Red Hat Enterprise Linux 8

Managing, monitoring and updating the kernel

A guide to managing the Linux kernel on Red Hat Enterprise Linux 8

Last Updated: 2020-07-29

Red Hat Enterprise Linux 8 Managing, monitoring and updating the kernel

A guide to managing the Linux kernel on Red Hat Enterprise Linux 8

Legal Notice

Abstract

This document provides the users and administrators with necessary information about configuring their workstations on the Linux kernel level. Such adjustments bring performance enhancements, easier troubleshooting or optimized system.

Table of Contents

PROVIDING FEEDBACK ON RED HAT DOCUMENTATION

We appreciate your input on our documentation. Please let us know how we could make it better. To do so:

- For simple comments on specific passages:

  1. Make sure you are viewing the documentation in the *Multi-page HTML* format. In addition, ensure you see the **Feedback** button in the upper right corner of the document.
  2. Use your mouse cursor to highlight the part of text that you want to comment on.
  3. Click the **Add Feedback** pop-up that appears below the highlighted text.
  4. Follow the displayed instructions.

- For submitting more complex feedback, create a Bugzilla ticket:

  1. Go to the Bugzilla website.
  2. As the Component, use **Documentation**.
  3. Fill in the **Description** field with your suggestion for improvement. Include a link to the relevant part(s) of documentation.
  4. Click **Submit Bug**.

CHAPTER 1. THE LINUX KERNEL RPM

The following sections describe the Linux kernel RPM package provided and maintained by Red Hat.

1.1. WHAT AN RPM IS

An RPM package is a file containing other files and their metadata (information about the files that are needed by the system).

Specifically, an RPM package consists of the **cpio** archive.

The **cpio** archive contains:

- Files
- RPM header (package metadata)
  The **rpm** package manager uses this metadata to determine dependencies, where to install files, and other information.

Types of RPM packages

There are two types of RPM packages. Both types share the file format and tooling, but have different contents and serve different purposes:

- Source RPM (SRPM)
  An SRPM contains source code and a SPEC file, which describes how to build the source code into a binary RPM. Optionally, the patches to source code are included as well.
- Binary RPM
  A binary RPM contains the binaries built from the sources and patches.

1.2. THE LINUX KERNEL RPM PACKAGE OVERVIEW

The **kernel** RPM is a meta package that does not contain any files, but rather ensures that the following sub-packages are properly installed:

- **kernel-core** - contains a minimal number of kernel modules needed for core functionality. This sub-package alone could be used in virtualized and cloud environments to provide a Red Hat Enterprise Linux 8 kernel with a quick boot time and a small disk size footprint.
- **kernel-modules** - contains further kernel modules.
- **kernel-modules-extra** - contains kernel modules for rare hardware.

The small set of **kernel** sub-packages above aims to provide a reduced maintenance surface to system administrators especially in virtualized and cloud environments.

The other common kernel packages are for example:

- **kernel-debug** — Contains a kernel with numerous debugging options enabled for kernel diagnosis, at the expense of reduced performance.
- **kernel-tools** — Contains tools for manipulating the Linux kernel and supporting documentation.
- **kernel-devel** — Contains the kernel headers and makefiles sufficient to build modules against the **kernel** package.

- **kernel-abi-whitelists** – Contains information pertaining to the Red Hat Enterprise Linux kernel ABI, including a list of kernel symbols that are needed by external Linux kernel modules and a **yum** plug-in to aid enforcement.

- **kernel-headers** – Includes the C header files that specify the interface between the Linux kernel and user-space libraries and programs. The header files define structures and constants that are needed for building most standard programs.

1.3. DISPLAYING CONTENTS OF THE KERNEL PACKAGE

The following procedure describes how to view the contents of the kernel package and its sub-packages without installing them using the **rpm** command.

Prerequisites

- Obtained **kernel**, **kernel-core**, **kernel-modules**, **kernel-modules-extra** RPM packages for your CPU architecture

Procedure

- List modules for **kernel**:

```
$ rpm -qlp <kernel_rpm>
(contains no files)
…
```

- List modules for **kernel-core**:

```
$ rpm -qlp <kernel-core_rpm>
…
/lib/modules/4.18.0-80.el8.x86_64/kernel/fs/udf/udf.ko.xz
/lib/modules/4.18.0-80.el8.x86_64/kernel/fs/xfs
/lib/modules/4.18.0-80.el8.x86_64/kernel/fs/xfs/xfs.ko.xz
/lib/modules/4.18.0-80.el8.x86_64/kernel/kernel
/lib/modules/4.18.0-80.el8.x86_64/kernel/kernel/trace
/lib/modules/4.18.0-80.el8.x86_64/kernel/kernel/trace/ring_buffer_benchmark.ko.xz
/lib/modules/4.18.0-80.el8.x86_64/kernel/lib
/lib/modules/4.18.0-80.el8.x86_64/kernel/lib/cordic.ko.xz
…
```

- List modules for **kernel-modules**:

```
$ rpm -qlp <kernel-modules_rpm>
…
/lib/modules/4.18.0-80.el8.x86_64/kernel/drivers/infiniband/hw/mlx4/mlx4_ib.ko.xz
/lib/modules/4.18.0-80.el8.x86_64/kernel/drivers/infiniband/hw/mlx5/mlx5_ib.ko.xz
/lib/modules/4.18.0-80.el8.x86_64/kernel/drivers/infiniband/hw/qedr/qedr.ko.xz
/lib/modules/4.18.0-80.el8.x86_64/kernel/drivers/infiniband/hw/usnic/usnic_verbs.ko.xz
/lib/modules/4.18.0-
80.el8.x86_64/kernel/drivers/infiniband/hw/vmw_pvrdma/vmw_pvrdma.ko.xz
…
```

- List modules for **kernel-modules-extra**:

```
$ rpm -qlp <kernel-modules-extra_rpm>
…
/lib/modules/4.18.0-80.el8.x86_64/extra/net/sched/sch_cbq.ko.xz
/lib/modules/4.18.0-80.el8.x86_64/extra/net/sched/sch_choke.ko.xz
/lib/modules/4.18.0-80.el8.x86_64/extra/net/sched/sch_drr.ko.xz
/lib/modules/4.18.0-80.el8.x86_64/extra/net/sched/sch_dsmark.ko.xz
/lib/modules/4.18.0-80.el8.x86_64/extra/net/sched/sch_gred.ko.xz
…
```

Additional resources

- For information on how to use the **rpm** command on already installed **kernel** RPM, including its sub-packages, see the **rpm(8)** manual page.
- Introduction to *RPM packages*

CHAPTER 2. UPDATING KERNEL WITH YUM

The following sections bring information about the Linux kernel provided and maintained by Red Hat (Red Hat kernel), and how to keep the Red Hat kernel updated. As a consequence, the operating system will have all the latest bug fixes, performance enhancements, and patches ensuring compatibility with new hardware.

2.1. WHAT IS THE KERNEL

The kernel is a core part of a Linux operating system, which manages the system resources, and provides interface between hardware and software applications. The Red Hat kernel is a custom-built kernel based on the upstream Linux mainline kernel that Red Hat engineers further develop and harden with a focus on stability and compatibility with the latest technologies and hardware.

Before Red Hat releases a new kernel version, the kernel needs to pass a set of rigorous quality assurance tests.

The Red Hat kernels are packaged in the RPM format so that they are easy to upgrade and verify by the **yum** package manager.

> **WARNING**
>
> Kernels that have not been compiled by Red Hat are **not** supported by Red Hat.

2.2. WHAT IS YUM

This section refers to description of the **yum** *package manager*.

Additional resources

- For more information on **yum** see the relevant sections of *Configuring basic system settings*.

2.3. UPDATING THE KERNEL

The following procedure describes how to update the kernel using the **yum** package manager.

Procedure

1. To update the kernel, use the following:

   ```
   # yum update kernel
   ```

   This command updates the kernel along with all dependencies to the latest available version.

2. Reboot your system for the changes to take effect.

NOTE

When upgrading from Red Hat Enterprise Linux 7 to Red Hat Enterprise Linux 8, follow relevant sections of the *Upgrading from RHEL 7 to RHEL 8* document.

2.4. INSTALLING THE KERNEL

The following procedure describes how to install new kernels using the **yum** package manager.

Procedure

- To install a specific kernel version, use the following:

```
# yum install kernel-{version}
```

Additional resources

- For a list of available kernels, refer to *Red Hat Code Browser*.
- For a list of release dates of specific kernel versions, see *this article*.

CHAPTER 3. MANAGING KERNEL MODULES

The following sections explain what kernel modules are, how to display their information, and how to perform basic administrative tasks with kernel modules.

3.1. INTRODUCTION TO KERNEL MODULES

The Red Hat Enterprise Linux kernel can be extended with optional, additional pieces of functionality, called kernel modules, without having to reboot the system. On Red Hat Enterprise Linux 8, kernel modules are extra kernel code which is built into compressed **<KERNEL_MODULE_NAME>.ko.xz** object files.

The most common functionality enabled by kernel modules are:

- Device driver which adds support for new hardware
- Support for a file system such as **GFS2** or **NFS**
- System calls

On modern systems, kernel modules are automatically loaded when needed. However, in some cases it is necessary to load or unload modules manually.

Like the kernel itself, the modules can take parameters that customize their behavior if needed.

Tooling is provided to inspect which modules are currently running, which modules are available to load into the kernel and which parameters a module accepts. The tooling also provides a mechanism to load and unload kernel modules into the running kernel.

3.2. INTRODUCTION TO BOOTLOADER SPECIFICATION

The BootLoader Specification (BLS) defines a scheme and the file format to manage the bootloader configuration for each boot option in the drop-in directory without the need to manipulate the bootloader configuration files. Unlike earlier approaches, each boot entry is now represented by a separate configuration file in the drop-in directory. The drop-in directory extends its configuration without having the need to edit or regenerate the configuration files. The BLS extends this concept for the boot menu entries.

Using BLS, you can manage the bootloader menu options by adding, removing, or editing individual boot entry files in a directory. This makes the kernel installation process significantly simpler and consistent across the different architectures.

The **grubby** tool is a thin wrapper script around the BLS and it supports the same **grubby** arguments and options. It runs the **dracut** to create an initial ramdisk image. With this setup, the core bootloader configuration files are static and are not modified after kernel installation.

This premise is particularly relevant in Red Hat Enterprise Linux 8 because the same bootloader is not used in all architectures. **GRUB2** is used in most of them such as the 64-bit ARM, but little-endian variants of IBM Power Systems with Open Power Abstraction Layer (OPAL) uses **Petitboot** and the IBM Z architecture uses **zipl**.

Additional Resources

- For more information about **grubby** utility, see What is grubby.

- For more details for boot entries, see What are Boot Entries
- For more details, see the **grubby(8)** manual page.

3.3. KERNEL MODULE DEPENDENCIES

Certain kernel modules sometimes depend on one or more other kernel modules. The **/lib/modules/<KERNEL_VERSION>/modules.dep** file contains a complete list of kernel module dependencies for the respective kernel version.

The dependency file is generated by the **depmod** program, which is a part of the **kmod** package. Many of the utilities provided by **kmod** take module dependencies into account when performing operations so that **manual** dependency-tracking is rarely necessary.

WARNING

The code of kernel modules is executed in kernel-space in the unrestricted mode. Because of this, you should be mindful of what modules you are loading.

Additional resources

- For more information about **/lib/modules/<KERNEL_VERSION>/modules.dep**, refer to the **modules.dep(5)** manual page.
- For further details including the synopsis and options of **depmod**, see the **depmod(8)** manual page.

3.4. LISTING CURRENTLY LOADED KERNEL MODULES

The following procedure describes how to view the currently loaded kernel modules.

Prerequisites

- The **kmod** package is installed.

Procedure

- To list all currently loaded kernel modules, execute:

```
$ lsmod

Module                  Size  Used by
fuse                  126976  3
uinput                 20480  1
xt_CHECKSUM            16384  1
ipt_MASQUERADE         16384  1
xt_conntrack           16384  1
ipt_REJECT             16384  1
nft_counter            16384  16
nf_nat_tftp            16384  0
```

```
nf_conntrack_tftp      16384  1 nf_nat_tftp
tun                    49152  1
bridge                192512  0
stp                    16384  1 bridge
llc                    16384  2 bridge,stp
nf_tables_set          32768  5
nft_fib_inet           16384  1
…
```

In the example above:

- The first column provides the **names** of currently loaded modules.
- The second column displays the amount of **memory** per module in kilobytes.
- The last column shows the number, and optionally the names of modules that are **dependent** on a particular module.

Additional resources

- For more information about **kmod**, refer to the **/usr/share/doc/kmod/README** file or the **lsmod(8)** manual page.

3.5. LISTING ALL INSTALLED KERNELS

The following procedure describes how to use the command line tool **grubby** to list the **GRUB2** boot entries.

Procedure

- To list the boot entries of the kernel, execute:

  ```
  # grubby --info=ALL | grep title
  ```

  The following output displays the boot entries of the kernel. The **kernel** field shows the kernel path.

  ```
  title=Red Hat Enterprise Linux (4.18.0-20.el8.x86_64) 8.0 (Ootpa)
  title=Red Hat Enterprise Linux (4.18.0-19.el8.x86_64) 8.0 (Ootpa)
  title=Red Hat Enterprise Linux (4.18.0-12.el8.x86_64) 8.0 (Ootpa)
  title=Red Hat Enterprise Linux (4.18.0) 8.0 (Ootpa)
  title=Red Hat Enterprise Linux (0-rescue-2fb13ddde2e24fde9e6a246a942caed1) 8.0 (Ootpa)
  ```

3.6. SETTING A KERNEL AS DEFAULT

The following procedure describes how to set a specific kernel as default using the **grubby** command-line tool and **GRUB2**.

Procedure

Setting the kernel as default, using thegrubby tool

- Execute the following command to set the kernel as default using the **grubby** tool:

```
# grubby --set-default $kernel_path
```

The command uses a machine ID without the **.conf** suffix as an argument.

NOTE

The machine ID is located in the **/boot/loader/entries/** directory.

Setting the kernel as default, using theid argument

- List the boot entries using the **id** argument and then set an intended kernel as default:

```
# grubby --info ALL | grep id
# grubby --set-default /boot/vmlinuz-<version>.<architecture>
```

NOTE

To list the boot entries using the **title** argument, execute the **# grubby --info=ALL | grep title** command.

Setting the default kernel for only the next boot

- Execute the following command to set the default kernel for only the next reboot using the **grub2-reboot** command:

```
# grub2-reboot <index|title|id>
```

WARNING

Set the default kernel for only the next boot with care. Installing new kernel RPM’s, self-built kernels, and manually adding the entries to the **/boot/loader/entries/** directory may change the index values.

3.7. DISPLAYING INFORMATION ABOUT KERNEL MODULES

When working with a kernel module, you may want to see further information about that module. This procedure describes how to display extra information about kernel modules.

Prerequisites

- The **kmod** package is installed.

Procedure

- To display information about any kernel module, execute:

```
$ modinfo <KERNEL_MODULE_NAME>

For example:
$ modinfo virtio_net

filename:       /lib/modules/4.18.0-94.el8.x86_64/kernel/drivers/net/virtio_net.ko.xz
license:        GPL
description:    Virtio network driver
rhelversion:    8.1
srcversion:     2E9345B281A898A91319773
alias:          virtio:d00000001v*
depends:        net_failover
intree:         Y
name:           virtio_net
vermagic:       4.18.0-94.el8.x86_64 SMP mod_unload modversions
…
parm:           napi_weight:int
parm:           csum:bool
parm:           gso:bool
parm:           napi_tx:bool
```

The **modinfo** command displays some detailed information about the specified kernel module. You can query information about all available modules, regardless of whether they are loaded or not. The **parm** entries show parameters the user is able to set for the module, and what type of value they expect.

> **NOTE**
>
> When entering the name of a kernel module, do not append the **.ko.xz** extension to the end of the name. Kernel module names do not have extensions; their corresponding files do.

Additional resources

- For more information about the **modinfo**, refer to the **modinfo(8)** manual page.

3.8. LOADING KERNEL MODULES AT SYSTEM RUNTIME

The optimal way to expand the functionality of the Linux kernel is by loading kernel modules. The following procedure describes how to use the **modprobe** command to find and load a kernel module into the currently running kernel.

Prerequisites

- Root permissions
- The **kmod** package is installed.
- The respective kernel module is not loaded. To ensure this is the case, list the loaded kernel modules.

Procedure

1. Select a kernel module you want to load.
   The modules are located in the **/lib/modules/$(uname -r)/kernel/<SUBSYSTEM>/** directory.

2. Load the relevant kernel module:

```
# modprobe <MODULE_NAME>
```

> **NOTE**
>
> When entering the name of a kernel module, do not append the **.ko.xz** extension to the end of the name. Kernel module names do not have extensions; their corresponding files do.

3. Optionally, verify the relevant module was loaded:

```
$ lsmod | grep <MODULE_NAME>
```

If the module was loaded correctly, this command displays the relevant kernel module. For example:

```
$ lsmod | grep serio_raw
serio_raw              16384  0
```

> **IMPORTANT**
>
> The changes described in this procedure **will not persist** after rebooting the system.

Additional resources

- For further details about **modprobe**, see the **modprobe(8)** manual page.

3.9. UNLOADING KERNEL MODULES AT SYSTEM RUNTIME

At times, you find that you need to unload certain kernel modules from the running kernel. The following procedure describes how to use the **modprobe** command to find and unload a kernel module at system runtime from the currently loaded kernel.

Prerequisites

- Root permissions
- The **kmod** package is installed.

Procedure

1. Execute the **lsmod** command and select a kernel module you want to unload.
   If a kernel module has dependencies, unload those prior to unloading the kernel module. For details on identifying modules with dependencies, see Section 3.4, “Listing currently loaded kernel modules”.

2. Unload the relevant kernel module:

```
# modprobe -r <MODULE_NAME>
```

When entering the name of a kernel module, do not append the **.ko.xz** extension to the end of the name. Kernel module names do not have extensions; their corresponding files do.

> **WARNING**
>
> Do not unload kernel modules when they are used by the running system. Doing so can lead to an unstable or non-operational system.

3. Optionally, verify the relevant module was unloaded:

```
$ lsmod | grep <MODULE_NAME>
```

If the module was unloaded successfully, this command does not display any output.

> **IMPORTANT**
>
> After finishing this procedure, the kernel modules that are defined to be automatically loaded on boot, **will not stay unloaded** after rebooting the system. For information on how to counter this outcome, see Preventing kernel modules from being automatically loaded at system boot time.

Additional resources

- For further details about **modprobe**, see the **modprobe(8)** manual page.

3.10. LOADING KERNEL MODULES AUTOMATICALLY AT SYSTEM BOOT TIME

The following procedure describes how to configure a kernel module so that it is loaded automatically during the boot process.

Prerequisites

- Root permissions
- The **kmod** package is installed.

Procedure

1. Select a kernel module you want to load during the boot process.
   The modules are located in the **/lib/modules/$(uname -r)/kernel/<SUBSYSTEM>/** directory.

2. Create a configuration file for the module:

```
# echo <MODULE_NAME> > /etc/modules-load.d/<MODULE_NAME>.conf
```

NOTE

When entering the name of a kernel module, do not append the **.ko.xz** extension to the end of the name. Kernel module names do not have extensions; their corresponding files do.

3. Optionally, after reboot, verify the relevant module was loaded:

   ```
   $ lsmod | grep <MODULE_NAME>
   ```

   The example command above should succeed and display the relevant kernel module.

IMPORTANT

The changes described in this procedure **will persist** after rebooting the system.

Additional resources

- For further details about loading kernel modules during the boot process, see the **modules-load.d(5)** manual page.

3.11. PREVENTING KERNEL MODULES FROM BEING AUTOMATICALLY LOADED AT SYSTEM BOOT TIME

The following procedure describes how to add a kernel module to a blacklist so that it will not be automatically loaded during the boot process.

Prerequisites

- Root permissions
- The **kmod** package is installed.
- Ensure that a blacklisted kernel module is not vital for your current system configuration.

Procedure

1. Select a kernel module that you want to blacklist:

   ```
   $ lsmod

   Module                  Size  Used by
   fuse                  126976  3
   xt_CHECKSUM            16384  1
   ipt_MASQUERADE         16384  1
   uinput                 20480  1
   xt_conntrack           16384  1
   …
   ```

   The **lsmod** command displays a list of modules loaded to the currently running kernel.

   - Alternatively, identify an unloaded kernel module you want to prevent from potentially loading.

All kernel modules are located in the **/lib/modules/<KERNEL_VERSION>/kernel/<SUBSYSTEM>/** directory.

2. Create a blacklist configuration file:

```
# vim /etc/modprobe.d/blacklist.conf

 # Blacklists <KERNEL_MODULE_1>
 blacklist <MODULE_NAME_1>
 install <MODULE_NAME_1> /bin/false

 # Blacklists <KERNEL_MODULE_2>
 blacklist <MODULE_NAME_2>
 install <MODULE_NAME_2> /bin/false

 # Blacklists <KERNEL_MODULE_n>
 blacklist <MODULE_NAME_n>
 install <MODULE_NAME_n> /bin/false
 …
```

The example shows the contents of the **blacklist.conf** file, edited by the **vim** editor. The **blacklist** line ensures that the relevant kernel module will not be automatically loaded during the boot process. The **blacklist** command, however, does not prevent the module from being loaded as a dependency for another kernel module that is not blacklisted. Therefore the **install** line causes the **/bin/false** to run instead of installing a module.

The lines starting with a hash sign are comments to make the file more readable.

NOTE

When entering the name of a kernel module, do not append the **.ko.xz** extension to the end of the name. Kernel module names do not have extensions; their corresponding files do.

3. Create a backup copy of the current initial ramdisk image before rebuilding:

```
# cp /boot/initramfs-$(uname -r).img /boot/initramfs-$(uname -r).bak.$(date +%m-%d-
%H%M%S).img
```

The command above creates a backup **initramfs** image in case the new version has an unexpected problem.

- Alternatively, create a backup copy of other initial ramdisk image which corresponds to the kernel version for which you want to blacklist kernel modules:

```
# cp /boot/initramfs-<SOME_VERSION>.img /boot/initramfs-
<SOME_VERSION>.img.bak.$(date +%m-%d-%H%M%S)
```

4. Generate a new initial ramdisk image to reflect the changes:

```
# dracut -f -v
```

- If you are building an initial ramdisk image for a different kernel version than you are currently booted into, specify both target **initramfs** and kernel version:

```
# dracut -f -v /boot/initramfs-<TARGET_VERSION>.img
<CORRESPONDING_TARGET_KERNEL_VERSION>
```

5. Reboot the system:

```
$ reboot
```

> **IMPORTANT**
>
> The changes described in this procedure **will take effect and persist**after rebooting the system. Improper blacklisting of a key kernel module can result in an unstable or non-operational system.

Additional resources

- For further details concerning the **dracut** utility, refer to the **dracut(8)** manual page.

3.12. SIGNING KERNEL MODULES FOR SECURE BOOT

You can enhance the security of your system by using signed kernel modules. The following sections describe how to self-sign privately built kernel modules for use with RHEL 8 on UEFI-based build systems where Secure Boot is enabled. These sections also provide an overview of available options for importing your public key into a target system where you want to deploy your kernel modules.

To sign and load kernel modules, you need to:

1. Authenticate a kernel module .
2. Generate a public and private key pair .
3. Import the public key on the target system .
4. Sign the kernel module with the private key .
5. Load the signed kernel module .

If Secure Boot is enabled, the UEFI operating system boot loaders, the Red Hat Enterprise Linux kernel, and all kernel modules have to be signed with a private key and authenticated with the corresponding public key. If they are not signed and authenticated, the system will not be allowed to finish the booting process.

The RHEL 8 distribution includes:

- Signed boot loaders
- Signed kernels
- Signed kernel modules

In addition, the signed first-stage boot loader and the signed kernel include embedded Red Hat public keys. These signed executable binaries and embedded keys enable RHEL 8 to install, boot, and run with the Microsoft UEFI Secure Boot Certification Authority keys that are provided by the UEFI firmware on systems that support UEFI Secure Boot. Note that not all UEFI-based systems include support for Secure Boot.

Prerequisites

To be able to sign externally built kernel modules, install the utilities listed in the following table on the build system.

Table 3.1. Required utilities

Utility	Provided by package	Used on	Purpose
openssl	**openssl**	Build system	Generates public and private X.509 key pair
sign-file	**kernel-devel**	Build system	Executable file used to sign a kernel module with the private key
mokutil	**mokutil**	Target system	Optional utility used to manually enroll the public key
keyctl	**keyutils**	Target system	Optional utility used to display public keys in the system keyring

NOTE

The build system, where you build and sign your kernel module, does not need to have UEFI Secure Boot enabled and does not even need to be a UEFI-based system.

3.12.1. Authenticating kernel modules with X.509 keys

In RHEL 8, when a kernel module is loaded, the module's signature is checked using the public X.509 keys on the kernel's system keyring, excluding keys on the kernel's system black list keyring. The following sections provide an overview of sources of keys/keyrings, examples of loaded keys from different sources in the system. Also, the user can see what it takes to authenticate a kernel module.

3.12.1.1. Authentication requirements

This section explains what conditions have to be met for loading kernel modules on systems with enabled UEFI Secure Boot functionality.

If UEFI Secure Boot is enabled or if the **module.sig_enforce** kernel parameter has been specified, you can only load signed kernel modules that are authenticated using a key on the system keyring. In addition, the public key must not be on the system black list keyring.

If UEFI Secure Boot is disabled and if the **module.sig_enforce** kernel parameter has not been specified, you can load unsigned kernel modules and signed kernel modules without a public key. This is summarized in the table below.

Table 3.2. Kernel module authentication requirements for loading

Module signed	Public key found and signature valid	UEFI Secure Boot state	sig_enforce	Module load	Kernel tainted
Unsigned	–	Not enabled	Not enabled	Succeeds	Yes
		Not enabled	Enabled	Fails	–
		Enabled	–	Fails	–
Signed	No	Not enabled	Not enabled	Succeeds	Yes
		Not enabled	Enabled	Fails	–
		Enabled	–	Fails	–
Signed	Yes	Not enabled	Not enabled	Succeeds	No
		Not enabled	Enabled	Succeeds	No
		Enabled	–	Succeeds	No

3.12.1.2. Sources for public keys

During boot, the kernel loads X.509 keys into the system keyring or the system black list keyring from a set of persistent key stores as shown in the table below.

Table 3.3. Sources for system keyrings

Source of X.509 keys	User ability to add keys	UEFI Secure Boot state	Keys loaded during boot
Embedded in kernel	No	–	**.system_keyring**
UEFI Secure Boot "db"	Limited	Not enabled	No
		Enabled	**.system_keyring**
UEFI Secure Boot "dbx"	Limited	Not enabled	No
		Enabled	**.system_keyring**
Embedded in **shim.efi** boot loader	No	Not enabled	No
		Enabled	**.system_keyring**
Machine Owner Key (MOK) list	Yes	Not enabled	No

Source of X.509 keys	User ability to add keys	UEFI Secure Boot state	Keys loaded during boot
		Enabled	**.system_keyring**

If the system is not UEFI-based or if UEFI Secure Boot is not enabled, then only the keys that are embedded in the kernel are loaded onto the system keyring. In that case you have no ability to augment that set of keys without rebuilding the kernel.

The system black list keyring is a list of X.509 keys which have been revoked. If your module is signed by a key on the black list then it will fail authentication even if your public key is in the system keyring.

You can display information about the keys on the system keyrings using the **keyctl** utility. The following is a shortened example output from a RHEL 8 system where UEFI Secure Boot is not enabled.

```
# keyctl list %:.system_keyring
3 keys in keyring:
...asymmetric: Red Hat Enterprise Linux Driver Update Program (key 3): bf57f3e87...
...asymmetric: Red Hat Enterprise Linux kernel signing key: 4249689eefc77e95880b...
...asymmetric: Red Hat Enterprise Linux kpatch signing key: 4d38fd864ebe18c5f0b7...
```

The following is a shortened example output from a RHEL 8 system where UEFI Secure Boot is enabled.

```
# keyctl list %:.system_keyring
6 keys in keyring:
...asymmetric: Red Hat Enterprise Linux Driver Update Program (key 3): bf57f3e87...
...asymmetric: Red Hat Secure Boot (CA key 1): 4016841644ce3a810408050766e8f8a29...
...asymmetric: Microsoft Corporation UEFI CA 2011: 13adbf4309bd82709c8cd54f316ed...
...asymmetric: Microsoft Windows Production PCA 2011: a92902398e16c49778cd90f99e...
...asymmetric: Red Hat Enterprise Linux kernel signing key: 4249689eefc77e95880b...
...asymmetric: Red Hat Enterprise Linux kpatch signing key: 4d38fd864ebe18c5f0b7...
```

The above output shows the addition of two keys from the UEFI Secure Boot "db" keys as well as the **Red Hat Secure Boot (CA key 1)**, which is embedded in the **shim.efi** boot loader. You can also look for the kernel console messages that identify the keys with an UEFI Secure Boot related source. These include UEFI Secure Boot db, embedded shim, and MOK list.

```
# dmesg | grep 'EFI: Loaded cert'
[5.160660] EFI: Loaded cert 'Microsoft Windows Production PCA 2011: a9290239...
[5.160674] EFI: Loaded cert 'Microsoft Corporation UEFI CA 2011: 13adbf4309b...
[5.165794] EFI: Loaded cert 'Red Hat Secure Boot (CA key 1): 4016841644ce3a8...
```

3.12.1.3. Generating a public and private key pair

You need to generate a public and private X.509 key pair to succeed in your efforts of using kernel modules on a Secure Boot-enabled system. You will later use the private key to sign the kernel module. You will also have to add the corresponding public key to the Machine Owner Key (MOK) for Secure Boot to validate the signed module.

Some of the parameters for this key pair generation are best specified with a configuration file.

Procedure

1. Create a configuration file with parameters for the key pair generation:

   ```
   # cat << EOF > configuration_file.config
   [ req ]
   default_bits = 4096
   distinguished_name = req_distinguished_name
   prompt = no
   string_mask = utf8only
   x509_extensions = myexts

   [ req_distinguished_name ]
   O = Organization
   CN = Organization signing key
   emailAddress = E-mail address

   [ myexts ]
   basicConstraints=critical,CA:FALSE
   keyUsage=digitalSignature
   subjectKeyIdentifier=hash
   authorityKeyIdentifier=keyid
   EOF
   ```

2. Create an X.509 public and private key pair as shown in the following example:

   ```
   # openssl req -x509 -new -nodes -utf8 -sha256 -days 36500 \
   -batch -config configuration_file.config -outform DER \
   -out my_signing_key_pub.der \
   -keyout my_signing_key.priv
   ```

   The public key will be written to the ***my_signing_key_pub*.der** file and the private key will be written to the ***my_signing_key*.priv** file.

   In RHEL 8, the validity dates of the key pair matter. The key does not expire, but the kernel module must be signed within the validity period of its signing key. For example, a key that is only valid in 2019 can be used to authenticate a kernel module signed in 2019 with that key. However, users cannot use that key to sign a kernel module in 2020.

3. Optionally, you can review the validity dates of your public keys like in the example below:

   ```
   # openssl x509 -inform der -text -noout -in <my_signing_key_pub.der>

   Validity
               Not Before: Feb 14 16:34:37 2019 GMT
               Not After : Feb 11 16:34:37 2029 GMT
   ```

4. Enroll your public key on all systems where you want to authenticate and load your kernel module.

WARNING

Apply strong security measures and access policies to guard the contents of your private key. In the wrong hands, the key could be used to compromise any system which is authenticated by the corresponding public key.

Additional resources

- For details about enrolling public keys into target systems, see Section 3.12.2.2, “Manually adding public key to the MOK list”.

3.12.2. Enrolling public key on target system

When RHEL 8 boots on a UEFI-based system with Secure Boot enabled, the kernel loads onto the system keyring all public keys that are in the Secure Boot db key database, but not in the dbx database of revoked keys. The sections below describe different ways of importing a public key on a target system so that the system keyring is able to use the public key to authenticate a kernel module.

3.12.2.1. Factory firmware image including public key

To facilitate authentication of your kernel module on your systems, consider requesting your system vendor to incorporate your public key into the UEFI Secure Boot key database in their factory firmware image.

3.12.2.2. Manually adding public key to the MOK list

The Machine Owner Key (MOK) facility feature can be used to expand the UEFI Secure Boot key database. When RHEL 8 boots on a UEFI-enabled system with Secure Boot enabled, the keys on the MOK list are also added to the system keyring in addition to the keys from the key database. The MOK list keys are also stored persistently and securely in the same fashion as the Secure Boot database keys, but these are two separate facilities. The MOK facility is supported by **shim.efi**, **MokManager.efi**, **grubx64.efi**, and the **mokutil** utility.

Enrolling a MOK key requires manual interaction by a user at the UEFI system console on each target system. Nevertheless, the MOK facility provides a convenient method for testing newly generated key pairs and testing kernel modules signed with them.

Procedure

1. Request the addition of your public key to the MOK list:

   ```
   # mokutil --import my_signing_key_pub.der
   ```

   You will be asked to enter and confirm a password for this MOK enrollment request.

2. Reboot the machine.
   The pending MOK key enrollment request will be noticed by **shim.efi** and it will launch **MokManager.efi** to allow you to complete the enrollment from the UEFI console.

3. Enter the password you previously associated with this request and confirm the enrollment.

Your public key is added to the MOK list, which is persistent.

Once a key is on the MOK list, it will be automatically propagated to the system keyring on this and subsequent boots when UEFI Secure Boot is enabled.

3.12.3. Signing kernel modules with the private key

Users are able to obtain enhanced security benefits on their systems by loading signed kernel modules if the UEFI Secure Boot mechanism is enabled. The following sections describe how to sign kernel modules with the private key.

Prerequisites

- You generated a public and private key pair and know the validity dates of your public keys. For details, see Section 3.12.1.3, “Generating a public and private key pair” .
- You enrolled your public key on the target system. For details, see Section 3.12.2, “Enrolling public key on target system”.
- You have a kernel module available for signing.

Procedure

- Execute the **sign-file** utility with parameters as shown in the example below:

```
# /usr/src/kernels/$(uname -r)/scripts/sign-file \
sha256 \
my_signing_key.priv \
my_signing_key_pub.der \
my_module.ko
```

Your kernel module is in ELF image format and **sign-file** computes and appends the signature directly to the ELF image in your kernel module file. The **modinfo** utility can be used to display information about the kernel module’s signature, if it is present.

The appended signature is not contained in an ELF image section and is not a formal part of the ELF image. Therefore, utilities such as **readelf** will not be able to display the signature on your kernel module.

Your kernel module is now ready for loading. Note that your signed kernel module is also loadable on systems where UEFI Secure Boot is disabled or on a non-UEFI system. That means you do not need to provide both a signed and unsigned version of your kernel module.

IMPORTANT

In RHEL 8, the validity dates of the key pair matter. The key does not expire, but the kernel module must be signed within the validity period of its signing key. The **sign-file** utility will not warn you of this. For example, a key that is only valid in 2019 can be used to authenticate a kernel module signed in 2019 with that key. However, users cannot use that key to sign a kernel module in 2020.

Additional resources

- For details on using **modinfo** to obtain information about kernel modules, see Section 3.7, “Displaying information about kernel modules”.

3.12.4. Loading signed kernel modules

Once your public key is enrolled and is in the system keyring, use the **mokutil** command to add your public key to the MOK list. Then sign the respective kernel module with your private key. In the end, manually load your signed kernel module with the **modprobe** command as described in the following section.

Prerequisites

- You generated the public and private keypair. For details, see Section 3.12.1.3, "Generating a public and private key pair".
- You enrolled the public key into the system keyring. For details, see Section 3.12.2.2, "Manually adding public key to the MOK list".

Procedure

1. Optionally, verify that your kernel module will not load before you have enrolled your public key. For details on how to list currently loaded kernel modules, see Section 3.4, "Listing currently loaded kernel modules".
2. Verify what keys have been added to the system keyring on the current boot:

   ```
   # keyctl list %:.system_keyring
   ```

   Since your public key has not been enrolled yet, it should not be displayed in the output of the command.
3. Request enrollment of your public key:

   ```
   # mokutil --import my_signing_key_pub.der
   ```
4. Reboot, and complete the enrollment at the UEFI console:

   ```
   # reboot
   ```
5. Verify the keys on the system keyring again:

   ```
   # keyctl list %:.system_keyring
   ```
6. Copy the kernel module into the **/extra/** directory of the kernel you want:

   ```
   # cp my_module.ko /lib/modules/$(uname -r)/extra/
   ```
7. Update the modular dependency list:

   ```
   # depmod -a
   ```
8. Load the kernel module and verify that it was successfully loaded:

   ```
   # modprobe -v my_module
   # lsmod | grep my_module
   ```

a. Optionally, to load the module on boot, add it to the **/etc/modules-loaded.d/my_module.conf** file:

```
# echo "my_module" > /etc/modules-load.d/my_module.conf
```

Additional resources

- For further information about loading kernel modules, see the relevant sections of Chapter 3, *Managing kernel modules*.

CHAPTER 4. CONFIGURING KERNEL COMMAND-LINE PARAMETERS

Kernel command-line parameters are a way to change the behavior of certain aspects of the Red Hat Enterprise Linux kernel at boot time. As a system administrator, you have full control over what options get set at boot. Certain kernel behaviors are only able to be set at boot time, so understanding how to make this changes is a key administration skill.

IMPORTANT

Opting to change the behavior of the system by modifying kernel command-line parameters may have negative effects on your system. You should therefore test changes prior to deploying them in production. For further guidance, contact Red Hat Support.

4.1. UNDERSTANDING KERNEL COMMAND-LINE PARAMETERS

Kernel command-line parameters are used for boot time configuration of:

- The Red Hat Enterprise Linux kernel
- The initial RAM disk
- The user space features

Kernel boot time parameters are often used to overwrite default values and for setting specific hardware settings.

By default, the kernel command-line parameters for systems using the GRUB2 bootloader are defined in the **kernelopts** variable of the **/boot/grub2/grubenv** file for all kernel boot entries.

NOTE

For IBM Z, the kernel command-line parameters are stored in the boot entry configuration file because the zipl bootloader does not support environment variables. Thus, the **kernelopts** environment variable cannot be used.

Additional resources

- For more information about what kernel command-line parameters you can modify, see **kernel-command-line(7)**, **bootparam(7)** and **dracut.cmdline(7)** manual pages.
- For more information about the **kernelopts** variable, see the knowledge base article, How to install and boot custom kernels in Red Hat Enterprise Linux 8.

4.2. WHAT GRUBBY IS

grubby is a utility for manipulating bootloader-specific configuration files.

You can use **grubby** also for changing the default boot entry, and for adding/removing arguments from a GRUB2 menu entry.

For more details see the **grubby(8)** manual page.

4.3. WHAT BOOT ENTRIES ARE

A boot entry is a collection of options which are stored in a configuration file and tied to a particular kernel version. In practice, you have at least as many boot entries as your system has installed kernels. The boot entry configuration file is located in the **/boot/loader/entries/** directory and can look like this:

```
6f9cc9cb7d7845d49698c9537337cedc-4.18.0-5.el8.x86_64.conf
```

The file name above consists of a machine ID stored in the **/etc/machine-id** file, and a kernel version.

The boot entry configuration file contains information about the kernel version, the initial ramdisk image, and the **kernelopts** environment variable, which contains the kernel command-line parameters. The contents of a boot entry config can be seen below:

```
title Red Hat Enterprise Linux (4.18.0-74.el8.x86_64) 8.0 (Ootpa)
version 4.18.0-74.el8.x86_64
linux /vmlinuz-4.18.0-74.el8.x86_64
initrd /initramfs-4.18.0-74.el8.x86_64.img $tuned_initrd
options $kernelopts $tuned_params
id rhel-20190227183418-4.18.0-74.el8.x86_64
grub_users $grub_users
grub_arg --unrestricted
grub_class kernel
```

The **kernelopts** environment variable is defined in the **/boot/grub2/grubenv** file.

Additional resources

For more information about **kernelopts** variable, see knowledge base article How to install and boot custom kernels in Red Hat Enterprise Linux 8.

4.4. SETTING KERNEL COMMAND-LINE PARAMETERS

To adjust the behavior of your system from the early stages of the booting process, you need to set certain kernel command-line parameters.

This section explains how to change kernel command-line parameters on various CPU architectures.

4.4.1. Changing kernel command-line parameters for all boot entries

This procedure describes how to change kernel command-line parameters for all boot entries on your system.

Prerequisites

- Verify that the **grubby** and **zipl** utilities are installed on your system.

Procedure

- To add a parameter:

  ```
  # grubby --update-kernel=ALL --args="<NEW_PARAMETER>"
  ```

For systems that use the GRUB2 bootloader, the command updates the **/boot/grub2/grubenv** file by adding a new kernel parameter to the **kernelopts** variable in that file.

On IBM Z that use the zIPL bootloader, the command adds a new kernel parameter to each **/boot/loader/entries/<*ENTRY*>.conf** file.

  - On IBM Z, execute the **zipl** command with no options to update the boot menu.

- To remove a parameter:

```
# grubby --update-kernel=ALL --remove-args="<PARAMETER_TO_REMOVE>"
```

  - On IBM Z, execute the **zipl** command with no options to update the boot menu.

Additional resources

- For more information about kernel command-line parameters, see Section 4.1, “Understanding kernel command-line parameters”.
- For information on the **grubby** utility, see the **grubby(8)** manual page.
- For further examples on how to use **grubby**, see the *grubby tool*.
- For information about the **zipl** utility, see the **zipl(8)** manual page.

4.4.2. Changing kernel command-line parameters for a single boot entry

This procedure describes how to change kernel command-line parameters for a single boot entry on your system.

Prerequisites

- Verify that the **grubby** and **zipl** utilities are installed on your system.

Procedure

- To add a parameter:

```
# grubby --update-kernel=/boot/vmlinuz-$(uname -r) --args="<NEW_PARAMETER>"
```

  - On IBM Z, execute the **zipl** command with no options to update the boot menu.

- To remove a parameter use the following:

```
# grubby --update-kernel=/boot/vmlinuz-$(uname -r) --remove-args="
<PARAMETER_TO_REMOVE>"
```

  - On IBM Z, execute the **zipl** command with no options to update the boot menu.

> **NOTE**
>
> On systems that use the **grub.cfg** file, there is, by default, the **options** parameter for each kernel boot entry, which is set to the **kernelopts** variable. This variable is defined in the **/boot/grub2/grubenv** configuration file.

IMPORTANT

On GRUB2 systems:

- If the kernel command-line parameters are modified for all boot entries, the **grubby** utility updates the **kernelopts** variable in the **/boot/grub2/grubenv** file.
- If kernel command-line parameters are modified for a single boot entry, the **kernelopts** variable is expanded, the kernel parameters are modified, and the resulting value is stored in the respective boot entry's **/boot/loader/entries/<RELEVANT_KERNEL_BOOT_ENTRY.conf>** file.

On zIPL systems:

- **grubby** modifies and stores the kernel command-line parameters of an individual kernel boot entry in the **/boot/loader/entries/<*ENTRY*>.conf** file.

Additional resources

- For more information about kernel command-line parameters, see Section 4.1, "Understanding kernel command-line parameters".
- For information on the **grubby** utility, see the **grubby(8)** manual page.
- For further examples on how to use **grubby**, see the *grubby tool*.
- For information about the **zipl** utility, see the **zipl(8)** manual page.

CHAPTER 5. CONFIGURING KERNEL PARAMETERS AT RUNTIME

As a system administrator, you can modify many facets of the Red Hat Enterprise Linux kernel's behavior at runtime. This section describes how to configure kernel parameters at runtime by using the **sysctl** command and by modifying the configuration files in the **/etc/sysctl.d/** and **/proc/sys/** directories.

5.1. WHAT ARE KERNEL PARAMETERS

Kernel parameters are tunable values which you can adjust while the system is running. There is no requirement to reboot or recompile the kernel for changes to take effect.

It is possible to address the kernel parameters through:

+ * The **sysctl** command

+ * The virtual file system mounted at the **/proc/sys/** directory

+ * The configuration files in the **/etc/sysctl.d/** directory

Tunables are divided into classes by the kernel subsystem. Red Hat Enterprise Linux has the following tunable classes:

Table 5.1. Table of sysctl classes

Tunable class	Subsystem
abi	Execution domains and personalities
crypto	Cryptographic interfaces
debug	Kernel debugging interfaces
dev	Device-specific information
fs	Global and specific file system tunables
kernel	Global kernel tunables
net	Network tunables
sunrpc	Sun Remote Procedure Call (NFS)
user	User Namespace limits
vm	Tuning and management of memory, buffers, and cache

Additional resources

- For more information about **sysctl**, see **sysctl(8)** manual pages.
- For more information about /**etc/sysctl.d**/ see, **sysctl.d(5)** manual pages.

5.2. SETTING KERNEL PARAMETERS AT RUNTIME

IMPORTANT

Configuring kernel parameters on a production system requires careful planning. Unplanned changes may render the kernel unstable, requiring a system reboot. Verify that you are using valid options before changing any kernel values.

5.2.1. Configuring kernel parameters temporarily with sysctl

The following procedure describes how to use the **sysctl** command to temporarily set kernel parameters at runtime. The command is also useful for listing and filtering tunables.

Prerequisites

- Kernel parameters introduction
- Root permissions

Procedure

1. To list all parameters and their values, use the following:

   ```
   # sysctl -a
   ```

   NOTE

   The **# sysctl -a** command displays kernel parameters, which can be adjusted at runtime and at boot time.

2. To configure a parameter temporarily, use the command as in the following example:

   ```
   # sysctl <TUNABLE_CLASS>.<PARAMETER>=<TARGET_VALUE>
   ```

   The sample command above changes the parameter value while the system is running. The changes take effect immediately, without a need for restart.

   NOTE

   The changes return back to default after your system reboots.

Additional resources

- For more information about **sysctl**, see the **sysctl(8)** manual page.
- To permanently modify kernel parameters, either use the **sysctl** command to write the values to the /**etc/sysctl.conf** file or make manual changes to the configuration files in the /**etc/sysctl.d**/ directory.

5.2.2. Configuring kernel parameters permanently with sysctl

The following procedure describes how to use the **sysctl** command to permanently set kernel parameters.

Prerequisites

- Kernel parameters introduction
- Root permissions

Procedure

1. To list all parameters, use the following:

   ```
   # sysctl -a
   ```

   The command displays all kernel parameters that can be configured at runtime.

2. To configure a parameter permanently:

   ```
   # sysctl -w <TUNABLE_CLASS>.<PARAMETER>=<TARGET_VALUE> >>
   /etc/sysctl.conf
   ```

   The sample command changes the tunable value and writes it to the **/etc/sysctl.conf** file, which overrides the default values of kernel parameters. The changes take effect immediately and persistently, without a need for restart.

> **NOTE**
>
> To permanently modify kernel parameters you can also make manual changes to the configuration files in the **/etc/sysctl.d/** directory.

Additional resources

- For more information about **sysctl**, see the **sysctl(8)** and **sysctl.conf(5)** manual pages.
- For more information about using the configuration files in the **/etc/sysctl.d/** directory to make permanent changes to kernel parameters, see Using configuration files in /etc/sysctl.d/ to adjust kernel parameters section.

5.2.3. Using configuration files in /etc/sysctl.d/ to adjust kernel parameters

The following procedure describes how to manually modify configuration files in the **/etc/sysctl.d/** directory to permanently set kernel parameters.

Prerequisites

- Kernel parameters introduction
- Root permissions

Procedure

1. Create a new configuration file in **/etc/sysctl.d/**:

   ```
   # vim /etc/sysctl.d/<some_file.conf>
   ```

2. Include kernel parameters, one per line, as follows:

   ```
   <TUNABLE_CLASS>.<PARAMETER>=<TARGET_VALUE>
   <TUNABLE_CLASS>.<PARAMETER>=<TARGET_VALUE>
   ```

3. Save the configuration file.

4. Reboot the machine for the changes to take effect.

   - Alternatively, to apply changes without rebooting, execute:

     ```
     # sysctl -p /etc/sysctl.d/<some_file.conf>
     ```

     The command enables you to read values from the configuration file, which you created earlier.

Additional resources

- For more information about **sysctl**, see the **sysctl(8)** manual page.
- For more information about **/etc/sysctl.d/**, see the **sysctl.d(5)** manual page.

5.2.4. Configuring kernel parameters temporarily through /proc/sys/

The following procedure describes how to set kernel parameters temporarily through the files in the virtual file system **/proc/sys/** directory.

Prerequisites

- Kernel parameters introduction
- Root permissions

Procedure

1. Identify a kernel parameter you want to configure:

   ```
   # ls -l /proc/sys/<TUNABLE_CLASS>/
   ```

   The writable files returned by the command can be used to configure the kernel. The files with read-only permissions provide feedback on the current settings.

2. Assign a target value to the kernel parameter:

   ```
   # echo <TARGET_VALUE> > /proc/sys/<TUNABLE_CLASS>/<PARAMETER>
   ```

   The command makes configuration changes that will disappear once the system is restarted.

3. Optionally, verify the value of the newly set kernel parameter:

```
# cat /proc/sys/<TUNABLE_CLASS>/<PARAMETER>
```

Additional resources

- To permanently modify kernel parameters, either use the **sysctl** command or make manual changes to the configuration files in the **/etc/sysctl.d/** directory.

5.3. KEEPING KERNEL PANIC PARAMETERS DISABLED IN VIRTUALIZED ENVIRONMENTS

When configuring a virtualized environment in Red Hat Enterprise Linux 8 (RHEL 8), you should not enable the **softlockup_panic** and **nmi_watchdog** kernel parameters, as the virtualized environment may trigger a spurious soft lockup that should not require a system panic.

The following sections explain the reasons behind this advice by summarizing:

- What causes a soft lockup.
- Describing the kernel parameters that control a system’s behavior on a soft lockup.
- Explaining how soft lockups may be triggered in a virtualized environment.

5.3.1. What is a soft lockup

A soft lockup is a situation usually caused by a bug, when a task is executing in kernel space on a CPU without rescheduling. The task also does not allow any other task to execute on that particular CPU. As a result, a warning is displayed to a user through the system console. This problem is also referred to as the soft lockup firing.

Additional resources

- For a technical reason behind a soft lockup, example log messages, and other details, see the following *Knowledge Article*.

5.3.2. Parameters controlling kernel panic

The following kernel parameters can be set to control a system’s behavior when a soft lockup is detected.

softlockup_panic

Controls whether or not the kernel will panic when a soft lockup is detected.

Type	Value	Effect
Integer	0	kernel does not panic on soft lockup
Integer	1	kernel panics on soft lockup

By default, on RHEL8 this value is 0.

In order to panic, the system needs to detect a hard lockup first. The detection is controlled by the **nmi_watchdog** parameter.

nmi_watchdog

Controls whether lockup detection mechanisms (**watchdogs**) are active or not. This parameter is of integer type.

Value	Effect
0	disables lockup detector
1	enables lockup detector

The hard lockup detector monitors each CPU for its ability to respond to interrupts.

watchdog_thresh

Controls frequency of watchdog **hrtimer**, NMI events, and soft/hard lockup thresholds.

Default threshold	Soft lockup threshold
10 seconds	2 * **watchdog_thresh**

Setting this parameter to zero disables lockup detection altogether.

Additional resources

- For further information about **nmi_watchdog** and **softlockup_panic**, see the *Softlockup detector and hardlockup detector* document.
- For more details about **watchdog_thresh**, see the *Kernel sysctl* document.

5.3.3. Spurious soft lockups in virtualized environments

The soft lockup firing on physical hosts, as described in Section 5.3.1, “What is a soft lockup”, usually represents a kernel or hardware bug. The same phenomenon happening on guest operating systems in virtualized environments may represent a false warning.

Heavy work-load on a host or high contention over some specific resource such as memory, usually causes a spurious soft lockup firing. This is because the host may schedule out the guest CPU for a period longer than 20 seconds. Then when the guest CPU is again scheduled to run on the host, it experiences a *time jump* which triggers due timers. The timers include also watchdog **hrtimer**, which can consequently report a soft lockup on the guest CPU.

Because a soft lockup in a virtualized environment may be spurious, you should not enable the kernel parameters that would cause a system panic when a soft lockup is reported on a guest CPU.

IMPORTANT

To understand soft lockups in guests, it is essential to know that the host schedules the guest as a task, and the guest then schedules its own tasks.

Additional resources

- For soft lockup definition and technicalities behind its functioning, see Section 5.3.1, “What is a soft lockup”.
- To learn about components of RHEL 8 virtualized environments and their interaction, see *RHEL 8 virtual machine components and their interaction*.

5.4. ADJUSTING KERNEL PARAMETERS FOR DATABASE SERVERS

There are different sets of kernel parameters which can affect performance of specific database applications. The following sections explain what kernel parameters to configure to secure efficient operation of database servers and databases.

5.4.1. Introduction to database servers

A database server is a hardware device which has a certain amount of main memory, and a database (DB) application installed. This DB application provides services as a means of writing the cached data from the main memory, which is usually small and expensive, to DB files (database). These services are provided to multiple clients on a network. There can be as many DB servers as a machine’s main memory and storage allows.

Red Hat Enterprise Linux 8 provides the following database applications:

- MariaDB 10.3
- MySQL 8.0
- PostgreSQL 10
- PostgreSQL 9.6
- PostgreSQL 12 - available since RHEL 8.1.1

5.4.2. Parameters affecting performance of database applications

The following kernel parameters affect performance of database applications.

fs.aio-max-nr

Defines the maximum number of asynchronous I/O operations the system can handle on the server.

NOTE

Raising the **fs.aio-max-nr** parameter produces no additional changes beyond increasing the aio limit.

fs.file-max

Defines the maximum number of file handles (temporary file names or IDs assigned to open files) the system supports at any instance.
The kernel dynamically allocates file handles whenever a file handle is requested by an application. The kernel however does not free these file handles when they are released by the application. The kernel recycles these file handles instead. This means that over time the total number of allocated file handles will increase even though the number of currently used file handles may be low.

kernel.shmall

Defines the total number of shared memory pages that can be used system-wide. To use the entire main memory, the value of the **kernel.shmall** parameter should be ≤ total main memory size.

kernel.shmmax

Defines the maximum size in bytes of a single shared memory segment that a Linux process can allocate in its virtual address space.

kernel.shmmni

Defines the maximum number of shared memory segments the database server is able to handle.

net.ipv4.ip_local_port_range

Defines the port range the system can use for programs which want to connect to a database server without a specific port number.

net.core.rmem_default

Defines the default receive socket memory through Transmission Control Protocol (TCP).

net.core.rmem_max

Defines the maximum receive socket memory through Transmission Control Protocol (TCP).

net.core.wmem_default

Defines the default send socket memory through Transmission Control Protocol (TCP).

net.core.wmem_max

Defines the maximum send socket memory through Transmission Control Protocol (TCP).

vm.dirty_bytes / vm.dirty_ratio

Defines a threshold in bytes / in percentage of dirty-able memory at which a process generating dirty data is started in the **write()** function.

NOTE

Either **vm.dirty_bytes** or **vm.dirty_ratio** can be specified at a time.

vm.dirty_background_bytes / vm.dirty_background_ratio

Defines a threshold in bytes / in percentage of dirty-able memory at which the kernel tries to actively write dirty data to hard-disk.

NOTE

Either **vm.dirty_background_bytes** or **vm.dirty_background_ratio** can be specified at a time.

vm.dirty_writeback_centisecs

Defines a time interval between periodic wake-ups of the kernel threads responsible for writing dirty data to hard-disk.
This kernel parameters measures in 100th's of a second.

vm.dirty_expire_centisecs

Defines the time after which dirty data is old enough to be written to hard-disk.
This kernel parameters measures in 100th's of a second.

Additional resources

- For explanation of dirty data writebacks, how they work, and what kernel parameters relate to them, see the *Dirty pagecache writeback and vm.dirty parameters* document.

CHAPTER 6. GETTING STARTED WITH KERNEL LOGGING

Log files are files that contain messages about the system, including the kernel, services, and applications running on it. The logging system in Red Hat Enterprise Linux is based on the built-in **syslog** protocol. Various utilities use this system to record events and organize them into log files. These files are useful when auditing the operating system or troubleshooting problems.

6.1. WHAT IS THE KERNEL RING BUFFER

During the boot process, the console provides a lot of important information about the initial phase of the system startup. To avoid loss of the early messages the kernel utilizes what is called a ring buffer. This buffer stores all messages, including boot messages, generated by the **printk()** function within the kernel code. The messages from the kernel ring buffer are then read and stored in log files on permanent storage, for example, by the **syslog** service.

The buffer mentioned above is a cyclic data structure which has a fixed size, and is hard-coded into the kernel. Users can display data stored in the kernel ring buffer through the **dmesg** command or the **/var/log/boot.log** file. When the ring buffer is full, the new data overwrites the old.

Additional resources

- For more information about **syslog**, see the **syslog(2)** manual page.
- For more details on how to examine or control boot log messages with **dmesg**, see the **dmesg(1)** manual page.

6.2. ROLE OF PRINTK ON LOG-LEVELS AND KERNEL LOGGING

Each message the kernel reports has a log-level associated with it that defines the importance of the message. The kernel ring buffer, as described in Section 6.1, “What is the kernel ring buffer” , collects kernel messages of all log-levels. It is the **kernel.printk** parameter that defines what messages from the buffer are printed to the console.

The log-level values break down in this order:

- 0 – Kernel emergency. The system is unusable.
- 1 – Kernel alert. Action must be taken immediately.
- 2 – Condition of the kernel is considered critical.
- 3 – General kernel error condition.
- 4 – General kernel warning condition.
- 5 – Kernel notice of a normal but significant condition.
- 6 – Kernel informational message.
- 7 – Kernel debug-level messages.

By default, **kernel.printk** in RHEL 8 contains the following four values:

```
# sysctl kernel.printk
kernel.printk = 7 4 1 7
```

The four values define the following:

1. value. Console log-level, defines the lowest priority of messages printed to the console.
2. value. Default log-level for messages without an explicit log-level attached to them.
3. value. Sets the lowest possible log-level configuration for the console log-level.
4. value. Sets default value for the console log-level at boot time.
   Each of these values above defines a different rule for handling error messages.

NOTE

Certain kernel command line parameters, such as **quiet** or **debug**, change the default **kernel.printk** values.

Additional resources

- For more information on **kernel.printk** and log-levels, see the **syslog(2)** manual page.

CHAPTER 7. INSTALLING AND CONFIGURING KDUMP

7.1. WHAT IS KDUMP

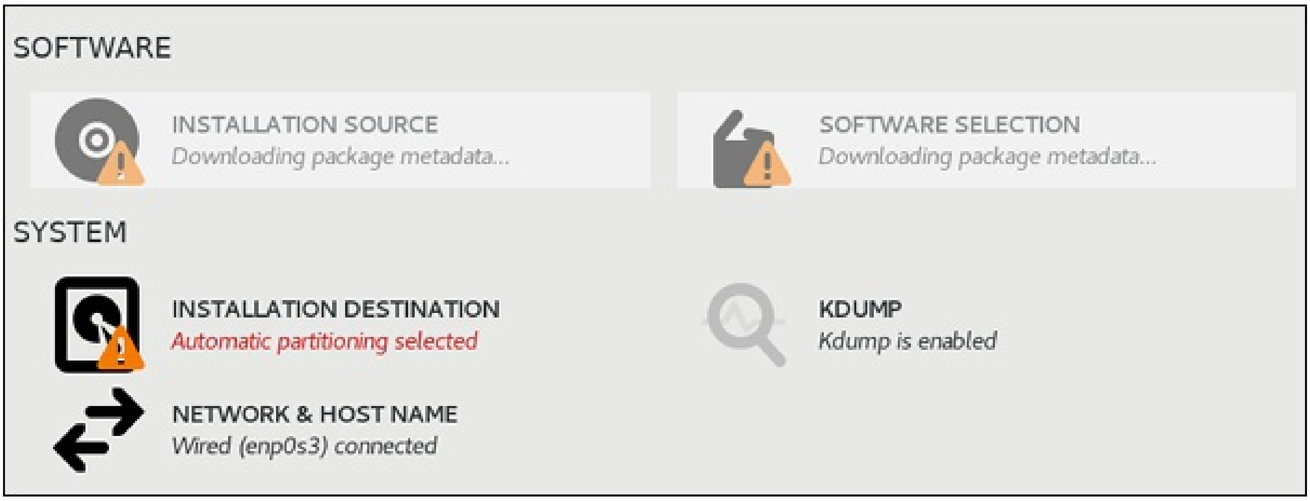
kdump is a service providing a crash dumping mechanism. The service enables you to save the contents of the system's memory for later analysis. **kdump** uses the **kexec** system call to boot into the second kernel (a *capture kernel*) without rebooting; and then captures the contents of the crashed kernel's memory (a *crash dump* or a *vmcore*) and saves it. The second kernel resides in a reserved part of the system memory.

IMPORTANT

A kernel crash dump can be the only information available in the event of a system failure (a critical bug). Therefore, ensuring that **kdump** is operational is important in mission-critical environments. Red Hat advise that system administrators regularly update and test **kexec-tools** in your normal kernel update cycle. This is especially important when new kernel features are implemented.

7.2. INSTALLING KDUMP

In many cases, the **kdump** service is installed and activated by default on the new Red Hat Enterprise Linux installations. The **Anaconda** installer provides a screen for **kdump** configuration when performing an interactive installation using the graphical or text interface. The installer screen is titled **Kdump** and is available from the main **Installation Summary** screen, and only allows limited configuration - you can only select whether **kdump** is enabled and how much memory is reserved.



SOFTWARE

INSTALLATION SOURCE
Downloading package metadata...

SOFTWARE SELECTION
Downloading package metadata...

SYSTEM

INSTALLATION DESTINATION
Automatic partitioning selected

KDUMP
Kdump is enabled

NETWORK & HOST NAME
Wired (enp0s3) connected

Some installation options, such as custom Kickstart installations, in some cases do not install or enable **kdump** by default. If this is the case on your system, follow the procedure below to install **kdump**.

Prerequisites

- An active Red Hat Enterprise Linux subscription
- A repository containing the **kexec-tools** package for your system CPU architecture
- Fulfilled **kdump** requirements

Procedure

1. Execute the following command to check whether **kdump** is installed on your system:

   ```
   $ rpm -q kexec-tools
   ```

   Output if the package is installed:

   ```
   kexec-tools-2.0.17-11.el8.x86_64
   ```

   Output if the package is not installed:

   ```
   package kexec-tools is not installed
   ```

2. Install **kdump** and other necessary packages by:

   ```
   # yum install kexec-tools
   ```

IMPORTANT

Starting with Red Hat Enterprise Linux 7.4 (kernel-3.10.0-693.el7) the **Intel IOMMU** driver is supported with **kdump**. For prior versions, Red Hat Enterprise Linux 7.3 (kernel-3.10.0-514[.XYZ].el7) and earlier, it is advised that **Intel IOMMU** support is disabled, otherwise kdump kernel is likely to become unresponsive.

Additional resources

- Information about memory requirements for **kdump** is available in Section 7.5.1, “Memory requirements for kdump”.

7.3. CONFIGURING KDUMP ON THE COMMAND LINE

7.3.1. Configuring kdump memory usage

The memory reserved for the **kdump** feature is always reserved during the system boot. The amount of memory is specified in the system’s Grand Unified Bootloader (GRUB) 2 configuration. The procedure below describes how to configure the memory reserved for **kdump** through the command line.

Prerequisites

- Fulfilled **kdump** requirements

Procedure

1. Edit the **/etc/default/grub** file using the root permissions.

2. Set the **crashkernel=** option to the required value.
   For example, to reserve 128 MB of memory, use the following:

   ```
   crashkernel=128M
   ```

   Alternatively, you can set the amount of reserved memory to a variable depending on the total amount of installed memory. The syntax for memory reservation into a variable is **crashkernel=*<range1>*:*<size1>*,*<range2>*:*<size2>***. For example:

```
crashkernel=512M-2G:64M,2G-:128M
```

The above example reserves 64 MB of memory if the total amount of system memory is 512 MB or higher and lower than 2 GB. If the total amount of memory is more than 2 GB, 128 MB is reserved for **kdump** instead.

- Offset the reserved memory.
  Some systems require to reserve memory with a certain fixed offset since crashkernel reservation is very early, and it wants to reserve some area for special usage. If the offset is set, the reserved memory begins there. To offset the reserved memory, use the following syntax:

  ```
  crashkernel=128M@16M
  ```

  The example above means that **kdump** reserves 128 MB of memory starting at 16 MB (physical address 0x01000000). If the offset parameter is set to 0 or omitted entirely, **kdump** offsets the reserved memory automatically. This syntax can also be used when setting a variable memory reservation as described above; in this case, the offset is always specified last (for example, **crashkernel=512M-2G:64M,2G-:128M@16M**).

3. Use the following command to update the GRUB2 configuration file:

   ```
   # grub2-mkconfig -o /boot/grub2/grub.cfg
   ```

NOTE

The alternative way to configure memory for **kdump** is to append the **crashkernel=<*SOME_VALUE*>** parameter to the **kernelopts** variable with the **grub2-editenv** which will update all of your boot entries. Or you can use the **grubby** utility to update kernel command line parameters of just one entry.

Additional resources

- The **crashkernel=** option can be defined in multiple ways. The **auto** value enables automatic configuration of reserved memory based on the total amount of memory in the system, following the guidelines described in Section 7.5.1, “Memory requirements for kdump”.
- For more information on boot entries, **kernelopts**, and how to work with **grub2-editenv** and **grubby** see Chapter 4, *Configuring kernel command-line parameters*.

7.3.2. Configuring the kdump target

When a kernel crash is captured, the core dump can be either stored as a file in a local file system, written directly to a device, or sent over a network using the **NFS** (Network File System) or **SSH** (Secure Shell) protocol. Only one of these options can be set at a time, and the default behavior is to store the vmcore file in the **/var/crash/** directory of the local file system.

Prerequisites

- Fulfilled **kdump** requirements

Procedure

To store the **vmcore** file in **/var/crash/** directory of the local file system:

- Edit the **/etc/kdump.conf** file and specify the path:

  ```
  path /var/crash
  ```

  The option **path /var/crash** represents the path to the file system in which **kdump** saves the **vmcore** file. When you specify a dump target in the **/etc/kdump.conf** file, then the **path** is relative to the specified dump target.

  If you do not specify a dump target in the **/etc/kdump.conf** file, then the **path** represents the absolute path from the root directory. Depending on what is mounted in the current system, the dump target and the adjusted dump path are taken automatically.

WARNING

kdump saves the **vmcore** file in **/var/crash/var/crash** directory, when the dump target is mounted at **/var/crash** and the option **path** is also set as **/var/crash** in the **/etc/kdump.conf** file. For example, in the following instance, the **ext4** file system is already mounted at **/var/crash** and the **path** are set as **/var/crash**:

```
grep -v ^# etc/kdump.conf | grep -v ^$
ext4 /dev/mapper/vg00-varcrashvol
path /var/crash
core_collector makedumpfile -c --message-level 1 -d 31
```

This results in the **/var/crash/var/crash** path. To solve this problem, use the option **path** / instead of **path /var/crash**

To change the local directory in which the core dump is to be saved, as **root**, edit the **/etc/kdump.conf** configuration file as described below.

1. Remove the hash sign ("#") from the beginning of the **#path /var/crash** line.

2. Replace the value with the intended directory path. For example:

   ```
   path /usr/local/cores
   ```

IMPORTANT

In Red Hat Enterprise Linux 8, the directory defined as the kdump target using the **path** directive must exist when the **kdump** systemd service is started - otherwise the service fails. This behavior is different from earlier releases of Red Hat Enterprise Linux, where the directory was being created automatically if it did not exist when starting the service.

To write the file to a different partition, as **root**, edit the **/etc/kdump.conf** configuration file as described below.

1. Remove the hash sign ("#") from the beginning of the **#ext4** line, depending on your choice.

   - device name (the **#ext4 /dev/vg/lv kdump** line)

- file system label (the **#ext4 LABEL=/boot** line)
- UUID (the **#ext4 UUID=03138356-5e61-4ab3-b58e-27507ac41937** line)

2. Change the file system type as well as the device name, label or UUID to the desired values. For example:

```
ext4 UUID=03138356-5e61-4ab3-b58e-27507ac41937
```

IMPORTANT

It is recommended to specify storage devices using a **LABEL=** or **UUID=**. Disk device names such as **/dev/sda3** are not guaranteed to be consistent across reboot.

IMPORTANT

When dumping to Direct Access Storage Device (DASD) on IBM Z hardware, it is essential that the dump devices are correctly specified in **/etc/dasd.conf** before proceeding.

To write the dump directly to a device:

1. Remove the hash sign ("#") from the beginning of the **#raw /dev/vg/lv_kdump** line.
2. Replace the value with the intended device name. For example:

```
raw /dev/sdb1
```

To store the dump to a remote machine using the **NFS** protocol:

1. Remove the hash sign ("#") from the beginning of the **#nfs my.server.com:/export/tmp** line.
2. Replace the value with a valid hostname and directory path. For example:

```
nfs penguin.example.com:/export/cores
```

To store the dump to a remote machine using the **SSH** protocol:

1. Remove the hash sign ("#") from the beginning of the **#ssh user@my.server.com** line.
2. Replace the value with a valid username and hostname.
3. Include your **SSH** key in the configuration.
   - Remove the hash sign from the beginning of the **#sshkey /root/.ssh/kdump_id_rsa** line.
   - Change the value to the location of a key valid on the server you are trying to dump to. For example:

```
ssh john@penguin.example.com
sshkey /root/.ssh/mykey
```

Additional resources

- For a complete list of currently supported and unsupported targets sorted by type, see Section 7.5.3, "Supported kdump targets" .
- For information on how to configure an SSH server and set up a key-based authentication, see *Configuring basic system settings* in Red Hat Enterprise Linux.

7.3.3. Configuring the core collector

kdump uses a program specified as **core collector** to capture the vmcore. Currently, the only fully supported **core collector** is the **makedumpfile** utility. It has several configurable options, which affect the collection process. For example the extent of collected data, or whether the resulting vmcore should be compressed.

To enable and configure the **core collector**, follow the procedure below.

Prerequisites

- Fulfilled **kdump** requirements

Procedure

1. As **root**, edit the **/etc/kdump.conf** configuration file and remove the hash sign ("#") from the beginning of the **#core_collector makedumpfile -l --message-level 1 -d 31**.
2. Add the **-c** parameter. For example:

   ```
   core_collector makedumpfile -c
   ```

   The command above enables the dump file compression.
3. Add the **-d *value*** parameter. For example:

   ```
   core_collector makedumpfile -d 17 -c
   ```

   The command above removes both zero and free pages from the dump. The *value* represents a bitmask, where each bit is associated with a certain type of memory pages and determines whether that type of pages will be collected. For description of respective bits see Section 7.5.4, "Supported kdump filtering levels" .

Additional resources

- See the **makedumpfile(8)** man page for a complete list of available options.

7.3.4. Configuring the kdump default failure responses

By default, when **kdump** fails to create a vmcore dump file at the target location specified in Section 7.3.2, "Configuring the kdump target" , the system reboots, and the dump is lost in the process. To change this behavior, follow the procedure below.

Prerequisites

- Fulfilled **kdump** requirements

Procedure

1. As **root**, remove the hash sign ("#") from the beginning of the **#default shell** line in the **/etc/kdump.conf** configuration file.

2. Replace the value with a desired action as described in Section 7.5.5, “Supported default failure responses”. For example:

   ```
   default poweroff
   ```

7.3.5. Enabling and disabling the kdump service

To start the **kdump** service at boot time, follow the procedure below.

Prerequisites

- Fulfilled **kdump** requirements.

- All configuration is set up according to your needs.

Procedure

1. To enable the **kdump** service, use the following command:

   ```
   # systemctl enable kdump.service
   ```

   This enables the service for **multi-user.target**.

2. To start the service in the current session, use the following command:

   ```
   # systemctl start kdump.service
   ```

3. To stop the **kdump** service, type the following command:

   ```
   # systemctl stop kdump.service
   ```

4. To disable the **kdump** service, execute the following command:

   ```
   # systemctl disable kdump.service
   ```

> **WARNING**
>
> It is recommended to set **kptr_restrict=1** as default. When **kptr_restrict** is set to (1) as default, the **kdumpctl** service loads the crash kernel even if Kernel Address Space Layout (KASLR) is enabled or not enabled.

Troubleshooting step

When **kptr_restrict** is not set to (1), and if KASLR is enabled, the contents of **/proc/kore** file are generated as all zeros. Consequently, the **kdumpctl** service fails to access the **/proc/kcore** and load the crash kernel.

To work around this problem, the **kexec-kdump-howto.txt** file displays a warning message, which specifies to keep the recommended setting as **kptr_restrict=1**.

To ensure that **kdumpctl** service loads the crash kernel, verify that:

- Kernel **kptr_restrict=1** in the **sysctl.conf** file.

Additional resources

- For more information on **systemd** and configuring services in general, see *Configuring basic system settings* in Red Hat Enterprise Linux.

7.4. CONFIGURING KDUMP IN THE WEB CONSOLE

The following sections provide an overview of how to setup and test the **kdump** configuration through the Red Hat Enterprise Linux web console. The web console is part of a default installation of Red Hat Enterprise Linux 8 and enables or disables the **kdump** service at boot time. Further, the web console conveniently enables you to configure the reserved memory for **kdump**; or to select the *vmcore* saving location in an uncompressed or compressed format.

Prerequisites

- See *Red Hat Enterprise Linux web console* for further details.

7.4.1. Configuring kdump memory usage and target location in web console

The procedure below shows you how to use the **Kernel Dump** tab in the Red Hat Enterprise Linux web console interface to configure the amount of memory that is reserved for the kdump kernel. The procedure also describes how to specify the target location of the vmcore dump file and how to test your configuration.

Prerequisites

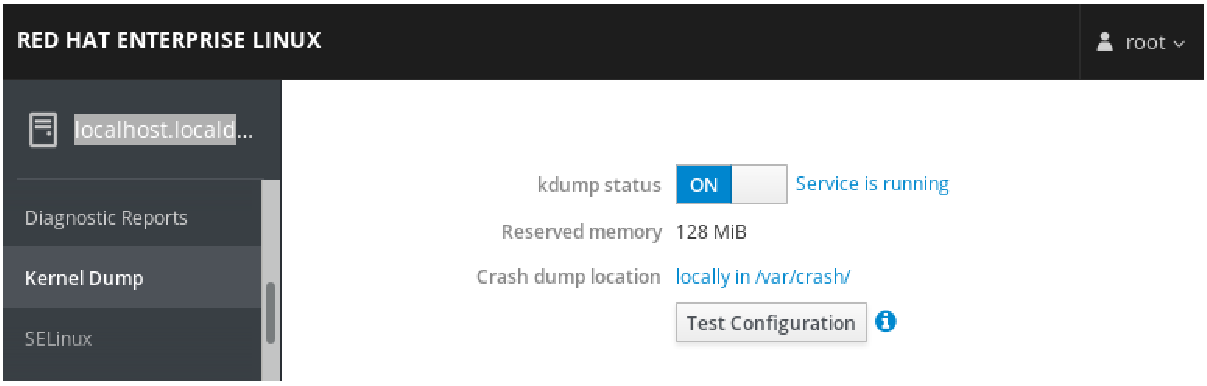
- Introduction to operating the *web console*

Procedure

1. Open the **Kernel Dump** tab and start the **kdump** service.

2. Configure the **kdump** memory usage through the *command line*.

3. Click the link next to the **Crash dump location** option.

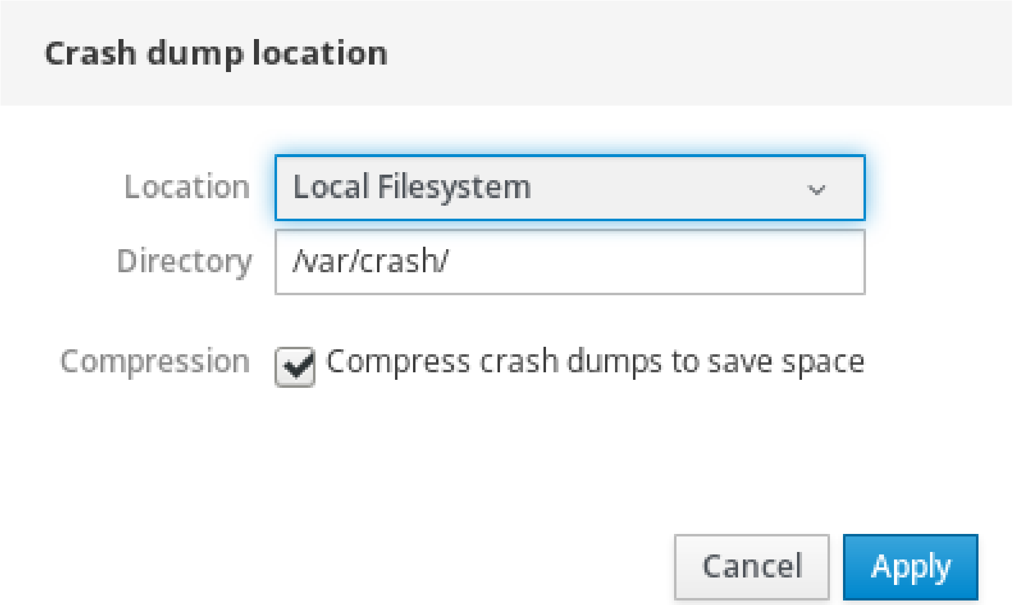


kdump status ON Service is running

Reserved memory 128 MiB

Crash dump location locally in /var/crash/

4. Select the **Local Filesystem** option from the drop-down and specify the directory you want to save the dump in.



Crash dump location

Location: Local Filesystem

Directory: /var/crash/

Compression: ☑ Compress crash dumps to save space

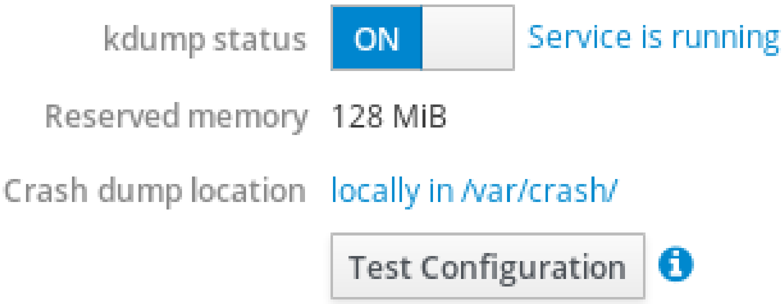
- Alternatively, select the **Remote over SSH** option from the drop-down to send the vmcore to a remote machine using the SSH protocol.
  Fill the **Server**, **ssh key**, and **Directory** fields with the remote machine address, ssh key location, and a target directory.

- Another choice is to select the **Remote over NFS** option from the drop-down and fill the **Mount** field to send the vmcore to a remote machine using the NFS protocol.

NOTE

Tick the **Compression** check box to reduce the size of the vmcore file.

5. Test your configuration by crashing the kernel.



kdump status ON Service is running

Reserved memory 128 MiB

Crash dump location locally in /var/crash/

> **WARNING**
>
> This step disrupts execution of the kernel and results in a system crash and loss of data.

Additional resources

- For a complete list of currently supported targets for **kdump**, see *Supported kdump targets* .
- For information on how to configure an SSH server and set up a key-based authentication, see *Using secure communications between two systems with OpenSSH* .

7.5. SUPPORTED KDUMP CONFIGURATIONS AND TARGETS

7.5.1. Memory requirements for kdump

In order for **kdump** to be able to capture a kernel crash dump and save it for further analysis, a part of the system memory has to be permanently reserved for the capture kernel. When reserved, this part of the system memory is not available to the main kernel.

The memory requirements vary based on certain system parameters. One of the major factors is the system's hardware architecture. To find out the exact machine architecture (such as Intel 64 and AMD64, also known as x86_64) and print it to standard output, use the following command:

```
$ uname -m
```

The table below contains a list of minimum memory requirements to automatically reserve a memory size for **kdump**. The size changes according to the system's architecture and total available physical memory.

Table 7.1. Minimum Amount of Reserved Memory Required for kdump

Architecture	Available Memory	Minimum Reserved Memory
AMD64 and Intel 64 (**x86_64**)	1 GB to 64 GB	160 MB of RAM.

Architecture	Available Memory	Minimum Reserved Memory
	64 GB to 1 TB	256 MB of RAM.
	1 TB and more	512 MB of RAM.
64-bit ARM architecture (**arm64**)	2 GB and more	512 MB of RAM.
IBM Power Systems (**ppc64le**)	2 GB to 4 GB	384 MB of RAM.
	4 GB to 16 GB	512 MB of RAM.
	16 GB to 64 GB	1 GB of RAM.
	64 GB to 128 GB	2 GB of RAM.
	128 GB and more	4 GB of RAM.
IBM Z (**s390x**)	4 GB to 64 GB	160 MB of RAM.
	64 GB to 1 TB	256 MB of RAM.
	1 TB and more	512 MB of RAM.

On many systems, **kdump** is able to estimate the amount of required memory and reserve it automatically. This behavior is enabled by default, but only works on systems that have more than a certain amount of total available memory , which varies based on the system architecture.

IMPORTANT

The automatic configuration of reserved memory based on the total amount of memory in the system is a best effort estimation. The actual required memory may vary due to other factors such as I/O devices. Using not enough of memory might cause that a debug kernel is not able to boot as a capture kernel in case of a kernel panic. To avoid this problem, sufficiently increase the crash kernel memory.

Additional resources

- For information on how to change memory settings on the command line, see Section 7.3.1, “Configuring kdump memory usage”.
- For instructions on how to set up the amount of reserved memory through the web console, see Section 7.4.1, “Configuring kdump memory usage and target location in web console” .
- For more information about various Red Hat Enterprise Linux technology capabilities and limits, see the *technology capabilities and limits tables* .

7.5.2. Minimum threshold for automatic memory reservation

On some systems, it is possible to allocate memory for **kdump** automatically, either by using the **crashkernel=auto** parameter in the boot loader configuration file, or by enabling this option in the graphical configuration utility. For this automatic reservation to work, however, a certain amount of total memory needs to be available in the system. The amount differs based on the system's architecture.

The table below lists the thresholds for automatic memory allocation. If the system has less memory than specified in the table, the memory needs to be reserved manually.

Table 7.2. Minimum Amount of Memory Required for Automatic Memory Reservation

Architecture	Required Memory
AMD64 and Intel 64 (**x86_64**)	2 GB
IBM Power Systems (**ppc64le**)	2 GB
IBM Z (**s390x**)	4 GB

Additional resources

- For information on how to manually change these settings on the command line, see Section 7.3.1, "Configuring kdump memory usage" .
- For instructions on how to manually change the amount of reserved memory through the web console, see Section 7.4.1, "Configuring kdump memory usage and target location in web console".

7.5.3. Supported kdump targets

When a kernel crash is captured, the vmcore dump file can be either written directly to a device, stored as a file on a local file system, or sent over a network. The table below contains a complete list of dump targets that are currently supported or explicitly unsupported by **kdump**.

Table 7.3. Supported kdump Targets

Type	Supported Targets	Unsupported Targets
Raw device	All locally attached raw disks and partitions.	
Local file system	**ext2**, **ext3**, **ext4**, and **xfs** file systems on directly attached disk drives, hardware RAID logical drives, LVM devices, and **mdraid** arrays.	Any local file system not explicitly listed as supported in this table, including the **auto** type (automatic file system detection).
Remote directory	Remote directories accessed using the **NFS** or **SSH** protocol over **IPv4**.	Remote directories on the **rootfs** file system accessed using the **NFS** protocol.

Type	Supported Targets	Unsupported Targets
Remote directories accessed using the **iSCSI** protocol over both hardware and software initiators.	Remote directories accessed using the **iSCSI** protocol on **be2iscsi** hardware.	Multipath-based storages.
		Remote directories accessed over **IPv6**.
		Remote directories accessed using the **SMB** or **CIFS** protocol.
		Remote directories accessed using the **FCoE** (*Fibre Channel over Ethernet*) protocol.
		Remote directories accessed using wireless network interfaces.

IMPORTANT

Utilizing firmware assisted dump (**fadump**) to capture a vmcore and store it to a remote machine using SSH or NFS protocol causes renaming of the network interface to **kdump-<interface-name>**. The renaming happens if the **<interface-name>** is generic, for example ***eth#**, **net#**, and so on. This problem occurs because the vmcore capture scripts in the initial RAM disk (**initrd**) add the *kdump-* prefix to the network interface name to secure persistent naming. Since the same **initrd** is used also for a regular boot, the interface name is changed for the production kernel too.

Additional resources

- For information on how to configure the target type on the command line, see Section 7.3.2, “Configuring the kdump target”.
- For information on how to configure the target through the web console, see Section 7.4.1, “Configuring kdump memory usage and target location in web console”.

7.5.4. Supported kdump filtering levels

To reduce the size of the dump file, **kdump** uses the **makedumpfile** core collector to compress the data and optionally to omit unwanted information. The table below contains a complete list of filtering levels that are currently supported by the **makedumpfile** utility.

Table 7.4. Supported Filtering Levels

Option	Description
1	Zero pages

Option	Description
2	Cache pages
4	Cache private
8	User pages
16	Free pages

NOTE

The **makedumpfile** command supports removal of transparent huge pages and hugetlbfs pages. Consider both these types of hugepages User Pages and remove them using the **-8** level.

Additional resources

- For instructions on how to configure the core collector on the command line, see Section 7.3.3, “Configuring the core collector”.

7.5.5. Supported default failure responses

By default, when **kdump** fails to create a core dump, the operating system reboots. You can, however, configure **kdump** to perform a different operation in case it fails to save the core dump to the primary target. The table below lists all default actions that are currently supported.

Table 7.5. Supported Default Actions

Option	Description
dump_to_rootfs	Attempt to save the core dump to the root file system. This option is especially useful in combination with a network target: if the network target is unreachable, this option configures kdump to save the core dump locally. The system is rebooted afterwards.
reboot	Reboot the system, losing the core dump in the process.
halt	Halt the system, losing the core dump in the process.
poweroff	Power off the system, losing the core dump in the process.
shell	Run a shell session from within the initramfs, allowing the user to record the core dump manually.

Additional resources

- For detailed information on how to set up the default failure responses on the command line, see Section 7.3.4, "Configuring the kdump default failure responses" .

7.5.6. Estimating kdump size

When planning and building your kdump environment, it is necessary to know how much space is required for the dump file before one is produced.

The **makedumpfile --mem-usage** command provides a useful report about excludable pages, and can be used to determine which dump level you want to assign. Run this command when the system is under representative load, otherwise **makedumpfile --mem-usage** returns a smaller value than is expected in your production environment.

```
[root@hostname ~]# makedumpfile --mem-usage /proc/kcore

TYPE        PAGES                   EXCLUDABLE   DESCRIPTION
----------------------------------------------------------------------
ZERO        501635              yes          Pages filled with zero
CACHE         51657             yes          Cache pages
CACHE_PRIVATE   5442                 yes          Cache pages + private
USER         16301             yes          User process pages
FREE         77738211             yes          Free pages
KERN_DATA      1333192              no          Dumpable kernel data
```

IMPORTANT

The **makedumpfile --mem-usage** command reports in **pages**. This means that you have to calculate the size of memory in use against the kernel page size. By default the Red Hat Enterprise Linux kernel uses 4 KB sized pages for AMD64 and Intel 64 architectures, and 64 KB sized pages for IBM POWER architectures.

7.6. TESTING THE KDUMP CONFIGURATION

The following procedure describes how to test that the kernel dump process works and is valid before the machine enters production.

WARNING

The commands below cause the kernel to crash. Use caution when following these steps, and never carelessly use them on active production system.

Procedure

1. Reboot the system with **kdump** enabled.

2. Make sure that **kdump** is running:

```
~]# systemctl is-active kdump
active
```

3. Force the Linux kernel to crash:

```
echo 1 > /proc/sys/kernel/sysrq
echo c > /proc/sysrq-trigger
```

WARNING

The command above crashes the kernel and a reboot is required.

Once booted again, the ***address-YYYY-MM-DD-HH:MM:SS*****/vmcore** file is created at the location you have specified in **/etc/kdump.conf** (by default to **/var/crash/**).

NOTE

In addition to confirming the validity of the configuration, it is possible to use this action to record how long it takes for a crash dump to complete, while a representative load was running.

7.7. USING KEXEC TO REBOOT THE KERNEL

The **kexec** system call enables loading and booting into another kernel from the currently running kernel, thus performing a function of a boot loader from within the kernel.

The **kexec** utility loads the kernel and the **initramfs** image for the **kexec** system call to boot into another kernel.

The following procedure describes how to manually invoke the **kexec** system call when using the **kexec** utility to reboot into another kernel.

Procedure

1. Execute the **kexec** utility:

```
# kexec -l /boot/vmlinuz-3.10.0-1040.el7.x86_64 --initrd=/boot/initramfs-3.10.0-
1040.el7.x86_64.img --reuse-cmdline
```

   The command manually loads the kernel and the initramfs image for the **kexec** system call.

2. Reboot the system:

```
# reboot
```

   The command detects the kernel, shuts down all services and then calls the **kexec** system call to reboot into the kernel you provided in the previous step.

WARNING

When you use the **kexec -e** command to reboot the kernel, the system does not go through the standard shutdown sequence before starting the next kernel, which may cause data loss or an unresponsive system.

7.8. BLACKLISTING KERNEL DRIVERS FOR KDUMP

Blacklisting kernel drivers for kdump is a mechanism to prevent the intended kernel drivers from loading. Blacklisting kernel drivers prevents the **oom killer** or other crash kernel failures.

To blacklist the kernel drivers, you may update the **KDUMP_COMMANDLINE_APPEND=** variable in the **/etc/sysconfig/kdump** file and specify one of the following blacklisting option:

- **rd.driver.blacklist=<modules>**
- **modprobe.blacklist=<modules>**

When you blacklist drivers in **/etc/sysconfig/kdump** file, it prevents the **kdump initramfs** from loading the blacklisted modules.

The following procedure describes how to blacklist a kernel driver to prevent crash kernel failures.

Procedure

1. Select the kernel module that you intend to blacklist:

```
$ lsmod

Module                  Size  Used by
fuse                  126976  3
xt_CHECKSUM            16384  1
ipt_MASQUERADE         16384  1
uinput                 20480  1
xt_conntrack           16384  1
```

   The **lsmod** command displays a list of modules that are loaded to the currently running kernel.

2. Update the **KDUMP_COMMANDLINE_APPEND=** line in the **/etc/sysconfig/kdump** file as follows:

```
KDUMP_COMMANDLINE_APPEND="rd.driver.blacklist=hv_vmbus,hv_storvsc,hv_utils,hv_netv
sc,hid-hyperv"
```

3. You can also update the **KDUMP_COMMANDLINE_APPEND=** line in the **/etc/sysconfig/kdump** file as follows:

```
KDUMP_COMMANDLINE_APPEND="modprobe.blacklist=emcp modprobe.blacklist=bnx2fc
modprobe.blacklist=libfcoe modprobe.blacklist=fcoe"
```

4. Restart the **kdump** service:

   ```
   $ systemctl restart kdump
   ```

Additional resources

- For more information concerning the **oom killer**, see the following Knowledge Article.
- The **dracut.cmdline** manpage for modules blacklist options.

7.9. RUNNING KDUMP ON SYSTEMS WITH ENCRYPTED DISK

When running an encrypted partition created by the Logical Volume Manager (LVM) tool, systems require a certain amount of available memory. If the system has less than the required amount of available memory, the **cryptsetup** utility fails to mount the partition. As a result, capturing the **vmcore** file to a local **kdump** target location (with LVM and enabled encryption), fails in the second kernel (capture kernel).

This procedure describes the running **kdump** mechanism by increasing the **crashkernel=** value, using a remote **kdump** target, or using a key derivation function (KDF).

Procedure

Run the **kdump** mechanism using one of the following procedures:

- To run the **kdump** define one of the following:
  - Configure a remote **kdump** target.
  - Define the dump to an unencrypted partition.
  - Specify an increased **crashkernel=** value to the required level.
- Add an extra key slot by using a key derivation function (KDF):
  1. **cryptsetup luksAddKey --pbkdf pbkdf2 /dev/vda2**
  2. **cryptsetup config --key-slot 1 --priority prefer /dev/vda2**
  3. **cryptsetup luksDump /dev/vda2**

Using the default KDF of the encrypted partition may consume a lot of memory. You must manually provide the password in the second kernel (capture), even if you encounter an Out of Memory (OOM) error message.

> **WARNING**
>
> Adding an extra key slot can have a negative effect on security, as multiple keys can decrypt an encrypted volume. This may cause a potential risk to the volume.

7.10. ANALYZING A CORE DUMP

To determine the cause of the system crash, you can use the **crash** utility, which provides an interactive prompt very similar to the GNU Debugger (GDB). This utility allows you to interactively analyze a core dump created by **kdump**, **netdump**, **diskdump** or **xendump** as well as a running Linux system. Alternatively, you have the option to use the *Kdump Helper* or *Kernel Oops Analyzer*.

7.10.1. Installing the crash utility

The following procedure describes how to install the **crash** analyzing tool.

Procedure

1. Enable the relevant **baseos** and **appstream** repositories:

   ```
   # subscription-manager repos --enable baseos repository
   ```

   ```
   # subscription-manager repos --enable appstream repository
   ```

2. Install the **crash** package:

   ```
   # yum install crash
   ```

3. Install the **kernel-debuginfo** package:

   ```
   # yum install kernel-debuginfo
   ```

   The package corresponds to your running kernel and provides the data necessary for the dump analysis.

Additional resources

- For more information about how to work with repositories using the **subscription-manager** utility, see *Configuring basic system settings*.

7.10.2. Running and exiting the crash utility

The following procedure describes how to start the crash utility for analyzing the cause of the system crash.

Prerequisites

- Identify the currently running kernel (for example **4.18.0-5.el8.x86_64**).

Procedure

1. To start the **crash** utility, two necessary parameters need to be passed to the command:
   - The debug-info (a decompressed vmlinuz image), for example **/usr/lib/debug/lib/modules/4.18.0-5.el8.x86_64/vmlinux** provided through a specific **kernel-debuginfo** package.
   - The actual vmcore file, for example **/var/crash/127.0.0.1-2018-10-06-14:05:33/vmcore**
     The resulting **crash** command then looks like this:

```
# crash /usr/lib/debug/lib/modules/4.18.0-5.el8.x86_64/vmlinux /var/crash/127.0.0.1-
2018-10-06-14:05:33/vmcore
```

Use the same *<kernel>* version that was captured by **kdump**.

> **Example 7.1. Running the crash utility**
>
> The following example shows analyzing a core dump created on October 6 2018 at 14:05 PM, using the 4.18.0-5.el8.x86_64 kernel.
>
> ```
> ...
> WARNING: kernel relocated [202MB]: patching 90160 gdb minimal_symbol values
>
>       KERNEL: /usr/lib/debug/lib/modules/4.18.0-5.el8.x86_64/vmlinux
>     DUMPFILE: /var/crash/127.0.0.1-2018-10-06-14:05:33/vmcore  [PARTIAL DUMP]
>         CPUS: 2
>         DATE: Sat Oct  6 14:05:16 2018
>       UPTIME: 01:03:57
> LOAD AVERAGE: 0.00, 0.00, 0.00
>        TASKS: 586
>     NODENAME: localhost.localdomain
>      RELEASE: 4.18.0-5.el8.x86_64
>      VERSION: #1 SMP Wed Aug 29 11:51:55 UTC 2018
>      MACHINE: x86_64  (2904 Mhz)
>       MEMORY: 2.9 GB
>        PANIC: "sysrq: SysRq : Trigger a crash"
>          PID: 10635
>      COMMAND: "bash"
>         TASK: ffff8d6c84271800  [THREAD_INFO: ffff8d6c84271800]
>          CPU: 1
>        STATE: TASK_RUNNING (SYSRQ)
>
> crash>
> ```

2. To exit the interactive prompt and terminate **crash**, type **exit** or **q**.

   > **Example 7.2. Exiting the crash utility**
   >
   > ```
   > crash> exit
   > ~]#
   > ```

> **NOTE**
>
> The **crash** command can also be used as a powerful tool for debugging a live system. However use it with caution so as not to break your system.

7.10.3. Displaying various indicators in the crash utility

The following procedures describe how to use the crash utility and display various indicators, such as a kernel message buffer, a backtrace, a process status, virtual memory information and open files.

Displaying the message buffer

- To display the kernel message buffer, type the **log** command at the interactive prompt as displayed in the example below:

```
crash> log
... several lines omitted ...
EIP: 0060:[<c068124f>] EFLAGS: 00010096 CPU: 2
EIP is at sysrq_handle_crash+0xf/0x20
EAX: 00000063 EBX: 00000063 ECX: c09e1c8c EDX: 00000000
ESI: c0a09ca0 EDI: 00000286 EBP: 00000000 ESP: ef4dbf24
 DS: 007b ES: 007b FS: 00d8 GS: 00e0 SS: 0068
Process bash (pid: 5591, ti=ef4da000 task=f196d560 task.ti=ef4da000)
Stack:
 c068146b c0960891 c0968653 00000003 00000000 00000002 efade5c0 c06814d0
<0> fffffffb c068150f b7776000 f2600c40 c0569ec4 ef4dbf9c 00000002 b7776000
<0> efade5c0 00000002 b7776000 c0569e60 c051de50 ef4dbf9c f196d560 ef4dbfb4
Call Trace:
 [<c068146b>] ? __handle_sysrq+0xfb/0x160
 [<c06814d0>] ? write_sysrq_trigger+0x0/0x50
 [<c068150f>] ? write_sysrq_trigger+0x3f/0x50
 [<c0569ec4>] ? proc_reg_write+0x64/0xa0
 [<c0569e60>] ? proc_reg_write+0x0/0xa0
 [<c051de50>] ? vfs_write+0xa0/0x190
 [<c051e8d1>] ? sys_write+0x41/0x70
 [<c0409adc>] ? syscall_call+0x7/0xb
Code: a0 c0 01 0f b6 41 03 19 d2 f7 d2 83 e2 03 83 e0 cf c1 e2 04 09 d0 88 41 03 f3 c3 90 c7 05
c8 1b 9e c0 01 00 00 00 0f ae f8 89 f6 <c6> 05 00 00 00 00 01 c3 89 f6 8d bc 27 00 00 00 00 8d 50
d0 83
EIP: [<c068124f>] sysrq_handle_crash+0xf/0x20 SS:ESP 0068:ef4dbf24
CR2: 0000000000000000
```

Type **help log** for more information on the command usage.

NOTE

The kernel message buffer includes the most essential information about the system crash and, as such, it is always dumped first in to the **vmcore-dmesg.txt** file. This is useful when an attempt to get the full **vmcore** file failed, for example because of lack of space on the target location. By default, **vmcore-dmesg.txt** is located in the **/var/crash/** directory.

Displaying a backtrace

- To display the kernel stack trace, use the **bt** command.

```
crash> bt
PID: 5591   TASK: f196d560  CPU: 2   COMMAND: "bash"
 #0 [ef4dbdcc] crash_kexec at c0494922
 #1 [ef4dbe20] oops_end at c080e402
 #2 [ef4dbe34] no_context at c043089d
 #3 [ef4dbe58] bad_area at c0430b26
 #4 [ef4dbe6c] do_page_fault at c080fb9b
 #5 [ef4dbee4] error_code (via page_fault) at c080d809
    EAX: 00000063  EBX: 00000063  ECX: c09e1c8c  EDX: 00000000  EBP: 00000000
    DS:  007b      ESI: c0a09ca0  ES:  007b      EDI: 00000286  GS:  00e0
    CS:  0060      EIP: c068124f  ERR: ffffffff  EFLAGS: 00010096
 #6 [ef4dbf18] sysrq_handle_crash at c068124f
 #7 [ef4dbf24] __handle_sysrq at c0681469
 #8 [ef4dbf48] write_sysrq_trigger at c068150a
 #9 [ef4dbf54] proc_reg_write at c0569ec2
#10 [ef4dbf74] vfs_write at c051de4e
#11 [ef4dbf94] sys_write at c051e8cc
#12 [ef4dbfb0] system_call at c0409ad5
    EAX: ffffffda  EBX: 00000001  ECX: b7776000  EDX: 00000002
    DS:  007b      ESI: 00000002  ES:  007b      EDI: b7776000
    SS:  007b      ESP: bfcb2088  EBP: bfcb20b4  GS:  0033
    CS:  0073      EIP: 00edc416  ERR: 00000004  EFLAGS: 00000246
```

Type **bt *<pid>*** to display the backtrace of a specific process or type **help bt** for more information on **bt** usage.

Displaying a process status

- To display the status of processes in the system, use the **ps** command.

```
crash> ps
   PID    PPID  CPU   TASK    ST  %MEM     VSZ    RSS  COMM
>     0      0   0  c09dc560  RU   0.0       0      0  [swapper]
>     0      0   1  f7072030  RU   0.0       0      0  [swapper]
      0      0   2  f70a3a90  RU   0.0       0      0  [swapper]
>     0      0   3  f70ac560  RU   0.0       0      0  [swapper]
      1      0   1  f705ba90  IN   0.0    2828   1424  init
... several lines omitted ...
   5566      1   1  f2592560  IN   0.0   12876    784  auditd
   5567      1   2  ef427560  IN   0.0   12876    784  auditd
   5587   5132   0  f196d030  IN   0.0   11064   3184  sshd
>  5591   5587   2  f196d560  RU   0.0    5084   1648  bash
```

Use **ps *<pid>*** to display the status of a single specific process. Use *help ps* for more information on **ps** usage.

Displaying virtual memory information

- To display basic virtual memory information, type the **vm** command at the interactive prompt.

```
crash> vm
PID: 5591   TASK: f196d560  CPU: 2   COMMAND: "bash"
   MM       PGD      RSS    TOTAL_VM
f19b5900  ef9c6000  1648k    5084k
  VMA       START      END    FLAGS  FILE
f1bb0310    242000    260000 8000875  /lib/ld-2.12.so
f26af0b8    260000    261000 8100871  /lib/ld-2.12.so
efbc275c    261000    262000 8100873  /lib/ld-2.12.so
efbc2a18    268000    3ed000 8000075  /lib/libc-2.12.so
efbc23d8    3ed000    3ee000 8000070  /lib/libc-2.12.so
efbc2888    3ee000    3f0000 8100071  /lib/libc-2.12.so
efbc2cd4    3f0000    3f1000 8100073  /lib/libc-2.12.so
efbc243c    3f1000    3f4000 100073
efbc28ec    3f6000    3f9000 8000075  /lib/libdl-2.12.so
```

```
efbc2568   3f9000   3fa000 8100071  /lib/libdl-2.12.so
efbc2f2c   3fa000   3fb000 8100073  /lib/libdl-2.12.so
f26af888   7e6000   7fc000 8000075  /lib/libtinfo.so.5.7
f26aff2c   7fc000   7ff000 8100073  /lib/libtinfo.so.5.7
efbc211c   d83000   d8f000 8000075  /lib/libnss_files-2.12.so
efbc2504   d8f000   d90000 8100071  /lib/libnss_files-2.12.so
efbc2950   d90000   d91000 8100073  /lib/libnss_files-2.12.so
f26afe00   edc000   edd000 4040075
f1bb0a18  8047000  8118000 8001875  /bin/bash
f1bb01e4  8118000  811d000 8101873  /bin/bash
f1bb0c70  811d000  8122000 100073
f26afae0  9fd9000  9ffa000 100073
... several lines omitted ...
```

Use **vm *<pid>*** to display information on a single specific process, or use **help vm** for more information on **vm** usage.

Displaying open files

- To display information about open files, use the **files** command.

```
crash> files
PID: 5591   TASK: f196d560  CPU: 2   COMMAND: "bash"
ROOT: /    CWD: /root
 FD    FILE     DENTRY    INODE    TYPE  PATH
  0  f734f640  eedc2c6c  eecd6048  CHR   /pts/0
  1  efade5c0  eee14090  f00431d4  REG   /proc/sysrq-trigger
  2  f734f640  eedc2c6c  eecd6048  CHR   /pts/0
 10  f734f640  eedc2c6c  eecd6048  CHR   /pts/0
255  f734f640  eedc2c6c  eecd6048  CHR   /pts/0
```

Use **files *<pid>*** to display files opened by only one selected process, or use **help files** for more information on **files** usage.

7.10.4. Using Kernel Oops Analyzer

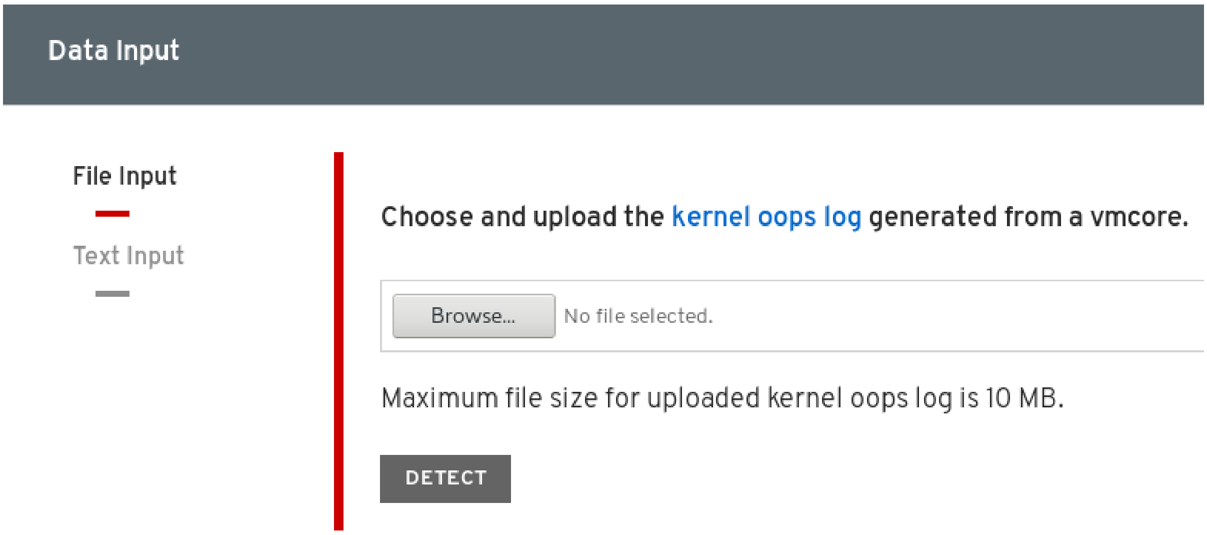
The Kernel Oops Analyzer is a tool that analyzes the crash dump by comparing the oops messages with known issues in the knowledge base.

Prerequisites

- Secure an oops message to feed the Kernel Oops Analyzer by following instructions in *Red Hat Labs*.

Procedure

1. Follow the *Kernel Oops Analyzer* link to access the tool.
2. Browse for the oops message by hitting the **Browse** button.



Choose and upload the kernel oops log generated from a vmcore.

No file selected.

Maximum file size for uploaded kernel oops log is 10 MB.

3. Click the **DETECT** button to compare the oops message based on information from **makedumpfile** against known solutions.

7.11. USING EARLY KDUMP TO CAPTURE BOOT TIME CRASHES

As a system administrator, you can utilize the **early kdump** support of the **kdump** service to capture a vmcore file of the crashing kernel during the early stages of the booting process. This section describes what **early kdump** is, how to configure it, and how to check the status of this mechanism.

7.11.1. What is early kdump

Kernel crashes during the booting phase occur when the **kdump** service is not yet started, and cannot facilitate capturing and saving the contents of the crashed kernel's memory. Therefore, the vital information for troubleshooting is lost.

To address this problem, RHEL 8 introduced the **early kdump** feature as a part of the **kdump** service.

Additional resources

- For more information about **early kdump** and its use, see the **/usr/share/doc/kexec-tools/early-kdump-howto.txt** file and What is early kdump support and how do I configure it? solution.
- For more information about the **kdump** service, see the Section 7.1, "What is kdump".

7.11.2. Enabling early kdump

This section describes how to enable the **early kdump** feature to eliminate the risk of losing information about the early boot kernel crashes.

Prerequisites

- An active Red Hat Enterprise Linux subscription
- A repository containing the **kexec-tools** package for your system CPU architecture
- Fulfilled **kdump** requirements

Procedure

1. Verify that the **kdump** service is enabled and active:

   ```
   # systemctl is-enabled kdump.service && systemctl is-active kdump.service
   enabled
   active
   ```

   If **kdump** is not enabled and running see, Section 7.3.5, “Enabling and disabling the kdump service”.

2. Rebuild the **initramfs** image of the booting kernel with the **early kdump** functionality:

   ```
   dracut -f --add earlykdump
   ```

3. Add the **rd.earlykdump** kernel command line parameter:

   ```
   grubby --update-kernel=/boot/vmlinuz-$(uname -r) --args="rd.earlykdump"
   ```

4. Reboot

5. Optionally, verify that **rd.earlykdump** was successfully added and **early kdump** feature was enabled:

   ```
   # cat /proc/cmdline
   BOOT_IMAGE=(hd0,msdos1)/vmlinuz-4.18.0-187.el8.x86_64 root=/dev/mapper/rhel-root ro
   crashkernel=auto resume=/dev/mapper/rhel-swap rd.lvm.lv=rhel/root rd.lvm.lv=rhel/swap
   rhgb quiet rd.earlykdump

   # journalctl -x | grep early-kdump
   Mar 20 15:44:41 redhat dracut-cmdline[304]: early-kdump is enabled.
   Mar 20 15:44:42 redhat dracut-cmdline[304]: kexec: loaded early-kdump kernel
   ```

Additional resources

- For more information on enabling **early kdump**, see the **/usr/share/doc/kexec-tools/early-kdump-howto.txt** file and What is early kdump support and how do I configure it? solution.

7.12. RELATED INFORMATION

The following section provides further information related to capturing crash information.

- **kdump.conf**(5) – a manual page for the **/etc/kdump.conf** configuration file containing the full documentation of available options.

- **zipl.conf**(5) – a manual page for the **/etc/zipl.conf** configuration file.

- **zipl**(8) – a manual page for the **zipl** boot loader utility for IBM System z.

- **makedumpfile**(8) – a manual page for the **makedumpfile** core collector.

- **kexec**(8) – a manual page for **kexec**.

- **crash**(8) – a manual page for the **crash** utility.

- **/usr/share/doc/kexec-tools/kexec-kdump-howto.txt** – an overview of the **kdump** and **kexec** installation and usage.
- For more information about the **kexec** and **kdump** configuration see the *Red Hat Knowledgebase article*.
- For more information about the supported **kdump** targets see the *Red Hat Knowledgebase article*.

CHAPTER 8. APPLYING PATCHES WITH KERNEL LIVE PATCHING

You can use the Red Hat Enterprise Linux kernel live patching solution to patch a running kernel without rebooting or restarting any processes.

With this solution, system administrators:

- Can immediately apply critical security patches to the kernel.
- Do not have to wait for long-running tasks to complete, for users to log off, or for scheduled downtime.
- Control the system's uptime more and do not sacrifice security or stability.

Note that not every critical or important CVE will be resolved using the kernel live patching solution. Our goal is to reduce the required reboots for security-related patches, not to eliminate them entirely. For more details about the scope of live patching, see the *Customer Portal Solutions article* .

> WARNING
>
> Some incompatibilities exist between kernel live patching and other kernel subcomponents. Read the Section 8.1, "Limitations of kpatch" carefully before using kernel live patching.

8.1. LIMITATIONS OF KPATCH

- The **kpatch** feature is not a general-purpose kernel upgrade mechanism. It is used for applying simple security and bug fix updates when rebooting the system is not immediately possible.
- Do not use the **SystemTap** or **kprobe** tools during or after loading a patch. The patch could fail to take effect until after such probes have been removed.

8.2. SUPPORT FOR THIRD-PARTY LIVE PATCHING

The **kpatch** utility is the only kernel live patching utility supported by Red Hat with the RPM modules provided by Red Hat repositories. Red Hat will not support any live patches which were not provided by Red Hat itself.

If you require support for an issue that arises with a third-party live patch, Red Hat recommends that you open a case with the live patching vendor at the outset of any investigation in which a root cause determination is necessary. This allows the source code to be supplied if the vendor allows, and for their support organization to provide assistance in root cause determination prior to escalating the investigation to Red Hat Support.

For any system running with third-party live patches, Red Hat reserves the right to ask for reproduction with Red Hat shipped and supported software. In the event that this is not possible, we require a similar system and workload be deployed on your test environment without live patches applied, to confirm if the same behavior is observed.

For more information about third-party software support policies, see How does Red Hat Global Support Services handle third-party software, drivers, and/or uncertified hardware/hypervisors or guest operating systems?

8.3. ACCESS TO KERNEL LIVE PATCHES

Kernel live patching capability is implemented as a kernel module (**kmod**) that is delivered as an RPM package.

All customers have access to kernel live patches, which are delivered through the usual channels. However, customers who do not subscribe to an extended support offering will lose access to new patches for the current minor release once the next minor release becomes available. For example, customers with standard subscriptions will only be able to live patch RHEL 8.2 kernel until the RHEL 8.3 kernel is released.

8.4. COMPONENTS OF KERNEL LIVE PATCHING

The components of kernel live patching are as follows:

Kernel patch module

- The delivery mechanism for kernel live patches.
- A kernel module which is built specifically for the kernel being patched.
- The patch module contains the code of the desired fixes for the kernel.
- The patch modules register with the **livepatch** kernel subsystem and provide information about original functions to be replaced, with corresponding pointers to the replacement functions. Kernel patch modules are delivered as RPMs.
- The naming convention is **kpatch_<kernel version>_<kpatch version>_<kpatch release>**. The "kernel version" part of the name has *dots* and *dashes* replaced with *underscores*.

The kpatch utility

A command-line utility for managing patch modules.

The kpatch service

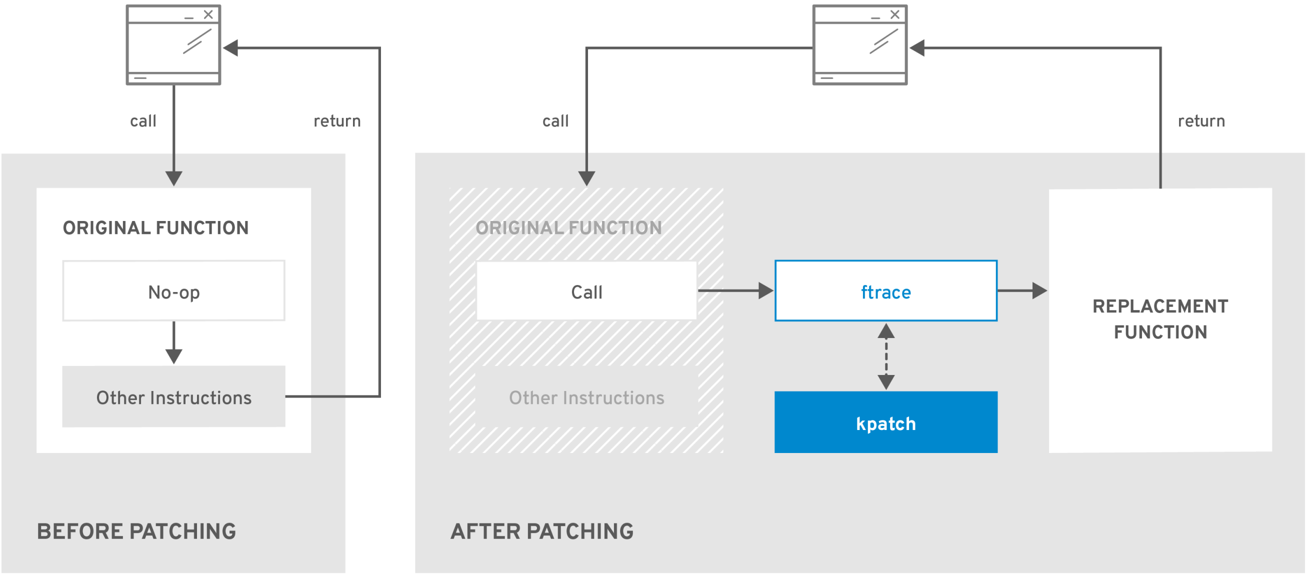
A **systemd** service required by **multiuser.target**. This target loads the kernel patch module at boot time.

8.5. HOW KERNEL LIVE PATCHING WORKS

The **kpatch** kernel patching solution uses the **livepatch** kernel subsystem to redirect old functions to new ones. When a live kernel patch is applied to a system, the following things happen:

1. The kernel patch module is copied to the **/var/lib/kpatch/** directory and registered for re-application to the kernel by **systemd** on next boot.
2. The kpatch module is loaded into the running kernel and the new functions are registered to the **ftrace** mechanism with a pointer to the location in memory of the new code.
3. When the kernel accesses the patched function, it is redirected by the **ftrace** mechanism which bypasses the original functions and redirects the kernel to patched version of the function.

Figure 8.1. How kernel live patching works



RHEL_424549_0119

8.6. ENABLING KERNEL LIVE PATCHING

A kernel patch module is delivered in an RPM package, specific to the version of the kernel being patched. Each RPM package will be cumulatively updated over time.

The following sections describe how to ensure you receive all future cumulative live patching updates for a given kernel.

WARNING

Red Hat does not support any third party live patches applied to a Red Hat supported system.

8.6.1. Subscribing to the live patching stream

This procedure describes installing a particular live patching package. By doing so, you subscribe to the live patching stream for a given kernel and ensure that you receive all future cumulative live patching updates for that kernel.

WARNING

Because live patches are cumulative, you cannot select which individual patches are deployed for a given kernel.

Prerequisites

- Root permissions

Procedure

1. Optionally, check your kernel version:

   ```
   # uname -r
   4.18.0-94.el8.x86_64
   ```

2. Search for a live patching package that corresponds to the version of your kernel:

   ```
   # yum search $(uname -r)
   ```

3. Install the live patching package:

   ```
   # yum install "kpatch-patch = $(uname -r)"
   ```

   The command above installs and applies the latest cumulative live patches for that specific kernel only.

   The live patching package contains a patch module, if the package’s version is 1-1 or higher. In that case the kernel will be automatically patched during the installation of the live patching package.

   The kernel patch module is also installed into the **/var/lib/kpatch/** directory to be loaded by the **systemd** system and service manager during the future reboots.

   > NOTE
   >
   > If there are not yet any live patches available for the given kernel, an empty live patching package will be installed. An empty live patching package will have a *kpatch_version-kpatch_release* of 0-0, for example **kpatch-patch-4_18_0-94-0-0.el8.x86_64.rpm**. The installation of the empty RPM subscribes the system to all future live patches for the given kernel.

4. Optionally, verify that the kernel is patched:

   ```
   # kpatch list
   Loaded patch modules:
   kpatch_4_18_0_94_1_1 [enabled]

   Installed patch modules:
   kpatch_4_18_0_94_1_1 (4.18.0-94.el8.x86_64)
   …
   ```

   The output shows that the kernel patch module has been loaded into the kernel, which is now patched with the latest fixes from the **kpatch-patch-4_18_0-94-1-1.el8.x86_64.rpm** package.

Additional resources

- For more information about the **kpatch** command-line utility, see the **kpatch(1)** manual page.

- Refer to the relevant sections of the *Configuring basic system settings* for further information about installing software packages in Red Hat Enterprise Linux 8.

8.7. UPDATING KERNEL PATCH MODULES

Since kernel patch modules are delivered and applied through RPM packages, updating a cumulative kernel patch module is like updating any other RPM package.

Prerequisites

- The system is subscribed to the live patching stream, as described in Section 8.6.1, "Subscribing to the live patching stream".

Procedure

- Update to a new cumulative version for the current kernel:

  ```
  # yum update "kpatch-patch = $(uname -r)"
  ```

  The command above automatically installs and applies any updates that are available for the currently running kernel. Including any future released cumulative live patches.

- Alternatively, update all installed kernel patch modules:

  ```
  # yum update "kpatch-patch*"
  ```

> **NOTE**
>
> When the system reboots into the same kernel, the kernel is automatically live patched again by the **kpatch.service** systemd service.

Additional resources

- For further information about updating software packages, see the relevant sections of *Configuring basic system settings* in Red Hat Enterprise Linux 8.

8.8. DISABLING KERNEL LIVE PATCHING

In case system administrators encountered some unanticipated negative effects connected with the Red Hat Enterprise Linux kernel live patching solution they have a choice to disable the mechanism. The following sections describe the ways how to disable the live patching solution.

> **IMPORTANT**
>
> Currently, Red Hat does not support reverting live patches without rebooting your system. In case of any issues, contact our support team.

8.8.1. Removing the live patching package

The following procedure describes how to disable the Red Hat Enterprise Linux kernel live patching solution by removing the live patching package.

Prerequisites

- Root permissions
- The live patching package is installed.

Procedure

1. Select the live patching package:

   ```
   # yum list installed | grep kpatch-patch
   kpatch-patch-4_18_0-94.x86_64        1-1.el8        @@commandline
   …
   ```

   The example output above lists live patching packages that you installed.

2. Remove the live patching package:

   ```
   # yum remove kpatch-patch-4_18_0-94.x86_64
   ```

   When a live patching package is removed, the kernel remains patched until the next reboot, but the kernel patch module is removed from disk. After the next reboot, the corresponding kernel will no longer be patched.

3. Reboot your system.

4. Verify that the live patching package was been removed:

   ```
   # yum list installed | grep kpatch-patch
   ```

   The command displays no output if the package has been successfully removed.

5. Optionally, verify that the kernel live patching solution is disabled:

   ```
   # kpatch list
   Loaded patch modules:
   ```

   The example output shows that the kernel is not patched and the live patching solution is not active because there are no patch modules that are currently loaded.

Additional resources

- For more information about the **kpatch** command-line utility, see the **kpatch(1)** manual page.
- For further information about removing software packages in RHEL 8, see relevant sections of *Configuring basic system settings*.

8.8.2. Uninstalling the kernel patch module

The following procedure describes how to prevent the Red Hat Enterprise Linux kernel live patching solution from applying a kernel patch module on subsequent boots.

Prerequisites

- Root permissions

- A live patching package is installed.
- A kernel patch module is installed and loaded.

Procedure

1. Select a kernel patch module:

   ```
   # kpatch list
   Loaded patch modules:
   kpatch_4_18_0_94_1_1 [enabled]

   Installed patch modules:
   kpatch_4_18_0_94_1_1 (4.18.0-94.el8.x86_64)
   …
   ```

2. Uninstall the selected kernel patch module:

   ```
   # kpatch uninstall kpatch_4_18_0_94_1_1
   uninstalling kpatch_4_18_0_94_1_1 (4.18.0-94.el8.x86_64)
   ```

   - Note that the uninstalled kernel patch module is still loaded:

     ```
     # kpatch list
     Loaded patch modules:
     kpatch_4_18_0_94_1_1 [enabled]

     Installed patch modules:
     <NO_RESULT>
     ```

     When the selected module is uninstalled, the kernel remains patched until the next reboot, but the kernel patch module is removed from disk.

3. Reboot your system.

4. Optionally, verify that the kernel patch module has been uninstalled:

   ```
   # kpatch list
   Loaded patch modules:
   …
   ```

   The example output above shows no loaded or installed kernel patch modules, therefore the kernel is not patched and the kernel live patching solution is not active.

Additional resources

- For more information about the **kpatch** command-line utility, refer to the **kpatch(1)** manual page.

8.8.3. Disabling kpatch.service

The following procedure describes how to prevent the Red Hat Enterprise Linux kernel live patching solution from applying all kernel patch modules globally on subsequent boots.

Prerequisites

- Root permissions
- A live patching package is installed.
- A kernel patch module is installed and loaded.

Procedure

1. Verify **kpatch.service** is enabled:

```
# systemctl is-enabled kpatch.service
enabled
```

2. Disable **kpatch.service**:

```
# systemctl disable kpatch.service
Removed /etc/systemd/system/multi-user.target.wants/kpatch.service.
```

   - Note that the applied kernel patch module is still loaded:

```
# kpatch list
Loaded patch modules:
kpatch_4_18_0_94_1_1 [enabled]

Installed patch modules:
kpatch_4_18_0_94_1_1 (4.18.0-94.el8.x86_64)
```

3. Reboot your system.

4. Optionally, verify the status of **kpatch.service**:

```
# systemctl status kpatch.service
● kpatch.service - "Apply kpatch kernel patches"
  Loaded: loaded (/usr/lib/systemd/system/kpatch.service; disabled; vendor preset: disabled)
  Active: inactive (dead)
```

   The example output testifies that **kpatch.service** has been disabled and is not running. Thereby, the kernel live patching solution is not active.

5. Verify that the kernel patch module has been unloaded:

```
# kpatch list
Loaded patch modules:
<NO_RESULT>

Installed patch modules:
kpatch_4_18_0_94_1_1 (4.18.0-94.el8.x86_64)
```

   The example output above shows that a kernel patch module is still installed but the kernel is not patched.

Additional resources

- For more information about the **kpatch** command-line utility, see the **kpatch(1)** manual page.

- For more information about the **systemd** system and service manager, unit configuration files, their locations, as well as a complete list of **systemd** unit types, see the relevant sections in *Configuring basic system settings*.

CHAPTER 9. SETTING LIMITS FOR APPLICATIONS

As a system administrator, use the control groups kernel functionality to set limits, prioritize or isolate the hardware resources of processes so that applications on your system are stable and do not run out of memory.

9.1. UNDERSTANDING CONTROL GROUPS

Control groups is a Linux kernel feature that enables you to organize processes into hierarchically ordered groups - **cgroups**. The hierarchy (control groups tree) is defined by providing structure to **cgroups** virtual file system, mounted by default on the **/sys/fs/cgroup/** directory. It is done manually by creating and removing sub-directories in **/sys/fs/cgroup/**. Alternatively, by using the **systemd** system and service manager.

The resource controllers (a kernel component) then modify the behavior of processes in **cgroups** by limiting, prioritizing or allocating system resources, (such as CPU time, memory, network bandwidth, or various combinations) of those processes.

The added value of **cgroups** is process aggregation which enables division of hardware resources among applications and users. Thereby an increase in overall efficiency, stability and security of users' environment can be achieved.

Control groups version 1

Control groups version 1 (**cgroups-v1**) provide a per-resource controller hierarchy. It means that each resource, such as CPU, memory, I/O, and so on, has its own control group hierarchy. It is possible to combine different control group hierarchies in a way that one controller can coordinate with another one in managing their respective resources. However, the two controllers may belong to different process hierarchies, which does not permit their proper coordination.
The **cgroups-v1** controllers were developed across a large time span and as a result, the behavior and naming of their control files is not uniform.

Control groups version 2

The problems with controller coordination, which stemmed from hierarchy flexibility, led to the development of *control groups version 2*.
Control groups version 2 (**cgroups-v2**) provides a single control group hierarchy against which all resource controllers are mounted.

The control file behavior and naming is consistent among different controllers.

> **WARNING**
>
> RHEL 8 provides **cgroups-v2** as a technology preview with a limited number of resource controllers. For more information about the relevant resource controllers, see the *cgroups-v2 release note*.

This sub-section was based on a Devconf.cz 2019 presentation.[1]

Additional resources

- For more information about resource controllers, see Section 9.2, "What kernel resource controllers are" section and **cgroups(7)** manual pages.
- For more information about **cgroups** hierarchies and **cgroups** versions, refer to **cgroups(7)** manual pages.
- For more information about **systemd** and **cgroups** cooperation, see Role of systemd in control groups section.

9.2. WHAT KERNEL RESOURCE CONTROLLERS ARE

The functionality of control groups is enabled by kernel resource controllers. RHEL 8 supports various controllers for *control groups version 1* (**cgroups-v1**) and *control groups version 2* (**cgroups-v2**).

A resource controller, also called a control group subsystem, is a kernel subsystem that represents a single resource, such as CPU time, memory, network bandwidth or disk I/O. The Linux kernel provides a range of resource controllers that are mounted automatically by the **systemd** system and service manager. Find a list of currently mounted resource controllers in the **/proc/cgroups** file.

The following controllers are available for **cgroups-v1**:

- **blkio** - can set limits on input/output access to and from block devices.
- **cpu** - can adjust the parameters of the Completely Fair Scheduler (CFS) scheduler for control group's tasks. It is mounted together with the **cpuacct** controller on the same mount.
- **cpuacct** - creates automatic reports on CPU resources used by tasks in a control group. It is mounted together with the **cpu** controller on the same mount.
- **cpuset** - can be used to restrict control group tasks to run only on a specified subset of CPUs and to direct the tasks to use memory only on specified memory nodes.
- **devices** - can control access to devices for tasks in a control group.
- **freezer** - can be used to suspend or resume tasks in a control group.
- **memory** - can be used to set limits on memory use by tasks in a control group and generates automatic reports on memory resources used by those tasks.
- **net_cls** - tags network packets with a class identifier (**classid**) that enables the Linux traffic controller (the **tc** command) to identify packets that originate from a particular control group task. A subsystem of **net_cls**, the **net_filter** (iptables), can also use this tag to perform actions on such packets. The **net_filter** tags network sockets with a firewall identifier (**fwid**) that allows the Linux firewall (through **iptables** command) to identify packets originating from a particular control group task.
- **net_prio** - sets the priority of network traffic.
- **pids** - can set limits for a number of processes and their children in a control group.
- **perf_event** - can group tasks for monitoring by the **perf** performance monitoring and reporting utility.
- **rdma** - can set limits on Remote Direct Memory Access/InfiniBand specific resources in a control group.

- **hugetlb** - can be used to limit the usage of large size virtual memory pages by tasks in a control group.

The following controllers are available for **cgroups-v2**:

- **io** - A follow-up to **blkio** of **cgroups-v1**.
- **memory** - A follow-up to **memory** of **cgroups-v1**.
- **pids** - Same as **pids** in **cgroups-v1**.
- **rdma** - Same as **rdma** in **cgroups-v1**.
- **cpu** - A follow-up to **cpu** and **cpuacct** of **cgroups-v1**.
- **cpuset** - Supports only the core functionality (**cpus{,.effective}**, **mems{,.effective}**) with a new partition feature.
- **perf_event** - Support is inherent, no explicit control file. You can specify a **v2 cgroup** as a parameter to the **perf** command that will profile all the tasks within that **cgroup**.

> IMPORTANT
>
> A resource controller can be used either in a **cgroups-v1** hierarchy or a **cgroups-v2** hierarchy, not simultaneously in both.

Additional resources

- For more information about resource controllers in general, refer to the **cgroups(7)** manual page.
- For detailed descriptions of specific resource controllers, see the documentation in the **/usr/share/doc/kernel-doc-<kernel_version>/Documentation/cgroups-v1/** directory.
- For more information about **cgroups-v2**, refer to the **cgroups(7)** manual page.

9.3. USING CONTROL GROUPS THROUGH A VIRTUAL FILE SYSTEM

You can use *control groups* (**cgroups**) to set limits, prioritize, or control access to hardware resources for groups of processes. This allows you to granularly control resource usage of applications to utilize them more efficiently. The following sections provide an overview of tasks related to management of **cgroups** for both version 1 and version 2 using a virtual file system.

9.3.1. Setting CPU limits to applications using cgroups-v1

Sometimes an application consumes a lot of CPU time, which may negatively impact the overall health of your environment. Use the **/sys/fs/** virtual file system to configure CPU limits to an application using *control groups version 1* (**cgroups-v1**).

Prerequisites

- An application whose CPU consumption you want to restrict.
- Verify that the **cgroups-v1** controllers were mounted:

```
# mount -l | grep cgroup
tmpfs on /sys/fs/cgroup type tmpfs (ro,nosuid,nodev,noexec,seclabel,mode=755)
cgroup on /sys/fs/cgroup/systemd type cgroup
(rw,nosuid,nodev,noexec,relatime,seclabel,xattr,release_agent=/usr/lib/systemd/systemd-
cgroups-agent,name=systemd)
cgroup on /sys/fs/cgroup/cpu,cpuacct type cgroup
(rw,nosuid,nodev,noexec,relatime,seclabel,cpu,cpuacct)
cgroup on /sys/fs/cgroup/perf_event type cgroup
(rw,nosuid,nodev,noexec,relatime,seclabel,perf_event)
cgroup on /sys/fs/cgroup/pids type cgroup (rw,nosuid,nodev,noexec,relatime,seclabel,pids)
...
```

Procedure

1. Identify the process ID (PID) of the application you want to restrict in CPU consumption:

```
# top
top - 11:34:09 up 11 min,  1 user,  load average: 0.51, 0.27, 0.22
Tasks: 267 total,   3 running, 264 sleeping,   0 stopped,   0 zombie
%Cpu(s): 49.0 us,  3.3 sy,  0.0 ni, 47.5 id,  0.0 wa,  0.2 hi,  0.0 si,  0.0 st
MiB Mem :   1826.8 total,    303.4 free,   1046.8 used,    476.5 buff/cache
MiB Swap:   1536.0 total,   1396.0 free,    140.0 used.    616.4 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 6955 root      20   0  228440   1752   1472 R  99.3   0.1   0:32.71 sha1sum
 5760 jdoe      20   0 3603868 205188  64196 S   3.7  11.0   0:17.19 gnome-shell
 6448 jdoe      20   0  743648  30640  19488 S   0.7   1.6   0:02.73 gnome-terminal-
    1 root      20   0  245300   6568   4116 S   0.3   0.4   0:01.87 systemd
  505 root      20   0       0      0      0 I   0.3   0.0   0:00.75 kworker/u4:4-events_unbound
...
```

The example output of the **top** program reveals that **PID 6955** (illustrative application **sha1sum**) consumes a lot of CPU resources.

2. Create a sub-directory in the **cpu** resource controller directory:

```
# mkdir /sys/fs/cgroup/cpu/Example/
```

The directory above represents a control group, where you can place specific processes and apply certain CPU limits to the processes. At the same time, some **cgroups-v1** interface files and **cpu** controller-specific files will be created in the directory.

3. Optionally, inspect the newly created control group:

```
# ll /sys/fs/cgroup/cpu/Example/
-rw-r—r--. 1 root root 0 Mar 11 11:42 cgroup.clone_children
-rw-r—r--. 1 root root 0 Mar 11 11:42 cgroup.procs
-r—r—r--. 1 root root 0 Mar 11 11:42 cpuacct.stat
-rw-r—r--. 1 root root 0 Mar 11 11:42 cpuacct.usage
-r—r—r--. 1 root root 0 Mar 11 11:42 cpuacct.usage_all
-r—r—r--. 1 root root 0 Mar 11 11:42 cpuacct.usage_percpu
-r—r—r--. 1 root root 0 Mar 11 11:42 cpuacct.usage_percpu_sys
-r—r—r--. 1 root root 0 Mar 11 11:42 cpuacct.usage_percpu_user
-r—r—r--. 1 root root 0 Mar 11 11:42 cpuacct.usage_sys
-r—r—r--. 1 root root 0 Mar 11 11:42 cpuacct.usage_user
```

```
-rw-r—r--. 1 root root 0 Mar 11 11:42 cpu.cfs_period_us
-rw-r—r--. 1 root root 0 Mar 11 11:42 cpu.cfs_quota_us
-rw-r—r--. 1 root root 0 Mar 11 11:42 cpu.rt_period_us
-rw-r—r--. 1 root root 0 Mar 11 11:42 cpu.rt_runtime_us
-rw-r—r--. 1 root root 0 Mar 11 11:42 cpu.shares
-r—r—r--. 1 root root 0 Mar 11 11:42 cpu.stat
-rw-r—r--. 1 root root 0 Mar 11 11:42 notify_on_release
-rw-r—r--. 1 root root 0 Mar 11 11:42 tasks
```

The example output shows files, such as **cpuacct.usage**, **cpu.cfs._period_us**, that represent specific configurations and/or limits, which can be set for processes in the **Example** control group. Notice that the respective file names are prefixed with the name of the control group controller to which they belong.

By default, the newly created control group inherits access to the system's entire CPU resources without a limit.

4. Configure CPU limits for the control group:

```
# echo "1000000" > /sys/fs/cgroup/cpu/Example/cpu.cfs_period_us
# echo "200000" > /sys/fs/cgroup/cpu/Example/cpu.cfs_quota_us
```

The **cpu.cfs_period_us** file represents a period of time in microseconds (µs, represented here as "us") for how frequently a control group's access to CPU resources should be reallocated. The upper limit is 1 second and the lower limit is 1000 microseconds.

The **cpu.cfs_quota_us** file represents the total amount of time in microseconds for which all processes collectively in a control group can run during one period (as defined by **cpu.cfs_period_us**). As soon as processes in a control group, during a single period, use up all the time specified by the quota, they are throttled for the remainder of the period and not allowed to run until the next period. The lower limit is 1000 microseconds.

The example commands above set the CPU time limits so that all processes collectively in the **Example** control group will be able to run only for 0.2 seconds (defined by **cpu.cfs_quota_us**) out of every 1 second (defined by **cpu.cfs_period_us**).

5. Optionally, verify the limits:

```
# cat /sys/fs/cgroup/cpu/Example/cpu.cfs_period_us
/sys/fs/cgroup/cpu/Example/cpu.cfs_quota_us
1000000
200000
```

6. Add the application's PID to the **Example** control group:

```
# echo "6955" > /sys/fs/cgroup/cpu/Example/cgroup.procs

or

# echo "6955" > /sys/fs/cgroup/cpu/Example/tasks
```

The previous command ensures that a desired application becomes a member of the **Example** control group and hence does not exceed the CPU limits configured for the **Example** control group. The PID should represent an existing process in the system. The **PID 6955** here was assigned to process **sha1sum /dev/zero &**, used to illustrate the use-case of the **cpu** controller.

7. Verify that the application runs in the specified control group:

```
# cat /proc/6955/cgroup
12:cpuset:/
11:hugetlb:/
10:net_cls,net_prio:/
9:memory:/user.slice/user-1000.slice/user@1000.service
8:devices:/user.slice
7:blkio:/
6:freezer:/
5:rdma:/
4:pids:/user.slice/user-1000.slice/user@1000.service
3:perf_event:/
2:cpu,cpuacct:/Example
1:name=systemd:/user.slice/user-1000.slice/user@1000.service/gnome-terminal-
server.service
```

The example output above shows that the process of the desired application runs in the **Example** control group, which applies CPU limits to the application's process.

8. Identify the current CPU consumption of your throttled application:

```
# top
top - 12:28:42 up  1:06,  1 user,  load average: 1.02, 1.02, 1.00
Tasks: 266 total,   6 running, 260 sleeping,   0 stopped,   0 zombie
%Cpu(s): 11.0 us,  1.2 sy,  0.0 ni, 87.5 id,  0.0 wa,  0.2 hi,  0.0 si,  0.2 st
MiB Mem :   1826.8 total,    287.1 free,   1054.4 used,    485.3 buff/cache
MiB Swap:   1536.0 total,   1396.7 free,    139.2 used.    608.3 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 6955 root      20   0  228440   1752   1472 R  20.6   0.1  47:11.43 sha1sum
 5760 jdoe      20   0 3604956 208832  65316 R   2.3  11.2   0:43.50 gnome-shell
 6448 jdoe      20   0  743836  31736  19488 S   0.7   1.7   0:08.25 gnome-terminal-
  505 root      20   0       0      0      0 I   0.3   0.0   0:03.39 kworker/u4:4-events_unbound
 4217 root      20   0   74192   1612   1320 S   0.3   0.1   0:01.19 spice-vdagentd
...
```

Notice that the CPU consumption of the **PID 6955** has decreased from 99% to 20%.

Additional resources

- For information about the control groups concept, see Section 9.1, "Understanding control groups".
- For more information about resource controllers, see the Section 9.2, "What kernel resource controllers are" section and the **cgroups(7)** manual page.
- For more information about the /**sys**/**fs**/ virtual filesystem, see the **sysfs(5)** manual page.

9.3.2. Setting CPU limits to applications using cgroups-v2

Sometimes an application uses a lot of CPU time, which may negatively impact the overall health of your environment. Use *control groups version 2* (**cgroups-v2**) to configure CPU limits to the application, and restrict its consumption.

Prerequisites

- An application whose CPU consumption you want to restrict.
- Section 9.1, "Understanding control groups"

Procedure

1. Prevent **cgroups-v1** from automatically mounting during the system boot:

   ```
   # grubby --update-kernel=/boot/vmlinuz-$(uname -r) --args="cgroup_no_v1=all"
   ```

   The command adds a kernel command-line parameter to the current boot entry. The **cgroup_no_v1=all** parameter prevents **cgroups-v1** from being automatically mounted.

   Alternatively, use the **systemd.unified_cgroup_hierarchy=1** kernel command-line parameter to mount **cgroups-v2** during the system boot by default.

   > NOTE
   >
   > RHEL 8 supports both **cgroups-v1** and **cgroups-v2**. However, **cgroups-v1** is enabled and mounted by default during the booting process.

2. Reboot the system for the changes to take effect.

3. Optionally, verify the **cgroups-v1** functionality has been disabled:

   ```
   # mount -l | grep cgroup
   tmpfs on /sys/fs/cgroup type tmpfs (ro,nosuid,nodev,noexec,seclabel,mode=755)
   cgroup on /sys/fs/cgroup/systemd type cgroup
   (rw,nosuid,nodev,noexec,relatime,seclabel,xattr,release_agent=/usr/lib/systemd/systemd-
   cgroups-agent,name=systemd)
   ```

   If **cgroups-v1** have been successfully disabled, the output does not show any "type cgroup" references, except for those which belong to **systemd**.

4. Mount **cgroups-v2** anywhere in the filesystem:

   ```
   # mount -t cgroup2 none <MOUNT_POINT>
   ```

5. Optionally, verify the **cgroups-v2** functionality has been mounted:

   ```
   # mount -l | grep cgroup2
   none on /cgroups-v2 type cgroup2 (rw,relatime,seclabel)
   ```

   The example output shows that **cgroups-v2** has been mounted to the **/cgroups-v2/** directory.

6. Optionally, inspect the contents of the **/cgroups-v2/** directory:

   ```
   # ll /cgroups-v2/
   -r—r—r--. 1 root root 0 Mar 13 11:57 cgroup.controllers
   -rw-r—r--. 1 root root 0 Mar 13 11:57 cgroup.max.depth
   -rw-r—r--. 1 root root 0 Mar 13 11:57 cgroup.max.descendants
   -rw-r—r--. 1 root root 0 Mar 13 11:57 cgroup.procs
   -r—r—r--. 1 root root 0 Mar 13 11:57 cgroup.stat
   ```

```
-rw-r—r--. 1 root root 0 Mar 13 11:58 cgroup.subtree_control
-rw-r—r--. 1 root root 0 Mar 13 11:57 cgroup.threads
-rw-r—r--. 1 root root 0 Mar 13 11:57 cpu.pressure
-r—r—r--. 1 root root 0 Mar 13 11:57 cpuset.cpus.effective
-r—r—r--. 1 root root 0 Mar 13 11:57 cpuset.mems.effective
-rw-r—r--. 1 root root 0 Mar 13 11:57 io.pressure
-rw-r—r--. 1 root root 0 Mar 13 11:57 memory.pressure
```

The **/cgroups-v2/** directory, also called the root control group, contains some interface files (starting with **cgroup**) and some controller-specific files such as **cpuset.cpus.effective**.

7. Identify the process IDs (PIDs) of applications you want to restrict in CPU consumption:

```
# top
top - 15:39:52 up  3:45,  1 user,  load average: 0.79, 0.20, 0.07
Tasks: 265 total,   3 running, 262 sleeping,   0 stopped,   0 zombie
%Cpu(s): 74.3 us,  6.1 sy,  0.0 ni, 19.4 id,  0.0 wa,  0.2 hi,  0.0 si,  0.0 st
MiB Mem :   1826.8 total,    243.8 free,   1102.1 used,    480.9 buff/cache
MiB Swap:   1536.0 total,   1526.2 free,      9.8 used.    565.6 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 5473 root      20   0  228440   1740   1456 R  99.7   0.1   0:12.11 sha1sum
 5439 root      20   0  222616   3420   3052 R  60.5   0.2   0:27.08 cpu_load_generator
 2170 jdoe      20   0 3600716 209960  67548 S   0.3  11.2   1:18.50 gnome-shell
 3051 root      20   0  274424   3976   3092 R   0.3   0.2   1:01.25 top
    1 root      20   0  245448  10256   5448 S   0.0   0.5   0:02.52 systemd
...
```

The example output of the **top** program reveals that **PID 5473** and **5439** (illustrative application **sha1sum** and **cpu_load_generator**) consume a lot of resources, namely CPU. Both are example applications used to demonstrate managing the **cgroups-v2** functionality.

8. Enable CPU-related controllers:

```
# echo "+cpu" > /cgroups-v2/cgroup.subtree_control
# echo "+cpuset" > /cgroups-v2/cgroup.subtree_control
```

The previous commands enable the **cpu** and **cpuset** controllers for the immediate sub-control groups of the **/cgroups-v2/** root control group.

9. Create a sub-directory in the previously created **/cgroups-v2/** directory:

```
# mkdir /cgroups-v2/Example/
```

The **/cgroups-v2/Example/** directory represents a sub-control group, where you can place specific processes and apply various CPU limits to the processes. Also, the previous step enabled the **cpu** and **cpuset** controllers for this sub-control group.

At the time of creation of **/cgroups-v2/Example/**, some **cgroups-v2** interface files and **cpu** and **cpuset** controller-specific files will be created in the directory.

10. Optionally, inspect the newly created control group:

```
# ll /cgroups-v2/Example/
-r—r—r--. 1 root root 0 Mar 13 14:48 cgroup.controllers
```

```
-r—r—r--. 1 root root 0 Mar 13 14:48 cgroup.events
-rw-r—r--. 1 root root 0 Mar 13 14:48 cgroup.freeze
-rw-r—r--. 1 root root 0 Mar 13 14:48 cgroup.max.depth
-rw-r—r--. 1 root root 0 Mar 13 14:48 cgroup.max.descendants
-rw-r—r--. 1 root root 0 Mar 13 14:48 cgroup.procs
-r—r—r--. 1 root root 0 Mar 13 14:48 cgroup.stat
-rw-r—r--. 1 root root 0 Mar 13 14:48 cgroup.subtree_control
-rw-r—r--. 1 root root 0 Mar 13 14:48 cgroup.threads
-rw-r—r--. 1 root root 0 Mar 13 14:48 cgroup.type
-rw-r—r--. 1 root root 0 Mar 13 14:48 cpu.max
-rw-r—r--. 1 root root 0 Mar 13 14:48 cpu.pressure
-rw-r—r--. 1 root root 0 Mar 13 14:48 cpuset.cpus
-r—r—r--. 1 root root 0 Mar 13 14:48 cpuset.cpus.effective
-rw-r—r--. 1 root root 0 Mar 13 14:48 cpuset.cpus.partition
-rw-r—r--. 1 root root 0 Mar 13 14:48 cpuset.mems
-r—r—r--. 1 root root 0 Mar 13 14:48 cpuset.mems.effective
-r—r—r--. 1 root root 0 Mar 13 14:48 cpu.stat
-rw-r—r--. 1 root root 0 Mar 13 14:48 cpu.weight
-rw-r—r--. 1 root root 0 Mar 13 14:48 cpu.weight.nice
-rw-r—r--. 1 root root 0 Mar 13 14:48 io.pressure
-rw-r—r--. 1 root root 0 Mar 13 14:48 memory.pressure
```

The example output shows files such as **cpuset.cpus** and **cpu.max**. The files are specific to the **cpuset** and **cpu** controllers that you enabled for the root's (**/cgroups-v2/**) direct child control groups using the **/cgroups-v2/cgroup.subtree_control** file. Also, there are general **cgroup** control interface files such as **cgroup.procs** or **cgroup.controllers**, which are common to all control groups, regardless of enabled controllers.

By default, the newly created sub-control group inherited access to the system's entire CPU resources without a limit.

11. Ensure the processes that you want to limit compete for CPU time on the same CPU:

```
# echo "1" > /cgroups-v2/Example/cpuset.cpus
```

The previous command secures processes that you placed in the **Example** sub-control group, compete on the same CPU. This setting is important for the **cpu** controller to activate.

IMPORTANT

The **cpu** controller is only activated if the relevant sub-control group has at least 2 processes, which compete for time on a single CPU.

12. Configure CPU limits of the control group:

```
# echo "200000 1000000" > /cgroups-v2/Example/cpu.max
```

The first value is the allowed time quota in microseconds for which all processes collectively in a sub-control group can run during one period (specified by the second value). During a single period, when processes in a control group collectively exhaust all the time specified by this quota, they are throttled for the remainder of the period and not allowed to run until the next period.

The example command sets the CPU time limits so that all processes collectively in the **Example** sub-control group are able to run on the CPU only for 0.2 seconds out of every 1 second.

13. Optionally, verify the limits:

```
# cat /cgroups-v2/Example/cpu.max
200000 1000000
```

14. Add the applications' PIDs to the **Example** sub-control group:

```
# echo "5473" > /cgroups-v2/Example/cgroup.procs
# echo "5439" > /cgroups-v2/Example/cgroup.procs
```

The example commands ensure that desired applications become members of the **Example** sub-control group and hence do not exceed the CPU limits configured for the **Example** sub-control group.

15. Verify that the applications run in the specified control group:

```
# cat /proc/5473/cgroup /proc/5439/cgroup
1:name=systemd:/user.slice/user-1000.slice/user@1000.service/gnome-terminal-
server.service
0::/Example
1:name=systemd:/user.slice/user-1000.slice/user@1000.service/gnome-terminal-
server.service
0::/Example
```

The example output above shows that the processes of the desired applications run in the **Example** sub-control group.

16. Inspect the current CPU consumption of your throttled applications:

```
# top
top - 15:56:27 up  4:02,  1 user,  load average: 0.03, 0.41, 0.55
Tasks: 265 total,   4 running, 261 sleeping,   0 stopped,   0 zombie
%Cpu(s):  9.6 us,  0.8 sy,  0.0 ni, 89.4 id,  0.0 wa,  0.2 hi,  0.0 si,  0.0 st
MiB Mem :   1826.8 total,    243.4 free,   1102.1 used,    481.3 buff/cache
MiB Swap:   1536.0 total,   1526.2 free,      9.8 used.    565.5 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 5439 root      20   0  222616   3420   3052 R  10.0   0.2   6:15.83 cpu_load_generator
 5473 root      20   0  228440   1740   1456 R  10.0   0.1   9:20.65 sha1sum
 2753 jdoe      20   0  743928  35328  20608 S   0.7   1.9   0:20.36 gnome-terminal-
 2170 jdoe      20   0 3599688 208820  67552 S   0.3  11.2   1:33.06 gnome-shell
 5934 root      20   0  274428   5064   4176 R   0.3   0.3   0:00.04 top
 ...
```

Notice that the CPU consumption for the **PID 5439** and **PID 5473** has decreased to 10%. The **Example** sub-control group limits its processes to 20% of the CPU time collectively. Since there are 2 processes in the control group, each can utilize 10% of the CPU time.

Additional resources

For information about the control groups, see Section [illegible] “Understanding control

- For information about the control groups concept, see Section 9.1, “Understanding control groups”.

- For more information about resource controllers, see the Section 9.2, “What kernel resource controllers are” section and the **cgroups(7)** manual page.

- For more information about the /**sys**/**fs**/ virtual filesystem, see the **sysfs(5)** manual page.

9.4. ROLE OF SYSTEMD IN CONTROL GROUPS VERSION 1

Red Hat Enterprise Linux 8 moves the resource management settings from the process level to the application level by binding the system of **cgroup** hierarchies with the **systemd** unit tree. Therefore, you can manage the system resources with the **systemctl** command, or by modifying the **systemd** unit files.

By default, the **systemd** system and service manager makes use of the **slice**, the **scope** and the **service** units to organize and structure processes in the control groups. The **systemctl** command enables you to further modify this structure by creating custom **slices**. Also, **systemd** automatically mounts hierarchies for important kernel resource controllers in the /**sys**/**fs**/**cgroup**/ directory.

Three **systemd** unit types are used for resource control:

- **Service** - A process or a group of processes, which **systemd** started according to a unit configuration file. Services encapsulate the specified processes so that they can be started and stopped as one set. Services are named in the following way:

  ```
  <name>.service
  ```

- **Scope** - A group of externally created processes. Scopes encapsulate processes that are started and stopped by the arbitrary processes through the **fork()** function and then registered by **systemd** at runtime. For example, user sessions, containers, and virtual machines are treated as scopes. Scopes are named as follows:

  ```
  <name>.scope
  ```

- **Slice** - A group of hierarchically organized units. Slices organize a hierarchy in which scopes and services are placed. The actual processes are contained in scopes or in services. Every name of a slice unit corresponds to the path to a location in the hierarchy. The dash ("-") character acts as a separator of the path components to a slice from the **-.slice** root slice. In the following example:

  ```
  <parent-name>.slice
  ```

  parent-name.slice is a sub-slice of **parent.slice**, which is a sub-slice of the **-.slice** root slice. **parent-name.slice** can have its own sub-slice named **parent-name-name2.slice**, and so on.

The **service**, the **scope**, and the **slice** units directly map to objects in the control group hierarchy. When these units are activated, they map directly to control group paths built from the unit names.

The following is an abbreviated example of a control group hierarchy:

```
Control group /:
-.slice
├─user.slice
│ ├─user-42.slice
│ │ ├─session-c1.scope
```

```
│ │ │ ├─ 967 gdm-session-worker [pam/gdm-launch-environment]
│ │ │ ├─1035 /usr/libexec/gdm-x-session gnome-session --autostart
/usr/share/gdm/greeter/autostart
│ │ │ ├─1054 /usr/libexec/Xorg vt1 -displayfd 3 -auth /run/user/42/gdm/Xauthority -background none
-noreset -keeptty -verbose 3
│ │ │ ├─1212 /usr/libexec/gnome-session-binary --autostart /usr/share/gdm/greeter/autostart
│ │ │ ├─1369 /usr/bin/gnome-shell
│ │ │ ├─1732 ibus-daemon --xim --panel disable
│ │ │ ├─1752 /usr/libexec/ibus-dconf
│ │ │ ├─1762 /usr/libexec/ibus-x11 --kill-daemon
│ │ │ ├─1912 /usr/libexec/gsd-xsettings
│ │ │ ├─1917 /usr/libexec/gsd-a11y-settings
│ │ │ ├─1920 /usr/libexec/gsd-clipboard
…
├─init.scope
│ └─1 /usr/lib/systemd/systemd --switched-root --system --deserialize 18
└─system.slice
  ├─rngd.service
  │ └─800 /sbin/rngd -f
  ├─systemd-udevd.service
  │ └─659 /usr/lib/systemd/systemd-udevd
  ├─chronyd.service
  │ └─823 /usr/sbin/chronyd
  ├─auditd.service
  │ ├─761 /sbin/auditd
  │ └─763 /usr/sbin/sedispatch
  ├─accounts-daemon.service
  │ └─876 /usr/libexec/accounts-daemon
  ├─example.service
  │ ├─ 929 /bin/bash /home/jdoe/example.sh
  │ └─4902 sleep 1
  …
```

The example above shows that services and scopes contain processes and are placed in slices that do not contain processes of their own.

Additional resources

- For more information about **systemd**, unit files, and a complete list of **systemd** unit types, see the relevant sections in *Configuring basic system settings*.
- For more information about resource controllers, see the What are kernel resource controllers section and the **systemd.resource-control(5)**, **cgroups(7)** manual pages.
- For more information about **fork()**, see the **fork(2)** manual pages.

9.5. USING CONTROL GROUPS VERSION 1 WITH SYSTEMD

The following sections provide an overview of tasks related to creation, modification and removal of the control groups (**cgroups**). The utilities provided by the **systemd** system and service manager are the preferred way of the **cgroups** management and will be supported in the future.

9.5.1. Creating control groups version 1 with systemd

You can use the **systemd** system and service manager to create transient and persistent control groups (**cgroups**) to set limits, prioritize, or control access to hardware resources for groups of processes.

9.5.1.1. Creating transient control groups

The transient **cgroups** set limits on resources consumed by a unit (service or scope) during its runtime.

Procedure

- To create a transient control group, use the **systemd-run** command in the following format:

  ```
  # systemd-run --unit=<name> --slice=<name>.slice <command>
  ```

  This command creates and starts a transient service or a scope unit and runs a custom command in such a unit.

  - The **--unit=<name>** option gives a name to the unit. If **--unit** is not specified, the name is generated automatically.
  - The **--slice=<*name*>.slice** option makes your service or scope unit a member of a specified slice. Replace **<*name*>.slice** with the name of an existing slice (as shown in the output of **systemctl -t slice**), or create a new slice by passing a unique name. By default, services and scopes are created as members of the **system.slice**.
  - Replace **<*command*>** with the command you wish to execute in the service or the scope unit.
    The following message is displayed to confirm that you created and started the service or the scope successfully:

    ```
    # Running as unit <name>.service
    ```

- Optionally, keep the unit running after its processes finished to collect run-time information:

  ```
  # systemd-run --unit=<name> --slice=<name>.slice --remain-after-exit <command>
  ```

  The command creates and starts a transient service unit and runs a custom command in such a unit. The **--remain-after-exit** option ensures that the service keeps running after its processes have finished.

Additional resources

- For more information about the concept of control groups, see Section 9.1, “Understanding control groups”.
- For more information about the role of **systemd** in control groups, see Section 9.4, “Role of systemd in control groups version 1”.
- For more information about **systemd**, unit configuration files and their locations, and a complete list of **systemd** unit types, see the relevant sections in *Configuring basic system settings*.
- For a detailed description of **systemd-run** including further options and examples, see the **systemd-run(1)** manual pages.

9.5.1.2. Creating persistent control groups

To assign a persistent control group to a service, it is necessary to edit its unit configuration file. The configuration is preserved after the system reboot, so it can be used to manage services that are started automatically.

Procedure

- To create a persistent control group, execute:

  ```
  # systemctl enable <name>.service
  ```

  The command above automatically creates a unit configuration file into the **/usr/lib/systemd/system/** directory and by default, it assigns **<*name*>.service** to the **system.slice** unit.

Additional resources

- For more information about the concept of control groups, see Section 9.1, "Understanding control groups".
- For more information about the role of **systemd** in control groups, see Section 9.4, "Role of systemd in control groups version 1".
- For more information about **systemd**, unit configuration files and their locations, and a complete list of **systemd** unit types, see the relevant sections in *Configuring basic system settings*.
- For a detailed description of **systemd-run** including further options and examples, see the **systemd-run(1)** manual pages.

9.5.2. Modifying control groups version 1 with systemd

Each persistent unit is supervised by the **systemd** system and service manager, and has a unit configuration file in the **/usr/lib/systemd/system/** directory. To change the resource control settings of the persistent units, modify its unit configuration file either manually in a text editor or from the command-line interface.

9.5.2.1. Configuring memory resource control settings on the command-line

Executing commands in the command-line interface is one of the ways how to set limits, prioritize, or control access to hardware resources for groups of processes.

Procedure

- To limit the memory usage of a service, run the following:

  ```
  # systemctl set-property example.service MemoryLimit=1500K
  ```

  The command instantly assigns the memory limit of 1,500 kilobytes to processes executed in a control group the **example.service** service belongs to. The **MemoryLimit** parameter, in this configuration variant, is defined in the **/etc/systemd/system.control/example.service.d/50-MemoryLimit.conf** file and controls the value of the **/sys/fs/cgroup/memory/system.slice/example.service/memory.limit_in_bytes** file.

- Optionally, to temporarily limit the memory usage of a service, run:

  ```
  # systemctl set-property --runtime example.service MemoryLimit=1500K
  ```

  The command instantly assigns the memory limit to the **example.service** service. The **MemoryLimit** parameter is defined until the next reboot in the **/run/systemd/system.control/example.service.d/50-MemoryLimit.conf** file. With a reboot, the whole **/run/systemd/system.control/** directory and **MemoryLimit** are removed.

NOTE

The **50-MemoryLimit.conf** file stores the memory limit as a multiple of 4096 bytes - one kernel page size specific for AMD64 and Intel 64. The actual number of bytes depends on a CPU architecture.

Additional resources

- For more information about the concept of control groups, see Section 9.1, "Understanding control groups".
- For more information about resource controllers, see Section 9.2, "What kernel resource controllers are" and **systemd.resource-control(5)**, **cgroups(7)** manual pages.
- For more information about the role of **systemd** in control groups, see Section 9.4, "Role of systemd in control groups version 1".

9.5.2.2. Configuring memory resource control settings with unit files

Manually modifying unit files is one of the ways how to set limits, prioritize, or control access to hardware resources for groups of processes.

Procedure

1. To limit the memory usage of a service, modify the **/usr/lib/systemd/system/example.service** file as follows:

   ```
   …
   [Service]
   MemoryLimit=1500K
   …
   ```

   The configuration above places a limit on maximum memory consumption of processes executed in a control group, which **example.service** is part of.

   NOTE

   Use suffixes K, M, G, or T to identify Kilobyte, Megabyte, Gigabyte, or Terabyte as a unit of measurement.

2. Reload all unit configuration files:

   ```
   # systemctl daemon-reload
   ```

3. Restart the service:

   ```
   # systemctl restart example.service
   ```

4. Reboot the system.

5. Optionally, check that the changes took effect:

   ```
   # cat /sys/fs/cgroup/memory/system.slice/example.service/memory.limit_in_bytes
   1536000
   ```

   The example output shows that the memory consumption was limited at around 1,500 Kilobytes.

   NOTE

   The **memory.limit_in_bytes** file stores the memory limit as a multiple of 4096 bytes - one kernel page size specific for AMD64 and Intel 64. The actual number of bytes depends on a CPU architecture.

Additional resources

- For more information about the concept of control groups, see Section 9.1, “Understanding control groups”.
- For more information about resource controllers, see Section 9.2, “What kernel resource controllers are” and **systemd.resource-control(5)**, **cgroups(7)** manual pages.
- For more information about **systemd**, unit configuration files and their locations, as well as a complete list of **systemd** unit types, see the relevant sections in *Configuring basic system settings*.
- For more information about the role of **systemd** in control groups, see Section 9.4, “Role of systemd in control groups version 1”.

9.5.3. Removing control groups version 1 with systemd

You can use the **systemd** system and service manager to remove transient and persistent control groups (**cgroups**) if you no longer need to limit, prioritize, or control access to hardware resources for groups of processes.

9.5.3.1. Removing transient control groups

Transient **cgroups** are automatically released once all the processes that a service or a scope unit contains, finish.

Procedure

- To stop the service unit with all its processes, execute:

  ```
  # systemctl stop <name>.service
  ```

- To terminate one or more of the unit processes, execute:

  ```
  # systemctl kill <name>.service --kill-who=PID,… --signal=signal
  ```

The command above uses the **--kill-who** option to select process(es) from the control group you wish to terminate. To kill multiple processes at the same time, pass a comma-separated list of PIDs. The **--signal** option determines the type of POSIX signal to be sent to the specified processes. The default signal is *SIGTERM*.

Additional resources

- For more information about the concept of control groups, see Section 9.1, "Understanding control groups".
- For more information about resource controllers, see Section 9.2, "What kernel resource controllers are" and **systemd.resource-control(5)**, **cgroups(7)** manual pages.
- For more information about the role of **systemd** in control groups, see Section 9.4, "Role of systemd in control groups version 1".
- For more information about **systemd**, unit configuration files and their locations, as well as a complete list of **systemd** unit types, see the relevant sections in *Configuring basic system settings*.

9.5.3.2. Removing persistent control groups

Persistent **cgroups** are released when a service or a scope unit is stopped or disabled and its configuration file is deleted.

Procedure

1. Stop the service unit:

```
# systemctl stop <name>.service
```

2. Disable the service unit:

```
# systemctl disable <name>.service
```

3. Remove the relevant unit configuration file:

```
# rm /usr/lib/systemd/system/<name>.service
```

4. Reload all unit configuration files so that changes take effect:

```
# systemctl daemon-reload
```

Additional resources

- For more information about the concept of control groups, see Section 9.1, "Understanding control groups".
- For more information about resource controllers, see Section 9.2, "What kernel resource controllers are" and **systemd.resource-control(5)**, **cgroups(7)** manual pages.
- For more information about the role of **systemd** in control groups, see Section 9.4, "Role of systemd in control groups version 1".

- For more information about **systemd**, unit configuration files and their locations, as well as a complete list of **systemd** unit types, see the relevant sections in *Configuring basic system settings*.

- For more information about killing processes with **systemd**, see the **systemd.kill(5)** manual page.

9.6. OBTAINING INFORMATION ABOUT CONTROL GROUPS VERSION 1

The following sections describe how to display various information about control groups (**cgroups**):

- Listing **systemd** units and viewing their status

- Viewing the **cgroups** hierarchy

- Monitoring resource consumption in real time

9.6.1. Listing systemd units

The following procedure describes how to use the **systemd** system and service manager to list its units.

Prerequisites

- Role of systemd in control groups

Procedure

- To list all active units on the system, execute the **# systemctl** command and the terminal will return an output similar to the following example:

```
UNIT                                        LOAD   ACTIVE SUB      DESCRIPTION
…
init.scope                                   loaded active running   System and Service Manager
session-2.scope                                  loaded active running   Session 2 of user jdoe
abrt-ccpp.service                                loaded active exited    Install ABRT coredump hook
abrt-oops.service                                loaded active running   ABRT kernel log watcher
abrt-vmcore.service                               loaded active exited    Harvest vmcores for ABRT
abrt-xorg.service                                loaded active running   ABRT Xorg log watcher
…
-.slice                                     loaded active active    Root Slice
machine.slice                                    loaded active active    Virtual Machine and Container
Slice system-getty.slice                                                  loaded active active
system-getty.slice
system-lvm2\x2dpvscan.slice                           loaded active active    system-
lvm2\x2dpvscan.slice
system-sshd\x2dkeygen.slice                           loaded active active    system-
sshd\x2dkeygen.slice
system-systemd\x2dhibernate\x2dresume.slice        loaded active active    system-
systemd\x2dhibernate\x2dresume>
system-user\x2druntime\x2ddir.slice                loaded active active    system-
user\x2druntime\x2ddir.slice
system.slice                                  loaded active active    System Slice
user-1000.slice                                  loaded active active    User Slice of UID 1000
```

```
user-42.slice                    loaded active active    User Slice of UID 42
user.slice                       loaded active active    User and Session Slice
…
```

- **UNIT** - a name of a unit that also reflects the unit position in a control group hierarchy. The units relevant for resource control are a *slice*, a *scope*, and a *service*.

- **LOAD** - indicates whether the unit configuration file was properly loaded. If the unit file failed to load, the field contains the state *error* instead of *loaded*. Other unit load states are: *stub*, *merged*, and *masked*.

- **ACTIVE** - the high-level unit activation state, which is a generalization of **SUB**.

- **SUB** - the low-level unit activation state. The range of possible values depends on the unit type.

- **DESCRIPTION** - the description of the unit content and functionality.

- To list inactive units, execute:

```
# systemctl --all
```

- To limit the amount of information in the output, execute:

```
# systemctl --type service,masked
```

The **--type** option requires a comma-separated list of unit types such as a *service* and a *slice*, or unit load states such as *loaded* and *masked*.

Additional resources

- For more information about **systemd**, unit files, and a complete list of **systemd** unit types, see the relevant sections in *Configuring basic system settings*.

9.6.2. Viewing a control group version 1 hierarchy

The following procedure describes how to display control groups (**cgroups**) hierarchy and processes running in specific **cgroups**.

Prerequisites

- Section 9.1, “Understanding control groups”

- Section 9.4, “Role of systemd in control groups version 1”

Procedure

- To display the whole **cgroups** hierarchy on your system, execute **# systemd-cgls**:

```
Control group /:
-.slice
├─user.slice
│ ├─user-42.slice
│ │ ├─session-c1.scope
│ │ │ ├─ 965 gdm-session-worker [pam/gdm-launch-environment]
│ │ │ ├─1040 /usr/libexec/gdm-x-session gnome-session --autostart
/usr/share/gdm/greeter/autostart
…
├─init.scope
│ └─1 /usr/lib/systemd/systemd --switched-root --system --deserialize 18
└─system.slice
  …
  ├─example.service
  │ ├─6882 /bin/bash /home/jdoe/example.sh
  │ └─6902 sleep 1
  ├─systemd-journald.service
    └─629 /usr/lib/systemd/systemd-journald
  …
```

The example output returns the entire **cgroups** hierarchy, where the highest level is formed by *slices*.

- To display the **cgroups** hierarchy filtered by a resource controller, execute **# systemd-cgls <*resource_controller*>**:

```
# systemd-cgls memory
Controller memory; Control group /:
├─1 /usr/lib/systemd/systemd --switched-root --system --deserialize 18
├─user.slice
│ ├─user-42.slice
│ │ ├─session-c1.scope
│ │ │ ├─ 965 gdm-session-worker [pam/gdm-launch-environment]
…
└─system.slice
  |
  …
  ├─chronyd.service
  │ └─844 /usr/sbin/chronyd
  ├─example.service
  │ ├─8914 /bin/bash /home/jdoe/example.sh
  │ └─8916 sleep 1
  …
```

The example output of the above command lists the services that interact with the selected controller.

- To display detailed information about a certain unit and its part of the **cgroups** hierarchy, execute **# systemctl status <*system_unit*>**:

```
# systemctl status example.service
● example.service - My example service
   Loaded: loaded (/usr/lib/systemd/system/example.service; enabled; vendor preset:
disabled)
   Active: active (running) since Tue 2019-04-16 12:12:39 CEST; 3s ago
 Main PID: 17737 (bash)
    Tasks: 2 (limit: 11522)
   Memory: 496.0K (limit: 1.5M)
   CGroup: /system.slice/example.service
           ├─17737 /bin/bash /home/jdoe/example.sh
           └─17743 sleep 1
```

```
Apr 16 12:12:39 redhat systemd[1]: Started My example service.
Apr 16 12:12:39 redhat bash[17737]: The current time is Tue Apr 16 12:12:39 CEST 2019
Apr 16 12:12:40 redhat bash[17737]: The current time is Tue Apr 16 12:12:40 CEST 2019
```

Additional resources

- For more information about resource controllers, see Section 9.2, “What kernel resource controllers are” section and **systemd.resource-control(5)**, **cgroups(7)** manual pages.

9.6.3. Viewing resource controllers

The following procedure describes how to learn which processes use which resource controllers.

Prerequisites

- Section 9.2, “What kernel resource controllers are”
- Section 9.1, “Understanding control groups”

Procedure

1. To view which resource controllers a process interacts with, execute the **# cat proc/<*PID*>/cgroup** command:

   ```
   # cat /proc/11269/cgroup
   12:freezer:/
   11:cpuset:/
   10:devices:/system.slice
   9:memory:/system.slice/example.service
   8:pids:/system.slice/example.service
   7:hugetlb:/
   6:rdma:/
   5:perf_event:/
   4:cpu,cpuacct:/
   3:net_cls,net_prio:/
   2:blkio:/
   1:name=systemd:/system.slice/example.service
   ```

   The example output relates to a process of interest. In this case, it is a process identified by **PID 11269**, which belongs to the **example.service** unit. You can determine whether the process was placed in a correct control group as defined by the **systemd** unit file specifications.

   NOTE

   By default, the items and their ordering in the list of resource controllers is the same for all units started by **systemd**, since it automatically mounts all the default resource controllers.

Additional resources

- For more information about resource controllers in general refer to the **cgroups(7)** manual pages.

- For a detailed description of specific resource controllers, see the documentation in the **/usr/share/doc/kernel-doc-<kernel_version>/Documentation/cgroups-v1/** directory.

9.6.4. Monitoring resource consumption

The following procedure describes how to view a list of currently running control groups (**cgroups**) and their resource consumption in real-time.

Prerequisites

- Section 9.1, “Understanding control groups”
- Section 9.2, “What kernel resource controllers are”
- Section 9.4, “Role of systemd in control groups version 1”

Procedure

1. To see a dynamic account of currently running **cgroups**, execute the **# systemd-cgtop** command:

```
Control Group                              Tasks   %CPU   Memory  Input/s Output/s
/                                      607   29.8     1.5G      -       -
/system.slice                              125      -   428.7M       -       -
/system.slice/ModemManager.service           3      -    8.6M       -       -
/system.slice/NetworkManager.service         3      -   12.8M       -       -
/system.slice/accounts-daemon.service        3      -    1.8M       -       -
/system.slice/boot.mount                     -      -   48.0K       -       -
/system.slice/chronyd.service                1      -    2.0M       -       -
/system.slice/cockpit.socket                 -      -    1.3M       -       -
/system.slice/colord.service                 3      -    3.5M       -       -
/system.slice/crond.service                  1      -    1.8M       -       -
/system.slice/cups.service                   1      -    3.1M       -       -
/system.slice/dev-hugepages.mount            -      -  244.0K       -       -
/system.slice/dev-mapper-rhel\x2dswap.swap   -      -  912.0K       -       -
/system.slice/dev-mqueue.mount               -      -   48.0K       -       -
/system.slice/example.service                2      -    2.0M       -       -
/system.slice/firewalld.service              2      -   28.8M       -       -
...
```

   The example output displays currently running **cgroups** ordered by their resource usage (CPU, memory, disk I/O load). The list refreshes every 1 second by default. Therefore, it offers a dynamic insight into the actual resource usage of each control group.

Additional resources

- For more information about dynamic monitoring of resource usage, see the **systemd-cgtop(1)** manual pages.

9.7. WHAT NAMESPACES ARE

Namespaces are one of the most important methods for organizing and identifying software objects.

A namespace wraps a global system resource (for example a mount point, a network device, or a hostname) in an abstraction that makes it appear to processes within the namespace that they have their own isolated instance of the global resource. One of the most common technologies that utilize namespaces are containers.

Changes to a particular global resource are visible only to processes in that namespace and do not affect the rest of the system or other namespaces.

To inspect which namespaces a process is a member of, you can check the symbolic links in the **/proc/<*PID*>/ns/** directory.

The following table shows supported namespaces and resources which they isolate:

Namespace	Isolates
Mount	Mount points
UTS	Hostname and NIS domain name
IPC	System V IPC, POSIX message queues
PID	Process IDs
Network	Network devices, stacks, ports, etc
User	User and group IDs
Control groups	Control group root directory

Additional resources

- For more information about namespaces, see the **namespaces(7)** and **cgroup_namespaces(7)** manual pages.
- For more information about **cgroups**, see Section 9.1, “Understanding control groups”.

[1] Linux Control Group v2 – An Introduction, Devconf.cz 2019 presentation by Waiman Long

CHAPTER 10. ANALYZING SYSTEM PERFORMANCE WITH BPF COMPILER COLLECTION

As a system administrator, use the BPF Compiler Collection (BCC) library to create tools for analyzing the performance of your Linux operating system and gathering information, which could be difficult to obtain through other interfaces.

10.1. A BRIEF INTRODUCTION TO BCC

BPF Compiler Collection (BCC) is a library, which facilitates the creation of the extended Berkeley Packet Filter (eBPF) programs. The main utility of eBPF programs is analyzing OS performance and network performance without experiencing overhead or security issues.

BCC removes the need for users to know deep technical details of eBPF, and provides many out-of-the-box starting points, such as the **bcc-tools** package with pre-created eBPF programs.

NOTE

The eBPF programs are triggered on events, such as disk I/O, TCP connections, and process creations. It is unlikely that the programs should cause the kernel to crash, loop or become unresponsive because they run in a safe virtual machine in the kernel.

Additional resources

- For more information about BCC, see the **/usr/share/doc/bcc/README.md** file.

10.2. INSTALLING THE BCC-TOOLS PACKAGE

This section describes how to install the **bcc-tools** package, which also installs the BPF Compiler Collection (BCC) library as a dependency.

Prerequisites

- An active *Red Hat Enterprise Linux subscription*
- An *enabled repository* containing the **bcc-tools** package
- Introduction to **yum** *package manager*
- Updated kernel

Procedure

1. Install **bcc-tools**:

   ```
   # yum install bcc-tools
   ```

   Once installed, the tools are placed in the **/usr/share/bcc/tools/** directory.

2. Optionally, inspect the tools:

   ```
   # ll /usr/share/bcc/tools/
   ...
   ```

```
-rwxr-xr-x. 1 root root  4198 Dec 14 17:53 dcsnoop
-rwxr-xr-x. 1 root root  3931 Dec 14 17:53 dcstat
-rwxr-xr-x. 1 root root 20040 Dec 14 17:53 deadlock_detector
-rw-r--r--. 1 root root  7105 Dec 14 17:53 deadlock_detector.c
drwxr-xr-x. 3 root root  8192 Mar 11 10:28 doc
-rwxr-xr-x. 1 root root  7588 Dec 14 17:53 execsnoop
-rwxr-xr-x. 1 root root  6373 Dec 14 17:53 ext4dist
-rwxr-xr-x. 1 root root 10401 Dec 14 17:53 ext4slower
...
```

The **doc** directory in the listing above contains documentation for each tool.

10.3. USING SELECTED BCC-TOOLS FOR PERFORMANCE ANALYSES

This section describes how to use certain pre-created programs from the BPF Compiler Collection (BCC) library to efficiently and securely analyze the system performance on the per-event basis. The set of pre-created programs in the BCC library can serve as examples for creation of additional programs.

Prerequisites

- Introduction to BCC
- Installed BCC library
- Root permissions

Using execsnoop to examine the system processes

1. Execute the **execsnoop** program in one terminal:

   ```
   # /usr/share/bcc/tools/execsnoop
   ```

2. In another terminal execute for example:

   ```
   $ ls /usr/share/bcc/tools/doc/
   ```

   The above creates a short-lived process of the **ls** command.

3. The terminal running **execsnoop** shows the output similar to the following:

   ```
   PCOMM PID    PPID   RET ARGS
   ls    8382   8287     0 /usr/bin/ls --color=auto /usr/share/bcc/tools/doc/
   sed  8385   8383     0 /usr/bin/sed s/^ *[0-9]\+ *//
   ...
   ```

   The **execsnoop** program prints a line of output for each new process, which consumes system resources. It even detects processes of programs that run very shortly, such as **ls**, and most monitoring tools would not register them.

   The result above shows a parent process name (**ls**), its process ID (**5076**), parent process ID (**2931**), the return value of the **exec()** system call (**0**), which loads program code into new processes. Finally, the output displays a location of the started program with arguments (**/usr/bin/ls --color=auto /usr/share/bcc/tools/doc/**).

To see more details, examples, and options for **execsnoop**, refer to the **/usr/share/bcc/tools/doc/execsnoop_example.txt** file.

For more information about **exec()**, see **exec(3)** manual pages.

Using opensnoop to track what files a command opens

1. Execute the **opensnoop** program in one terminal:

   ```
   # /usr/share/bcc/tools/opensnoop -n uname
   ```

   The above prints output for files, which are opened only by the process of the **uname** command.

2. In another terminal execute:

   ```
    $ uname
   ```

   The command above opens certain files, which are captured in the next step.

3. The terminal running **opensnoop** shows the output similar to the following:

   ```
   PID    COMM  FD ERR PATH
   8596   uname  3  0   /etc/ld.so.cache
   8596   uname  3  0   /lib64/libc.so.6
   8596   uname  3  0   /usr/lib/locale/locale-archive
   ...
   ```

   The **opensnoop** program watches the **open()** system call across the whole system, and prints a line of output for each file that **uname** tried to open along the way.

   The result above shows a process ID (**PID**), a process name (**COMM**), and a file descriptor (**FD**) - a value that **open()** returns to refer to the open file. Finally, the output displays a column for errors (**ERR**) and a location of files that **open()** tries to open (**PATH**).

   If a command tries to read a non-existent file, then the **FD** column returns **-1** and the **ERR** column prints a value corresponding to the relevant error. As a result, **opensnoop** can help you identify an application that does not behave properly.

To see more details, examples, and options for **opensnoop**, refer to the **/usr/share/bcc/tools/doc/opensnoop_example.txt** file.

For more information about **open()**, see **open(2)** manual pages.

Using biotop to examine the I/O operations on the disk

1. Execute the **biotop** program in one terminal:

   ```
   # /usr/share/bcc/tools/biotop 30
   ```

   The command enables you to monitor the top processes, which perform I/O operations on the disk. The argument ensures that the command will produce a 30 second summary.

   NOTE

   When no argument provided, the output screen by default refreshes every 1 second.

2. In another terminal execute for example :

   ```
   # dd if=/dev/vda of=/dev/zero
   ```

   The command above reads the content from the local hard disk device and writes the output to the **/dev/zero** file. This step generates certain I/O traffic to illustrate **biotop**.

3. The terminal running **biotop** shows the output similar to the following:

   ```
   PID    COMM             D MAJ MIN DISK       I/O  Kbytes     AVGms
   9568   dd               R 252 0   vda      16294 14440636.0  3.69
   48     kswapd0          W 252 0   vda       1763 120696.0    1.65
   7571   gnome-shell      R 252 0   vda        834 83612.0     0.33
   1891   gnome-shell      R 252 0   vda       1379 19792.0     0.15
   7515   Xorg             R 252 0   vda        280  9940.0     0.28
   7579   llvmpipe-1       R 252 0   vda        228  6928.0     0.19
   9515   gnome-control-c  R 252 0   vda         62  6444.0     0.43
   8112   gnome-terminal-  R 252 0   vda         67  2572.0     1.54
   7807   gnome-software   R 252 0   vda         31  2336.0     0.73
   9578   awk              R 252 0   vda         17  2228.0     0.66
   7578   llvmpipe-0       R 252 0   vda        156  2204.0     0.07
   9581   pgrep            R 252 0   vda         58  1748.0     0.42
   7531   InputThread      R 252 0   vda         30  1200.0     0.48
   7504   gdbus            R 252 0   vda          3  1164.0     0.30
   1983   llvmpipe-1       R 252 0   vda         39   724.0     0.08
   1982   llvmpipe-0       R 252 0   vda         36   652.0     0.06
   ...
   ```

   The results shows that the **dd** process, with the process ID 9568, performed 16,294 read operations from the **vda** disk. The read operations reached total of 14,440,636 Kbytes with an average I/O time 3.69 ms.

To see more details, examples, and options for **biotop**, refer to the **/usr/share/bcc/tools/doc/biotop_example.txt** file.

For more information about **dd**, see **dd(1)** manual pages.

Using xfsslower to expose unexpectedly slow file system operations

1. Execute the **xfsslower** program in one terminal:

   ```
   # /usr/share/bcc/tools/xfsslower 1
   ```

   The command above measures the time the XFS file system spends in performing read, write, open or sync (**fsync**) operations. The **1** argument ensures that the program shows only the operations that are slower than 1 ms.

   NOTE

   When no arguments provided, **xfsslower** by default displays operations slower than 10 ms.

2. In another terminal execute, for example, the following:

   ```
   $ vim text
   ```

```
_
```

The command above creates a text file in the **vim** editor to initiate certain interaction with the XFS file system.

3. The terminal running **xfsslower** shows something similar upon saving the file from the previous step:

```
TIME     COMM           PID    T BYTES   OFF_KB   LAT(ms) FILENAME
13:07:14 b'bash'        4754   R 256     0           7.11 b'vim'
13:07:14 b'vim'         4754   R 832     0           4.03 b'libgpm.so.2.1.0'
13:07:14 b'vim'         4754   R 32      20          1.04 b'libgpm.so.2.1.0'
13:07:14 b'vim'         4754   R 1982    0           2.30 b'vimrc'
13:07:14 b'vim'         4754   R 1393    0           2.52 b'getscriptPlugin.vim'
13:07:45 b'vim'         4754   S 0       0           6.71 b'text'
13:07:45 b'pool'        2588   R 16      0           5.58 b'text'
...
```

Each line above represents an operation in the file system, which took more time than a certain threshold. **xfsslower** is good at exposing possible file system problems, which can take form of unexpectedly slow operations.

The **T** column represents operation type (**R**ead/**W**rite/**S**ync), **OFF_KB** is a file offset in KB. **FILENAME** is the file the process (**COMM**) is trying to read, write, or sync.

To see more details, examples, and options for **xfsslower**, refer to the **/usr/share/bcc/tools/doc/xfsslower_example.txt** file.

For more information about **fsync**, see **fsync(2)** manual pages.

CHAPTER 11. ENHANCING SECURITY WITH THE KERNEL INTEGRITY SUBSYSTEM

You can increase the protection of your system by utilizing components of the kernel integrity subsystem. The following sections introduce the relevant components and provide guidance on their configuration.

11.1. THE KERNEL INTEGRITY SUBSYSTEM

The integrity subsystem is a part of the kernel which is responsible for maintaining the overall system's data integrity. This subsystem helps to keep the state of a certain system the same from the time it was built thereby it prevents undesired modification on specific system files from users.

The kernel integrity subsystem consists of two major components:

Integrity Measurement Architecture (IMA)

- Measures files' content whenever it is executed or opened. Users can change this behavior by applying custom policies.
- Places the measured values within the kernel's memory space thereby it prevents any modification from the users of the system.
- Allows local and remote parties to verify the measured values.

Extended Verification Module (EVM)

- Protects files' extended attributes (also known as *xattr*) that are related to the system's security, like IMA measurements and SELinux attributes, by cryptographically hashing their corresponding values.

Both IMA and EVM also contain numerous feature extensions that bring additional functionality. For example:

IMA-Appraisal

- Provides local validation of the current file's content against the values previously stored in the measurement file within the kernel memory. This extension forbids any operation to be performed over a specific file in case the current and the previous measure do not match.

EVM Digital Signatures

- Allows digital signatures to be used through cryptographic keys stored into the kernel's keyring.

NOTE

The feature extensions complement each other, but you can configure and use them independently of one another.

The kernel integrity subsystem can harness the Trusted Platform Module (TPM) to harden the system security even more. TPM is a specification by the Trusted Computing Group (TCG) for important cryptographic functions. TPMs are usually built as dedicated hardware that is attached to the platform's motherboard and prevents software-based attacks by providing cryptographic functions from a protected and tamper-proof area of the hardware chip. Some of the TPM features are:

- Random-number generator
- Generator and secure storage for cryptographic keys
- Hashing generator
- Remote attestation

Additional resources

- For details about the kernel integrity subsystem, see the *official upstream wiki page* .
- For further information about TPM, see the *Trusted Computing Group resources* .

11.2. INTEGRITY MEASUREMENT ARCHITECTURE

Integrity Measurement Architecture (IMA) is a component of the kernel integrity subsystem. IMA aims to maintain the contents of local files. Specifically, IMA measures, stores, and appraises files' hashes before they are accessed, which prevents the reading and execution of unreliable data. Thereby, IMA enhances the security of the system.

11.3. EXTENDED VERIFICATION MODULE

Extended Verification Module (EVM) is a component of the kernel integrity subsystem, which monitors changes in files' extended attributes (xattr). Many security-oriented technologies, including Integrity Measurement Architecture (IMA), store sensitive file information, such as content hashes, in the extended attributes. EVM creates another hash from these extended attributes and from a special key, which is loaded at boot time. The resulting hash is validated every time the extended attribute is used. For example, when IMA appraises the file.

RHEL 8 accepts the special encrypted key under the **evm-key** keyring. The key was created by a *master key* held in the kernel keyrings.

11.4. TRUSTED AND ENCRYPTED KEYS

The following section introduces trusted and encrypted keys as an important part of enhancing system security.

Trusted and *encrypted keys* are variable-length symmetric keys generated by the kernel that utilize the kernel keyring service. The fact that this type of keys never appear in the user space in an unencrypted form means that their integrity can be verified, which in turn means that they can be used, for example, by the extended verification module (EVM) to verify and confirm the integrity of a running system. User-level programs can only access the keys in the form of encrypted *blobs*.

Trusted keys need a hardware component: the Trusted Platform Module (TPM) chip, which is used to both create and encrypt (seal) the keys. The TPM seals the keys using a 2048-bit RSA key called the *storage root key* (SRK).

> NOTE
>
> To use a TPM 1.2 specification, enable and activate it through a setting in the machine firmware or by using the **tpm_setactive** command from the **tpm-tools** package of utilities. Also, the **TrouSers** software stack needs to be installed and the **tcsd** daemon needs to be running to communicate with the TPM (dedicated hardware). The **tcsd** daemon is part of the **TrouSers** suite, which is available through the **trousers** package. The more recent and backward incompatible TPM 2.0 uses a different software stack, where the **tpm2-tools** or **ibm-tss** utilities provide access to the dedicated hardware.

In addition to that, the user can seal the trusted keys with a specific set of the TPM's *platform configuration register* (PCR) values. PCR contains a set of integrity-management values that reflect the firmware, boot loader, and operating system. This means that PCR-sealed keys can only be decrypted by the TPM on the same system on which they were encrypted. However, once a PCR-sealed trusted key is loaded (added to a keyring), and thus its associated PCR values are verified, it can be updated with new (or future) PCR values, so that a new kernel, for example, can be booted. A single key can also be saved as multiple blobs, each with different PCR values.

Encrypted keys do not require a TPM, as they use the kernel Advanced Encryption Standard (AES), which makes them faster than trusted keys. Encrypted keys are created using kernel-generated random numbers and encrypted by a *master key* when they are exported into user-space blobs. The master key is either a trusted key or a user key. If the master key is not trusted, the encrypted key is only as secure as the user key used to encrypt it.

11.4.1. Working with trusted keys

The following section describes how to create, export, load or update trusted keys with the **keyctl** utility to improve the system security.

Prerequisites

- For the 64-bit ARM architecture and IBM Z, the **trusted** kernel module needs to be loaded. For more information on how to load kernel modules, see Chapter 3, *Managing kernel modules*.
- Trusted Platform Module (TPM) needs to be enabled and active. For more information about TPM see,] and xref:trusted-and-encrypted-keys_enhancing-security-with-the-kernel-integrity-subsystem[.

Procedure

1. To create a trusted key using a TPM, execute:

   ```
   # keyctl add trusted <name> "new <key_length> [options]" <key_ring>
   ```

   - Based on the syntax, construct an example command as follows:

     ```
     # keyctl add trusted kmk "new 32" @u
     642500861
     ```

     The command creates a trusted key called **kmk** with the length of 32 bytes (256 bits) and places it in the user keyring (**@u**). The keys may have a length of 32 to 128 bytes (256 to 1024 bits).

2. To list the current structure of the kernel keyrings:

```
# keyctl show
Session Keyring
       -3 --alswrv    500   500  keyring: ses 97833714 --alswrv 500 -1 \ keyring: uid.1000
642500861 --alswrv 500 500 \ trusted: kmk
```

3. To export the key to a user-space blob, execute:

   ```
   # keyctl pipe 642500861 > kmk.blob
   ```

   The command uses the **pipe** subcommand and the serial number of **kmk**.

4. To load the trusted key from the user-space blob, use the **add** subcommand with the blob as an argument:

   ```
   # keyctl add trusted kmk "load `cat kmk.blob`" @u
   268728824
   ```

5. Create secure encrypted keys based on the TPM-sealed trusted key:

   ```
   # keyctl add encrypted <name> "new [format] <key_type>:<master_key_name> <keylength>"
   <key_ring>
   ```

   - Based on the syntax, generate an encrypted key using the already created trusted key:

     ```
     # keyctl add encrypted encr-key "new trusted:kmk 32" @u
     159771175
     ```

     The command uses the TPM-sealed trusted key (**kmk**), produced in the previous step, as a *master key* for generating encrypted keys.

Additional resources

- For detailed information about using **keyctl**, see the **keyctl(1)** manual page.
- For more information about trusted and encrypted keys, see Section 11.4, "Trusted and encrypted keys".
- For more information about the kernel keyring service, see the upstream kernel documentation.
- For more information about the TPM, see Section 11.1, "The kernel integrity subsystem" .

11.4.2. Working with encrypted keys

The following section describes managing encrypted keys to improve the system security on systems where a Trusted Platform Module (TPM) is not available.

Prerequisites

- For the 64-bit ARM architecture and IBM Z, the **encrypted-keys** kernel module needs to be loaded. For more information on how to load kernel modules, see Chapter 3, *Managing kernel modules*.

Procedure

1. Use a random sequence of numbers to generate a user key:

   ```
   # keyctl add user kmk-user “dd if=/dev/urandom bs=1 count=32 2>/dev/null” @u
   427069434
   ```

   The command generates a user key called **kmk-user** which acts as a *master key* and is used to seal the actual encrypted keys.

2. Generate an encrypted key using the master key from the previous step:

   ```
   # keyctl add encrypted encr-key "new user:kmk-user 32" @u
   1012412758
   ```

3. Optionally, list all keys in the specified user keyring:

   ```
   # keyctl list @u
   2 keys in keyring:
   427069434: --alswrv  1000  1000 user: kmk-user
   1012412758: --alswrv  1000  1000 encrypted: encr-key
   ```

> **IMPORTANT**
>
> Keep in mind that encrypted keys that are not sealed by a master trusted key are only as secure as the user master key (random-number key) used to encrypt them. Therefore, the master user key should be loaded as securely as possible and preferably early during the boot process.

Additional resources

- For detailed information about using **keyctl**, see the **keyctl(1)** manual page.
- For more information about the kernel keyring service, see the upstream kernel documentation.

11.5. ENABLING INTEGRITY MEASUREMENT ARCHITECTURE AND EXTENDED VERIFICATION MODULE

Integrity measurement architecture (IMA) and extended verification module (EVM) belong to the kernel integrity subsystem and enhance the system security in various ways. The following section describes how to enable and configure IMA and EVM to improve the security of the operating system.

Prerequisites

- Verify that the **securityfs** filesystem is mounted on the **/sys/kernel/security/** directory and the **/sys/kernel/security/integrity/ima/** directory exists.

  ```
  # mount
  …
  securityfs on /sys/kernel/security type securityfs (rw,nosuid,nodev,noexec,relatime)
  …
  ```

- Verify that the **systemd** service manager is already patched to support IMA and EVM on boot time:

```
# dmesg | grep -i -e EVM -e IMA
[    0.000000] Command line: BOOT_IMAGE=(hd0,msdos1)/vmlinuz-4.18.0-167.el8.x86_64
root=/dev/mapper/rhel-root ro crashkernel=auto resume=/dev/mapper/rhel-swap
rd.lvm.lv=rhel/root rd.lvm.lv=rhel/swap rhgb quiet
[    0.000000] kvm-clock: cpu 0, msr 23601001, primary cpu clock
[    0.000000] Using crashkernel=auto, the size chosen is a best effort estimation.
[    0.000000] Kernel command line: BOOT_IMAGE=(hd0,msdos1)/vmlinuz-4.18.0-
167.el8.x86_64 root=/dev/mapper/rhel-root ro crashkernel=auto resume=/dev/mapper/rhel-
swap rd.lvm.lv=rhel/root rd.lvm.lv=rhel/swap rhgb quiet
[    0.911527] ima: No TPM chip found, activating TPM-bypass!
[    0.911538] ima: Allocated hash algorithm: sha1
[    0.911580] evm: Initialising EVM extended attributes:
[    0.911581] evm: security.selinux
[    0.911581] evm: security.ima
[    0.911582] evm: security.capability
[    0.911582] evm: HMAC attrs: 0x1
[    1.715151] systemd[1]: systemd 239 running in system mode. (+PAM +AUDIT +SELINUX
+IMA -APPARMOR +SMACK +SYSVINIT +UTMP +LIBCRYPTSETUP +GCRYPT
+GNUTLS +ACL +XZ +LZ4 +SECCOMP +BLKID +ELFUTILS +KMOD +IDN2 -IDN +PCRE2
default-hierarchy=legacy)
[    3.824198] fbcon: qxldrmfb (fb0) is primary device
[    4.673457] PM: Image not found (code -22)
[    6.549966] systemd[1]: systemd 239 running in system mode. (+PAM +AUDIT +SELINUX
+IMA -APPARMOR +SMACK +SYSVINIT +UTMP +LIBCRYPTSETUP +GCRYPT
+GNUTLS +ACL +XZ +LZ4 +SECCOMP +BLKID +ELFUTILS +KMOD +IDN2 -IDN +PCRE2
default-hierarchy=legacy)
```

Procedure

1. Add the following kernel command line parameters:

    ```
    # grubby --update-kernel=/boot/vmlinuz-$(uname -r) --args="ima_policy=appraise_tcb
    ima_appraise=fix evm=fix"
    ```

    The command enables IMA and EVM in the *fix* mode for the current boot entry and allows users to gather and update the IMA measurements.

    The **ima_policy=appraise_tcb** kernel command line parameter ensures that the kernel uses the default Trusted Computing Base (TCB) measurement policy and the appraisal step. The appraisal part forbids access to files, whose prior and current measures do not match.

2. Reboot to make the changes come into effect.

3. Optionally, verify that the parameters have been added to the kernel command line:

    ```
    # cat /proc/cmdline
    BOOT_IMAGE=(hd0,msdos1)/vmlinuz-4.18.0-167.el8.x86_64 root=/dev/mapper/rhel-root ro
    crashkernel=auto resume=/dev/mapper/rhel-swap rd.lvm.lv=rhel/root rd.lvm.lv=rhel/swap
    rhgb quiet ima_policy=appraise_tcb ima_appraise=fix evm=fix
    ```

4. Create a kernel master key to protect the EVM key:

    ```
    # keyctl add user kmk dd if=/dev/urandom bs=1 count=32 2> /dev/null @u
    748544121
    ```

The kernel master key (**kmk**) is kept entirely in the kernel space memory. The 32-byte long value of the kernel master key **kmk** is generated from random bytes from the **/dev/urandom** file and placed in the user (**@u**) keyring. The key serial number is on the second line of the previous output.

5. Create an encrypted EVM key based on the **kmk** key:

   ```
   # keyctl add encrypted evm-key "new user:kmk 64" @u
   641780271
   ```

   The command uses **kmk** to generate and encrypt a 64-byte long user key (named **evm-key**) and places it in the user (**@u**) keyring. The key serial number is on the second line of the previous output.

   > **IMPORTANT**
   >
   > It is necessary to name the user key as **evm-key** because that is the name the EVM subsystem is expecting and is working with.

6. Create a directory for exported keys:

   ```
   # mkdir -p /etc/keys/
   ```

7. Search for the **kmk** key and export its value into a file:

   ```
   # keyctl pipe keyctl search @u user kmk > /etc/keys/kmk
   ```

   The command places the unencrypted value of the kernel master key (**kmk**) into a file of previously defined location (**/etc/keys/**).

8. Search for the **evm-key** user key and export its value into a file:

   ```
   # keyctl pipe keyctl search @u encrypted evm-key > /etc/keys/evm-key
   ```

   The command places the encrypted value of the user **evm-key** key into a file of arbitrary location. The **evm-key** has been encrypted by the kernel master key earlier.

9. Optionally, view the newly created keys:

   ```
   # keyctl show
   Session Keyring
   974575405   --alswrv    0       0      keyring: ses 299489774 --alswrv 0 65534 \ keyring: uid.0
   748544121 --alswrv 0 0 \ user: kmk
   641780271   --alswrv    0       0          \_ encrypted: evm-key
   ```

   You should be able to see a similar output.

10. Activate EVM:

    ```
    # echo 1 > /sys/kernel/security/evm
    ```

11. Optionally, verify that EVM has been initialized:

```
# dmesg | tail -1
[…] evm: key initialized
```

Additional resources

- For more information about the kernel integrity subsystem, see Section 11.1, “The kernel integrity subsystem”.
- For more information about Integrity Measurement Architecture, see Section 11.2, “Integrity measurement architecture”.
- For more information about Extended Verification Module, see Section 11.3, “Extended verification module”.
- For more information about creating encrypted keys, see Section 11.4, “Trusted and encrypted keys”.

11.6. COLLECTING FILE HASHES WITH INTEGRITY MEASUREMENT ARCHITECTURE

The first level of operation of integrity measurement architecture (IMA) is the *measurement* phase, which allows to create file hashes and store them as extended attributes (xattrs) of those files. The following section describes how to create and inspect the files' hashes.

Prerequisites

- Enable integrity measurement architecture (IMA) and extended verification module (EVM) as described in Section 11.5, “Enabling integrity measurement architecture and extended verification module”.
- Verify that the **ima-evm-utils**, **attr**, and **keyutils** packages are already installed:

```
# yum install ima-evm-utils attr keyutils
Updating Subscription Management repositories.
This system is registered to Red Hat Subscription Management, but is not receiving updates.
You can use subscription-manager to assign subscriptions.
Last metadata expiration check: 0:58:22 ago on Fri 14 Feb 2020 09:58:23 AM CET.
Package ima-evm-utils-1.1-5.el8.x86_64 is already installed.
Package attr-2.4.48-3.el8.x86_64 is already installed.
Package keyutils-1.5.10-7.el8.x86_64 is already installed.
Dependencies resolved.
Nothing to do.
Complete!
```

Procedure

1. Create a test file:

   ```
   # echo <Test_text> > test_file
   ```

   IMA and EVM ensure that the example file **test_file** is assigned hash values, which are stored as its extended attributes.

2. Inspect extended attributes of the file:

   ```
   # getfattr -m . -d test_file
   # file: test_file
   security.evm=0sAnDIy4VPA0HArpPO/EqiutnNyBql
   security.ima=0sAQOEDeuUnWzwwKYk+n66h/vby3eD
   security.selinux="unconfined_u:object_r:admin_home_t:s0"
   ```

   The previous example output shows extended attributes related to SELinux and the IMA and EVM hash values. EVM actively adds a **security.evm** extended attribute and detects any offline tampering to xattrs of other files such as **security.ima** that are directly related to content integrity of files. The value of the **security.evm** field is in Hash-based Message Authentication Code (HMAC-SHA1), which was generated with the **evm-key** user key.

Additional resources

- For further information about general security concepts in Red Hat Enterprise Linux 8, see the relevant sections of *Security hardening*.
- For information about Integrity Measurement Architecture, see Section 11.2, “Integrity measurement architecture”.
- For information about Extended Verification Module, see Section 11.3, “Extended verification module”.

11.7. RELATED INFORMATION

- For further information about general security concepts in Red Hat Enterprise Linux 8, see the relevant sections of *Security hardening*.
- For details about IMA and EVM see the *official upstream wiki page* .
- Basic and advanced configuration of Security-Enhanced Linux (SELinux) describes the basic principles of SELinux and documents in detail how to configure and use SELinux with various services, such as the Apache HTTP Server.