



Red Hat Enterprise Linux 9

Managing, monitoring, and updating the kernel

A guide to managing the Linux kernel on Red Hat Enterprise Linux 9

Red Hat Enterprise Linux 9 Managing, monitoring, and updating the kernel

A guide to managing the Linux kernel on Red Hat Enterprise Linux 9

Legal Notice

Copyright © 2023 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux[®] is the registered trademark of Linus Torvalds in the United States and other countries.

Java[®] is a registered trademark of Oracle and/or its affiliates.

XFS[®] is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL[®] is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js[®] is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack[®] Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

As a system administrator, you can configure the Linux kernel to optimize the operating system. Changes to the Linux kernel can improve system performance, security, and stability, as well as your ability to audit the system and troubleshoot problems.

Table of Contents

MAKING OPEN SOURCE MORE INCLUSIVE	6
PROVIDING FEEDBACK ON RED HAT DOCUMENTATION	7
CHAPTER 1. THE LINUX KERNEL	8
1.1. WHAT THE KERNEL IS	8
1.2. RPM PACKAGES	8
Types of RPM packages	8
1.3. THE LINUX KERNEL RPM PACKAGE OVERVIEW	9
1.4. DISPLAYING CONTENTS OF THE KERNEL PACKAGE	9
1.5. INSTALLING SPECIFIC KERNEL VERSIONS	11
1.6. UPDATING THE KERNEL	11
1.7. SETTING A KERNEL AS DEFAULT	11
CHAPTER 2. THE 64K PAGE SIZE KERNEL	13
CHAPTER 3. MANAGING KERNEL MODULES	14
3.1. INTRODUCTION TO KERNEL MODULES	14
3.2. KERNEL MODULE DEPENDENCIES	14
3.3. LISTING INSTALLED KERNEL MODULES	15
3.4. LISTING CURRENTLY LOADED KERNEL MODULES	15
3.5. DISPLAYING INFORMATION ABOUT KERNEL MODULES	16
3.6. LOADING KERNEL MODULES AT SYSTEM RUNTIME	17
3.7. UNLOADING KERNEL MODULES AT SYSTEM RUNTIME	18
3.8. UNLOADING KERNEL MODULES AT EARLY STAGES OF THE BOOT PROCESS	19
3.9. LOADING KERNEL MODULES AUTOMATICALLY AT SYSTEM BOOT TIME	20
3.10. PREVENTING KERNEL MODULES FROM BEING AUTOMATICALLY LOADED AT SYSTEM BOOT TIME	21
3.11. COMPILING CUSTOM KERNEL MODULES	23
CHAPTER 4. CONFIGURING KERNEL COMMAND-LINE PARAMETERS	26
4.1. WHAT ARE KERNEL COMMAND-LINE PARAMETERS	26
4.2. UNDERSTANDING BOOT ENTRIES	26
4.3. CHANGING KERNEL COMMAND-LINE PARAMETERS FOR ALL BOOT ENTRIES	27
4.4. CHANGING KERNEL COMMAND-LINE PARAMETERS FOR A SINGLE BOOT ENTRY	28
4.5. CHANGING KERNEL COMMAND-LINE PARAMETERS TEMPORARILY AT BOOT TIME	28
4.6. CONFIGURING GRUB SETTINGS TO ENABLE SERIAL CONSOLE CONNECTION	29
CHAPTER 5. CONFIGURING KERNEL PARAMETERS AT RUNTIME	31
5.1. WHAT ARE KERNEL PARAMETERS	31
5.2. CONFIGURING KERNEL PARAMETERS TEMPORARILY WITH SYSCTL	32
5.3. CONFIGURING KERNEL PARAMETERS PERMANENTLY WITH SYSCTL	32
5.4. USING CONFIGURATION FILES IN /ETC/SYSCTL.D/ TO ADJUST KERNEL PARAMETERS	33
5.5. CONFIGURING KERNEL PARAMETERS TEMPORARILY THROUGH /PROC/SYS/	34
CHAPTER 6. CONFIGURING KERNEL PARAMETERS PERMANENTLY BY USING THE KERNEL_SETTINGS RHEL SYSTEM ROLE	35
6.1. INTRODUCTION TO THE KERNEL_SETTINGS ROLE	35
6.2. APPLYING SELECTED KERNEL PARAMETERS USING THE KERNEL_SETTINGS ROLE	36
CHAPTER 7. APPLYING PATCHES WITH KERNEL LIVE PATCHING	40
7.1. LIMITATIONS OF KPATCH	40
7.2. SUPPORT FOR THIRD-PARTY LIVE PATCHING	40
7.3. ACCESS TO KERNEL LIVE PATCHES	41

7.4. COMPONENTS OF KERNEL LIVE PATCHING	41
7.5. HOW KERNEL LIVE PATCHING WORKS	41
7.6. SUBSCRIBING THE CURRENTLY INSTALLED KERNELS TO THE LIVE PATCHING STREAM	42
7.7. AUTOMATICALLY SUBSCRIBING ANY FUTURE KERNEL TO THE LIVE PATCHING STREAM	44
7.8. DISABLING AUTOMATIC SUBSCRIPTION TO THE LIVE PATCHING STREAM	45
7.9. UPDATING KERNEL PATCH MODULES	46
7.10. REMOVING THE LIVE PATCHING PACKAGE	47
7.11. UNINSTALLING THE KERNEL PATCH MODULE	48
7.12. DISABLING KPATCH.SERVICE	49
CHAPTER 8. KEEPING KERNEL PANIC PARAMETERS DISABLED IN VIRTUALIZED ENVIRONMENTS	51
8.1. WHAT IS A SOFT LOCKUP	51
8.2. PARAMETERS CONTROLLING KERNEL PANIC	51
8.3. SPURIOUS SOFT LOCKUPS IN VIRTUALIZED ENVIRONMENTS	52
CHAPTER 9. ADJUSTING KERNEL PARAMETERS FOR DATABASE SERVERS	53
9.1. INTRODUCTION TO DATABASE SERVERS	53
9.2. PARAMETERS AFFECTING PERFORMANCE OF DATABASE APPLICATIONS	53
CHAPTER 10. GETTING STARTED WITH KERNEL LOGGING	55
10.1. WHAT IS THE KERNEL RING BUFFER	55
10.2. ROLE OF PRINTK ON LOG-LEVELS AND KERNEL LOGGING	55
CHAPTER 11. SIGNING A KERNEL AND MODULES FOR SECURE BOOT	57
11.1. PREREQUISITES	57
11.2. WHAT IS UEFI SECURE BOOT	58
11.3. UEFI SECURE BOOT SUPPORT	59
11.4. REQUIREMENTS FOR AUTHENTICATING KERNEL MODULES WITH X.509 KEYS	59
11.5. SOURCES FOR PUBLIC KEYS	60
11.6. GENERATING A PUBLIC AND PRIVATE KEY PAIR	61
11.7. EXAMPLE OUTPUT OF SYSTEM KEYRINGS	63
11.8. ENROLLING PUBLIC KEY ON TARGET SYSTEM BY ADDING THE PUBLIC KEY TO THE MOK LIST	64
11.9. SIGNING A KERNEL WITH THE PRIVATE KEY	65
11.10. SIGNING A GRUB BUILD WITH THE PRIVATE KEY	66
11.11. SIGNING KERNEL MODULES WITH THE PRIVATE KEY	67
11.12. LOADING SIGNED KERNEL MODULES	69
CHAPTER 12. UPDATING THE SECURE BOOT REVOCATION LIST	71
12.1. PREREQUISITES	71
12.2. WHAT IS UEFI SECURE BOOT	71
12.3. THE SECURE BOOT REVOCATION LIST	71
12.4. APPLYING AN ONLINE REVOCATION LIST UPDATE	72
12.5. APPLYING AN OFFLINE REVOCATION LIST UPDATE	73
CHAPTER 13. ENHANCING SECURITY WITH THE KERNEL INTEGRITY SUBSYSTEM	74
13.1. THE KERNEL INTEGRITY SUBSYSTEM	74
13.2. TRUSTED AND ENCRYPTED KEYS	75
13.3. WORKING WITH TRUSTED KEYS	76
13.4. WORKING WITH ENCRYPTED KEYS	77
13.5. ENABLING IMA AND EVM	78
13.6. COLLECTING FILE HASHES WITH INTEGRITY MEASUREMENT ARCHITECTURE	81
CHAPTER 14. USING SYSTEMD TO MANAGE RESOURCES USED BY APPLICATIONS	83
14.1. ALLOCATING SYSTEM RESOURCES USING SYSTEMD	83
14.2. ROLE OF SYSTEMD IN RESOURCE MANAGEMENT	84

14.3. OVERVIEW OF SYSTEMD HIERARCHY FOR CGROUPS	84
14.4. LISTING SYSTEMD UNITS	86
14.5. VIEWING SYSTEMD CONTROL GROUP HIERARCHY	87
14.6. VIEWING CGROUPS OF PROCESSES	89
14.7. MONITORING RESOURCE CONSUMPTION	90
14.8. USING SYSTEMD UNIT FILES TO SET LIMITS FOR APPLICATIONS	90
14.9. USING SYSTEMCTL COMMAND TO SET LIMITS TO APPLICATIONS	91
14.10. SETTING GLOBAL DEFAULT CPU AFFINITY THROUGH MANAGER CONFIGURATION	92
14.11. CONFIGURING NUMA POLICIES USING SYSTEMD	93
14.12. NUMA POLICY CONFIGURATION OPTIONS FOR SYSTEMD	94
14.13. CREATING TRANSIENT CGROUPS USING SYSTEMD-RUN COMMAND	94
14.14. REMOVING TRANSIENT CONTROL GROUPS	95
CHAPTER 15. UNDERSTANDING CONTROL GROUPS	97
15.1. INTRODUCING CONTROL GROUPS	97
15.2. INTRODUCING KERNEL RESOURCE CONTROLLERS	98
15.3. INTRODUCING NAMESPACES	100
CHAPTER 16. USING CGROUPFS TO MANUALLY MANAGE CGROUPS	101
16.1. CREATING CGROUPS AND ENABLING CONTROLLERS IN CGROUPS-V2 FILE SYSTEM	101
16.2. CONTROLLING DISTRIBUTION OF CPU TIME FOR APPLICATIONS BY ADJUSTING CPU WEIGHT	103
16.3. MOUNTING CGROUPS-V1	106
16.4. SETTING CPU LIMITS TO APPLICATIONS USING CGROUPS-V1	108
CHAPTER 17. ANALYZING SYSTEM PERFORMANCE WITH BPF COMPILER COLLECTION	112
17.1. INSTALLING THE BCC-TOOLS PACKAGE	112
17.2. USING SELECTED BCC-TOOLS FOR PERFORMANCE ANALYSES	112
Using execsnoop to examine the system processes	112
Using opensnoop to track what files a command opens	113
Using btop to examine the I/O operations on the disk	114
Using xfslower to expose unexpectedly slow file system operations	115
CHAPTER 18. INSTALLING KDUMP	117
18.1. WHAT IS KDUMP	117
18.2. INSTALLING KDUMP USING ANACONDA	117
18.3. INSTALLING KDUMP ON THE COMMAND LINE	118
CHAPTER 19. CONFIGURING KDUMP ON THE COMMAND LINE	119
19.1. CONFIGURING KDUMP MEMORY USAGE ON RHEL 9	119
19.2. CONFIGURING THE KDUMP TARGET	121
19.3. CONFIGURING THE CORE COLLECTOR	123
19.4. CONFIGURING THE KDUMP DEFAULT FAILURE RESPONSES	124
19.5. CONFIGURATION FILE FOR KDUMP	125
19.6. ENABLING AND DISABLING THE KDUMP SERVICE	125
19.7. TESTING THE KDUMP CONFIGURATION	126
19.8. PREVENTING KERNEL DRIVERS FROM LOADING FOR KDUMP	127
19.9. RUNNING KDUMP ON SYSTEMS WITH ENCRYPTED DISK	128
CHAPTER 20. ENABLING KDUMP	130
20.1. ENABLING KDUMP FOR ALL INSTALLED KERNELS	130
20.2. ENABLING KDUMP FOR A SPECIFIC INSTALLED KERNEL	130
20.3. DISABLING THE KDUMP SERVICE	131
CHAPTER 21. SUPPORTED KDUMP CONFIGURATIONS AND TARGETS	133
21.1. MEMORY REQUIREMENTS FOR KDUMP	133

21.2. MINIMUM THRESHOLD FOR MEMORY RESERVATION	134
21.3. SUPPORTED KDUMP TARGETS	134
21.4. SUPPORTED KDUMP FILTERING LEVELS	135
21.5. SUPPORTED DEFAULT FAILURE RESPONSES	136
21.6. USING FINAL_ACTION PARAMETER	137
21.7. USING FAILURE_ACTION PARAMETER	137
CHAPTER 22. FIRMWARE ASSISTED DUMP MECHANISMS	138
22.1. FIRMWARE ASSISTED DUMP ON IBM POWERPC HARDWARE	138
22.2. ENABLING FIRMWARE ASSISTED DUMP MECHANISM	138
22.3. FIRMWARE ASSISTED DUMP MECHANISMS ON IBM Z HARDWARE	139
22.4. USING SADUMP ON FUJITSU PRIMEQUEST SYSTEMS	140
CHAPTER 23. ANALYZING A CORE DUMP	142
23.1. INSTALLING THE CRASH UTILITY	142
23.2. RUNNING AND EXITING THE CRASH UTILITY	142
23.3. DISPLAYING VARIOUS INDICATORS IN THE CRASH UTILITY	143
23.4. USING KERNEL OOPS ANALYZER	146
23.5. THE KDUMP HELPER TOOL	147

MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see [our CTO Chris Wright's message](#).

PROVIDING FEEDBACK ON RED HAT DOCUMENTATION

We appreciate your feedback on our documentation. Let us know how we can improve it.

Submitting comments on specific passages

1. View the documentation in the **Multi-page HTML** format and ensure that you see the **Feedback** button in the upper right corner after the page fully loads.
2. Use your cursor to highlight the part of the text that you want to comment on.
3. Click the **Add Feedback** button that appears near the highlighted text.
4. Add your feedback and click **Submit**.

Submitting feedback through Jira (account required)

1. Log in to the [Jira](#) website.
2. Click **Create** in the top navigation bar
3. Enter a descriptive title in the **Summary** field.
4. Enter your suggestion for improvement in the **Description** field. Include links to the relevant parts of the documentation.
5. Click **Create** at the bottom of the dialogue.

CHAPTER 1. THE LINUX KERNEL

Learn about the Linux kernel and the Linux kernel RPM package provided and maintained by Red Hat (Red Hat kernel). Keep the Red Hat kernel updated, which ensures the operating system has all the latest bug fixes, performance enhancements, and patches, and is compatible with new hardware.

1.1. WHAT THE KERNEL IS

The kernel is a core part of a Linux operating system that 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 easily upgraded and verified by the **dnf** package manager.



WARNING

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

1.2. RPM PACKAGES

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.

1.3. THE LINUX KERNEL RPM PACKAGE OVERVIEW

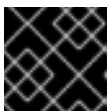
The **kernel** RPM is a meta package that does not contain any files, but rather ensures that the following required sub-packages are properly installed:

- **kernel-core** – contains the binary image of the Linux kernel (**vmlinuz**).
- **kernel-modules-core** – contains the basic kernel modules to ensure core functionality. This includes the modules essential for the proper functioning of the most commonly used hardware.
- **kernel-modules** – contains the remaining kernel modules that are not present in **kernel-core**.

The **kernel-core** and **kernel-modules-core** sub-packages together can be used in virtualized and cloud environments to provide a RHEL 9 kernel with a quick boot time and a small disk size footprint. **kernel-modules** sub-package is usually unnecessary for such deployments.

Optional kernel packages are for example:

- **kernel-modules-extra** – contains kernel modules for rare hardware and modules which loading is disabled by default.
- **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-stablelists** – contains information pertaining to the RHEL kernel ABI, including a list of kernel symbols that are needed by external Linux kernel modules and a **dnf** 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.
- **kernel-uki-virt** – contains the Unified Kernel Image (UKI) of the RHEL kernel. UKI combines the Linux kernel, **initramfs**, and the kernel command line into a single signed binary which can be booted directly from the UEFI firmware. **kernel-uki-virt** contains the required kernel modules to run in virtualized and cloud environments and can be used instead of the **kernel-core** sub-package.



IMPORTANT

kernel-uki-virt is provided as Technology Preview in RHEL 9.2.

Additional resources

- [What are the kernel-core, kernel-modules, and kernel-modules-extras packages?](#)

1.4. DISPLAYING CONTENTS OF THE KERNEL PACKAGE

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/5.14.0-1.el9.x86_64/kernel/fs/udf/udf.ko.xz
/lib/modules/5.14.0-1.el9.x86_64/kernel/fs/xfs
/lib/modules/5.14.0-1.el9.x86_64/kernel/fs/xfs/xfs.ko.xz
/lib/modules/5.14.0-1.el9.x86_64/kernel/kernel
/lib/modules/5.14.0-1.el9.x86_64/kernel/kernel/trace
/lib/modules/5.14.0-1.el9.x86_64/kernel/kernel/trace/ring_buffer_benchmark.ko.xz
/lib/modules/5.14.0-1.el9.x86_64/kernel/lib
/lib/modules/5.14.0-1.el9.x86_64/kernel/lib/cordic.ko.xz
...
```

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

```
$ rpm -qlp <kernel-modules_rpm>
...
/lib/modules/5.14.0-1.el9.x86_64/kernel/drivers/infiniband/hw/mlx4/mlx4_ib.ko.xz
/lib/modules/5.14.0-1.el9.x86_64/kernel/drivers/infiniband/hw/mlx5/mlx5_ib.ko.xz
/lib/modules/5.14.0-1.el9.x86_64/kernel/drivers/infiniband/hw/qedr/qedr.ko.xz
/lib/modules/5.14.0-1.el9.x86_64/kernel/drivers/infiniband/hw/usnic/usnic_verbs.ko.xz
/lib/modules/5.14.0-1.el9.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/5.14.0-1.el9.x86_64/extra/net/sched/sch_cbq.ko.xz
/lib/modules/5.14.0-1.el9.x86_64/extra/net/sched/sch_choke.ko.xz
/lib/modules/5.14.0-1.el9.x86_64/extra/net/sched/sch_drr.ko.xz
/lib/modules/5.14.0-1.el9.x86_64/extra/net/sched/sch_dsmark.ko.xz
/lib/modules/5.14.0-1.el9.x86_64/extra/net/sched/sch_gred.ko.xz
...
```

Additional resources

- The **rpm(8)** manual page
- [RPM packages](#)

1.5. INSTALLING SPECIFIC KERNEL VERSIONS

Install new kernels using the **dnf** package manager.

Procedure

- To install a specific kernel version, enter the following command:

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

Additional resources

- [Red Hat Code Browser](#)
- [Red Hat Enterprise Linux Release Dates](#)

1.6. UPDATING THE KERNEL

Update the kernel using the **dnf** package manager.

Procedure

1. To update the kernel, enter the following command:

```
# dnf 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.

Additional resources

- [package manager](#)
- The **dnf(8)** manual page

1.7. SETTING A KERNEL AS DEFAULT

Set a specific kernel as default using the **grubby** command-line tool and GRUB.

Procedure

- Setting the kernel as default, using the **grubby** tool
 - Enter 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 the **id** 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.

CHAPTER 2. THE 64K PAGE SIZE KERNEL

kernel-64k is an additional, optional 64-bit ARM architecture kernel package that supports 64k pages. This additional kernel exists alongside the RHEL 9 for ARM kernel which supports 4k pages.

Optimal system performance directly relates to different memory configuration requirements. These requirements are addressed by the two variants of kernel, each suitable for different workloads. RHEL 9 on 64-bit ARM hardware thus offers two MMU page sizes:

- 4k pages kernel for efficient memory usage in smaller environments,
- **kernel-64k** for workloads with large, contiguous memory working sets.

The 4k pages kernel and **kernel-64k** do not differ in the user experience as the user space is the same. You can choose the variant that addresses your situation the best.

4k pages kernel

Use 4k pages for more efficient memory usage in smaller environments, such as those in Edge and lower-cost, small cloud instances. In these environments, increasing the physical system memory amounts is not practical due to space, power, and cost constraints. Also, not all 64-bit ARM architecture processors support a 64k page size.

The 4k pages kernel supports graphical installation using Anaconda, system or cloud image-based installations, as well as advanced installations using Kickstart.

kernel-64k

The 64k page size kernel is a useful option for large datasets on ARM platforms. **kernel-64k** is suitable for memory-intensive workloads as it has significant gains in overall system performance, namely in large database, HPC, and high network performance.

You must choose page size on 64-bit ARM architecture systems at the time of installation. You can install **kernel-64k** only by Kickstart by adding the **kernel-64k** package to the package list in the **Kickstart** file.

Additional resources

- [Installing RHEL on ARM with Kernel-64k](#)

CHAPTER 3. MANAGING KERNEL MODULES

Learn about kernel modules, 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 9, 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. 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.

depmod

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.

weak-modules

In addition to **depmod**, Red Hat Enterprise Linux provides the **weak-modules** script shipped also with the **kmod** package. **weak-modules** determines which modules are kABI-compatible with installed

kernels. While checking modules kernel compatibility, **weak-modules** processes modules symbol dependencies from higher to lower release of kernel for which they were built. This means that **weak-modules** processes each module independently of kernel release they were built against.

Additional resources

- The **modules.dep(5)** manual page
- The **depmod(8)** manual page
- [What is the purpose of weak-modules script shipped with Red Hat Enterprise Linux?](#)
- [What is Kernel Application Binary Interface \(kABI\)?](#)

3.3. LISTING INSTALLED KERNEL MODULES

The **grubby --info=ALL** command displays an indexed list of installed kernels on **!BLS** and **BLS** installs.

Procedure

- List the installed kernels using the following command:

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

The list of all installed kernels is displayed as follows:

```
title="Red Hat Enterprise Linux (5.14.0-1.el9.x86_64) 9.0 (Plow)"
title="Red Hat Enterprise Linux (0-rescue-0d772916a9724907a5d1350bcd39ac92) 9.0
(Plow)"
```

The above example displays the installed kernels list of grubby-8.40-17, from the GRUB menu.

3.4. LISTING CURRENTLY LOADED KERNEL MODULES

View the currently loaded kernel modules.

Prerequisites

- The **kmod** package is installed.

Procedure

- To list all currently loaded kernel modules, enter:

```
$ 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

- The `/usr/share/doc/kmod/README` file
- The `lsmod(8)` manual page

3.5. DISPLAYING INFORMATION ABOUT KERNEL MODULES

Use the **modinfo** command to display some detailed information about the specified kernel module.

Prerequisites

- The **kmod** package is installed.

Procedure

- To display information about any kernel module, enter:

```
$ modinfo <KERNEL_MODULE_NAME>
```

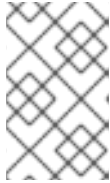
For example:

```
$ modinfo virtio_net

filename:    /lib/modules/5.14.0-1.el9.x86_64/kernel/drivers/net/virtio_net.ko.xz
license:     GPL
description: Virtio network driver
rhelversion: 9.0
srcversion:  8809CDDBE7202A1B00B9F1C
alias:       virtio:d00000001v*
depends:      net_failover
retpoline:   Y
intree:      Y
name:        virtio_net
vermagic:    5.14.0-1.el9.x86_64 SMP mod_unload modversions
```

```
...
parm:      napi_weight:int
parm:      csum:bool
parm:      gso:bool
parm:      napi_tx:bool
```

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

- The **modinfo(8)** manual page

3.6. LOADING KERNEL MODULES AT SYSTEM RUNTIME

The optimal way to expand the functionality of the Linux kernel is by loading kernel modules. 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

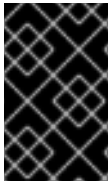
When entering the name of a kernel module, do not append the **.ko.xz** extension to the end of the name. Kernel module names do not have extensions; their corresponding files do.

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. For information about how to load kernel modules to **persist** across system reboots, see [Loading kernel modules automatically at system boot time](#).

Additional resources

- The **modprobe(8)** manual page

3.7. UNLOADING KERNEL MODULES AT SYSTEM RUNTIME

At times, you find that you need to unload certain kernel modules from the running kernel. 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. Enter 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 [Listing currently loaded kernel modules](#) and [Kernel module dependencies](#).
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

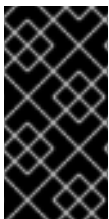
After finishing this procedure, the kernel modules that are defined to be automatically loaded on boot, **will not stay unloaded** after rebooting the system. For information about how to counter this outcome, see [Preventing kernel modules from being automatically loaded at system boot time](#).

Additional resources

- `modprobe(8)` manual page

3.8. UNLOADING KERNEL MODULES AT EARLY STAGES OF THE BOOT PROCESS

In certain situations it is necessary to unload a kernel module very early in the booting process. For example, when the kernel module contains a code, which causes the system to become unresponsive, and the user is not able to reach the stage to permanently disable the rogue kernel module. In that case it is possible to temporarily block the loading of the kernel module using a boot loader.



IMPORTANT

The changes described in this procedure **will not persist** after the next reboot. For information about how to add a kernel module to a denylist so that it will not be automatically loaded during the boot process, see [Preventing kernel modules from being automatically loaded at system boot time](#).

Prerequisites

- You have a loadable kernel module, which you want to prevent from loading for some reason.

Procedure

- Edit the relevant boot loader entry to unload the desired kernel module before the booting sequence continues.
 - Use the cursor keys to highlight the relevant boot loader entry.
 - Press **e** key to edit the entry.

Figure 3.1. Kernel boot menu

```

Red Hat Enterprise Linux (5.14.0-63.el9.x86_64) 9.0 (Plow)
Red Hat Enterprise Linux (5.14.0-1.7.1.el9.x86_64) 9.0 (Plow)
Red Hat Enterprise Linux (0-rescue-a36d6cc1dc7e4f59932e4352ddd01471) 9.0→

Use the ↑ and ↓ keys to change the selection.
Press 'e' to edit the selected item, or 'c' for a command prompt.

```

- Use the cursor keys to navigate to the line that starts with **linux**.
- Append **modprobe.blacklist=module_name** to the end of the line.

Figure 3.2. Kernel boot entry

```

load_video
set gfxpayload=keep
insmod gzio
linux ($root)/vmlinuz-5.14.0-63.el9.x86_64 root=/dev/mapper/rhel-root ro crash\
kernel=1G-4G:192M,4G-64G:256M,64G-:512M resume=/dev/mapper/rhel-swap rd.lvm.lv\
=rhel/root rd.lvm.lv=rhel/swap rhgb quiet modprobe.blacklist=serio_raw
initrd ($root)/initramfs-5.14.0-63.el9.x86_64.img

Press Ctrl-x to start, Ctrl-c for a command prompt or Escape to
discard edits and return to the menu. Pressing Tab lists
possible completions.

```

The **serio_raw** kernel module illustrates a rogue module to be unloaded early in the boot process.

- Press **CTRL+x** keys to boot using the modified configuration.

Verification

- Once the system fully boots, verify that the relevant kernel module is not loaded.

```
# lsmod | grep serio_raw
```

Additional resources

- [Managing kernel modules](#)

3.9. LOADING KERNEL MODULES AUTOMATICALLY AT SYSTEM BOOT TIME

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
```



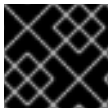
NOTE

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

- **modules-load.d(5)** manual page

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

You can prevent the system from loading a kernel module automatically during the boot process by listing the module in **modprobe** configuration file with a corresponding command.

Prerequisites

- The commands in this procedure require root privileges. Either use **su -** to switch to the root user or preface the commands with **sudo**.
- The **kmod** package is installed.
- Ensure that your current system configuration does not require a kernel module you plan to deny.

Procedure

1. List modules loaded to the currently running kernel by using the **lsmod** command:

```
$ lsmod
Module                Size  Used by
tls                   131072  0
uinput                20480   1
snd_seq_dummy         16384   0
snd_hrtimer           16384   1
...
```

In the output, identify the module you want to prevent from being loaded.

- Alternatively, identify an unloaded kernel module you want to prevent from potentially loading in the `/lib/modules/<KERNEL-VERSION>/kernel/<SUBSYSTEM>/` directory, for example:

```
$ ls /lib/modules/4.18.0-477.20.1.el8_8.x86_64/kernel/crypto/
ansi_cprng.ko.xz      chacha20poly1305.ko.xz  md4.ko.xz
serpent_generic.ko.xz
anubis.ko.xz          cmac.ko.xz...
```

2. Create a configuration file serving as a denylist:

```
# touch /etc/modprobe.d/denylist.conf
```

3. In a text editor of your choice, combine the names of modules you want to exclude from automatic loading to the kernel with the **blacklist** configuration command, for example:

```
# Prevents <KERNEL-MODULE-1> from being loaded
blacklist <MODULE-NAME-1>
install <MODULE-NAME-1> /bin/false

# Prevents <KERNEL-MODULE-2> from being loaded
blacklist <MODULE-NAME-2>
install <MODULE-NAME-2> /bin/false
...
```

Because the **blacklist** command does not prevent the module from being loaded as a dependency for another kernel module that is not in a denylist, you must also define the **install** line. In this case, the system runs **/bin/false** instead of installing the module. The lines starting with a hash sign are comments you can use 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.

4. Create a backup copy of the current initial RAM disk image before rebuilding:

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

- Alternatively, create a backup copy of an initial RAM disk image which corresponds to the kernel version for which you want to prevent kernel modules from automatic loading:

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

5. Generate a new initial RAM disk image to apply the changes:

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

- If you build an initial RAM disk image for a different kernel version than your system currently uses, specify both target **initramfs** and kernel version:

```
# dracut -f -v /boot/initramfs-<TARGET-VERSION>.img <CORRESPONDING-TARGET-KERNEL-VERSION>
```

6. Restart the system:

```
$ reboot
```



IMPORTANT

The changes described in this procedure **will take effect and persist** after rebooting the system. If you incorrectly list a key kernel module in the denylist, you can switch the system to an unstable or non-operational state.

Additional resources

- [How do I prevent a kernel module from loading automatically?](#) solution article
- **modprobe.d(5)** and **dracut(8)** man pages

3.11. COMPILING CUSTOM KERNEL MODULES

You can build a sampling kernel module as requested by various configurations at hardware and software level.

Prerequisites

- You installed the **kernel-devel**, **gcc**, and **elfutils-libelf-devel** packages.

```
# dnf install kernel-devel-$(uname -r) gcc elfutils-libelf-devel
```

- You have root permissions.
- You created the **/root/testmodule/** directory where you compile the custom kernel module.

Procedure

1. Create the **/root/testmodule/test.c** file with the following content.

```
#include <linux/module.h>
```

```
#include <linux/kernel.h>

int init_module(void)
{ printk("Hello World\n This is a test\n"); return 0; }

void cleanup_module(void)
{ printk("Good Bye World"); }

MODULE_LICENSE("GPL");
```

The **test.c** file is a source file that provides the main functionality to the kernel module. The file has been created in a dedicated **/root/testmodule/** directory for organizational purposes. After the module compilation, the **/root/testmodule/** directory will contain multiple files.

The **test.c** file includes from the system libraries:

- The **linux/kernel.h** header file is necessary for the **printk()** function in the example code.
- The **linux/module.h** file contains function declarations and macro definitions to be shared between several source files written in C programming language. Next follow the **init_module()** and **cleanup_module()** functions to start and end the kernel logging function **printk()**, which prints text.

2. Create the **/root/testmodule/Makefile** file with the following content.

```
obj-m := test.o
```

The Makefile contains instructions that the compiler has to produce an object file specifically named **test.o**. The **obj-m** directive specifies that the resulting **test.ko** file is going to be compiled as a loadable kernel module. Alternatively, the **obj-y** directive would instruct to build **test.ko** as a built-in kernel module.

3. Compile the kernel module.

```
# make -C /lib/modules/$(uname -r)/build M=/root/testmodule modules
make: Entering directory '/usr/src/kernels/5.14.0-70.17.1.el9_0.x86_64'
CC [M] /root/testmodule/test.o
MODPOST /root/testmodule/Module.symvers
CC [M] /root/testmodule/test.mod.o
LD [M] /root/testmodule/test.ko
BTF [M] /root/testmodule/test.ko
Skipping BTF generation for /root/testmodule/test.ko due to unavailability of vmlinux
make: Leaving directory '/usr/src/kernels/5.14.0-70.17.1.el9_0.x86_64'
```

The compiler creates an object file (**test.o**) for each source file (**test.c**) as an intermediate step before linking them together into the final kernel module (**test.ko**).

After a successful compilation, **/root/testmodule/** contains additional files that relate to the compiled custom kernel module. The compiled module itself is represented by the **test.ko** file.

Verification

1. Optional: check the contents of the **/root/testmodule/** directory:

```
# ls -l /root/testmodule/
```

```
total 152
-rw-r--r--. 1 root root  16 Jul 26 08:19 Makefile
-rw-r--r--. 1 root root  25 Jul 26 08:20 modules.order
-rw-r--r--. 1 root root   0 Jul 26 08:20 Module.symvers
-rw-r--r--. 1 root root 224 Jul 26 08:18 test.c
-rw-r--r--. 1 root root 62176 Jul 26 08:20 test.ko
-rw-r--r--. 1 root root  25 Jul 26 08:20 test.mod
-rw-r--r--. 1 root root  849 Jul 26 08:20 test.mod.c
-rw-r--r--. 1 root root 50936 Jul 26 08:20 test.mod.o
-rw-r--r--. 1 root root 12912 Jul 26 08:20 test.o
```

2. Copy the kernel module to the `/lib/modules/$(uname -r)/` directory:

```
# cp /root/testmodule/test.ko /lib/modules/$(uname -r)/
```

3. Update the modular dependency list:

```
# depmod -a
```

4. Load the kernel module:

```
# modprobe -v test
insmod /lib/modules/5.14.0-1.el9.x86_64/test.ko
```

5. Verify that the kernel module was successfully loaded:

```
# lsmod | grep test
test                16384  0
```

6. Read the latest messages from the kernel ring buffer:

```
# dmesg
[74422.545004] Hello World
                This is a test
```

Additional resources

- [Managing kernel modules](#)

CHAPTER 4. CONFIGURING KERNEL COMMAND-LINE PARAMETERS

With kernel command-line parameters, you can 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 can only be set at boot time, so understanding how to make these changes is a key administration skill.



IMPORTANT

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

4.1. WHAT ARE KERNEL COMMAND-LINE PARAMETERS

With kernel command-line parameters, you can overwrite default values and set specific hardware settings. At boot time, you can configure the following features:

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

By default, the kernel command-line parameters for systems using the GRUB boot loader are defined in the boot entry configuration file for each kernel boot entry.

You can manipulate boot loader configuration files by using the **grubby** utility. With **grubby**, you can perform these actions:

- Change the default boot entry.
- Add or remove arguments from a GRUB menu entry.

Additional resources

- **kernel-command-line(7)**, **bootparam(7)** and **dracut.cmdline(7)** manual pages
- [How to install and boot custom kernels in Red Hat Enterprise Linux 8](#)
- The **grubby(8)** manual page

4.2. UNDERSTANDING BOOT ENTRIES

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:

```
d8712ab6d4f14683c5625e87b52b6b6e-5.14.0-1.el9.x86_64.conf
```

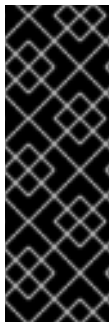
The file name above consists of a machine ID stored in the **/etc/machine-id** file, and a kernel version.

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

```
title Red Hat Enterprise Linux (5.14.0-1.el9.x86_64) 9.0 (Plow)
version 5.14.0-1.el9.x86_64
linux /vmlinuz-5.14.0-1.el9.x86_64
initrd /initramfs-5.14.0-1.el9.x86_64.img
options root=/dev/mapper/rhel_kvm--02--guest08-root ro crashkernel=1G-4G:192M,4G-64G:256M,64G-:512M resume=/dev/mapper/rhel_kvm--02--guest08-swap rd.lvm.lv=rhel_kvm-02-guest08/root rd.lvm.lv=rhel_kvm-02-guest08/swap console=ttyS0,115200
grub_users $grub_users
grub_arg --unrestricted
grub_class kernel
```

4.3. CHANGING KERNEL COMMAND-LINE PARAMETERS FOR ALL BOOT ENTRIES

Change kernel command-line parameters for all boot entries on your system.



IMPORTANT

When installing a newer version of the kernel in RHEL 9 systems, the **grubby** tool passes the kernel command-line arguments from the previous kernel version.

However, this does not apply to RHEL version 9.0 in which newly installed kernels lose previous command-line options. You must run the **grub2-mkconfig** command on the newly installed kernel to pass the parameters to your new kernel. For more information about this known issue, see [Boot loader](#).

Prerequisites

- Verify that the **grubby** utility is installed on your system.
- Verify that the **zipl** utility is installed on your IBM Z system.

Procedure

- To add a parameter:

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

For systems that use the GRUB boot loader and, on IBM Z that use the zipl boot loader, 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.

- After each update of your kernel package, propagate the configured kernel options to the new kernels:

```
# grub2-mkconfig -o /etc/grub2.cfg
```

Additional resources

- [What are kernel command-line parameters](#)
- **grubby(8)** and **zipl(8)** manual pages
- [grubby tool](#)

4.4. CHANGING KERNEL COMMAND-LINE PARAMETERS FOR A SINGLE BOOT ENTRY

Make changes in 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.



IMPORTANT

- **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

- [What are kernel command-line parameters](#)
- **grubby(8)** and **zipl(8)** manual pages
- [grubby tool](#)

4.5. CHANGING KERNEL COMMAND-LINE PARAMETERS TEMPORARILY AT BOOT TIME

Make temporary changes to a Kernel Menu Entry by changing the kernel parameters only during a single boot process.

Procedure

1. Select the kernel you want to start when the GRUB 2 boot menu appears and press the **e** key to edit the kernel parameters.
2. Find the kernel command line by moving the cursor down. The kernel command line starts with **linux** on 64-Bit IBM Power Series and x86-64 BIOS-based systems, or **linuxefi** on UEFI systems.
3. Move the cursor to the end of the line.



NOTE

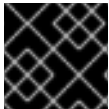
Press **Ctrl+a** to jump to the start of the line and **Ctrl+e** to jump to the end of the line. On some systems, **Home** and **End** keys might also work.

4. Edit the kernel parameters as required. For example, to run the system in emergency mode, add the *emergency* parameter at the end of the **linux** line:

```
linux ($root)/vmlinuz-5.14.0-63.el9.x86_64 root=/dev/mapper/rhel-root ro crashkernel=1G-4G:192M,4G-64G:256M,64G-:512M resume=/dev/mapper/rhel-swap rd.lvm.lv=rhel/root rd.lvm.lv=rhel/swap rhgb quiet pass:quotes[_emergency_]
```

+ To enable the system messages, remove the **rhgb** and **quiet** parameters.

1. Press **Ctrl+x** to boot with the selected kernel and the modified command line parameters.



IMPORTANT

Press **Esc** key to leave command line editing and it will drop all the user made changes.



NOTE

This procedure applies only for a single boot and does not persistently make the changes.

4.6. CONFIGURING GRUB SETTINGS TO ENABLE SERIAL CONSOLE CONNECTION

The serial console is beneficial when you need to connect to a headless server or an embedded system and the network is down. Or when you need to avoid security rules and obtain login access on a different system.

You need to configure some default GRUB settings to use the serial console connection.

Prerequisites

- You have root permissions.

Procedure

1. Add the following two lines to the **/etc/default/grub** file:

```
GRUB_TERMINAL="serial"  
GRUB_SERIAL_COMMAND="serial --speed=9600 --unit=0 --word=8 --parity=no --stop=1"
```

The first line disables the graphical terminal. The **GRUB_TERMINAL** key overrides values of **GRUB_TERMINAL_INPUT** and **GRUB_TERMINAL_OUTPUT** keys.

The second line adjusts the baud rate (**--speed**), parity and other values to fit your environment and hardware. Note that a much higher baud rate, for example 115200, is preferable for tasks such as following log files.

2. Update the GRUB configuration file.

- On BIOS-based machines:

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

- On UEFI-based machines:

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

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

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. 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



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.

Additional resources

- **sysctl(8)**, and **sysctl.d(5)** manual pages

5.2. CONFIGURING KERNEL PARAMETERS TEMPORARILY WITH SYSCTL

Use the **sysctl** command to temporarily set kernel parameters at runtime. The command is also useful for listing and filtering tunables.

Prerequisites

- Root permissions

Procedure

1. List all parameters and their values.

```
# 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, enter:

```
# 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

- The **sysctl(8)** manual page
- [Configuring kernel parameters permanently with sysctl](#)
- [Using configuration files in /etc/sysctl.d/ to adjust kernel parameters](#)

5.3. CONFIGURING KERNEL PARAMETERS PERMANENTLY WITH SYSCTL

Use the **sysctl** command to permanently set kernel parameters.

Prerequisites

- Root permissions

Procedure

1. List all parameters.

```
# sysctl -a
```

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

2. 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

- The **sysctl(8)** and **sysctl.conf(5)** manual pages
- [Using configuration files in /etc/sysctl.d/ to adjust kernel parameters](#)

5.4. USING CONFIGURATION FILES IN /ETC/SYSCTL.D/ TO ADJUST KERNEL PARAMETERS

Modify configuration files in the **/etc/sysctl.d/** directory manually to permanently set kernel parameters.

Prerequisites

- 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.

```
<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, enter:

```
# 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

- **sysctl(8), sysctl.d(5)** manual pages

5.5. CONFIGURING KERNEL PARAMETERS TEMPORARILY THROUGH /PROC/SYS/

Set kernel parameters temporarily through the files in the **/proc/sys/** virtual file system directory.

Prerequisites

- 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

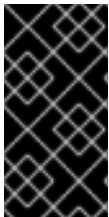
- [Configuring kernel parameters permanently with sysctl](#)
- [Using configuration files in /etc/sysctl.d/ to adjust kernel parameters](#)

CHAPTER 6. CONFIGURING KERNEL PARAMETERS PERMANENTLY BY USING THE `KERNEL_SETTINGS` RHEL SYSTEM ROLE

You can use the **`kernel_settings`** role to configure kernel parameters on multiple clients at once. This solution:

- Provides a friendly interface with efficient input setting.
- Keeps all intended kernel parameters in one place.

After you run the **`kernel_settings`** role from the control machine, the kernel parameters are applied to the managed systems immediately and persist across reboots.



IMPORTANT

Note that RHEL System Role delivered over RHEL channels are available to RHEL customers as an RPM package in the default AppStream repository. RHEL System Role are also available as a collection to customers with Ansible subscriptions over Ansible Automation Hub.

6.1. INTRODUCTION TO THE `KERNEL_SETTINGS` ROLE

RHEL System Roles is a set of roles that provide a consistent configuration interface to remotely manage multiple systems.

RHEL System Roles were introduced for automated configurations of the kernel using the **`kernel_settings`** System Role. The **`rhel-system-roles`** package contains this system role, and also the reference documentation.

To apply the kernel parameters on one or more systems in an automated fashion, use the **`kernel_settings`** role with one or more of its role variables of your choice in a playbook. A playbook is a list of one or more plays that are human-readable, and are written in the YAML format.

With the **`kernel_settings`** role you can configure:

- The kernel parameters using the **`kernel_settings_sysctl`** role variable
- Various kernel subsystems, hardware devices, and device drivers using the **`kernel_settings_sysfs`** role variable
- The CPU affinity for the **`systemd`** service manager and processes it forks using the **`kernel_settings_systemd_cpu_affinity`** role variable
- The kernel memory subsystem transparent hugepages using the **`kernel_settings_transparent_hugepages`** and **`kernel_settings_transparent_hugepages_defrag`** role variables

Additional resources

- **`README.md`** and **`README.html`** files in the `/usr/share/doc/rhel-system-roles/kernel_settings/` directory
- [Working with playbooks](#)

- [How to build your inventory](#)

6.2. APPLYING SELECTED KERNEL PARAMETERS USING THE KERNEL_SETTINGS ROLE

Follow these steps to prepare and apply an Ansible playbook to remotely configure kernel parameters with persisting effect on multiple managed operating systems.

Prerequisites

- You have **root** permissions.
- Entitled by your RHEL subscription, you installed the **ansible-core** and **rhel-system-roles** packages on the control machine.
- An inventory of managed hosts is present on the control machine and Ansible is able to connect to them.



IMPORTANT

RHEL 8.0 - 8.5 provided access to a separate Ansible repository that contains Ansible Engine 2.9 for automation based on Ansible. Ansible Engine contains command-line utilities such as **ansible**, **ansible-playbook**; connectors such as **docker** and **podman**; and the entire world of plugins and modules. For information about how to obtain and install Ansible Engine, refer to [How do I Download and Install Red Hat Ansible Engine?](#) .

RHEL 8.6 and 9.0 has introduced Ansible Core (provided as **ansible-core** RPM), which contains the Ansible command-line utilities, commands, and a small set of built-in Ansible plugins. The AppStream repository provides **ansible-core**, which has a limited scope of support. You can learn more by reviewing [Scope of support for the ansible-core package included in the RHEL 9 AppStream](#).

Procedure

1. Optionally, review the **inventory** file for illustration purposes:

```
# cat /home/jdoe/<ansible_project_name>/inventory
[testingservers]
pdoe@192.168.122.98
fdoe@192.168.122.226

[db-servers]
db1.example.com
db2.example.com

[webservers]
web1.example.com
web2.example.com
192.0.2.42
```

The file defines the **[testingservers]** group and other groups. It allows you to run Ansible more effectively against a specific set of systems.

2. Create a configuration file to set defaults and privilege escalation for Ansible operations.

- a. Create a new YAML file and open it in a text editor, for example:

```
# vi /home/jdoe/<ansible_project_name>/ansible.cfg
```

- b. Insert the following content into the file:

```
[defaults]
inventory = ./inventory

[privilege_escalation]
become = true
become_method = sudo
become_user = root
become_ask_pass = true
```

The **[defaults]** section specifies a path to the inventory file of managed hosts. The **[privilege_escalation]** section defines that user privileges be shifted to **root** on the specified managed hosts. This is necessary for successful configuration of kernel parameters. When Ansible playbook is run, you will be prompted for user password. The user automatically switches to **root** by means of **sudo** after connecting to a managed host.

3. Create an Ansible playbook that uses the **kernel_settings** role.

- a. Create a new YAML file and open it in a text editor, for example:

```
# vi /home/jdoe/<ansible_project_name>/kernel-roles.yml
```

This file represents a playbook and usually contains an ordered list of tasks, also called *plays*, that are run against specific managed hosts selected from your **inventory** file.

- b. Insert the following content into the file:

```
---
-
  hosts: testingservers
  name: "Configure kernel settings"
  roles:
    - rhel-system-roles.kernel_settings
  vars:
    kernel_settings_sysctl:
      - name: fs.file-max
        value: 400000
      - name: kernel.threads-max
        value: 65536
    kernel_settings_sysfs:
      - name: /sys/class/net/lo/mtu
        value: 65000
    kernel_settings_transparent_hugepages: madvise
```

The **name** key is optional. It associates an arbitrary string with the play as a label and identifies what the play is for. The **hosts** key in the play specifies the hosts against which the play is run. The value or values for this key can be provided as individual names of managed hosts or as groups of hosts as defined in the **inventory** file.

The **vars** section represents a list of variables containing selected kernel parameter names and values to which they have to be set.

The **roles** key specifies what system role is going to configure the parameters and values mentioned in the **vars** section.



NOTE

You can modify the kernel parameters and their values in the playbook to fit your needs.

4. Optionally, verify that the syntax in your play is correct.

```
# ansible-playbook --syntax-check kernel-roles.yml

playbook: kernel-roles.yml
```

This example shows the successful verification of a playbook.

5. Execute your playbook.

```
# ansible-playbook kernel-roles.yml

...

BECOME password:

PLAY [Configure kernel settings]
*****

PLAY RECAP
*****

fdoe@192.168.122.226    : ok=10  changed=4  unreachable=0  failed=0  skipped=6
rescued=0  ignored=0
pdoe@192.168.122.98    : ok=10  changed=4  unreachable=0  failed=0  skipped=6
rescued=0  ignored=0
```

Before Ansible runs your playbook, you are going to be prompted for your password and so that a user on managed hosts can be switched to **root**, which is necessary for configuring kernel parameters.

The recap section shows that the play finished successfully (**failed=0**) for all managed hosts, and that 4 kernel parameters have been applied (**changed=4**).

6. Restart your managed hosts and check the affected kernel parameters to verify that the changes have been applied and persist across reboots.

Additional resources

- [Preparing a control node and managed nodes to use RHEL System Roles](#)
- **README.html** and **README.md** files in the `/usr/share/doc/rhel-system-roles/kernel_settings/` directory

- [Build Your Inventory](#)
- [Configuring Ansible](#)
- [Working With Playbooks](#)
- [Using Variables](#)
- [Roles](#)

CHAPTER 7. 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

[Limitations of kpatch](#) carefully before using kernel live patching.

7.1. 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.

7.2. 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?](#)

7.3. 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 9.1 kernel until the RHEL 9.2 kernel is released.

7.4. 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* 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.

The kpatch-dnf package

A DNF plugin delivered in the form of an RPM package. This plugin manages automatic subscription to kernel live patches.

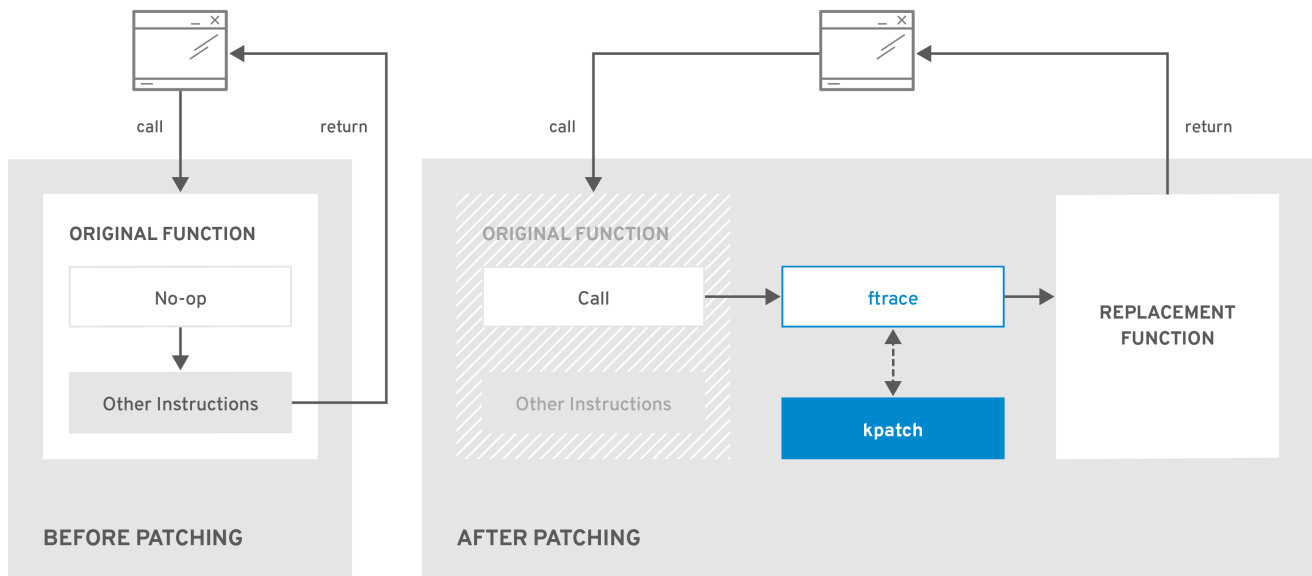
7.5. HOW KERNEL LIVE PATCHING WORKS

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.

- 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 7.1. How kernel live patching works



RHEL_424549_0119

7.6. SUBSCRIBING THE CURRENTLY INSTALLED KERNELS TO THE LIVE PATCHING STREAM

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 procedure explains how to subscribe to all future cumulative live patching updates for a given kernel. Because live patches are cumulative, you cannot select which individual patches are deployed for a given kernel.



WARNING

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

Prerequisites

- Root permissions

Procedure

- Optionally, check your kernel version:

```
# uname -r
5.14.0-1.el9.x86_64
```

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

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

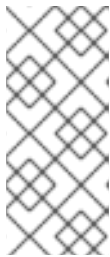
3. Install the live patching package:

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

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

If the version of a live patching package is 1-1 or higher, the package will contain a patch module. 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

An empty live patching package will be installed when there are no live patches available for a given kernel. An empty live patching package will have a *kpatch_version-kpatch_release* of 0-0, for example **kpatch-patch-5_14_0-1-0-0.x86_64.rpm**. The installation of the empty RPM subscribes the system to all future live patches for the given kernel.

Verification step

- Verify that all installed kernels have been patched:

```
# kpatch list
```

Loaded patch modules:

```
kpatch_5_14_0_1_0_1 [enabled]
```

Installed patch modules:

```
kpatch_5_14_0_1_0_1 (5.14.0-1.el9.x86_64)
```

```
...
```

The output shows that the kernel patch module has been loaded into the kernel that is now patched with the latest fixes from the **kpatch-patch-5_14_0-1-0-1.el9.x86_64.rpm** package.



NOTE

Entering the **kpatch list** command does not return an empty live patching package. Use the **rpm -qa | grep kpatch** command instead.

```
# rpm -qa | grep kpatch
```

```
kpatch-dnf-0.4-3.el9.noarch
```

```
kpatch-0.9.7-2.el9.noarch
```

```
kpatch-patch-5_14_0-284_25_1-0-0.el9_2.x86_64
```

Additional resources

- **kpatch(1)** manual page
- [Installing RHEL 9 content](#)

7.7. AUTOMATICALLY SUBSCRIBING ANY FUTURE KERNEL TO THE LIVE PATCHING STREAM

You can use the **kpatch-dnf** DNF plugin to subscribe your system to fixes delivered by the kernel patch module, also known as kernel live patches. The plugin enables **automatic** subscription for any kernel the system currently uses, and also for kernels **to-be-installed in the future**

Prerequisites

- You have root permissions.

Procedure

1. Optionally, check all installed kernels and the kernel you are currently running:

```
# dnf list installed | grep kernel
Updating Subscription Management repositories.
Installed Packages
...
kernel-core.x86_64      5.14.0-1.el9      @beaker-BaseOS
kernel-core.x86_64      5.14.0-2.el9      @@commandline
...

# uname -r
5.14.0-2.el9.x86_64
```

2. Install the **kpatch-dnf** plugin:

```
# dnf install kpatch-dnf
```

3. Enable automatic subscription to kernel live patches:

```
# dnf kpatch auto
Updating Subscription Management repositories.
Last metadata expiration check: 1:38:21 ago on Fri 17 Sep 2021 07:29:53 AM EDT.
Dependencies resolved.
=====
Package                                Architecture
=====
Installing:
kpatch-patch-5_14_0-1                  x86_64
kpatch-patch-5_14_0-2                  x86_64

Transaction Summary
=====
Install 2 Packages
...
```


This command subscribes all currently installed kernels to receiving kernel live patches. The command also installs and applies the latest cumulative live patches, if any, for all installed kernels.

In the future, when you update the kernel, live patches will automatically be installed during the new kernel installation process.

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



NOTE

An empty live patching package will be installed when there are no live patches available for a given kernel. An empty live patching package will have a *kpatch_version-kpatch_release* of 0-0, for example **kpatch-patch-5_14_0-1-0-0.el9.x86_64.rpm**. The installation of the empty RPM subscribes the system to all future live patches for the given kernel.

Verification step

- Verify that all installed kernels have been patched:

kpatch list

Loaded patch modules:

kpatch_5_14_0_2_0_1 [enabled]

Installed patch modules:

kpatch_5_14_0_1_0_1 (5.14.0-1.el9.x86_64)

kpatch_5_14_0_2_0_1 (5.14.0-2.el9.x86_64)

The output shows that both the kernel you are running, and the other installed kernel have been patched with fixes from **kpatch-patch-5_14_0-1-0-1.el9.x86_64.rpm** and **kpatch-patch-5_14_0-2-0-1.el9.x86_64.rpm** packages respectively.



NOTE

Entering the **kpatch list** command does not return an empty live patching package. Use the **rpm -qa | grep kpatch** command instead.

rpm -qa | grep kpatch

kpatch-dnf-0.4-3.el9.noarch

kpatch-0.9.7-2.el9.noarch

kpatch-patch-5_14_0-284_25_1-0-0.el9_2.x86_64

Additional resources

- **kpatch(1)** and **dnf-kpatch(8)** manual pages
- [Configuring basic system settings](#) in RHEL

7.8. DISABLING AUTOMATIC SUBSCRIPTION TO THE LIVE PATCHING STREAM

When you subscribe your system to fixes delivered by the kernel patch module, your subscription is **automatic**. You can disable this feature, and thus disable automatic installation of **kpatch-patch** packages.

Prerequisites

- You have root permissions.

Procedure

1. Optionally, check all installed kernels and the kernel you are currently running:

```
# dnf list installed | grep kernel
Updating Subscription Management repositories.
Installed Packages
...
kernel-core.x86_64      5.14.0-1.el9      @beaker-BaseOS
kernel-core.x86_64      5.14.0-2.el9      @@commandline
...

# uname -r
5.14.0-2.el9.x86_64
```

2. Disable automatic subscription to kernel live patches:

```
# dnf kpatch manual
Updating Subscription Management repositories.
```

Verification step

- You can check for the successful outcome:

```
# yum kpatch status
...
Updating Subscription Management repositories.
Last metadata expiration check: 0:30:41 ago on Tue Jun 14 15:59:26 2022.
Kpatch update setting: manual
```

Additional resources

- **kpatch(1)** and **dnf-kpatch(8)** manual pages

7.9. 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 [Subscribing the currently installed kernels to the live patching stream](#).

Procedure

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

```
# dnf 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:

```
# dnf 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

- [Configuring basic system settings](#) in RHEL

7.10. REMOVING THE LIVE PATCHING PACKAGE

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.

```
# dnf list installed | grep kpatch-patch
kpatch-patch-5_14_0-1.x86_64    0-1.el9    @@commandline
...
```

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

2. Remove the live patching package.

```
# dnf remove kpatch-patch-5_14_0-1.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. On future reboot, the corresponding kernel will no longer be patched.

3. Reboot your system.
4. Verify that the live patching package has been removed.

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

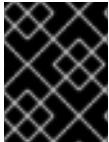
The command displays no output if the package has been successfully removed.

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.



IMPORTANT

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

Additional resources

- The **kpatch(1)** manual page
- [Configuring basic system settings](#) in RHEL

7.11. UNINSTALLING THE KERNEL PATCH MODULE

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_5_14_0_1_0_1 [enabled]

Installed patch modules:

kpatch_5_14_0_1_0_1 (5.14.0-1.el9.x86_64)

...

2. Uninstall the selected kernel patch module.

```
# kpatch uninstall kpatch_5_14_0_1_0_1
```

uninstalling kpatch_5_14_0_1_0_1 (5.14.0-1.el9.x86_64)

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

```
# kpatch list
Loaded patch modules:
kpatch_5_14_0_1_0_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.



IMPORTANT

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

Additional resources

- The **kpatch(1)** manual page

7.12. DISABLING KPATCH.SERVICE

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_5_14_0_1_0_1 [enabled]

Installed patch modules:

kpatch_5_14_0_1_0_1 (5.14.0-1.el9.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:

Installed patch modules:

kpatch_5_14_0_1_0_1 (5.14.0-1.el9.x86_64)

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

**IMPORTANT**

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

Additional resources

- The **kpatch(1)** manual page
- [Managing system services with systemctl](#)

CHAPTER 8. KEEPING KERNEL PANIC PARAMETERS DISABLED IN VIRTUALIZED ENVIRONMENTS

When configuring a virtualized environment in RHEL 9, you should not enable the **softlockup_panic** and **nmi_watchdog** kernel parameters, because the virtualized environment may trigger a spurious soft lockup that should not require a system panic.

Find the reasons behind this advice in the following sections.

8.1. 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

- [What is a CPU soft lockup?](#)

8.2. 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

- [Softlockup detector and hardlockup detector](#)
- [Kernel sysctl](#)

8.3. SPURIOUS SOFT LOCKUPS IN VIRTUALIZED ENVIRONMENTS

The soft lockup firing on physical hosts, as described in [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

- [What is a soft lockup](#)
- [Virtual machine components and their interaction](#)
- [Virtual machine reports a "BUG: soft lockup"](#)

CHAPTER 9. ADJUSTING KERNEL PARAMETERS FOR DATABASE SERVERS

There are different sets of kernel parameters which can affect performance of specific database applications. To secure efficient operation of database servers and databases, configure the respective kernel parameters accordingly.

9.1. INTRODUCTION TO DATABASE SERVERS

A database server is a service that provides features of a database management system (DBMS). DBMS provides utilities for database administration and interacts with end users, applications, and databases.

Red Hat Enterprise Linux 9 provides the following database management systems:

- MariaDB 10.5
- MySQL 8.0
- PostgreSQL 13
- Redis 6

9.2. 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 \leq 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

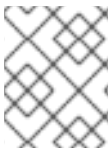
Defines a threshold in bytes / in percentage of dirty-able memory at which a process generating dirty data is started in the **write()** function.

**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

- [Dirty pagecache writeback and vm.dirty parameters](#)

CHAPTER 10. 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.

10.1. 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

- **syslog(2)** and **dmesg(1)** manual page

10.2. 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 [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 9 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.



IMPORTANT

The default **7 4 1 7****printk** value allows for better debugging of kernel activity. However, when coupled with a serial console, this **printk** setting is able to cause intense I/O bursts that could lead to a RHEL system becoming temporarily unresponsive. To avoid these situations, setting a **printk** value of **4 4 1 7** typically works, but at the expense of losing the extra debugging information.

Also note that certain kernel command line parameters, such as **quiet** or **debug**, change the default **kernel.printk** values.

Additional resources

- **syslog(2)** manual page

CHAPTER 11. SIGNING A KERNEL AND MODULES FOR SECURE BOOT

You can enhance the security of your system by using a signed kernel and signed kernel modules. On UEFI-based build systems where Secure Boot is enabled, you can self-sign a privately built kernel or kernel modules. Furthermore, you can import your public key into a target system where you want to deploy your kernel or kernel modules.

If Secure Boot is enabled, all of the following components have to be signed with a private key and authenticated with the corresponding public key:

- UEFI operating system boot loader
- The Red Hat Enterprise Linux kernel
- All kernel modules

If any of these components are not signed and authenticated, the system cannot finish the booting process.

Red Hat Enterprise Linux 9 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 Red Hat Enterprise Linux 9 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

- Not all UEFI-based systems include support for Secure Boot.
- 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.

11.1. PREREQUISITES

- To be able to sign externally built kernel modules, install the utilities from the following packages:

```
# dnf install pesign openssl kernel-devel mokutil keyutils
```

Table 11.1. Required utilities

Utility	Provided by package	Used on	Purpose
efikeygen	pesign	Build system	Generates public and private X.509 key pair

Utility	Provided by package	Used on	Purpose
openssl	openssl	Build system	Exports the unencrypted private key
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

11.2. WHAT IS UEFI SECURE BOOT

With the *Unified Extensible Firmware Interface* (UEFI) Secure Boot technology, you can prevent the execution of the kernel-space code that has not been signed by a trusted key. The system boot loader is signed with a cryptographic key. The database of public keys, which is contained in the firmware, authorizes the signing key. You can subsequently verify a signature in the next-stage boot loader and the kernel.

UEFI Secure Boot establishes a chain of trust from the firmware to the signed drivers and kernel modules as follows:

- An UEFI private key signs, and a public key authenticates the **shim** first-stage boot loader. A *certificate authority* (CA) in turn signs the public key. The CA is stored in the firmware database.
- The **shim** file contains the Red Hat public key **Red Hat Secure Boot (CA key 1)** to authenticate the GRUB boot loader and the kernel.
- The kernel in turn contains public keys to authenticate drivers and modules.

Secure Boot is the boot path validation component of the UEFI specification. The specification defines:

- Programming interface for cryptographically protected UEFI variables in non-volatile storage.
- Storing the trusted X.509 root certificates in UEFI variables.
- Validation of UEFI applications such as boot loaders and drivers.
- Procedures to revoke known-bad certificates and application hashes.

UEFI Secure Boot helps in the detection of unauthorized changes but does **not**:

- Prevent installation or removal of second-stage boot loaders.
- Require explicit user confirmation of such changes.
- Stop boot path manipulations. Signatures are verified during booting, not when the boot loader is installed or updated.

If the boot loader or the kernel are not signed by a system trusted key, Secure Boot prevents them from starting.

11.3. UEFI SECURE BOOT SUPPORT

You can install and run Red Hat Enterprise Linux 9 on systems with enabled UEFI Secure Boot if the kernel and all the loaded drivers are signed with a trusted key. Red Hat provides kernels and drivers that are signed and authenticated by the relevant Red Hat keys.

If you want to load externally built kernels or drivers, you must sign them as well.

Restrictions imposed by UEFI Secure Boot

- The system only runs the kernel-mode code after its signature has been properly authenticated.
- GRUB module loading is disabled because there is no infrastructure for signing and verification of GRUB modules. Allowing them to be loaded constitutes execution of untrusted code inside the security perimeter that Secure Boot defines.
- Red Hat provides a signed GRUB binary that contains all the supported modules on Red Hat Enterprise Linux 9.

Additional resources

- [Restrictions Imposed by UEFI Secure Boot](#)

11.4. REQUIREMENTS FOR AUTHENTICATING KERNEL MODULES WITH X.509 KEYS

In Red Hat Enterprise Linux 9, when a kernel module is loaded, the kernel checks the signature of the module against the public X.509 keys from the kernel system keyring (**.builtin_trusted_keys**) and the kernel platform keyring (**.platform**). The **.platform** keyring contains keys from third-party platform providers and custom public keys. The keys from the kernel system **.blacklist** keyring are excluded from verification.

You need to meet certain conditions to load 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 those signed kernel modules whose signatures were authenticated against keys from the system keyring (**.builtin_trusted_keys**) and the platform keyring (**.platform**).
 - The public key must not be on the system revoked keys keyring (**.blacklist**).
- If UEFI Secure Boot is disabled and the **module.sig_enforce** kernel parameter has not been specified:
 - You can load unsigned kernel modules and signed kernel modules without a public key.
- If the system is not UEFI-based or if UEFI Secure Boot is disabled:

- Only the keys embedded in the kernel are loaded onto **.builtin_trusted_keys** and **.platform**.
- You have no ability to augment that set of keys without rebuilding the kernel.

Table 11.2. 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

11.5. SOURCES FOR PUBLIC KEYS

During boot, the kernel loads X.509 keys from a set of persistent key stores into the following keyrings:

- The system keyring (**.builtin_trusted_keys**)
- The **.platform** keyring
- The system **.blacklist** keyring

Table 11.3. Sources for system keyrings

Source of X.509 keys	User can add keys	UEFI Secure Boot state	Keys loaded during boot
Embedded in kernel	No	-	.builtin_trusted_keys
UEFI db	Limited	Not enabled	No
		Enabled	.platform

Source of X.509 keys	User can add keys	UEFI Secure Boot state	Keys loaded during boot
Embedded in the shim boot loader	No	Not enabled	No
		Enabled	.platform
Machine Owner Key (MOK) list	Yes	Not enabled	No
		Enabled	.platform

.builtin_trusted_keys

- A keyring that is built on boot
- Contains trusted public keys
- **root** privileges are needed to view the keys

.platform

- A keyring that is built on boot
- Contains keys from third-party platform providers and custom public keys
- **root** privileges are needed to view the keys

.blacklist

- A keyring with X.509 keys which have been revoked
- A module signed by a key from **.blacklist** will fail authentication even if your public key is in **.builtin_trusted_keys**

UEFI Secure Bootdb

- A signature database
- Stores keys (hashes) of UEFI applications, UEFI drivers, and boot loaders
- The keys can be loaded on the machine

UEFI Secure Bootdbx

- A revoked signature database
- Prevents keys from being loaded
- The revoked keys from this database are added to the **.blacklist** keyring

11.6. GENERATING A PUBLIC AND PRIVATE KEY PAIR

To use a custom kernel or custom kernel modules on a Secure Boot-enabled system, you must generate a public and private X.509 key pair. You can use the generated private key to sign the kernel or the kernel modules. You can also validate the signed kernel or kernel modules by adding the corresponding public key to the Machine Owner Key (MOK) for Secure Boot.



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.

Procedure

- Create an X.509 public and private key pair:
 - If you only want to sign custom kernel *modules*:

```
# efkeygen --dbdir /etc/pki/pesign \  
--self-sign \  
--module \  
--common-name 'CN=Organization signing key' \  
--nickname 'Custom Secure Boot key'
```

- If you want to sign custom *kernel*:

```
# efkeygen --dbdir /etc/pki/pesign \  
--self-sign \  
--kernel \  
--common-name 'CN=Organization signing key' \  
--nickname 'Custom Secure Boot key'
```

- When the RHEL system is running FIPS mode:

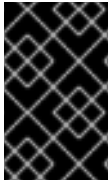
```
# efkeygen --dbdir /etc/pki/pesign \  
--self-sign \  
--kernel \  
--common-name 'CN=Organization signing key' \  
--nickname 'Custom Secure Boot key' \  
--token 'NSS FIPS 140-2 Certificate DB'
```



NOTE

In FIPS mode, you must use the **--token** option so that **efkeygen** finds the default "NSS Certificate DB" token in the PKI database.

The public and private keys are now stored in the **/etc/pki/pesign/** directory.



IMPORTANT

It is a good security practice to sign the kernel and the kernel modules within the validity period of its signing key. However, the **sign-file** utility does not warn you and the key will be usable in Red Hat Enterprise Linux 9 regardless of the validity dates.

Additional resources

- **openssl(1)** manual page
- [RHEL Security Guide](#)
- [Enrolling public key on target system by adding the public key to the MOK list](#)

11.7. EXAMPLE OUTPUT OF SYSTEM KEYRINGS

You can display information about the keys on the system keyrings using the **keyctl** utility from the **keyutils** package.

Prerequisites

- You have root permissions.
- You have installed the **keyctl** utility from the **keyutils** package.

Example 11.1. Keyrings output

The following is a shortened example output of **.builtin_trusted_keys**, **.platform**, and **.blacklist** keyrings from a Red Hat Enterprise Linux 9 system where UEFI Secure Boot is enabled.

```
# keyctl list %:.builtin_trusted_keys
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...

# keyctl list %:.platform
4 keys in keyring:
...asymmetric: VMware, Inc.: 4ad8da0472073...
...asymmetric: Red Hat Secure Boot CA 5: cc6fafa72...
...asymmetric: Microsoft Windows Production PCA 2011: a929f298e1...
...asymmetric: Microsoft Corporation UEFI CA 2011: 13adbf4e0bd82...

# keyctl list %:.blacklist
4 keys in keyring:
...blacklist: bin:f5ff83a...
...blacklist: bin:0dfdbec...
...blacklist: bin:38f1d22...
...blacklist: bin:51f831f...
```

The **.builtin_trusted_keys** keyring in the example 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** boot loader.

Example 11.2. Kernel console output

The following example shows the kernel console output. The messages identify the keys with an UEFI Secure Boot related source. These include UEFI Secure Boot **db**, embedded **shim**, and MOK list.

```
# dmesg | egrep 'integrity.*cert'
[1.512966] integrity: Loading X.509 certificate: UEFI:db
[1.513027] integrity: Loaded X.509 cert 'Microsoft Windows Production PCA 2011: a929023...
[1.513028] integrity: Loading X.509 certificate: UEFI:db
[1.513057] integrity: Loaded X.509 cert 'Microsoft Corporation UEFI CA 2011: 13adbf4309...
[1.513298] integrity: Loading X.509 certificate: UEFI:MokListRT (MOKvar table)
[1.513549] integrity: Loaded X.509 cert 'Red Hat Secure Boot CA 5: cc6fa5e72868ba494e93...
```

Additional resources

- **keyctl(1)**, **dmesg(1)** manual pages

11.8. ENROLLING PUBLIC KEY ON TARGET SYSTEM BY ADDING THE PUBLIC KEY TO THE MOK LIST

You must enroll your public key on all systems where you want to authenticate and load your kernel or kernel modules. You can import the public key on a target system in different ways so that the platform keyring (**.platform**) is able to use the public key to authenticate the kernel or kernel modules.

When RHEL 9 boots on a UEFI-based system with Secure Boot enabled, the kernel loads onto the platform keyring (**.platform**) all public keys that are in the Secure Boot **db** key database. At the same time, the kernel excludes the keys in the **dbx** database of revoked keys.

You can use the Machine Owner Key (MOK) facility feature to expand the UEFI Secure Boot key database. When RHEL 9 boots on an UEFI-enabled system with Secure Boot enabled, the keys on the MOK list are also added to the platform keyring (**.platform**) 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**, **MokManager**, **GRUB**, and the **mokutil** utility.



NOTE

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.

Prerequisites

- You have generated a public and private key pair and know the validity dates of your public keys. For details, see [Generating a public and private key pair](#).

Procedure

Procedure

1. Export your public key to the **sb_cert.cer** file:

```
# certutil -d /etc/pki/pesign \
-n 'Custom Secure Boot key' \
-Lr \
> sb_cert.cer
```

2. Import your public key into the MOK list:

```
# mokutil --import sb_cert.cer
```

3. Enter a new password for this MOK enrollment request.
4. Reboot the machine.
The **shim** boot loader notices the pending MOK key enrollment request and it launches **MokManager.efi** to enable you to complete the enrollment from the UEFI console.
5. Choose **Enroll MOK**, enter the password you previously associated with this request when prompted, 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 **.platform** keyring on this and subsequent boots when UEFI Secure Boot is enabled.

11.9. SIGNING A KERNEL WITH THE PRIVATE KEY

You can obtain enhanced security benefits on your system by loading a signed kernel if the UEFI Secure Boot mechanism is enabled.

Prerequisites

- You have generated a public and private key pair and know the validity dates of your public keys. For details, see [Generating a public and private key pair](#).
- You have enrolled your public key on the target system. For details, see [Enrolling public key on target system by adding the public key to the MOK list](#).
- You have a kernel image in the ELF format available for signing.

Procedure

- On the x64 architecture:
 - a. Create a signed image:

```
# pesign --certificate 'Custom Secure Boot key' \
--in vmlinuz-version \
--sign \
--out vmlinuz-version.signed
```

Replace **version** with the version suffix of your **vmlinuz** file, and **Custom Secure Boot key** with the name that you chose earlier.

- b. Optional: Check the signatures:

```
# pesign --show-signature \  
--in vmlinuz-version.signed
```

- c. Overwrite the unsigned image with the signed image:

```
# mv vmlinuz-version.signed vmlinuz-version
```

- On the 64-bit ARM architecture:

- a. Decompress the **vmlinuz** file:

```
# zcat vmlinuz-version > vmlinux-version
```

- b. Create a signed image:

```
# pesign --certificate 'Custom Secure Boot key' \  
--in vmlinux-version \  
--sign \  
--out vmlinux-version.signed
```

- c. Optional: Check the signatures:

```
# pesign --show-signature \  
--in vmlinux-version.signed
```

- d. Compress the **vmlinux** file:

```
# gzip --to-stdout vmlinux-version.signed > vmlinuz-version
```

- e. Remove the uncompressed **vmlinux** file:

```
# rm vmlinux-version*
```

11.10. SIGNING A GRUB BUILD WITH THE PRIVATE KEY

On a system where the UEFI Secure Boot mechanism is enabled, you can sign a GRUB build with a custom existing private key. You must do this if you are using a custom GRUB build, or if you have removed the Microsoft trust anchor from your system.

Prerequisites

- You have generated a public and private key pair and know the validity dates of your public keys. For details, see [Generating a public and private key pair](#).
- You have enrolled your public key on the target system. For details, see [Enrolling public key on target system by adding the public key to the MOK list](#).
- You have a GRUB EFI binary available for signing.

Procedure

- On the x64 architecture:

- a. Create a signed GRUB EFI binary:

```
# pesign --in /boot/efi/EFI/redhat/grubx64.efi \
--out /boot/efi/EFI/redhat/grubx64.efi.signed \
--certificate 'Custom Secure Boot key' \
--sign
```

Replace **Custom Secure Boot key** with the name that you chose earlier.

- b. Optional: Check the signatures:

```
# pesign --in /boot/efi/EFI/redhat/grubx64.efi.signed \
--show-signature
```

- c. Overwrite the unsigned binary with the signed binary:

```
# mv /boot/efi/EFI/redhat/grubx64.efi.signed \
/boot/efi/EFI/redhat/grubx64.efi
```

- On the 64-bit ARM architecture:

- a. Create a signed GRUB EFI binary:

```
# pesign --in /boot/efi/EFI/redhat/grubaa64.efi \
--out /boot/efi/EFI/redhat/grubaa64.efi.signed \
--certificate 'Custom Secure Boot key' \
--sign
```

Replace **Custom Secure Boot key** with the name that you chose earlier.

- b. Optional: Check the signatures:

```
# pesign --in /boot/efi/EFI/redhat/grubaa64.efi.signed \
--show-signature
```

- c. Overwrite the unsigned binary with the signed binary:

```
# mv /boot/efi/EFI/redhat/grubaa64.efi.signed \
/boot/efi/EFI/redhat/grubaa64.efi
```

11.11. SIGNING KERNEL MODULES WITH THE PRIVATE KEY

You can enhance the security of your system by loading signed kernel modules if the UEFI Secure Boot mechanism is enabled.

Your signed kernel module is also loadable on systems where UEFI Secure Boot is disabled or on a non-UEFI system. As a result, you do not need to provide both a signed and unsigned version of your kernel module.

Prerequisites

- You have generated a public and private key pair and know the validity dates of your public keys. For details, see [Generating a public and private key pair](#).
- You have enrolled your public key on the target system. For details, see [Enrolling public key on target system by adding the public key to the MOK list](#).
- You have a kernel module in ELF image format available for signing.

Procedure

1. Export your public key to the **sb_cert.cer** file:

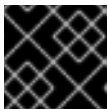
```
# certutil -d /etc/pki/pesign \  
-n 'Custom Secure Boot key' \  
-Lr \  
> sb_cert.cer
```

2. Extract the key from the NSS database as a PKCS #12 file:

```
# pk12util -o sb_cert.p12 \  
-n 'Custom Secure Boot key' \  
-d /etc/pki/pesign
```

3. When the previous command prompts you, enter a new password that encrypts the private key.
4. Export the unencrypted private key:

```
# openssl pkcs12 \  
-in sb_cert.p12 \  
-out sb_cert.priv \  
-nocerts \  
-noenc
```



IMPORTANT

Handle the unencrypted private key with care.

5. Sign your kernel module. The following command appends the signature directly to the ELF image in your kernel module file:

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

Your kernel module is now ready for loading.



IMPORTANT

In Red Hat Enterprise Linux 9, 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 2021 can be used to authenticate a kernel module signed in 2021 with that key. However, users cannot use that key to sign a kernel module in 2022.

Verification

1. Display information about the kernel module's signature:

```
# modinfo my_module.ko | grep signer
signer:    Your Name Key
```

Check that the signature lists your name as entered during generation.



NOTE

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** cannot display the signature on your kernel module.

2. Load the module:

```
# insmod my_module.ko
```

3. Remove (unload) the module:

```
# modprobe -r my_module.ko
```

Additional resources

- [Displaying information about kernel modules](#)

11.12. LOADING SIGNED KERNEL MODULES

Once your public key is enrolled in the system keyring (**.builtin_trusted_keys**) and the MOK list, and after you have signed the respective kernel module with your private key, you can load your signed kernel module with the **modprobe** command.

Prerequisites

- You have generated the public and private key pair. For details, see [Generating a public and private key pair](#).
- You have enrolled the public key into the system keyring. For details, see [Enrolling public key on target system by adding the public key to the MOK list](#).
- You have signed a kernel module with the private key. For details, see [Signing kernel modules with the private key](#).

- Install the **kernel-modules-extra** package, which creates the **/lib/modules/\$(uname -r)/extra/** directory:

```
# dnf -y install kernel-modules-extra
```

Procedure

1. Verify that your public keys are on the system keyring:

```
# keyctl list %:.platform
```

2. Copy the kernel module into the **extra/** directory of the kernel that you want:

```
# cp my_module.ko /lib/modules/$(uname -r)/extra/
```

3. Update the modular dependency list:

```
# depmod -a
```

4. Load the kernel module:

```
# modprobe -v my_module
```

5. 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
```

Verification

- Verify that the module was successfully loaded:

```
# lsmod | grep my_module
```

Additional resources

- [Managing kernel modules](#)

CHAPTER 12. UPDATING THE SECURE BOOT REVOCATION LIST

You can update the UEFI Secure Boot Revocation List on your system so that Secure Boot identifies software with known security issues and prevents it from compromising your boot process.

12.1. PREREQUISITES

- Secure Boot is enabled on your system.

12.2. WHAT IS UEFI SECURE BOOT

With the *Unified Extensible Firmware Interface* (UEFI) Secure Boot technology, you can prevent the execution of the kernel-space code that has not been signed by a trusted key. The system boot loader is signed with a cryptographic key. The database of public keys, which is contained in the firmware, authorizes the signing key. You can subsequently verify a signature in the next-stage boot loader and the kernel.

UEFI Secure Boot establishes a chain of trust from the firmware to the signed drivers and kernel modules as follows:

- An UEFI private key signs, and a public key authenticates the **shim** first-stage boot loader. A *certificate authority* (CA) in turn signs the public key. The CA is stored in the firmware database.
- The **shim** file contains the Red Hat public key **Red Hat Secure Boot (CA key 1)** to authenticate the GRUB boot loader and the kernel.
- The kernel in turn contains public keys to authenticate drivers and modules.

Secure Boot is the boot path validation component of the UEFI specification. The specification defines:

- Programming interface for cryptographically protected UEFI variables in non-volatile storage.
- Storing the trusted X.509 root certificates in UEFI variables.
- Validation of UEFI applications such as boot loaders and drivers.
- Procedures to revoke known-bad certificates and application hashes.

UEFI Secure Boot helps in the detection of unauthorized changes but does **not**:

- Prevent installation or removal of second-stage boot loaders.
- Require explicit user confirmation of such changes.
- Stop boot path manipulations. Signatures are verified during booting, not when the boot loader is installed or updated.

If the boot loader or the kernel are not signed by a system trusted key, Secure Boot prevents them from starting.

12.3. THE SECURE BOOT REVOCATION LIST

The UEFI Secure Boot Revocation List, or the Secure Boot Forbidden Signature Database (**dbx**), is a list that identifies software that Secure Boot no longer allows to run.

When a security issue or a stability problem is found in software that interfaces with Secure Boot, such as in the GRUB boot loader, the Revocation List stores its hash signature. Software with such a recognized signature cannot run during boot, and the system boot fails in order to prevent compromising the system.

For example, a certain version of GRUB might contain a security issue that allows an attacker to bypass the Secure Boot mechanism. When the issue is found, the Revocation List adds hash signatures of all GRUB versions that contain the issue. As a result, only secure GRUB versions can boot on the system.

The Revocation List requires regular updates to recognize newly found issues. When updating the Revocation List, make sure to use a safe update method that does not cause your currently installed system to no longer boot.

12.4. APPLYING AN ONLINE REVOCATION LIST UPDATE

You can update the Secure Boot Revocation List on your system so that Secure Boot prevents known security issues. This procedure is safe and ensures that the update does not prevent your system from booting.

Prerequisites

- Your system can access the internet for updates.

Procedure

1. Determine the current version of the Revocation List:

```
# fwupdmgr get-devices
```

See the **Current version** field under **UEFI dbx**.

2. Enable the LVFS Revocation List repository:

```
# fwupdmgr enable-remote lvfs
```

3. Refresh the repository metadata:

```
# fwupdmgr refresh
```

4. Apply the Revocation List update:

- On the command line:

```
# fwupdmgr update
```

- In the graphical interface:

- i. Open the **Software** application
- ii. Navigate to the **Updates** tab.
- iii. Find the **Secure Boot dbx Configuration Update** entry.

- iv. Click **Update**.
5. At the end of the update, **fwupdmgr** or **Software** asks you to reboot the system. Confirm the reboot.

Verification

- After the reboot, check the current version of the Revocation List again:

```
# fwupdmgr get-devices
```

12.5. APPLYING AN OFFLINE REVOCATION LIST UPDATE

On a system with no internet connection, you can update the Secure Boot Revocation List from RHEL so that Secure Boot prevents known security issues. This procedure is safe and ensures that the update does not prevent your system from booting.

Procedure

1. Determine the current version of the Revocation List:

```
# fwupdmgr get-devices
```

See the **Current version** field under **UEFI dbx**.

2. List the updates available from RHEL:

```
# ls /usr/share/dbxtool/
```

3. Select the most recent update file for your architecture. The file names use the following format:

```
DBXUpdate-date-architecture.cab
```

4. Install the selected update file:

```
# fwupdmgr install /usr/share/dbxtool/DBXUpdate-date-architecture.cab
```

5. At the end of the update, **fwupdmgr** asks you to reboot the system. Confirm the reboot.

Verification

- After the reboot, check the current version of the Revocation List again:

```
# fwupdmgr get-devices
```

CHAPTER 13. ENHANCING SECURITY WITH THE KERNEL INTEGRITY SUBSYSTEM

You can improve the protection of your system by using components of the kernel integrity subsystem. Learn more about the relevant components and their configuration.



NOTE

You can use the features with cryptographic signatures only for Red Hat products because the kernel keyring system includes only the certificates for Red Hat signature keys. Using other hash features results in incomplete tamper-proofing.

13.1. THE KERNEL INTEGRITY SUBSYSTEM

The integrity subsystem is the part of the kernel that maintains overall integrity of system data. This subsystem helps to keep the state of a system the same from the time it was built. By using this subsystem, you can prevent undesired modification of specific system files.

The kernel integrity subsystem consists of two major components:

Integrity Measurement Architecture (IMA)

- IMA measures file content whenever it is executed or opened by cryptographically hashing or signing with cryptographic keys. The keys are stored in the kernel keyring subsystem.
- IMA places the measured values within the kernel's memory space. This prevents users of the system from modifying the measured values.
- IMA allows local and remote parties to verify the measured values.
- IMA provides local validation of the current content of files against the values previously stored in the measurement list within the kernel memory. This extension forbids performing any operation on a specific file in case the current and the previous measures do not match.

Extended Verification Module (EVM)

- EVM protects extended attributes of files (also known as *xattr*) that are related to system security, such as IMA measurements and SELinux attributes. EVM cryptographically hashes their corresponding values or signs them with cryptographic keys. The keys are stored in the kernel keyring subsystem.

The kernel integrity subsystem can use the Trusted Platform Module (TPM) to further harden system security.

A TPM is a hardware, firmware, or virtual component with integrated cryptographic keys, which is built according to the TPM specification by the Trusted Computing Group (TCG) for important cryptographic functions. TPMs are usually built as dedicated hardware attached to the platform's motherboard. By providing cryptographic functions from a protected and tamper-proof area of the hardware chip, TPMs are protected from software-based attacks. TPMs provide the following features:

- Random-number generator
- Generator and secure storage for cryptographic keys

- Hashing generator
- Remote attestation

Additional resources

- [Security hardening](#)
- [Basic and advanced configuration of Security-Enhanced Linux \(SELinux\)](#)

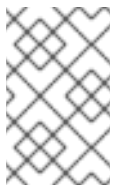
13.2. TRUSTED AND ENCRYPTED KEYS

Trusted keys and *encrypted keys* are an important part of enhancing system security.

Trusted and encrypted keys are variable-length symmetric keys generated by the kernel that use the kernel keyring service. The integrity of the keys can be verified, which 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

Trusted keys need the Trusted Platform Module (TPM) chip, which is used to both create and encrypt (seal) the keys. Each TPM has a master wrapping key, called the storage root key, which is stored within the TPM itself.



NOTE

RHEL 9 supports only TPM 2.0. If you must use TPM 1.2, use RHEL 8. For more information, see the [Is Trusted Platform Module \(TPM\) supported by Red Hat?](#) solution.

You can verify that a TPM 2.0 chip has been enabled by entering the following command:

```
$ cat /sys/class/tpm/tpm0/tpm_version_major
2
```

You can also enable a TPM 2.0 chip and manage the TPM 2.0 device through settings in the machine firmware.

In addition to that, you 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, when 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. You can save a single key also as multiple blobs, each with a different PCR value.

Encrypted keys

Encrypted keys do not require a TPM, because 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.

13.3. WORKING WITH TRUSTED KEYS

You can improve system security by using the **keyctl** utility to create, export, load and update trusted keys.

Prerequisites

- Trusted Platform Module (TPM) is enabled and active. See [The kernel integrity subsystem](#) and [Trusted and encrypted keys](#).

You can verify that your system has a TPM by entering the **tpm2_pcrread** command. If the output from this command displays several hashes, you have a TPM.

Procedure

- Create a 2048-bit RSA key with an SHA-256 primary storage key with a persistent handle of, for example, *81000001*, by using one of the following utilities:

- By using the **tss2** package:

```
# TPM_DEVICE=/dev/tpm0 tsscreateprimary -hi o -st
Handle 80000000
# TPM_DEVICE=/dev/tpm0 tssevictcontrol -hi o -ho 80000000 -hp 81000001
```

- By using the **tpm2-tools** package:

```
# tpm2_createprimary --key-algorithm=rsa2048 --key-context=key.ctx
name-alg:
value: sha256
raw: 0xb
...
sym-keybits: 128
rsa: xxxxxx...

# tpm2_evictcontrol -c key.ctx 0x81000001
persistentHandle: 0x81000001
action: persisted
```

- Create a trusted key by using a TPM 2.0 with the syntax of **keyctl add trusted <NAME> "new <KEY_LENGTH> keyhandle=<PERSISTENT-HANDLE> [options]" <KEYRING>**. In this example, the persistent handle is *81000001*.

```
# keyctl add trusted kmk "new 32 keyhandle=0x81000001" @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).

- 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
```


- Export the key to a user-space blob by using the serial number of the trusted key:

```
# keyctl pipe 642500861 > kmk.blob
```

The command uses the **pipe** subcommand and the serial number of **kmk**.

- Load the trusted key from the user-space blob:

```
# keyctl add trusted kmk "load `cat kmk.blob`" @u
268728824
```

- Create secure encrypted keys that use the TPM-sealed trusted key (**kmk**). Follow this syntax:
`keyctl add encrypted <NAME> "new [FORMAT] <KEY_TYPE>:<PRIMARY_KEY_NAME>
<KEY_LENGTH>" <KEYRING>`

```
# keyctl add encrypted encr-key "new trusted:kmk 32" @u
159771175
```

Additional resources

- the **keyctl(1)** manual page
- [Trusted and encrypted keys](#)
- [Kernel Key Retention Service](#)
- [The kernel integrity subsystem](#)

13.4. WORKING WITH ENCRYPTED KEYS

You can improve system security on systems where a Trusted Platform Module (TPM) is not available by managing encrypted keys.

Procedure

- Generate a user key by using a random sequence of numbers.

```
# 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 *primary key* and is used to seal the actual encrypted keys.

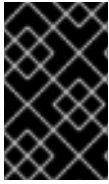
- Generate an encrypted key using the primary key from the previous step:

```
# keyctl add encrypted encr-key "new user:kmk-user 32" @u
1012412758
```

- 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

Encrypted keys that are not sealed by a trusted primary key are only as secure as the user primary key (random-number key) that was used to encrypt them. Therefore, load the primary user key as securely as possible and preferably early during the boot process.

Additional resources

- The **keyctl(1)** manual page
- [Kernel Key Retention Service](#)

13.5. ENABLING IMA AND EVM

You can enable and configure Integrity measurement architecture (IMA) and extended verification module (EVM) to improve the security of the operating system.

Prerequisites

- Secure Boot is temporarily disabled.



NOTE

When Secure Boot is enabled, the **ima_appraise=fix** kernel command-line parameter does not work.

- The **securityfs** file system is mounted on the **/sys/kernel/security/** directory and the **/sys/kernel/security/integrity/ima/** directory exists. You can verify where **securityfs** is mounted by using the **mount** command:

```
# mount
...
securityfs on /sys/kernel/security type securityfs (rw,nosuid,nodev,noexec,relatime)
...
```

- The **systemd** service manager is patched to support IMA and EVM on boot time. You can verify this by using the following command:

```
# dmesg | grep -i -e EVM -e IMA
[ 0.000000] Command line: BOOT_IMAGE=(hd0,msdos1)/vmlinuz-5.14.0-1.el9.x86_64
root=/dev/mapper/rhel-root ro crashkernel=1G-4G:192M,4G-64G:256M,64G-:512M
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=1G-4G:192M,4G-64G:256M,64G-:512M, the size chosen is
a best effort estimation.
[ 0.000000] Kernel command line: BOOT_IMAGE=(hd0,msdos1)/vmlinuz-5.14.0-
1.el9.x86_64 root=/dev/mapper/rhel-root ro crashkernel=1G-4G:192M,4G-64G:256M,64G-
:512M 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. Enable IMA and EVM in the *fix* mode for the current boot entry and allow users to gather and update the IMA measurements by adding 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 step forbids access to files whose prior and current measures do not match.

2. Reboot to make the changes come into effect.
3. Optional: Verify that the parameters have been added to the kernel command line:

```
# cat /proc/cmdline
BOOT_IMAGE=(hd0,msdos1)/vmlinuz-5.14.0-1.el9.x86_64 root=/dev/mapper/rhel-root ro
crashkernel=1G-4G:192M,4G-64G:256M,64G-:512M 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:

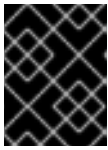
```
# keyctl add user kmk "$((dd if=/dev/urandom bs=1 count=32 2> /dev/null))" @u
748544121
```

The **kmk** is kept entirely in the kernel space memory. The 32-byte long value of the **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 first line of the previous output.

5. Create an encrypted EVM key based on the **kmk**:

```
# keyctl add encrypted evm-key "new user:kmk 64" @u
641780271
```

The command uses the **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 first 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** and export its unencrypted value into the new directory.

```
# keyctl pipe $(keyctl search @u user kmk) > /etc/keys/kmk
```

8. Search for the **evm-key** and export its encrypted value into the new directory.

```
# keyctl pipe $(keyctl search @u encrypted evm-key) > /etc/keys/evm-key
```

The **evm-key** has been encrypted by the kernel master key earlier.

9. Optional: 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

# ls -l /etc/keys/
total 8
-rw-r--r--. 1 root root 246 Jun 24 12:44 evm-key
-rw-r--r--. 1 root root 32 Jun 24 12:43 kmk
```

10. Optional: If the keys have been removed from the keyring, for example after system reboot, you can import the already exported **kmk** and **evm-key** instead of creating new ones.

- a. Import the **kmk**.

```
# keyctl add user kmk "$(cat /etc/keys/kmk)" @u
451342217
```

- b. Import the **evm-key**.

```
# keyctl add encrypted evm-key "load $(cat /etc/keys/evm-key)" @u
924537557
```

11. Activate EVM.

```
# echo 1 > /sys/kernel/security/evm
```

12. Relabel the whole system.

```
# find / -fstype xfs -type f -uid 0 -exec head -n 1 '{}' >/dev/null \;
```



WARNING

Enabling IMA and EVM without relabeling the system might make the majority of the files on the system inaccessible.

Verification

- Verify that EVM has been initialized.

```
# dmesg | tail -1
[...] evm: key initialized
```

Additional resources

- [The kernel integrity subsystem](#)
- [Trusted and encrypted keys](#).

13.6. COLLECTING FILE HASHES WITH INTEGRITY MEASUREMENT ARCHITECTURE

In the *measurement* phase, you can create file hashes and store them as extended attributes (*xattrs*) of those files. With the file hashes, you can generate either an RSA-based digital signature or a Hash-based Message Authentication Code (HMAC-SHA1) and thus prevent offline tampering attacks on the extended attributes.

Prerequisites

- IMA and EVM are enabled. For more information, see [Enabling integrity measurement architecture and extended verification module](#).
- A valid trusted key or encrypted key is stored in the kernel keyring.
- The **ima-evm-utils**, **attr**, and **keyutils** packages are installed.

Procedure

1. Create a test file:

```
# echo <Test_text> > test_file
```

IMA and EVM ensure that the **test_file** example file has assigned hash values that are stored as its extended attributes.

2. Inspect the file's extended attributes:

```
# getfattr -m . -d test_file
# file: test_file
security.evm=0sAnDly4VPA0HArpPO/EqiutnNyBql
security.ima=0sAQOEDeuUnWzwwKYk+n66h/vby3eD
```

The example output shows extended attributes with the IMA and EVM hash values and SELinux context. EVM adds a **security.evm** extended attribute related to the other attributes. At this point, you can use the **evmctl** utility on **security.evm** to generate either an RSA-based digital signature or a Hash-based Message Authentication Code (HMAC-SHA1).

Additional resources

- [Security hardening](#)

CHAPTER 14. USING SYSTEMD TO MANAGE RESOURCES USED BY APPLICATIONS

RHEL 9 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.

To achieve this, **systemd** takes various configuration options from the unit files or directly via the **systemctl** command. Then **systemd** applies those options to specific process groups by utilizing the Linux kernel system calls and features like **cgroups** and **namespaces**.



NOTE

You can review the full set of configuration options for **systemd** in the following manual pages:

- **systemd.resource-control(5)**
- **systemd.exec(5)**

14.1. ALLOCATING SYSTEM RESOURCES USING SYSTEMD

To modify the distribution of system resources, you can apply one or more of the following distribution models:

Weights

You can distribute the resource by adding up the weights of all sub-groups and giving each sub-group the fraction matching its ratio against the sum.

For example, if you have 10 cgroups, each with weight of value 100, the sum is 1000. Each cgroup receives one tenth of the resource.

Weight is usually used to distribute stateless resources. For example the *CPUWeight=* option is an implementation of this resource distribution model.

Limits

A cgroup can consume up to the configured amount of the resource. The sum of sub-group limits can exceed the limit of the parent cgroup. Therefore it is possible to overcommit resources in this model.

For example the *MemoryMax=* option is an implementation of this resource distribution model.

Protections

You can set up a protected amount of a resource for a cgroup. If the resource usage is below the protection boundary, the kernel will try not to penalize this cgroup in favor of other cgroups that compete for the same resource. An overcommit is also possible.

For example the *MemoryLow=* option is an implementation of this resource distribution model.

Allocations

Exclusive allocations of an absolute amount of a finite resource. An overcommit is not possible. An example of this resource type in Linux is the real-time budget.

unit file option

A setting for resource control configuration.

For example, you can configure CPU resource with options like *CPUAccounting=*, or *CPUQuota=*. Similarly, you can configure memory or I/O resources with options like *AllowedMemoryNodes=* and *IOAccounting=*.

Procedure

To change the required value of the unit file option of your service, you can adjust the value in the unit file, or use **systemctl** command:

1. Check the assigned values for the service of your choice.

```
# systemctl show --property <unit file option> <service name>
```

2. Set the required value of the CPU time allocation policy option:

```
# systemctl set-property <service name> <unit file option>=<value>
```

Verification steps

- Check the newly assigned values for the service of your choice.

```
# systemctl show --property <unit file option> <service name>
```

Additional resources

- **systemd.resource-control(5)**, **systemd.exec(5)** manual pages

14.2. ROLE OF SYSTEMD IN RESOURCE MANAGEMENT

The core function of **systemd** is service management and supervision. The **systemd** system and service manager ensures that managed services start at the right time and in the correct order during the boot process. The services have to run smoothly to use the underlying hardware platform optimally. Therefore, **systemd** also provides capabilities to define resource management policies, and to tune various options, which can improve the performance of the service.



IMPORTANT

In general, Red Hat recommends you use **systemd** for controlling the usage of system resources. You should manually configure the **cgroups** virtual file system only in special cases. For example, when you need to use **cgroup-v1** controllers that have no equivalents in **cgroup-v2** hierarchy.

14.3. OVERVIEW OF SYSTEMD HIERARCHY FOR CGROUPS

On the backend, 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. You can further modify this hierarchy by creating custom unit files or using the **systemctl** command. Also, **systemd** automatically mounts hierarchies for important kernel resource controllers at 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/sedispach
├─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

- [Configuring basic system settings](#) in Red Hat Enterprise Linux
- [What are kernel resource controllers](#)
- **systemd.resource-control(5), systemd.exec(5), cgroups(7), fork(), fork(2)** manual pages
- [Understanding cgroups](#)

14.4. LISTING SYSTEMD UNITS

Use the **systemd** system and service manager to list its units.

Procedure

- List all active units on the system with the **# systemctl** command. The terminal will return an output similar to the following example:

```

# systemctl
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.
- List inactive units.

```
# systemctl --all
```

- Limit the amount of information in the output.

```
# 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

- [Configuring basic system settings](#) in RHEL
- The **systemd.resource-control(5)**, **systemd.exec(5)** manual pages

14.5. VIEWING SYSTEMD CONTROL GROUP HIERARCHY

Display control groups (**cgroups**) hierarchy and processes running in specific **cgroups**.

Procedure

- Display the whole **cgroups** hierarchy on your system with the **systemd-cgls** command.

```

# 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*.

- Display the **cgroups** hierarchy filtered by a resource controller with the **systemd-cgls** **<resource_controller>** command.

```

# 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.

- Display detailed information about a certain unit and its part of the **cgroups** hierarchy with the **systemctl status <system_unit>** command.

```

# 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

- [What are kernel resource controllers](#)
- The **systemd.resource-control(5)**, **cgroups(7)** manual pages

14.6. VIEWING CGROUPS OF PROCESSES

The following procedure describes how to learn which *control group* (**cgroup**) a process belongs to. Then you can check the **cgroup** to learn which controllers and controller-specific configurations it uses.

Procedure

1. To view which **cgroup** a process belongs to, run the **# cat proc/<PID>/cgroup** command:

```
# cat /proc/2467/cgroup
0::/system.slice/example.service
```

The example output relates to a process of interest. In this case, it is a process identified by **PID 2467**, 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.

2. To display what controllers the **cgroup** utilizes and the respective configuration files, check the **cgroup** directory:

```
# cat /sys/fs/cgroup/system.slice/example.service/cgroup.controllers
memory pids

# ls /sys/fs/cgroup/system.slice/example.service/
cgroup.controllers
cgroup.events
...
cpu.pressure
cpu.stat
io.pressure
memory.current
memory.events
...
pids.current
pids.events
pids.max
```



NOTE

The version 1 hierarchy of **cgroups** uses a per-controller model. Therefore the output from the **/proc/PID/cgroup** file shows, which **cgroups** under each controller the PID belongs to. You can find the respective **cgroups** under the controller directories at **/sys/fs/cgroup/<controller_name>/**.

Additional resources

- **cgroups(7)** manual page

- [What are kernel resource controllers](#)
- Documentation in the `/usr/share/doc/kernel-doc-<kernel_version>/Documentation/admin-guide/cgroup-v2.rst` file (after installing the **kernel-doc** package)

14.7. MONITORING RESOURCE CONSUMPTION

View a list of currently running control groups (**cgroups**) and their resource consumption in real-time.

Procedure

1. Display a dynamic account of currently running **cgroups** with the **systemd-cgtop** command.

```
# systemd-cgtop
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-rhelx2dswap.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

- The **systemd-cgtop(1)** manual page

14.8. USING SYSTEMD UNIT FILES TO SET LIMITS FOR APPLICATIONS

Each existing or running unit is supervised by the **systemd**, which also creates control groups for them. The units have configuration files in the `/usr/lib/systemd/system/` directory. You can manually modify the unit files to set limits, prioritize, or control access to hardware resources for groups of processes.

Prerequisites

- You have the **root** privileges.

Procedure

1. Modify the `/usr/lib/systemd/system/example.service` file to limit the memory usage of a service:

```
...
[Service]
MemoryMax=1500K
...
```

The configuration above places a maximum memory limit, which the processes in a control group cannot exceed. The **example.service** service is part of such a control group which has imposed limitations. You can 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
```

NOTE

You can review the full set of configuration options for **systemd** in the following manual pages:

- **systemd.resource-control(5)**
- **systemd.exec(5)**

Verification

1. Check that the changes took effect:

```
# cat /sys/fs/cgroup/system.slice/example.service/memory.max
1536000
```

The example output shows that the memory consumption was limited at around 1,500 KB.

Additional resources

- [Understanding cgroups](#)
- [Configuring basic system settings](#) in Red Hat Enterprise Linux
- **systemd.resource-control(5)**, **systemd.exec(5)**, **cgroups(7)** manual pages

14.9. USING SYSTEMCTL COMMAND TO SET LIMITS TO APPLICATIONS

CPU affinity settings help you restrict the access of a particular process to some CPUs. Effectively, the CPU scheduler never schedules the process to run on the CPU that is not in the affinity mask of the process.

The default CPU affinity mask applies to all services managed by **systemd**.

To configure CPU affinity mask for a particular **systemd** service, **systemd** provides **CPUAffinity=** both as a unit file option and a manager configuration option in the **/etc/systemd/system.conf** file.

The **CPUAffinity= unit file option** sets a list of CPUs or CPU ranges that are merged and used as the affinity mask.

After configuring CPU affinity mask for a particular **systemd** service, you must restart the service to apply the changes.

Procedure

To set CPU affinity mask for a particular **systemd** service using the **CPUAffinity unit file** option:

1. Check the values of the **CPUAffinity** unit file option in the service of your choice:

```
$ systemctl show --property <CPU affinity configuration option> <service name>
```

2. As a root, set the required value of the **CPUAffinity** unit file option for the CPU ranges used as the affinity mask:

```
# systemctl set-property <service name> CPUAffinity=<value>
```

3. Restart the service to apply the changes.

```
# systemctl restart <service name>
```



NOTE

You can review the full set of configuration options for **systemd** in the following manual pages:

- **systemd.resource-control(5)**
- **systemd.exec(5)**

14.10. SETTING GLOBAL DEFAULT CPU AFFINITY THROUGH MANAGER CONFIGURATION

The **CPUAffinity** option in the **/etc/systemd/system.conf** file defines an affinity mask for the process identification number (PID) 1 and all processes forked off of PID1. You can then override the **CPUAffinity** on a per-service basis.

To set default CPU affinity mask for all systemd services using the **manager configuration** option:

1. Set the CPU numbers for the **CPUAffinity=** option in the **/etc/systemd/system.conf** file.
2. Save the edited file and reload the **systemd** service:

```
# systemctl daemon-reload
```

3. Reboot the server to apply the changes.



NOTE

You can review the full set of configuration options for **systemd** in the following manual pages:

- **systemd.resource-control(5)**
- **systemd.exec(5)**

14.11. CONFIGURING NUMA POLICIES USING SYSTEMD

Non-uniform memory access (NUMA) is a computer memory subsystem design, in which the memory access time depends on the physical memory location relative to the processor.

Memory close to the CPU has lower latency (local memory) than memory that is local for a different CPU (foreign memory) or is shared between a set of CPUs.

In terms of the Linux kernel, NUMA policy governs where (for example, on which NUMA nodes) the kernel allocates physical memory pages for the process.

systemd provides unit file options **NUMAPolicy** and **NUMAMask** to control memory allocation policies for services.

Procedure

To set the NUMA memory policy through the **NUMAPolicy** unit file option:

1. Check the values of the **NUMAPolicy** unit file option in the service of your choice:

```
$ systemctl show --property <NUMA policy configuration option> <service name>
```

2. As a root, set the required policy type of the **NUMAPolicy** unit file option:

```
# systemctl set-property <service name> NUMAPolicy=<value>
```

3. Restart the service to apply the changes.

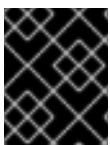
```
# systemctl restart <service name>
```

To set a global **NUMAPolicy** setting through the **manager configuration** option:

1. Search in the **/etc/systemd/system.conf** file for the **NUMAPolicy** option.
2. Edit the policy type and save the file.
3. Reload the **systemd** configuration:

```
# systemd daemon-reload
```

4. Reboot the server.



IMPORTANT

When you configure a strict NUMA policy, for example **bind**, make sure that you also appropriately set the **CPUAffinity=** unit file option.

Additional resources

- [Using `systemctl` command to set limits to applications](#)
- The **`systemd.resource-control(5)`**, **`systemd.exec(5)`**, **`set_mempolicy(2)`** manual pages.

14.12. NUMA POLICY CONFIGURATION OPTIONS FOR SYSTEMD

Systemd provides the following options to configure the NUMA policy:

NUMAPolicy

Controls the NUMA memory policy of the executed processes. The following policy types are possible:

- default
- preferred
- bind
- interleave
- local

NUMAMask

Controls the NUMA node list which is associated with the selected NUMA policy.

Note that the **NUMAMask** option is not required to be specified for the following policies:

- default
- local

For the preferred policy, the list specifies only a single NUMA node.

Additional resources

- **`systemd.resource-control(5)`**, **`systemd.exec(5)`**, and **`set_mempolicy(2)`** manual pages

14.13. CREATING TRANSIENT CGROUPS USING SYSTEMD-RUN COMMAND

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

- [What are control groups](#)
- [Configuring basic system settings](#) in RHEL
- the **systemd-run(1)** manual page

14.14. REMOVING TRANSIENT CONTROL GROUPS

You can use the **systemd** system and service manager to remove transient control groups (**cgroups**) if you no longer need to limit, prioritize, or control access to hardware resources for groups of processes.

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

- [What are control groups](#)
- [What are kernel resource controllers](#)
- **systemd.resource-control(5), cgroups(7)** manual pages
- [Configuring basic system settings](#) in RHEL

CHAPTER 15. UNDERSTANDING CONTROL GROUPS

Using the control groups (**cgroups**) kernel functionality, you can control resource usage of applications to use them more efficiently.

You can use **cgroups** for the following tasks:

- Setting limits for system resource allocation.
- Prioritizing the allocation of hardware resources to specific processes.
- Isolating certain processes from obtaining hardware resources.

15.1. INTRODUCING CONTROL GROUPS

Using the *control groups* Linux kernel feature, you can organize processes into hierarchically ordered groups – **cgroups**. You define the hierarchy (control groups tree) by providing structure to **cgroups** virtual file system, mounted by default on the **/sys/fs/cgroup/** directory.

The **systemd** service manager uses **cgroups** to organize all units and services that it governs. Manually, you can manage the hierarchies of **cgroups** by creating and removing sub-directories in the **/sys/fs/cgroup/** directory.

The resource controllers in the kernel then modify the behavior of processes in **cgroups** by limiting, prioritizing or allocating system resources, of those processes. These resources include the following:

- CPU time
- Memory
- Network bandwidth
- Combinations of these resources

The primary use case of **cgroups** is aggregating system processes and dividing hardware resources among applications and users. This makes it possible to increase the efficiency, stability, and security of your environment.

Control groups version 1

Control groups version 1 (**cgroups-v1**) provide a per-resource controller hierarchy. This means that each resource (such as CPU, memory, or I/O) has its own control group hierarchy. You can combine different control group hierarchies in a way that one controller can coordinate with another in managing their respective resources. However, when the two controllers belong to different process hierarchies, proper coordination is limited.

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

Control groups version 2 (**cgroups-v2**) provide a single control group hierarchy against which all resource controllers are mounted.

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

**IMPORTANT**

RHEL 9, by default, mounts and uses **cgroups-v2**.

Additional resources

- [Introducing kernel resource controllers](#)
- The **cgroups(7)** manual page
- [cgroups-v1](#)
- [cgroups-v2](#)

15.2. INTRODUCING KERNEL RESOURCE CONTROLLERS

Kernel resource controllers enable the functionality of control groups. RHEL 9 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** service manager. You can find a list of the currently mounted resource controllers in the **/proc/cgroups** file.

Table 15.1. Controllers available for cgroups-v1:

blkio	Sets limits on input/output access to and from block devices.
cpu	Adjusts the parameters of the Completely Fair Scheduler (CFS) for a control group's tasks. The cpu controller 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. The cpuacct controller is mounted together with the cpu controller on the same mount.
cpuset	Restricts 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	Controls access to devices for tasks in a control group.
freezer	Suspends or resumes tasks in a control group.
memory	Sets 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 to identify packets that originate from a particular control group task (by using the iptables command).

net_prio	Sets the priority of network traffic.
pids	Sets limits for a number of processes and their children in a control group.
perf_event	Groups tasks for monitoring by the perf performance monitoring and reporting utility.
rdma	Sets 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.

Table 15.2. Controllers available for **cgroups-v2**:

io	Sets limits on input/output access to and from block devices.
memory	Sets limits on memory use by tasks in a control group and generates automatic reports on memory resources used by those tasks.
pids	Sets limits for a number of processes and their children in a control group.
rdma	Sets limits on Remote Direct Memory Access/InfiniBand specific resources in a control group.
cpu	Adjusts the parameters of the Completely Fair Scheduler (CFS) for a control group's tasks and creates automatic reports on CPU resources used by tasks in a control group.
cpuset	Restricts 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. Supports only the core functionality (cpus{,.effective} , mems{,.effective}) with a new partition feature.
perf_event	Groups tasks for monitoring by the perf performance monitoring and reporting utility. perf_event is enabled automatically on the v2 hierarchy.

**IMPORTANT**

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

Additional resources

- The **cgroups(7)** manual page
- Documentation in **/usr/share/doc/kernel-doc-<kernel_version>/Documentation/cgroups-v1/** directory (after installing the **kernel-doc** package).

15.3. INTRODUCING NAMESPACES

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 use 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.

Table 15.3. 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

- The **namespaces(7)** and **cgroup_namespaces(7)** manual pages
- [Introducing control groups](#)

CHAPTER 16. USING CGROUPFS TO MANUALLY MANAGE CGROUPS

You can manage **cgroup** hierarchies on your system by creating directories on the **cgroupfs** virtual file system. The file system is mounted by default on the **/sys/fs/cgroup/** directory and you can specify desired configurations in dedicated control files.



IMPORTANT

In general, Red Hat recommends you use **systemd** for controlling the usage of system resources. You should manually configure the **cgroups** virtual file system only in special cases. For example, when you need to use **cgroup-v1** controllers that have no equivalents in **cgroup-v2** hierarchy.

16.1. CREATING CGROUPS AND ENABLING CONTROLLERS IN CGROUPS-V2 FILE SYSTEM

You can manage the *control groups* (**cgroups**) by creating or removing directories and by writing to files in the **cgroups** virtual file system. The file system is by default mounted on the **/sys/fs/cgroup/** directory. To use settings from the **cgroups** controllers, you also need to enable the desired controllers for child **cgroups**. The root **cgroup** has, by default, enabled the **memory** and **pids** controllers for its child **cgroups**. Therefore, Red Hat recommends to create at least two levels of child **cgroups** inside the **/sys/fs/cgroup/** root **cgroup**. This way you optionally remove the **memory** and **pids** controllers from the child **cgroups** and maintain better organizational clarity of **cgroup** files.

Prerequisites

- You have root permissions.

Procedure

1. Create the **/sys/fs/cgroup/Example/** directory:

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

The **/sys/fs/cgroup/Example/** directory defines a child group. When you create the **/sys/fs/cgroup/Example/** directory, some **cgroups-v2** interface files are automatically created in the directory. The **/sys/fs/cgroup/Example/** directory contains also controller-specific files for the **memory** and **pids** controllers.

2. Optionally, inspect the newly created child control group:

```
# ll /sys/fs/cgroup/Example/
-r--r--r--. 1 root root 0 Jun  1 10:33 cgroup.controllers
-r--r--r--. 1 root root 0 Jun  1 10:33 cgroup.events
-rw-r--r--. 1 root root 0 Jun  1 10:33 cgroup.freeze
-rw-r--r--. 1 root root 0 Jun  1 10:33 cgroup.procs
...
-rw-r--r--. 1 root root 0 Jun  1 10:33 cgroup.subtree_control
-r--r--r--. 1 root root 0 Jun  1 10:33 memory.events.local
-rw-r--r--. 1 root root 0 Jun  1 10:33 memory.high
-rw-r--r--. 1 root root 0 Jun  1 10:33 memory.low
...
```

```
-r—r—r--. 1 root root 0 Jun  1 10:33 pids.current
-r—r—r--. 1 root root 0 Jun  1 10:33 pids.events
-rw-r—r--. 1 root root 0 Jun  1 10:33 pids.max
```

The example output shows general **cgroup** control interface files such as **cgroup.procs** or **cgroup.controllers**. These files are common to all control groups, regardless of enabled controllers.

The files such as **memory.high** and **pids.max** relate to the **memory** and **pids** controllers, which are in the root control group (**/sys/fs/cgroup/**), and are enabled by default by **systemd**.

By default, the newly created child group inherits all settings from the parent **cgroup**. In this case, there are no limits from the root **cgroup**.

3. Verify that the desired controllers are available in the **/sys/fs/cgroup/cgroup.controllers** file:

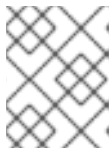
```
# cat /sys/fs/cgroup/cgroup.controllers
cpuset cpu io memory hugetlb pids rdma
```

4. Enable the desired controllers. In this example it is **cpu** and **cpuset** controllers:

```
# echo "+cpu" >> /sys/fs/cgroup/cgroup.subtree_control
# echo "+cpuset" >> /sys/fs/cgroup/cgroup.subtree_control
```

These commands enable the **cpu** and **cpuset** controllers for the immediate child groups of the **/sys/fs/cgroup/** root control group. Including the newly created **Example** control group. A *child group* is where you can specify processes and apply control checks to each of the processes based on your criteria.

Users can read the contents of the **cgroup.subtree_control** file at any level to get an idea of what controllers are going to be available for enablement in the immediate child group.



NOTE

By default, the **/sys/fs/cgroup/cgroup.subtree_control** file in the root control group contains **memory** and **pids** controllers.

5. Enable the desired controllers for child **cgroups** of the **Example** control group:

```
# echo "+cpu +cpuset" >> /sys/fs/cgroup/Example/cgroup.subtree_control
```

This command ensures that the immediate child control group will *only* have controllers relevant to regulate the CPU time distribution – not to **memory** or **pids** controllers.

6. Create the **/sys/fs/cgroup/Example/tasks/** directory:

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

The **/sys/fs/cgroup/Example/tasks/** directory defines a child group with files that relate purely to **cpu** and **cpuset** controllers. You can now assign processes to this control group and utilize **cpu** and **cpuset** controller options for your processes.

7. Optionally, inspect the child control group:

```
# ll /sys/fs/cgroup/Example/tasks
-r--r--r--. 1 root root 0 Jun  1 11:45 cgroup.controllers
-r--r--r--. 1 root root 0 Jun  1 11:45 cgroup.events
-rw-r--r--. 1 root root 0 Jun  1 11:45 cgroup.freeze
-rw-r--r--. 1 root root 0 Jun  1 11:45 cgroup.max.depth
-rw-r--r--. 1 root root 0 Jun  1 11:45 cgroup.max.descendants
-rw-r--r--. 1 root root 0 Jun  1 11:45 cgroup.procs
-r--r--r--. 1 root root 0 Jun  1 11:45 cgroup.stat
-rw-r--r--. 1 root root 0 Jun  1 11:45 cgroup.subtree_control
-rw-r--r--. 1 root root 0 Jun  1 11:45 cgroup.threads
-rw-r--r--. 1 root root 0 Jun  1 11:45 cgroup.type
-rw-r--r--. 1 root root 0 Jun  1 11:45 cpu.max
-rw-r--r--. 1 root root 0 Jun  1 11:45 cpu.pressure
-rw-r--r--. 1 root root 0 Jun  1 11:45 cpuset.cpus
-r--r--r--. 1 root root 0 Jun  1 11:45 cpuset.cpus.effective
-rw-r--r--. 1 root root 0 Jun  1 11:45 cpuset.cpus.partition
-rw-r--r--. 1 root root 0 Jun  1 11:45 cpuset.mems
-r--r--r--. 1 root root 0 Jun  1 11:45 cpuset.mems.effective
-r--r--r--. 1 root root 0 Jun  1 11:45 cpu.stat
-rw-r--r--. 1 root root 0 Jun  1 11:45 cpu.weight
-rw-r--r--. 1 root root 0 Jun  1 11:45 cpu.weight.nice
-rw-r--r--. 1 root root 0 Jun  1 11:45 io.pressure
-rw-r--r--. 1 root root 0 Jun  1 11:45 memory.pressure
```



IMPORTANT

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

Verification steps

- Optional: confirm that you have created a new **cgroup** with only the desired controllers active:

```
# cat /sys/fs/cgroup/Example/tasks/cgroup.controllers
cpuset cpu
```

Additional resources

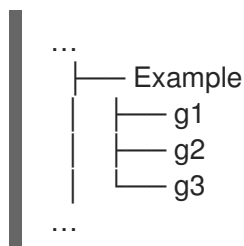
- [Understanding control groups](#)
- [What are kernel resource controllers](#)
- [Mounting cgroups-v1](#)
- **cgroups(7)**, **sysfs(5)** manual pages

16.2. CONTROLLING DISTRIBUTION OF CPU TIME FOR APPLICATIONS BY ADJUSTING CPU WEIGHT

You need to assign values to the relevant files of the **cpu** controller to regulate distribution of the CPU time to applications under the specific cgroup tree.

Prerequisites

- You have root permissions.
- You have applications for which you want to control distribution of CPU time.
- You created a two level hierarchy of *child control groups* inside the **/sys/fs/cgroup/** root control group as in the following example:



- You enabled the **cpu** controller in the parent control group and in child control groups similarly as described in [Creating cgroups and enabling controllers in cgroups-v2 file system](#) .

Procedure

1. Configure desired CPU weights to achieve resource restrictions within the control groups:

```
# echo "150" > /sys/fs/cgroup/Example/g1/cpu.weight
# echo "100" > /sys/fs/cgroup/Example/g2/cpu.weight
# echo "50" > /sys/fs/cgroup/Example/g3/cpu.weight
```

2. Add the applications' PIDs to the **g1**, **g2**, and **g3** child groups:

```
# echo "33373" > /sys/fs/cgroup/Example/g1/cgroup.procs
# echo "33374" > /sys/fs/cgroup/Example/g2/cgroup.procs
# echo "33377" > /sys/fs/cgroup/Example/g3/cgroup.procs
```

The example commands ensure that desired applications become members of the **Example/g*** child cgroups and will get their CPU time distributed as per the configuration of those cgroups.

The weights of the children cgroups (**g1**, **g2**, **g3**) that have running processes are summed up at the level of the parent cgroup (**Example**). The CPU resource is then distributed proportionally based on the respective weights.

As a result, when all processes run at the same time, the kernel allocates to each of them the proportionate CPU time based on their respective cgroup's **cpu.weight** file:

Child cgroup	cpu.weight file	CPU time allocation
g1	150	~50% (150/300)
g2	100	~33% (100/300)
g3	50	~16% (50/300)

The value of the **cpu.weight** controller file is not a percentage.

If one process stopped running, leaving cgroup **g2** with no running processes, the calculation would omit the cgroup **g2** and only account weights of cgroups **g1** and **g3**:

Child cgroup	cpu.weight file	CPU time allocation
g1	150	~75% (150/200)
g3	50	~25% (50/200)



IMPORTANT

If a child cgroup had multiple running processes, the CPU time allocated to the respective cgroup would be distributed equally to the member processes of that cgroup.

Verification

1. Verify that the applications run in the specified control groups:

```
# cat /proc/33373/cgroup /proc/33374/cgroup /proc/33377/cgroup
0::/Example/g1
0::/Example/g2
0::/Example/g3
```

The command output shows the processes of the specified applications that run in the **Example/g*** child cgroups.

2. Inspect the current CPU consumption of the throttled applications:

```
# top
top - 05:17:18 up 1 day, 18:25, 1 user, load average: 3.03, 3.03, 3.00
Tasks: 95 total, 4 running, 91 sleeping, 0 stopped, 0 zombie
%Cpu(s): 18.1 us, 81.6 sy, 0.0 ni, 0.0 id, 0.0 wa, 0.3 hi, 0.0 si, 0.0 st
MiB Mem : 3737.0 total, 3233.7 free, 132.8 used, 370.5 buff/cache
MiB Swap: 4060.0 total, 4060.0 free, 0.0 used. 3373.1 avail Mem

  PID USER   PR NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 33373 root    20  0 18720 1748 1460 R  49.5  0.0 415:05.87 sha1sum
 33374 root    20  0 18720 1756 1464 R  32.9  0.0 412:58.33 sha1sum
 33377 root    20  0 18720 1860 1568 R  16.3  0.0 411:03.12 sha1sum
   760 root    20  0 416620 28540 15296 S  0.3  0.7  0:10.23 tuned
     1 root    20  0 186328 14108 9484 S  0.0  0.4  0:02.00 systemd
     2 root    20  0     0     0  0 S  0.0  0.0  0:00.01 kthread
...
```



NOTE

We forced all the example processes to run on a single CPU for clearer illustration. The CPU weight applies the same principles also when used on multiple CPUs.

Notice that the CPU resource for the **PID 33373**, **PID 33374**, and **PID 33377** was allocated based on the weights, 150, 100, 50, you assigned to the respective child cgroups. The weights correspond to around 50%, 33%, and 16% allocation of CPU time for each application.

Additional resources

- [Understanding control groups](#)
- [What are kernel resource controllers](#)
- [Creating cgroups and enabling controllers in cgroups-v2 file system](#)
- [Resource Distribution Models](#)
- **cgroups(7)**, **sysfs(5)** manual pages

16.3. MOUNTING CGROUPS-V1

During the boot process, RHEL 9 mounts the **cgroup-v2** virtual filesystem by default. To utilize **cgroup-v1** functionality in limiting resources for your applications, manually configure the system.



NOTE

Both **cgroup-v1** and **cgroup-v2** are fully enabled in the kernel. There is no default control group version from the kernel point of view, and is decided by **systemd** to mount at startup.

Prerequisites

- You have root permissions.

Procedure

1. Configure the system to mount **cgroups-v1** by default during system boot by the **systemd** system and service manager:

```
# grubby --update-kernel=/boot/vmlinuz-$(uname -r) --
args="systemd.unified_cgroup_hierarchy=0
systemd.legacy_systemd_cgroup_controller"
```

This adds the necessary kernel command-line parameters to the current boot entry.

To add the same parameters to all kernel boot entries:

```
# grubby --update-kernel=ALL --args="systemd.unified_cgroup_hierarchy=0
systemd.legacy_systemd_cgroup_controller"
```

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

Verification

1. Optionally, verify that the **cgroups-v1** filesystem was mounted:

```
# mount -l | grep cgroup
```

```

tmpfs on /sys/fs/cgroup type tmpfs
(ro,nosuid,nodev,noexec,seclabel,size=4096k,nr_inodes=1024,mode=755,inode64)
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/perf_event type cgroup
(rw,nosuid,nodev,noexec,relatime,seclabel,perf_event)
cgroup on /sys/fs/cgroup/cpu,cpuacct type cgroup
(rw,nosuid,nodev,noexec,relatime,seclabel,cpu,cpuacct)
cgroup on /sys/fs/cgroup/pids type cgroup (rw,nosuid,nodev,noexec,relatime,seclabel,pids)
cgroup on /sys/fs/cgroup/cpuset type cgroup
(rw,nosuid,nodev,noexec,relatime,seclabel,cpuset)
cgroup on /sys/fs/cgroup/net_cls,net_prio type cgroup
(rw,nosuid,nodev,noexec,relatime,seclabel,net_cls,net_prio)
cgroup on /sys/fs/cgroup/hugetlb type cgroup
(rw,nosuid,nodev,noexec,relatime,seclabel,hugetlb)
cgroup on /sys/fs/cgroup/memory type cgroup
(rw,nosuid,nodev,noexec,relatime,seclabel,memory)
cgroup on /sys/fs/cgroup/blkio type cgroup (rw,nosuid,nodev,noexec,relatime,seclabel,blkio)
cgroup on /sys/fs/cgroup/devices type cgroup
(rw,nosuid,nodev,noexec,relatime,seclabel,devices)
cgroup on /sys/fs/cgroup/misc type cgroup (rw,nosuid,nodev,noexec,relatime,seclabel,misc)
cgroup on /sys/fs/cgroup/freezer type cgroup
(rw,nosuid,nodev,noexec,relatime,seclabel,freezer)
cgroup on /sys/fs/cgroup/rdma type cgroup (rw,nosuid,nodev,noexec,relatime,seclabel,rdma)

```

The **cgroups-v1** filesystems that correspond to various **cgroup-v1** controllers, were successfully mounted on the **/sys/fs/cgroup/** directory.

2. Optionally, inspect the contents of the **/sys/fs/cgroup/** directory:

```

# ll /sys/fs/cgroup/
dr-xr-xr-x. 10 root root 0 Mar 16 09:34 blkio
lrwxrwxrwx. 1 root root 11 Mar 16 09:34 cpu → cpu,cpuacct
lrwxrwxrwx. 1 root root 11 Mar 16 09:34 cpuacct → cpu,cpuacct
dr-xr-xr-x. 10 root root 0 Mar 16 09:34 cpu,cpuacct
dr-xr-xr-x. 2 root root 0 Mar 16 09:34 cpuset
dr-xr-xr-x. 10 root root 0 Mar 16 09:34 devices
dr-xr-xr-x. 2 root root 0 Mar 16 09:34 freezer
dr-xr-xr-x. 2 root root 0 Mar 16 09:34 hugetlb
dr-xr-xr-x. 10 root root 0 Mar 16 09:34 memory
dr-xr-xr-x. 2 root root 0 Mar 16 09:34 misc
lrwxrwxrwx. 1 root root 16 Mar 16 09:34 net_cls → net_cls,net_prio
dr-xr-xr-x. 2 root root 0 Mar 16 09:34 net_cls,net_prio
lrwxrwxrwx. 1 root root 16 Mar 16 09:34 net_prio → net_cls,net_prio
dr-xr-xr-x. 2 root root 0 Mar 16 09:34 perf_event
dr-xr-xr-x. 10 root root 0 Mar 16 09:34 pids
dr-xr-xr-x. 2 root root 0 Mar 16 09:34 rdma
dr-xr-xr-x. 11 root root 0 Mar 16 09:34 systemd

```

The **/sys/fs/cgroup/** directory, also called the *root control group*, by default, contains controller-specific directories such as **cpuset**. In addition, there are some directories related to **systemd**.

Additional resources

- [Understanding control groups](#)

- [What are kernel resource controllers](#)
- **cgroups(7)**, **sysfs(5)** manual pages
- [cgroup-v2 enabled by default in RHEL 9](#)

16.4. SETTING CPU LIMITS TO APPLICATIONS USING CGROUPS-V1

To configure CPU limits to an application by using *control groups version 1* (**cgroups-v1**), use the **/sys/fs/** virtual file system.

Prerequisites

- You have root permissions.
- You have an application whose CPU consumption you want to restrict.
- You configured the system to mount **cgroups-v1** by default during system boot by the **systemd** system and service manager:

```
# grubby --update-kernel=/boot/vmlinuz-$(uname -r) --
args="systemd.unified_cgroup_hierarchy=0
systemd.legacy_systemd_cgroup_controller"
```

This adds the necessary kernel command-line parameters to the current boot entry.

Procedure

1. Identify the process ID (PID) of the application that 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 0.3  0.0   0:00.75 kworker/u4:4-events_unbound
...
```

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

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

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

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

3. *Optional*: 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
```

This 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. Note 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 (μ s, represented here as "us") for how frequently a control group's access to CPU resources should be reallocated. The upper limit is 1 000 000 microsecond and the lower limit is 1 000 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**). When 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 1 000 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. *Optional*: Verify the limits:

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

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

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

This command ensures that a specific 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.

Verification

1. 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
```

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

2. 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
...
```

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



NOTE

The **cgroups-v2** counterpart for **cpu.cfs_period_us** and **cpu.cfs_quota_us** is the **cpu.max** file. The **cpu.max** file is available through the **cpu** controller.

Additional resources

- [Understanding control groups](#)
- [What kernel resource controllers are](#)
- **cgroups(7), sysfs(5)** manual pages

CHAPTER 17. ANALYZING SYSTEM PERFORMANCE WITH BPF COMPILER COLLECTION

As a system administrator, you can 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.

17.1. INSTALLING THE BCC-TOOLS PACKAGE

Install the **bcc-tools** package, which also installs the BPF Compiler Collection (BCC) library as a dependency.

Procedure

1. Install **bcc-tools**.

```
# dnf install bcc-tools
```

The BCC tools are installed 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.

17.2. USING SELECTED BCC-TOOLS FOR PERFORMANCE ANALYSES

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

- [Installed bcc-tools package](#)
- Root permissions

Using execsnoop to examine the system processes

1. Run the **execsnoop** program in one terminal:

```
# /usr/share/bcc/tools/execsnoop
```

2. In another terminal run, 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/
...
```

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 **execsnoop** output displays the following fields:

- **PCOMM** - The parent process name. (**ls**)
- **PID** - The process ID. (**8382**)
- **PPID** - The parent process ID. (**8287**)
- **RET** - The return value of the **exec()** system call (**0**), which loads program code into new processes.
- **ARGS** - The location of the started program with arguments.

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. Run 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, enter:

```
$ 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 **opensnoop** output displays the following fields:

- **PID** - The process ID. (**8596**)
- **COMM** - The process name. (**uname**)
- **FD** - The file descriptor - a value that **open()** returns to refer to the open file. (**3**)
- **ERR** - Any errors.
- **PATH** - The location of files that **open()** tried to open.
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. Run 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 enter, 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 **biotop** output displays the following fields:

- **PID** - The process ID. (**9568**)
- **COMM** - The process name. (**dd**)
- **DISK** - The disk performing the read operations. (**vda**)
- **I/O** - The number of read operations performed. (16294)
- **Kbytes** - The amount of Kbytes reached by the read operations. (14,440,636)
- **AVGms** - The average I/O time of read operations. (3.69)

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. Run 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 enter, 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'getsriptPlugin.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 **xfsslower** output displays the following fields:

- **COMM** - The process name. (**b'bash'**)
- **T** - The operation type. (**R**)
 - **Read**
 - **Write**
 - **Sync**
- **OFF_KB** - The file offset in KB. (0)
- **FILENAME** - The file being read, written, or synced.

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 18. INSTALLING KDUMP

The **kdump** service is installed and activated by default on the new Red Hat Enterprise Linux installations. Learn about **kdump** and how to install **kdump** when it is not enabled by default.

18.1. WHAT IS KDUMP

kdump is a service which provides a crash dumping mechanism. The service enables you to save the contents of the system memory for 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 into a file. 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, operational **kdump** 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.

You can enable **kdump** for all installed kernels on a machine or only for specified kernels. This is useful when there are multiple kernels used on a machine, some of which are stable enough that there is no concern that they could crash.

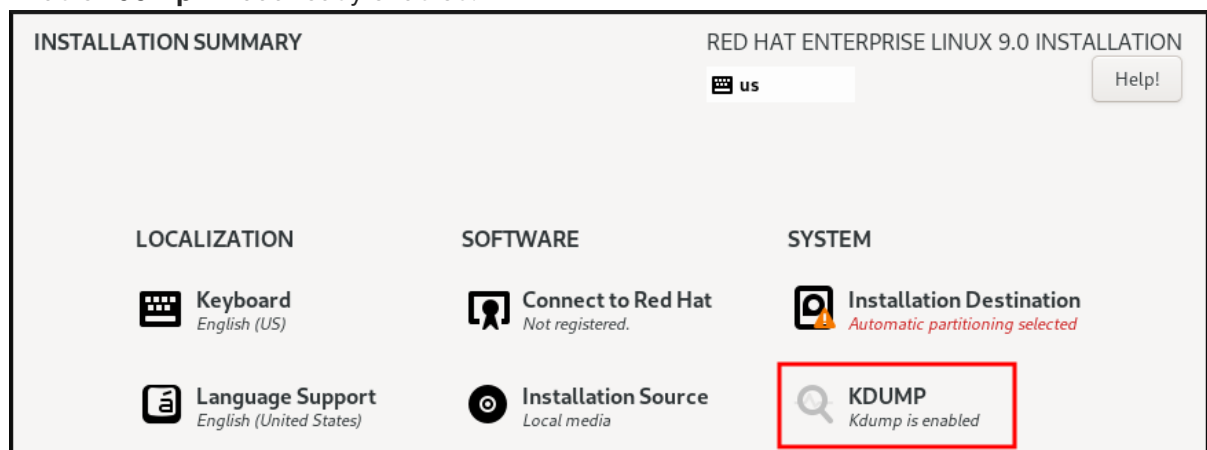
When **kdump** is installed, a default **/etc/kdump.conf** file is created. The file includes the default minimum **kdump** configuration. You can edit this file to customize the **kdump** configuration, but it is not required.

18.2. INSTALLING KDUMP USING ANACONDA

The **Anaconda** installer provides a graphical interface screen for **kdump** configuration during an interactive installation. The installer screen is titled as **KDUMP** and is available from the main **Installation Summary** screen. You can enable **kdump** and reserve the required amount of memory.

Procedure

1. Go to the **Kdump** field.
2. Enable **kdump** if not already enabled.



3. Define how much memory should be reserved for **kdump**.

18.3. INSTALLING KDUMP ON THE COMMAND LINE

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

Prerequisites

- An active RHEL subscription
- The **kexec-tools** package
- Fulfilled requirements for **kdump** configurations and targets. For details, see [Supported kdump configurations and targets](#).

Procedure

1. Check whether **kdump** is installed on your system:

```
# rpm -q kexec-tools
```

Output if the package is installed:

```
# kexec-tools-2.0.22-13.el9.x86_64
```

Output if the package is not installed:

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

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

```
# dnf install kexec-tools
```

CHAPTER 19. CONFIGURING KDUMP ON THE COMMAND LINE

The memory for **kdump** is reserved during the system boot. The memory size is configured in the system's Grand Unified Bootloader (GRUB) configuration file. The memory size depends on the **crashkernel=** value specified in the configuration file and the size of the system's physical memory.

19.1. CONFIGURING KDUMP MEMORY USAGE ON RHEL 9

The **kexec-tools** package maintains the default **crashkernel=** memory reservation values. The **kdump** service uses the default value to reserve the crash kernel memory for each kernel. The default value can also serve as the reference base value to estimate the required memory size when you set the **crashkernel=** value manually. The minimum size of the crash kernel can vary depending on the hardware and machine specifications.

The automatic memory allocation for **kdump** also varies based on the system hardware architecture and available memory size. For example, on AMD and Intel 64-bit architectures, the default value for the **crashkernel=** parameter will work only when the available memory is more than 1 GB. The **kexec-tools** utility configures the following default memory reserves on AMD64 and Intel 64-bit architecture:

```
crashkernel=1G-4G:192M,4G-64G:256M,64G:512M
```

You can also run **kdumpctl estimate** to query a rough estimate value without triggering a crash. The estimated **crashkernel=** value might not be an accurate one but can serve as a reference to set an appropriate **crashkernel=** value.



NOTE

The **crashkernel=auto** option in the boot command line is no longer supported on RHEL 9 and later releases.

Prerequisites

- You have root permissions on the system.
- You have fulfilled **kdump** requirements for configurations and targets. For details, see [Supported kdump configurations and targets](#)
- You have installed the **zipl** utility if it is the IBM Z system.

Procedure

1. Configure the default value for crash kernel.

```
# kdumpctl reset-crashkernel --kernel=ALL
```

When configuring the **crashkernel=** value, test the configuration by rebooting with **kdump** enabled. If the **kdump** kernel fails to boot, increase the memory size gradually to set an acceptable value.

2. To use a custom **crashkernel=** value:
 - a. Configure the required memory reserve.

■

```
# crashkernel=192M
```

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

```
# crashkernel=1G-4G:192M,2G-64G:256M
```

The example reserves 192 MB of memory if the total amount of system memory is 1 GB or higher and lower than 4 GB. If the total amount of memory is more than 4 GB, 256 MB is reserved for **kdump**.

- b. (Optional) 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=192M@16M
```

The example reserves 192 MB of memory starting at 16 MB (physical address 0x01000000). If you offset to 0 or do not specify a value, **kdump** offsets the reserved memory automatically. You can also offset memory when setting a variable memory reservation by specifying the offset as the last value. For example, **crashkernel=1G-4G:192M,2G-64G:256M@16M**.

- c. Update the bootloader configuration.

```
# grubby --update-kernel ALL --args "crashkernel=<custom-value>"
```

The **<custom-value>** must contain the custom **crashkernel=** value that you have configured for the crash kernel.

3. Reboot for changes to take effect.

```
# reboot
```

Verification

Cause the kernel to crash by activating the **sysrq** key. The **address-YYYY-MM-DD-HH:MM:SS/vmcore** file is saved to the target location as specified in the **/etc/kdump.conf** file. If you choose the default target location, the **vmcore** file is saved in the partition mounted under **/var/crash/**.



WARNING

The commands to test **kdump** configuration will cause the kernel to crash with data loss. Follow the instructions with care and do not use an active production system to test the kdump configuration

1. Activate the **sysrq** key to boot into the **kdump** kernel.

```
echo c > /proc/sysrq-trigger
```

The command causes the kernel to crash and reboots the kernel if required.

2. Display the `/etc/kdump.conf` file and check if the **vmcore** file is saved in the target destination.

Additional resources

- [How to manually modify the boot parameter in grub before the system boots](#)
- **grubby(8)** man page

19.2. CONFIGURING THE KDUMP TARGET

The crash dump is usually stored as a file in a local file system, written directly to a device. Alternatively, you can set up for the crash dump to be sent over a network using the **NFS** or **SSH** protocols. Only one of these options to preserve a crash dump file can be set at a time. The default behavior is to store it in the `/var/crash/` directory of the local file system.

Prerequisites

- **Root** permissions.
- Fulfilled requirements for **kdump** configurations and targets. For details, see [Supported kdump configurations and targets](#).

Procedure

- To store the crash dump 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 crash dump file.



NOTE

- When you specify a dump target in the `/etc/kdump.conf` file, then the path is **relative** to the specified dump target.
- When 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.

Example 19.1. The kdump target configuration

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

Here, the dump target is specified (**ext4 /dev/mapper/vg00-varcrashvol**), and thus mounted at **/var/crash**. The **path** option is also set to **/var/crash**, so the **kdump** saves the **vmcore** file in the **/var/crash/var/crash** directory.

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

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 RHEL 9, the directory defined as the kdump target using the **path** directive must exist when the **kdump systemd** service is started – otherwise the service fails.

- To write the file to a different partition, edit the **/etc/kdump.conf** configuration file:
 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
```

NOTE

The correct syntax for specifying UUID values is both **UUID="correct-uuid"** and **UUID=correct-uuid**.



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 crash dump directly to a device, edit the **/etc/kdump.conf** configuration file:
 1. Remove the hash sign ("**#**") from the beginning of the **#raw /dev/vg/lv_kdump** line

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

```
raw /dev/sdb1
```

- To store the crash 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 crash 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
```

19.3. CONFIGURING THE CORE COLLECTOR

The **kdump** service uses the **core_collector** program to capture the **vmcore** image. In RHEL, the **makedumpfile** utility is the default core collector. **makedumpfile** is a dump program that helps to copy only the required pages using various dump levels and compress the size of a dump file.

Using **makedumpfile**, you can create a small size dump file either by compressing dump data or by excluding pages or both. It needs the first kernel debug information to distinguish the not necessary pages by analyzing how the first kernel uses the memory.

Syntax

```
core_collector makedumpfile -z -d 31 --message-level 1
```

Options

- **-c, -l, -z, or -p**: specifies the compress dump file format by each page when you use one of these options: **-c** for **zlib**, **-l** for **lzo**, **-z** for **zstd**, or **-p** for **snappy**.
- **-d (dump_level)**: excludes pages so that they do not get copied to the dump file.
- **--message-level**: specifies the message types.

Using **--message-level**, you can restrict the outputs to print. For example, specifying 7 as the message level prints common messages and error messages. The maximum value for **--message_level** is 31.

Prerequisites

- Fulfilled **kdump** requirements for configurations and targets.

Procedure

1. As **root** user, edit the **/etc/kdump.conf** configuration file to remove the hash sign ("**#**") from the beginning of the following command:

```
core_collector makedumpfile -z -d 31 --message-level 1
```

2. To enable dump file compression, specify one of the **makedumpfile** options:

```
core_collector makedumpfile -z -d 31 --message-level 1
```

where,

- **-z** specifies the **dump** compressed file format.
- **-d** specifies dump level as 31.
- **--message-level** specifies message level as 1.

Also, consider the following example that uses **-l**:

- To compress a dump file using **-l**:

```
core_collector makedumpfile -l -d 31 --message-level 1
```

Additional resources

- **makedumpfile(8)** manual page

19.4. CONFIGURING THE KDUMP DEFAULT FAILURE RESPONSES

By default, when **kdump** fails to create a crash dump file at the configured target location, the system reboots and the dump is lost in the process. To change this behavior, follow the procedure below.

Prerequisites

- You have root permissions on the system.
- Fulfilled requirements for **kdump** configurations and targets.

Procedure

1. As **root**, remove the hash sign ("**#**") from the beginning of the **#failure_action** line in the **/etc/kdump.conf** configuration file.
2. Replace the value with a desired action.


```
failure_action poweroff
```

19.5. CONFIGURATION FILE FOR KDUMP

The configuration file for **kdump** kernel is **/etc/sysconfig/kdump**. This file controls the **kdump** kernel command line parameters.

For most configurations, use the default options. However, in some scenarios you might need to modify certain parameters to control the **kdump** kernel behavior. For example, modifying the **KDUMP_COMMANDLINE_APPEND** option to append the **kdump** kernel command-line to obtain a detailed debugging output or the **KDUMP_COMMANDLINE_REMOVE** option to remove arguments from the **kdump** command line.

For information about additional configuration options refer to **Documentation/admin-guide/kernel-parameters.txt** or the **/etc/sysconfig/kdump** file.

- **KDUMP_COMMANDLINE_REMOVE**

This option removes arguments from the current **kdump** command line. It removes parameters that may cause **kdump** errors or **kdump** kernel boot failures. These parameters may have been parsed from the previous **KDUMP_COMMANDLINE** process or inherited from the **/proc/cmdline** file. When this variable is not configured, it inherits all values from the **/proc/cmdline** file. Configuring this option also provides information that is helpful in debugging an issue.

Example

To remove certain arguments, add them to **KDUMP_COMMANDLINE_REMOVE** as follows:

```
# KDUMP_COMMANDLINE_REMOVE="hugepages hugepagesz slub_debug quiet
log_buf_len swiotlb"
```

- **KDUMP_COMMANDLINE_APPEND**

This option appends arguments to the current command line. These arguments may have been parsed by the previous **KDUMP_COMMANDLINE_REMOVE** variable.

For the **kdump** kernel, disabling certain modules such as **mce**, **cgroup**, **numa**, **hest_disable** can help prevent kernel errors. These modules may consume a significant portion of the kernel memory reserved for **kdump** or cause **kdump** kernel boot failures.

Example

To disable memory **cgroups** on the **kdump** kernel command line, run the command as follows:

```
# KDUMP_COMMANDLINE_APPEND="cgroup_disable=memory"
```

Additional resources

- **Documentation/admin-guide/kernel-parameters.txt** file
- **/etc/sysconfig/kdump** file

19.6. ENABLING AND DISABLING THE KDUMP SERVICE

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

Prerequisites

- Fulfilled **kdump** requirements for configurations and targets.
- All configurations for installing **kdump** are 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/kcore** 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.

19.7. TESTING THE KDUMP CONFIGURATION

Testing the **kdump** configuration validates the configuration and also records the time taken for a crash dump to complete with the specified workload.



WARNING

The commands to test **kdump** configuration will cause the kernel to crash with loss of data. Follow the instructions with care and do not use an active production system to test the **kdump** configuration.

Procedure

1. Reboot the system with **kdump** enabled.
2. Check if **kdump** is active.

```
# systemctl is-active kdump
active
```

3. Force a kernel crash.

```
echo c > /proc/sysrq-trigger
```



WARNING

The command causes the kernel to crash and reboots the kernel if required.

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

Additional resources

- [Configuring the kdump target](#)

19.8. PREVENTING KERNEL DRIVERS FROM LOADING FOR KDUMP

You can control the capture kernel from loading certain kernel drivers by adding the **KDUMP_COMMANDLINE_APPEND=** variable in the **/etc/sysconfig/kdump** configuration file. By using this method, you can prevent the **kdump** initial RAM disk image **initramfs** from loading the specified kernel module. This helps to prevent the out-of-memory (oom) killer errors or other crash kernel failures.

You can append the **KDUMP_COMMANDLINE_APPEND=** variable using one of the following configuration options:

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

Procedure

1. Select a kernel module that you intend to block from loading.

```
$ 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=** variable in the **/etc/sysconfig/kdump** file.

```
#
KDUMP_COMMANDLINE_APPEND="rd.driver.blacklist=hv_vmbus,hv_storvsc,hv_utils,
hv_netvsc,hid-hyperv"
```

Also, consider the following example using the **modprobe.blacklist=<modules>** configuration option.

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

3. Restart the **kdump** service.

```
# systemctl restart kdump
```

Additional resources

- **dracut.cmdline** man page

19.9. RUNNING KDUMP ON SYSTEMS WITH ENCRYPTED DISK

When you run a Linux Unified Key Setup (LUKS) encrypted partition, the system requires a certain amount of available memory. If the system has less than the required amount of available memory, the **systemd-cryptsetup** service fails to mount the partition. As a result, capturing the **vmcore** file to an encrypted target location fails in the second kernel (capture kernel).

The **kdumpctl estimate** command helps you estimate the amount of memory you need for **kdump**. It prints the recommended **crashkernel** value, which is the most suitable memory size required for **kdump**.

The recommended **crashkernel** value is calculated based on the current kernel size, kernel modules, **initramfs**, and the LUKS encrypted target memory requirement.

In case you use the custom **crashkernel** option, **kdumpctl estimate** prints the **LUKS required size** value. The value is the memory size required for LUKS encrypted target.

Procedure

1. Print the estimate **crashkernel** value:

kdumpctl estimate

Encrypted kdump target requires extra memory, assuming using the keyslot with minimum memory requirement

Reserved crashkernel: 256M

Recommended crashkernel: 652M

Kernel image size: 47M

Kernel modules size: 8M

Initramfs size: 20M

Runtime reservation: 64M

LUKS required size: 512M

Large modules: none

WARNING: Current crashkernel size is lower than recommended size 652M.

2. Configure the amount of required memory by increasing **crashkernel** to the desired value.

```
# grubby --args="crashkernel=652M" --update-kernel=ALL
```

3. Reboot for changes to take effect.

```
# reboot
```



NOTE

If the **kdump** service still fails to save the dump file to the encrypted target, increase the **crashkernel** value gradually to configure an appropriate amount of memory.

CHAPTER 20. ENABLING KDUMP

By using the procedure, you can enable or disable the **kdump** service for all installed kernels or for a specific kernel.

20.1. ENABLING KDUMP FOR ALL INSTALLED KERNELS

You can enable and start the **kdump** service for all kernels installed on the machine.

Prerequisites

- Administrator privileges

Procedure

1. Add the **crashkernel=auto** command-line parameter to all installed kernels:

```
# grubby --update-kernel=ALL --args="crashkernel=auto"
```

2. Enable the **kdump** service.

```
# systemctl enable --now kdump.service
```

Verification

- Check that the **kdump** service is running:

```
# systemctl status kdump.service

○ kdump.service - Crash recovery kernel arming
  Loaded: loaded (/usr/lib/systemd/system/kdump.service; enabled; vendor preset: disabled)
  Active: active (live)
```

20.2. ENABLING KDUMP FOR A SPECIFIC INSTALLED KERNEL

You can enable the **kdump** service for a specific kernel on the machine.

Prerequisites

- Administrator privileges

Procedure

1. List the kernels installed on the machine.

```
# ls -a /boot/vmlinuz-*
/boot/vmlinuz-0-rescue-2930657cd0dc43c2b75db480e5e5b4a9 /boot/vmlinuz-4.18.0-330.el8.x86_64 /boot/vmlinuz-4.18.0-330.rt7.111.el8.x86_64
```

2. Add a specific **kdump** kernel to the system's Grand Unified Bootloader (GRUB) configuration file.

For example:

```
# grubby --update-kernel=vmlinuz-4.18.0-330.el8.x86_64 --args="crashkernel=auto"
```

3. Enable the **kdump** service.

```
# systemctl enable --now kdump.service
```

Verification

- Check that the **kdump** service is running:

```
# systemctl status kdump.service

○ kdump.service - Crash recovery kernel arming
  Loaded: loaded (/usr/lib/systemd/system/kdump.service; enabled; vendor preset: disabled)
  Active: active (live)
```

20.3. DISABLING THE KDUMP SERVICE

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

Prerequisites

- Fulfilled requirements for **kdump** configurations and targets. For details, see [Supported kdump configurations and targets](#).
- All configurations for installing **kdump** are set up according to your needs. For details, see [Installing kdump](#).

Procedure

1. To stop the **kdump** service in the current session:

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

2. To disable the **kdump** service:

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



WARNING

It is recommended to set **kptr_restrict=1**. In that case, the **kdumpectl** service loads the crash kernel regardless of Kernel Address Space Layout (KASLR) being enabled or not.

Troubleshooting step

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

To work around this problem, the **/usr/share/doc/kexec-tools/kexec-kdump-howto.txt** file displays a warning message, which recommends the **kpтр_restrict=1** setting.

To ensure that **kdumрctI** service loads the crash kernel, verify that **kernel.kpтр_restrict = 1** is listed in the **sysctl.conf** file.

Additional resources

- [Managing system services with systemctl](#)

CHAPTER 21. SUPPORTED KDUMP CONFIGURATIONS AND TARGETS

21.1. 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
```

With the stated list of minimum memory requirements, you set the appropriate memory size to automatically reserve a memory for **kdump** on the latest available versions. The memory size depends on the system's architecture and total available physical memory.

Table 21.1. Minimum amount of reserved memory required for kdump

Architecture	Available Memory	Minimum Reserved Memory
AMD64 and Intel 64 (x86_64)	1 GB to 4 GB	160 MB of RAM.
	4 GB to 64 GB	192 MB of RAM.
	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	448 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)	1 GB to 4 GB	160 MB of RAM.
	4 GB to 64 GB	192 MB of RAM.
	64 GB to 1 TB	256 MB of RAM.

Architecture	Available Memory	Minimum Reserved Memory
	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. The automatic memory reservation mechanism is enabled by default and requires the systems to have available memory more than the minimum threshold, which varies based on the system architecture.



IMPORTANT

The 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

- [How has the crashkernel parameter changed between RHEL8 minor releases?](#)
- [Technology capabilities and limits tables](#)

21.2. MINIMUM THRESHOLD FOR MEMORY RESERVATION

The **kexec-tools** utility, by default, configures the **crashkernel** command line parameter and reserves a certain amount of memory for **kdump**. For the default memory reservation to work, a certain amount of total memory must be available in the system. The amount of memory required differs based on the system's architecture. If the system has memory less than the specified threshold value, you must configure the memory manually.

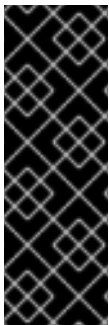
Table 21.2. Minimum amount of memory required for memory reservation

Architecture	Required Memory
AMD64 and Intel 64 (x86_64)	1 GB
IBM Power Systems (ppc64le)	2 GB
IBM Z (s390x)	1 GB
ARM (aarch64)	2 GB

21.3. SUPPORTED KDUMP TARGETS

When a kernel crash is captured, the **vmcore** dump file can be saved directly to a device, stored as a file on a local file system, or sent over a network. With the list of dump targets, you can understand the targets that are currently supported or not supported by **kdump**.

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.
Remote directories accessed using the iSCSI protocol over both hardware and software initiators.	Remote directories accessed using the iSCSI protocol on be2iscsi hardware.	Remote directories accessed over IPv6 .
		Remote directories accessed using the SMB or CIFS protocol.
		Remote directories accessed using wireless network interfaces.



IMPORTANT

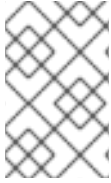
Utilizing firmware assisted dump (**fadump**) to capture a vmcore and store it to a remote machine using SSH or NFS protocol causes renaming of the network interface to **kdump-*<interface-name>***. The renaming happens if the **<interface-name>** is generic, for example ***eth#**, **net#**, and so on. 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.

21.4. 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. With the list of filtering levels, you can understand the levels that are currently supported by the **makedumpfile** utility.

Option	Description
1	Zero pages
2	Cache pages
4	Cache private

Option	Description
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.

21.5. 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. With the lists of default actions, you can understand the responses that are currently supported.

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.
final_action	Enable additional operations such as reboot , halt , and poweroff actions after a successful kdump or when shell or dump_to_rootfs failure action completes. The default final_action option is reboot .
failure_action	Specifies the action to perform when a dump might fail in the event of a kernel crash. The default failure_action option is reboot .

21.6. USING FINAL_ACTION PARAMETER

The **final_action** parameter enables you to use certain additional operations such as **reboot**, **halt**, and **poweroff** actions after a successful **kdump** or when an invoked **failure_action** mechanism using **shell** or **dump_to_rootfs** completes. If the **final_action** option is not specified, it defaults to **reboot**.

Procedure

1. To configure **final_action**, edit the **/etc/kdump.conf** file and add one of the following options:

```
# final_action <reboot | halt | poweroff>
```

2. Restart the **kdump** service for the changes to take effect:

```
# kdumpctl restart
```

21.7. USING FAILURE_ACTION PARAMETER

The **failure_action** parameter specifies the action to perform when a dump fails in the event of a kernel crash. The default action for **failure_action** is **reboot**, which reboots the system.

failure_action specifies one of the following actions to take:

- **reboot**: reboots the system after a dump failure.
- **dump_to_rootfs**: saves the dump file on a root file system when a non-root dump target is configured.
- **halt**: halts the system.
- **poweroff**: stops the running operations on the system.
- **shell**: starts a shell session inside **initramfs**, from which you can manually perform additional recovery actions.

Procedure:

1. To configure an action to take if the dump fails, edit the **/etc/kdump.conf** file and specify one of the **failure_action** options:

```
# failure_action <reboot | halt | poweroff | shell | dump_to_rootfs>
```

2. Restart the **kdump** service for the changes to take effect:

```
# kdumpctl restart
```

CHAPTER 22. FIRMWARE ASSISTED DUMP MECHANISMS

Firmware assisted dump (fadump) is a dump capturing mechanism, provided as an alternative to the **kdump** mechanism on IBM POWER systems. The **kexec** and **kdump** mechanisms are useful for capturing core dumps on AMD64 and Intel 64 systems. However, some hardware such as mini systems and mainframe computers, leverage the onboard firmware to isolate regions of memory and prevent any accidental overwriting of data that is important to the crash analysis. The **fadump** utility, is optimized for the **fadump** mechanisms and their integration with RHEL on IBM POWER systems.

22.1. FIRMWARE ASSISTED DUMP ON IBM POWERPC HARDWARE

The **fadump** utility captures the **vmcore** file from a fully-reset system with PCI and I/O devices. This mechanism uses firmware to preserve memory regions during a crash and then reuses the **kdump** userspace scripts to save the **vmcore** file. The memory regions consist of all system memory contents, except the boot memory, system registers, and hardware Page Table Entries (PTEs).

The **fadump** mechanism offers improved reliability over the traditional dump type, by rebooting the partition and using a new kernel to dump the data from the previous kernel crash. The **fadump** requires an IBM POWER6 processor-based or later version hardware platform.

For further details about the **fadump** mechanism, including PowerPC specific methods of resetting hardware, see the `/usr/share/doc/kexec-tools/fadump-howto.txt` file.



NOTE

The area of memory that is not preserved, known as boot memory, is the amount of RAM required to successfully boot the kernel after a crash event. By default, the boot memory size is 256MB or 5% of total system RAM, whichever is larger.

Unlike **kexec-initiated** event, the **fadump** mechanism uses the production kernel to recover a crash dump. When booting after a crash, PowerPC hardware makes the device node `/proc/device-tree/rtas/ibm.kernel-dump` available to the **proc** filesystem (**procfs**). The **fadump-aware kdump** scripts, check for the stored **vmcore**, and then complete the system reboot cleanly.

22.2. ENABLING FIRMWARE ASSISTED DUMP MECHANISM

You can enhance the crash dumping capabilities of IBM POWER systems by enabling the firmware assisted dump (**fadump**) mechanism.

In the Secure Boot environment, the **GRUB2** boot loader allocates a boot memory region, known as the Real Mode Area (RMA). The RMA has a size of 512 MB, which is divided among the boot components and, if a component exceeds its size allocation, **GRUB2** fails with an out-of-memory (**OOM**) error.



WARNING

Do not enable firmware assisted dump (**fadump**) mechanism in the Secure Boot environment on RHEL 9.1 and early versions. The **GRUB2** boot loader fails with the following error:

```
error: ../../grub-core/kern/mm.c:376:out of memory.
Press any key to continue...
```

The system is recoverable only if you increase the default **initramfs** size due to the **fadump** configuration.

For information about workaround methods to recover the system, see the [System boot ends in GRUB Out of Memory \(OOM\)](#) article.

Prerequisites

- You have root privileges.

Procedure

1. Install the **kexec-tools** package.
2. Configure the default value for **crashkernel**.

```
# kdumpctl reset-crashkernel --fadump=on --kernel=ALL
```

3. (Optional) Reserve boot memory instead of the default value.

```
# grubby --update-kernel ALL --args="fadump=on crashkernel=xxM"
```

where, **xx** is the required memory size in megabytes.



NOTE

When specifying boot configuration options, test the configurations by rebooting the kernel with **kdump** enabled. If the **kdump** kernel fails to boot, increase the **crashkernel** value gradually to set an appropriate value.

4. Reboot for changes to take effect.

```
# reboot
```

22.3. FIRMWARE ASSISTED DUMP MECHANISMS ON IBM Z HARDWARE

IBM Z systems supports two firmware assisted dump mechanisms: Stand-alone dump (**sadump**) and **VMDUMP** dump file.

The **kdump** infrastructure is supported and utilized on IBM Z systems. However, using one of the firmware assisted dump (**fadump**) methods for IBM Z can provide various benefits:

- The **sadump** mechanism is initiated and controlled from the system console, and is stored on an **IPL** bootable device.
- The **VMDUMP** mechanism is similar to **sadump**. This tool is initiated from the system console, but retrieves the resulting dump from hardware and copies it to the system for analysis.
- These methods, similar to other hardware-based dump mechanisms, have the ability to capture the state of a machine in the early boot phase, before the **kdump** service starts.
- Although **VMDUMP** contains a mechanism to receive the dump file into a Red Hat Enterprise Linux system, the configuration and control of **VMDUMP** is managed from the IBM Z Hardware console.

Additional resources

- [Using the Dump Tools on Red Hat Enterprise Linux 8.5](#)
- [Stand-alone dump](#)
- [Creating dumps on z/VM with VMDUMP](#)

22.4. USING SADUMP ON FUJITSU PRIMEQUEST SYSTEMS

The Fujitsu **sadump** mechanism is designed to provide a **fallback** dump capture in an event when **kdump** is unable to complete successfully. The **sadump** mechanism is invoked manually from the system Management Board (MMB) interface. Using MMB, configure **kdump** like for an Intel 64 or AMD 64 server and then proceed to enable **sadump**.

Procedure

1. Add or edit the following lines in the **/etc/sysctl.conf** file to ensure that **kdump** starts as expected for **sadump**:

```
kernel.panic=0
kernel.unknown_nmi_panic=1
```



WARNING

In particular, ensure that after **kdump**, the system does not reboot. If the system reboots after **kdump** has fails to save the **vmcore** file, then it is not possible to invoke the **sadump**.

2. Set the **failure_action** parameter in **/etc/kdump.conf** appropriately as **halt** or **shell**.

```
failure_action shell
```


Additional resources

- The FUJITSU Server PRIMEQUEST 2000 Series Installation Manual

CHAPTER 23. 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 Kernel Oops Analyzer or the Kdump Helper tool.

23.1. INSTALLING THE CRASH UTILITY

Install the **crash** tool to obtain the core analysis suite.

Procedure

1. Enable the relevant repositories:

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

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

```
# subscription-manager repos --enable rhel-9-for-x86_64-baseos-debug-rpms
```

2. Install the **crash** package:

```
# dnf install crash
```

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

```
# dnf install kernel-debuginfo
```

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

23.2. RUNNING AND EXITING THE CRASH UTILITY

Start the **crash** utility for analyzing the cause of the system crash.

Prerequisites

- Identify the currently running kernel (for example **5.14.0-1.el9.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/5.14.0-1.el9.x86_64/vmlinux** provided through a specific **kernel-debuginfo** package.
 - The actual vmcore file, for example **/var/crash/127.0.0.1-2021-09-13-14:05:33/vmcore**
The resulting **crash** command then looks like this:

```
# crash /usr/lib/debug/lib/modules/5.14.0-1.el9.x86_64/vmlinux /var/crash/127.0.0.1-2021-09-13-14:05:33/vmcore
```

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

Example 23.1. Running the crash utility

The following example shows analyzing a core dump created on September 13 2021 at 14:05 PM, using the 5.14.0-1.el9.x86_64 kernel.

```
...
WARNING: kernel relocated [202MB]: patching 90160 gdb minimal_symbol values

    KERNEL: /usr/lib/debug/lib/modules/5.14.0-1.el9.x86_64/vmlinux
    DUMPFILE: /var/crash/127.0.0.1-2021-09-13-14:05:33/vmcore [PARTIAL DUMP]
    CPUS: 2
    DATE: Mon Sep 13 14:05:16 2021
    UPTIME: 01:03:57
    LOAD AVERAGE: 0.00, 0.00, 0.00
    TASKS: 586
    NODENAME: localhost.localdomain
    RELEASE: 5.14.0-1.el9.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 23.2. 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.

Additional resources

- [A Guide to Unexpected System Restarts](#)

23.3. DISPLAYING VARIOUS INDICATORS IN THE CRASH UTILITY

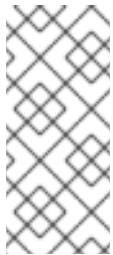
Use the **crash** utility to display various indicators, such as a kernel message buffer, a backtrace, a process status, virtual memory information and open files.

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> ffffffff 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 about 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 [ef4dbd0c] 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: ffffffff EBX: 00000001 ECX: b7776000 EDX: 00000002
    DS: 007b   ESI: 00000002 ES: 007b   EDI: b7776000
    SS: 007b   ESP: bfc2088 EBP: bfc20b4 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 about **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 about **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 about a single specific process, or use **help vm** for more information about **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 about **files** usage.

23.4. USING KERNEL OOPS ANALYZER

The Kernel Oops Analyzer tool 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.

Procedure

- Access the Kernel Oops Analyzer tool.
- To diagnose a kernel crash issue, upload a kernel oops log generated in **vmcore**.

- Alternatively you can also diagnose a kernel crash issue by providing a text message or a **vmcore-dmesg.txt** as an input.

Option 1: File Input

Choose File No file chosen

Choose and upload the [kernel oops log](#) generated from a vmcore.
Maximum file size for uploaded kernel oops log is 10 MB.

Detect

Option 2: Text Input

Detect **Clear**

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

Additional resources

- [The Kernel Oops Analyzer](#) article
- [A Guide to Unexpected System Restarts](#)

23.5. THE KDUMP HELPER TOOL

The Kdump Helper tool helps to set up the **kdump** using the provided information. Kdump Helper generates a configuration script based on your preferences. Initiating and running the script on your server sets up the **kdump** service.

Additional resources

- [Kdump Helper](#)