote login to the root account, and you might want to prohibit password-based authentication (in favor of SSH private key authentication).

# Prohibit the Superuser From Logging in Using SSH

It is a good practice to prohibit direct login to the root user account from remote systems. Some of the risks of allowing direct login as root include:

* The user name root exists on every Linux system by default, so a potential attacker only has to guess the password, instead of a valid user name and password combination. This reduces complexity for an attacker.
* The root user has unrestricted privileges, so its compromise can lead to maximum damage to the system.
* From an auditing perspective, it can be hard to track which authorized user logged in as root and made changes. If users have to log in as a regular user and switch to the root account, this generates a log event that can be used to help provide accountability.

The OpenSSH server uses the PermitRootLogin configuration setting in the /etc/ssh/sshd\_config configuration file to allow or prohibit users logging in to the system as root.

**PermitRootLogin yes**

With the PermitRootLogin parameter to yes, as it is by default, people are permitted to log in as root. To prevent this, set the value to no. Alternatively, to prevent password-based authentication but allow private key-based authentication for root, set the PermitRootLogin parameter to without-password.

The SSH server (sshd) must be reloaded for any changes to take effect.

**[root@host ~]# systemctl reload sshd**

# Prohibiting Password-Based Authentication for SSH

Allowing only private key-based logins to the remote command line has various advantages:

* Attackers cannot use password guessing attacks to remotely break into known accounts on the system.
* With passphrase-protected private keys, an attacker needs both the passphrase and a copy of the private key. With passwords, an attacker just needs the password.
* By using passphrase-protected private keys in conjunction with ssh-agent, the passphrase is exposed less frequently since it is entered less frequently, and logging in is more convenient for the user.

The OpenSSH server uses the PasswordAuthentication parameter in the /etc/ssh/sshd\_config configuration file to control whether users can use password-based authentication to log in to the system.

**PasswordAuthentication yes**

The default value of yes for the PasswordAuthentication parameter in the /etc/ssh/sshd\_config configuration file causes the SSH server to allow users to use password-based authentication while logging in. The value of no for PasswordAuthentication prevents users from using password-based authentication.

Keep in mind that whenever you change the /etc/ssh/sshd\_config file, you must reload the sshd service for changes to take effect.

## Important

Remember, if you turn off password-based authentication for ssh, you need to have a way to ensure that the user's ~/.ssh/authorized\_keys file on the remote server is populated with their public key, so that they can log in.

# References

ssh(1), sshd\_config(5) man pages

# Guided Exercise: Customizing OpenSSH Service Configuration

In this exercise, you will disable direct logins as root and password-based authentication for the OpenSSH service on one of your servers.

## Outcomes

You should be able to:

* Disable direct logins as root over ssh.
* Disable password-based authentication for remote users to connect over SSH.

## Log in to workstation as student using student as the password.

## On workstation, run lab ssh-customize start to start the exercise. This script creates the necessary user accounts and files.

**[student@workstation ~]$ lab ssh-customize start**

### From workstation, open an SSH session to serverb as student.

**[student@workstation ~]$ ssh student@serverb**

**...output omitted...**

**[student@serverb ~]$**

### Use the su command to switch to operator2 on serverb. Use redhat as the password of operator2.

**[student@serverb ~]$ su - operator2**

**Password: redhat**

**[operator2@serverb ~]$**

### Use the ssh-keygen command to generate SSH keys. Do not enter any passphrase for the keys.

**[operator2@serverb ~]$ ssh-keygen**

**Generating public/private rsa key pair.**

**Enter file in which to save the key (/home/operator2/.ssh/id\_rsa): Enter**

**Created directory '/home/operator2/.ssh'.**

**Enter passphrase (empty for no passphrase): Enter**

**Enter same passphrase again: Enter**

**Your identification has been saved in /home/operator2/.ssh/id\_rsa.**

**Your public key has been saved in /home/operator2/.ssh/id\_rsa.pub.**

**The key fingerprint is:**

**SHA256:JainiQdnRosC+xXhOqsJQQLzBNUldb+jJbyrCZQBERI operator1@serverb.lab.example.com**

**The key's randomart image is:**

**+---[RSA 2048]----+**

**|E+\*+ooo . |**

**|.= o.o o . |**

**|o.. = . . o |**

**|+. + \* . o . |**

**|+ = X . S + |**

**| + @ + = . |**

**|. + = o |**

**|.o . . . . |**

**|o o.. |**

**+----[SHA256]-----+**

### Use the ssh-copy-id command to send the public key of the SSH key pair to operator2 on servera. Use redhat as the password of operator2 on servera.

**[operator2@serverb ~]$ ssh-copy-id operator2@servera**

**/usr/bin/ssh-copy-id: INFO: Source of key(s) to be installed: "/home/operator1/.ssh/id\_rsa.pub"**

**The authenticity of host 'servera (172.25.250.10)' can't be established.**

**ECDSA key fingerprint is SHA256:ERTdjooOIrIwVSZQnqD5or+JbXfidg0udb3DXBuHWzA.**

**Are you sure you want to continue connecting (yes/no)? yes**

**/usr/bin/ssh-copy-id: INFO: attempting to log in with the new key(s), to filter out any that are already installed**

**/usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- if you are prompted now it is to install the new keys**

**operator2@servera's password: redhat**

**Number of key(s) added: 1**

**Now try logging into the machine, with: "ssh 'operator2@servera'"**

**and check to make sure that only the key(s) you wanted were added.**

### Confirm that you can successfully log in to servera as operator2 using the SSH keys.

Open an SSH session to servera as operator2.

**[operator2@serverb ~]$ ssh operator2@servera**

**...output omitted...**

**[operator2@servera ~]$**

Notice that the preceding ssh command used SSH keys for authentication.

Log out of servera.

**[operator2@servera ~]$ exit**

**logout**

**Connection to servera closed.**

### Confirm that you can successfully log in to servera as root using redhat as the password.

Open an SSH session to servera as root using redhat as the password.

**[operator2@serverb ~]$ ssh root@servera**

**root@servera's password: redhat**

**...output omitted...**

**[root@servera ~]#**

Notice that the preceding ssh command used the password of the superuser for authentication because SSH keys do not exist for the superuser.

Log out of servera.

**[root@servera ~]# exit**

**logout**

**Connection to servera closed.**

**[operator2@serverb ~]$**

### Confirm that you can successfully log in to servera as operator3 using redhat as the password.

Open an SSH session to servera as operator3 using redhat as the password.

**[operator2@serverb ~]$ ssh operator3@servera**

**operator3@servera's password: redhat**

**...output omitted...**

**[operator3@servera ~]$**

Notice that the preceding ssh command used the password of operator3 for authentication because SSH keys do not exist for operator3.

Log out of servera.

**[operator3@servera ~]$ exit**

**logout**

**Connection to servera closed.**

**[operator2@serverb ~]$**

### Configure sshd on servera to prevent users logging in as root. Use redhat as the password of the superuser when required.

Open an SSH session to servera as operator2 using the SSH keys.

**[operator2@serverb ~]$ ssh operator2@servera**

**...output omitted...**

**[operator2@servera ~]$**

On servera, switch to root. Use redhat as the password of the root user.

**[operator2@servera ~]$ su -**

**Password: redhat**

**[root@servera ~]#**

Set PermitRootLogin to no in /etc/ssh/sshd\_config and reload sshd. You may use vim /etc/ssh/sshd\_config to edit the configuration file of sshd.

**...output omitted...**

**PermitRootLogin no**

**...output omitted...**

**[root@servera ~]# systemctl reload sshd**

Open another terminal on workstation and open an SSH session to serverb as operator2. From serverb, try logging in to servera as root. This should fail because you disabled root user login over SSH in the preceding step.

Note

For your convenience, password-less login is already configured between workstation and serverb in the classroom environment.

**[student@workstation ~]$ ssh operator2@serverb**

**...output omitted...**

**[operator2@serverb ~]$ ssh root@servera**

**root@servera's password: redhat**

**Permission denied, please try again.**

**root@servera's password: redhat**

**Permission denied, please try again.**

**root@servera's password: redhat**

**root@servera: Permission denied (publickey,gssapi-keyex,gssapi-with-mic,password).**

By default, the ssh command attempts to authenticate using key-based authentication first and then, if that fails, password-based authentication.

### Configure sshd on servera to allow users to authenticate using SSH keys only, rather than their passwords.

Return to the first terminal that has the root user's shell active on servera. Set PasswordAuthentication to no in /etc/ssh/sshd\_config and reload sshd. You may use vim /etc/ssh/sshd\_config to edit the configuration file of sshd.

**...output omitted...**

**PasswordAuthentication no**

**...output omitted...**

**[root@servera ~]# systemctl reload sshd**

Go to the second terminal that has operator2 user's shell active on serverb and try logging in to servera as operator3. This should fail because SSH keys are not configured for operator3, and the sshd service on servera does not allow the use of passwords for authentication.

**[operator2@serverb ~]$ ssh operator3@servera**

**operator3@servera: Permission denied (publickey,gssapi-keyex,gssapi-with-mic).**

Note

For more granularity, you may use the explicit -o PubkeyAuthentication=no and -o PasswordAuthentication=yes options with the ssh command. This allows you to override the ssh command's defaults and confidently determine that the preceding command fails based on the settings you adjusted in /etc/ssh/sshd\_config in the preceding step.

Return to the first terminal that has the root user's shell active on servera. Verify that PubkeyAuthentication is enabled in /etc/ssh/sshd\_config. You may use vim /etc/ssh/sshd\_config to view the configuration file of sshd.

**...output omitted...**

**#PubkeyAuthentication yes**

**...output omitted...**

Notice that the PubkeyAuthentication line is commented. Any commented line in this file uses the default value. Commented lines indicate the default values of a parameter. The public key authentication of SSH is active by default, as the commented line indicates.

Return to the second terminal that has operator2 user's shell active on serverb and try logging in to servera as operator2. This should succeed because the SSH keys are configured for operator2 to log in to servera from serverb.

**[operator2@serverb ~]$ ssh operator2@servera**

**...output omitted...**

**[operator2@servera ~]$**

From the second terminal, exit the operator2 user's shell on both servera and serverb.

**[operator2@servera ~]$ exit**

**logout**

**Connection to servera closed.**

**[operator2@serverb ~]$ exit**

**logout**

**Connection to serverb closed.**

**[student@workstation ~]$**

Close the second terminal on workstation.

**[student@workstation ~]$ exit**

From the first terminal, exit the root user's shell on servera.

**[root@servera ~]# exit**

**logout**

From the first terminal, exit the operator2 user's shell on both servera and serverb.

**[operator2@servera ~]$ exit**

**logout**

**Connection to servera closed.**

**[operator2@serverb ~]$ exit**

**logout**

**[student@serverb ~]$**

Log out of serverb and return to the student user's shell on workstation.

**[student@serverb ~]$ exit**

**logout**

**Connection to serverb closed.**

**[student@workstation ~]$**

# Finish

On workstation, run lab ssh-customize finish to complete this exercise.

**[student@workstation ~]$ lab ssh-customize finish**

This concludes the guided exercise.

# Summary

## In this chapter, you learned:

* The **ssh** command allows users to access remote systems securely using the SSH protocol.
* A client system stores remote servers' identities in ~/.ssh/known\_hosts and /etc/ssh/ssh\_known\_hosts.
* SSH supports both password-based and key-based authentication.
* The **ssh-keygen** command generates an SSH key pair for authentication. The **ssh-copy-id** command exports the public key to remote systems.
* The sshd service implements the SSH protocol on Red Hat Enterprise Linux systems.
* It is a recommended practice to configure sshd to disable remote logins as root and to require public key authentication rather than password-based authentication.