Red Hat Enterprise Linux 9.0 Beta

Managing, monitoring, and updating the kernel

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

Last Updated: 2022-04-06
Abstract

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

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>RHEL BETA RELEASE</td>
<td>6</td>
</tr>
<tr>
<td>MAKING OPEN SOURCE MORE INCLUSIVE</td>
<td>7</td>
</tr>
<tr>
<td>PROVIDING FEEDBACK ON RED HAT DOCUMENTATION</td>
<td>8</td>
</tr>
<tr>
<td><strong>CHAPTER 1. THE LINUX KERNEL RPM</strong></td>
<td>9</td>
</tr>
<tr>
<td>1.1. WHAT AN RPM IS</td>
<td>9</td>
</tr>
<tr>
<td>Types of RPM packages</td>
<td>9</td>
</tr>
<tr>
<td>1.2. THE LINUX KERNEL RPM PACKAGE OVERVIEW</td>
<td>9</td>
</tr>
<tr>
<td>1.3. DISPLAYING CONTENTS OF THE KERNEL PACKAGE</td>
<td>10</td>
</tr>
<tr>
<td><strong>CHAPTER 2. UPDATING KERNEL WITH DNF</strong></td>
<td>12</td>
</tr>
<tr>
<td>2.1. WHAT IS THE KERNEL</td>
<td>12</td>
</tr>
<tr>
<td>2.2. WHAT IS YUM</td>
<td>12</td>
</tr>
<tr>
<td>2.3. UPDATING THE KERNEL</td>
<td>12</td>
</tr>
<tr>
<td>2.4. INSTALLING THE KERNEL</td>
<td>12</td>
</tr>
<tr>
<td><strong>CHAPTER 3. MANAGING KERNEL MODULES</strong></td>
<td>14</td>
</tr>
<tr>
<td>3.1. INTRODUCTION TO KERNEL MODULES</td>
<td>14</td>
</tr>
<tr>
<td>3.2. KERNEL MODULE DEPENDENCIES</td>
<td>14</td>
</tr>
<tr>
<td>3.3. LISTING CURRENTLY LOADED KERNEL MODULES</td>
<td>15</td>
</tr>
<tr>
<td>3.4. SETTING A KERNEL AS DEFAULT</td>
<td>15</td>
</tr>
<tr>
<td>3.5. DISPLAYING INFORMATION ABOUT KERNEL MODULES</td>
<td>16</td>
</tr>
<tr>
<td>3.6. LOADING KERNEL MODULES AT SYSTEM RUNTIME</td>
<td>17</td>
</tr>
<tr>
<td>3.7. UNLOADING KERNEL MODULES AT SYSTEM RUNTIME</td>
<td>18</td>
</tr>
<tr>
<td>3.8. UNLOADING KERNEL MODULES AT EARLY STAGES OF THE BOOT PROCESS</td>
<td>19</td>
</tr>
<tr>
<td>3.9. LOADING KERNEL MODULES AUTOMATICALLY AT SYSTEM BOOT TIME</td>
<td>21</td>
</tr>
<tr>
<td>3.10. PREVENTING KERNEL MODULES FROM BEING AUTOMATICALLY LOADED AT SYSTEM BOOT TIME</td>
<td>21</td>
</tr>
<tr>
<td>3.11. COMPILING CUSTOM KERNEL MODULES</td>
<td>23</td>
</tr>
<tr>
<td><strong>CHAPTER 4. SIGNING KERNEL MODULES FOR SECURE BOOT</strong></td>
<td>26</td>
</tr>
<tr>
<td>4.1. REQUIREMENTS FOR AUTHENTICATING KERNEL MODULES WITH X.509 KEYS</td>
<td>27</td>
</tr>
<tr>
<td>4.2. SOURCES FOR PUBLIC KEYS</td>
<td>28</td>
</tr>
<tr>
<td>4.3. GENERATING A PUBLIC AND PRIVATE KEY PAIR</td>
<td>29</td>
</tr>
<tr>
<td>4.4. EXAMPLE OUTPUT OF SYSTEM KEYRINGS</td>
<td>30</td>
</tr>
<tr>
<td>4.5. ENROLLING PUBLIC KEY ON TARGET SYSTEM BY ADDING THE PUBLIC KEY TO THE MOK LIST</td>
<td>31</td>
</tr>
<tr>
<td>4.6. SIGNING KERNEL MODULES WITH THE PRIVATE KEY</td>
<td>32</td>
</tr>
<tr>
<td>4.7. LOADING SIGNED KERNEL MODULES</td>
<td>33</td>
</tr>
<tr>
<td><strong>CHAPTER 5. CONFIGURING KERNEL COMMAND-LINE PARAMETERS</strong></td>
<td>35</td>
</tr>
<tr>
<td>5.1. UNDERSTANDING KERNEL COMMAND-LINE PARAMETERS</td>
<td>35</td>
</tr>
<tr>
<td>5.2. WHAT GRUBBY IS</td>
<td>35</td>
</tr>
<tr>
<td>5.3. WHAT BOOT ENTRIES ARE</td>
<td>35</td>
</tr>
<tr>
<td>5.4. CHANGING KERNEL COMMAND-LINE PARAMETERS FOR ALL BOOT ENTRIES</td>
<td>36</td>
</tr>
<tr>
<td>5.5. CHANGING KERNEL COMMAND-LINE PARAMETERS FOR A SINGLE BOOT ENTRY</td>
<td>37</td>
</tr>
<tr>
<td>5.6. CHANGING KERNEL COMMAND-LINE PARAMETERS TEMPORARILY AT BOOT TIME</td>
<td>38</td>
</tr>
<tr>
<td><strong>CHAPTER 6. CONFIGURING KERNEL PARAMETERS AT RUNTIME</strong></td>
<td>39</td>
</tr>
<tr>
<td>6.1. WHAT ARE KERNEL PARAMETERS</td>
<td>39</td>
</tr>
<tr>
<td>6.2. CONFIGURING KERNEL PARAMETERS TEMPORARILY WITH SYSCTL</td>
<td>40</td>
</tr>
<tr>
<td>6.3. CONFIGURING KERNEL PARAMETERS PERMANENTLY WITH SYSCTL</td>
<td>40</td>
</tr>
<tr>
<td>6.4. USING CONFIGURATION FILES IN /ETC/SYSCTL.D/ TO ADJUST KERNEL PARAMETERS</td>
<td>41</td>
</tr>
</tbody>
</table>
RHEL BETA RELEASE

Red Hat provides Red Hat Enterprise Linux Beta access to all subscribed Red Hat accounts. The purpose of Beta access is to:

- Provide an opportunity to customers to test major features and capabilities prior to the general availability release and provide feedback or report issues.

- Provide Beta product documentation as a preview. Beta product documentation is under development and is subject to substantial change.

Note that Red Hat does not support the usage of RHEL Beta releases in production use cases. For more information, see What does Beta mean in Red Hat Enterprise Linux and can I upgrade a RHEL Beta installation to a General Availability (GA) release?.

Red Hat Enterprise Linux 9.0 Beta Managing, monitoring, and updating the kernel 6
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 input on our documentation. Please let us know how we could make it better.

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

- For submitting feedback via Bugzilla, create a new ticket:
  1. Go to the [Bugzilla](https://bugzilla.redhat.com/) website.
  2. As the Component, use **Documentation**.
  3. Fill in the **Description** field with your suggestion for improvement. Include a link to the relevant part(s) of documentation.
  4. Click **Submit Bug**.
CHAPTER 1. THE LINUX KERNEL RPM

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

1.1. WHAT AN RPM IS

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

Specifically, an RPM package consists of the cpio archive.

The cpio archive contains:

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

Types of RPM packages

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

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

- Binary RPM
  A binary RPM contains the binaries built from the sources and patches.

1.2. THE LINUX KERNEL RPM PACKAGE OVERVIEW

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

- kernel-core - contains the binary image of the kernel, all initramfs-related objects to bootstrap the system, and a minimal number of kernel modules to ensure core functionality. This sub-package alone could be used in virtualized and cloud environments to provide a Red Hat Enterprise Linux 9 kernel with a quick boot time and a small disk size footprint.

- kernel-modules - contains the remaining kernel modules that are not present in kernel-core.

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

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-whitelists** – 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.

### Additional resources

- *What are the kernel-core, kernel-modules, and kernel-modules-extras packages?*

### 1.3. DISPLAYING CONTENTS OF THE KERNEL PACKAGE

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

#### Prerequisites

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

#### Procedure

- List modules for **kernel**:

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

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

  ```bash
  $ 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**:

  ```bash
  $ 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
  ```
List modules for `kernel-modules-extra`:

```bash
$ rpm -qlp <kernel-modules-extra_rpm>
```

Additional resources

- `rpm(8)` manual page
- `RPM packages`
- `RPM packages`
- `RPM packages`
CHAPTER 2. UPDATING KERNEL WITH DNF

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

2.1. WHAT IS THE KERNEL

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

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

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

WARNING

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

2.2. WHAT IS YUM

This section refers to description of the dnf package manager.

Additional resources

- Configuring basic system settings in RHEL

2.3. UPDATING THE KERNEL

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

Procedure

1. To update the kernel, use the following:

```
# 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.

2.4. INSTALLING THE KERNEL
The following procedure describes how to install new kernels using the `dnf` package manager.

### Procedure

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

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

### Additional resources

- Red Hat Code Browser
- Red Hat Enterprise Linux Release Dates
CHAPTER 3. MANAGING KERNEL MODULES

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

3.1. INTRODUCTION TO KERNEL MODULES

The Red Hat Enterprise Linux kernel can be extended with optional, additional pieces of functionality, called kernel modules, without having to reboot the system. On Red Hat Enterprise Linux 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.

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

**WARNING**

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

Additional resources

- `modules.dep(5)` manual page
- `depmod(8)` manual page
3.3. LISTING CURRENTLY LOADED KERNEL MODULES

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

Prerequisites

- The `kmod` package is installed.

Procedure

- To list all currently loaded kernel modules, execute:

```
$ lsmod
```

<table>
<thead>
<tr>
<th>Module</th>
<th>Size</th>
<th>Used by</th>
</tr>
</thead>
<tbody>
<tr>
<td>fuse</td>
<td>126976</td>
<td>3</td>
</tr>
<tr>
<td>uinput</td>
<td>20480</td>
<td>1</td>
</tr>
<tr>
<td>xt_CHECKSUM</td>
<td>16384</td>
<td>1</td>
</tr>
<tr>
<td>ipt_MASQUERADE</td>
<td>16384</td>
<td>1</td>
</tr>
<tr>
<td>xt_contrack</td>
<td>16384</td>
<td>1</td>
</tr>
<tr>
<td>ipt_REJECT</td>
<td>16384</td>
<td>1</td>
</tr>
<tr>
<td>nft_counter</td>
<td>16384</td>
<td>16</td>
</tr>
<tr>
<td>nf_nat_tftp</td>
<td>16384</td>
<td>0</td>
</tr>
<tr>
<td>nf_connttrack_tftp</td>
<td>16384</td>
<td>1 nf_nat_tftp</td>
</tr>
<tr>
<td>tun</td>
<td>49152</td>
<td>1</td>
</tr>
<tr>
<td>bridge</td>
<td>192512</td>
<td>0</td>
</tr>
<tr>
<td>stp</td>
<td>16384</td>
<td>1 bridge</td>
</tr>
<tr>
<td>llc</td>
<td>16384</td>
<td>2 bridge,stp</td>
</tr>
<tr>
<td>nf_tables_set</td>
<td>32768</td>
<td>5</td>
</tr>
<tr>
<td>nft_fib_inet</td>
<td>16384</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

3.4. SETTING A KERNEL AS DEFAULT

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

Procedure

Setting the kernel as default, using the `grubby` tool
Execute the following command to set the kernel as default using the `grubby` tool:

```bash
# 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 `id` argument

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

  ```bash
  # 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:

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

**WARNING**

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

### 3.5. Displaying Information About Kernel Modules

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

**Prerequisites**

- The `kmod` package is installed.

**Procedure**
To display information about any kernel module, execute:

```bash
$ modinfo <KERNEL_MODULE_NAME>
```

For example:

```bash
$ 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
```

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

+ **NOTE**

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

Additional resources

- `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. The following procedure describes how to use the `modprobe` command to find and load a kernel module into the currently running kernel.

**Prerequisites**

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

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

2. Load the relevant kernel module:

   ```
   # modprobe <MODULE_NAME>
   ```

   **NOTE**

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

3. Optionally, verify the relevant module was loaded:

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

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

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

   **IMPORTANT**

   The changes described in this procedure will not persist after rebooting the system. For information on how to load kernel modules to persist across system reboots, see Loading kernel modules automatically at system boot time.

Additional resources

- `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. The following procedure describes how to use the `modprobe` command to find and unload a kernel module at system runtime from the currently loaded kernel.

Prerequisites

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

Procedure

1. Execute the `lsmod` command and select a kernel module you want to unload.

   If a kernel module has dependencies, unload those prior to unloading the kernel module. For details on identifying modules with dependencies, see 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 on 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 bootloader.

**IMPORTANT**

The changes described in this procedure will not persist after the next reboot. For information on 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 bootloader entry to unload the desired kernel module before the booting sequence continues.
  - Use the cursor keys to highlight the relevant bootloader entry.
  - Press e key to edit the entry.

Figure 3.1. Kernel boot menu

![Kernel boot menu]

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
textwidth=10
insmod grilo
linux (/boot/vmlinuz-5.14.0-63.el9.x86_64 root=UUID=0e3d6a9d-5a9d-4942-aad2-44f05c4333e8 rdinit=/dev/sda4) ro root=UUID=0e3d6a9d-5a9d-4942-aad2-44f05c4333e8 rdinit=/dev/sda4 quiet modprobe.blacklist=serio_raw
initrd (/boot/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
```
3.9. LOADING KERNEL MODULES AUTOMATICALLY AT SYSTEM BOOT TIME

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

Prerequisites

- Root permissions
- The kmod package is installed.

Procedure

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

2. Create a configuration file for the module:

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

   **NOTE**

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

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

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

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

   **IMPORTANT**

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

Additional resources

- modules-load.d(5) manual page

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

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

Prerequisites

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

• The `kmod` package is installed.

• Ensure that a kernel module in a denylist is not vital for your current system configuration.

**Procedure**

1. Select a kernel module that you want to put in a denylist:

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

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

   • Alternatively, identify an unloaded kernel module you want to prevent from potentially loading.
     All kernel modules are located in the `/lib/modules/<KERNEL_VERSION>/kernel/<SUBSYSTEM>/` directory.

2. Create a configuration file for a denylist:

   ```
   # vim /etc/modprobe.d/blacklist.conf
   # Blacklists <KERNEL_MODULE_1>
   blacklist <MODULE_NAME_1>
   install <MODULE_NAME_1> /bin/false

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5. Reboot the system:

```
$ reboot
```

IMPORTANT

The changes described in this procedure will take effect and persist after rebooting the system. If you improperly put a key kernel module in a denylist, you can face an unstable or non-operational system.

Additional resources

- How do I prevent a kernel module from loading automatically?
- dracut(8) manual page

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.

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:

```c
#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");
}
```

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/5.14.0-1.3.1.el9.x86_64'
CC [M] /root/testmodule/test.o
MODPOST /root/testmodule/Module.symvers
ERROR: modpost: missing MODULE_LICENSE() in /root/testmodule/test.o
make[1]: * [scripts/Makefile.modpost:150: /root/testmodule/Module.symvers] Error 1
make[1]: * [scripts/Makefile.modpost:150: /root/testmodule/Module.symvers] Error 1
make[1]: * Deleting file '/root/testmodule/Module.symvers'
make: * [Makefile:1776: modules] Error 2
make: Leaving directory '/usr/src/kernels/5.14.0-1.3.1.el9.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 452
   -rw-r--r-- 1 root root    16 Jul 12 10:16 Makefile
   -rw-r--r-- 1 root root   32 Jul 12 10:16 modules.order
   -rw-r--r-- 1 root root    0 Jul 12 10:16 Module.symvers
   -rw-r--r-- 1 root root   197 Jul 12 10:15 test.c
   -rw-r--r-- 1 root root  219736 Jul 12 10:16 test.ko
   -rw-r--r-- 1 root root   826 Jul 12 10:16 test.mod.c
   -rw-r--r-- 1 root root  113760 Jul 12 10:16 test.mod.o
   -rw-r--r-- 1 root root  107424 Jul 12 10:16 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
   ```

   1. Verify that the kernel module was successfully loaded:

   ```
   # Ismod | grep test
   test 16384 0
   ```

   2. 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. SIGNING KERNEL MODULES FOR SECURE BOOT

You can enhance the security of your system by using signed kernel modules. The following sections describe how to self-sign privately built kernel modules for use with RHEL 9 on UEFI-based build systems where Secure Boot is enabled. These sections also provide an overview of available options for importing your public key into a target system where you want to deploy your kernel modules.

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

The RHEL 9 distribution includes:

- Signed boot loaders
- Signed kernels
- Signed kernel modules

In addition, the signed first-stage boot loader and the signed kernel include embedded Red Hat public keys. These signed executable binaries and embedded keys enable RHEL 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 that not all UEFI-based systems include support for Secure Boot.

Prerequisites

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

Table 4.1. Required utilities

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

NOTE

The build system, where you build and sign your kernel module, does not need to have UEFI Secure Boot enabled and does not even need to be a UEFI-based system.
4.1. REQUIREMENTS FOR AUTHENTICATING KERNEL MODULES WITH X.509 KEYS

In RHEL 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. The following sections provide an overview of sources of keys, keyrings and examples of loaded keys from different sources in the system. Also, you can see how to authenticate a kernel module.

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 4.2. Kernel module authentication requirements for loading

<table>
<thead>
<tr>
<th>Module signed</th>
<th>Public key found and signature valid</th>
<th>UEFI Secure Boot state</th>
<th>sig_enforce</th>
<th>Module load</th>
<th>Kernel tainted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsigned</td>
<td>-</td>
<td>Not enabled</td>
<td>Not enabled</td>
<td>Succeeds</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Not enabled</td>
<td>Enabled</td>
<td>Fails</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Enabled</td>
<td>-</td>
<td>Fails</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Signed</td>
<td>No</td>
<td>Not enabled</td>
<td>Not enabled</td>
<td>Succeeds</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Not enabled</td>
<td>Enabled</td>
<td>Fails</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Enabled</td>
<td>-</td>
<td>Fails</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Signed</td>
<td>Yes</td>
<td>Not enabled</td>
<td>Not enabled</td>
<td>Succeeds</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Not enabled</td>
<td>Enabled</td>
<td>Succeeds</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

CHAPTER 4. SIGNING KERNEL MODULES FOR SECURE BOOT
4.2. 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 4.3. Sources for system keyrings

<table>
<thead>
<tr>
<th>Source of X.509 keys</th>
<th>User can add keys</th>
<th>UEFI Secure Boot state</th>
<th>Keys loaded during boot</th>
</tr>
</thead>
<tbody>
<tr>
<td>Embedded in kernel</td>
<td>No</td>
<td>-</td>
<td>.builtin_trusted_keys</td>
</tr>
<tr>
<td>UEFI Secure Boot &quot;db&quot;</td>
<td>Limited</td>
<td>Not enabled</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Enabled</td>
<td>.platform</td>
</tr>
<tr>
<td>Embedded in shim.efi boot</td>
<td>No</td>
<td>Not enabled</td>
<td>No</td>
</tr>
<tr>
<td>loader</td>
<td></td>
<td>Enabled</td>
<td>.builtin_trusted_keys</td>
</tr>
<tr>
<td>Machine Owner Key (MOK)</td>
<td>Yes</td>
<td>Not enabled</td>
<td>No</td>
</tr>
<tr>
<td>list</td>
<td></td>
<td>Enabled</td>
<td>.platform</td>
</tr>
</tbody>
</table>

.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 Boot db:
- a signature database
- stores keys (hashes) of UEFI applications, UEFI drivers, and bootloaders
- the keys can be loaded on the machine

UEFI Secure Boot dbx:
- a revoked signature database
- prevents keys from being loaded
- the revoked keys from this database are added to the blacklist keyring

4.3. GENERATING A PUBLIC AND PRIVATE KEY PAIR

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

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

Procedure

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

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

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

[ myexts ]
basicConstraints=critical,CA:FALSE
keyUsage=digitalSignature
subjectKeyIdentifier=hash
authorityKeyIdentifier=keyid
EOF
```
2. Create an X.509 public and private key pair as shown in the following example:

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

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

**IMPORTANT**

In RHEL 9, the validity dates of the key pair do not matter. The key does not expire, but it is recommended for good security practices that the kernel module be signed within the validity period of its signing key. However, the `sign-file` utility will not warn you and the key will be usable regardless of the validity dates.

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

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

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

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

**WARNING**

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

Additional resources

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

4.4. EXAMPLE OUTPUT OF SYSTEM KEYRINGS

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

The following is a shortened example output of `.builtin_trusted_keys`, `.platform`, and `.blacklist` keyrings from a RHEL 9 system where UEFI Secure Boot is enabled.
Prerequisites

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

```
# 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): 4016841644ce3a810408050766ed8fa29...
...asymmetric: Microsoft Corporation UEFI CA 2011: 13adbef3039b4309b870c904316e58a25...
...asymmetric: Microsoft Windows Production PCA 2011: a92902398e16cd4977818d90f99e5a...
...asymmetric: Red Hat Enterprise Linux kernel signing key: 4249689eefc77e95880b7f0b7...
...asymmetric: Red Hat Enterprise Linux kpatch signing key: 4d38fd864ebe18c5f0b7...
```

```
# keyctl list %:.platform
4 keys in keyring:
...asymmetric: VMware, Inc.: 4adb8da0472073...
...asymmetric: Red Hat Secure Boot CA 5: cc6faf6e72...
...asymmetric: Microsoft Windows Production PCA 2011: a929f298e1...
...asymmetric: Microsoft Corporation UEFI CA 2011: 13adbef4e0bd82...
```

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

The `.builtin_trusted_keys` keyring above shows the addition of two keys from the UEFI Secure Boot "db" keys as well as the **Red Hat Secure Boot (CA key 1)**, which is embedded in the `shim.efi` boot loader.

The following example shows the kernel console output. The messages identify the keys with a 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: 13adbef4309...
[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

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

When RHEL 9 boots on a UEFI-based system with Secure Boot enabled, the kernel loads onto the system keyring `.builtin_trusted_keys` 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. The sections below
describe different ways of importing a public key on a target system so that the system keyring
(.builtin_trusted_keys) is able to use the public key to authenticate a kernel module.

The Machine Owner Key (MOK) facility feature can be used to expand the UEFI Secure Boot key
database. When RHEL 9 boots on a UEFI-enabled system with Secure Boot enabled, the keys on the
MOK list are also added to the system keyring (.builtin_trusted_keys) in addition to the keys from the
key database. The MOK list keys are also stored persistently and securely in the same fashion as the
Secure Boot database keys, but these are two separate facilities. The MOK facility is supported by
shim.efi, MokManager.efi, grubx64.efi, and the mokutil utility.

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

Procedure

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

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

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

2. Reboot the machine.

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

3. Choose “Enroll MOK” and 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 system keyring on this and
subsequent boots when UEFI Secure Boot is enabled.

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.

4.6. SIGNING KERNEL MODULES WITH THE PRIVATE KEY

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

Prerequisites

- You generated a public and private key pair and know the validity dates of your public keys. For
details, see Generating a public and private key pair.

- You 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

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

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

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

**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` will not be able to display the signature on your kernel module.

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

**IMPORTANT**

In RHEL 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.

Additional resources

- Displaying information about kernel modules

### 4.7. 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 finally load your signed kernel module with the the `modprobe` command as described in the following section.

**Prerequisites**

- You have generated the public and private keypair. 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](#).

**Procedure**

1. Verify that your public keys are on the system keyring:
2. Install the `kernel-modules-extra` package which will create the `/lib/modules/$(uname -r)/extra/` directory:

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

3. Copy the kernel module into the `/extra/` directory of the kernel you want:

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

4. Update the modular dependency list:

```
# depmod -a
```

5. Load the kernel module and verify that it was successfully loaded:

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

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

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

Additional resources

- Managing kernel modules
CHAPTER 5. CONFIGURING KERNEL COMMAND-LINE PARAMETERS

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

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

5.1. UNDERSTANDING KERNEL COMMAND-LINE PARAMETERS

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

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

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

By default, the kernel command-line parameters for systems using the GRUB2 bootloader are defined in the boot entry configuration file for each kernel boot 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

5.2. WHAT GRUBBY IS

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

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

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

5.3. WHAT BOOT ENTRIES ARE

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

d8712ab6d4f14683c5625e87b52b66b6e-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
```

Additional resources

- How to install and boot custom kernels in Red Hat Enterprise Linux 8

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

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

Prerequisites

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

**IMPORTANT**

Newly installed kernels do not inherit the kernel command-line parameters from your previously configured kernels. However, the kernel command-line parameters are stored in the /etc/default/grub file. As a result, you can run the **grub2-mkconfig** command on the newly installed kernel to get the needed parameters propagated to your new kernel.

Procedure

- To add a parameter:

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

  For systems that use the GRUB2 bootloader and, on IBM Z that use the zIPL bootloader, the command adds a new kernel parameter to each /boot/loader/entries/<ENTRY>.conf file.

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

  - On IBM Z, execute the **zipl** command with no options to update the boot menu.
To remove a parameter:

```
# grubby --update-kernel=ALL --remove-args="<PARAMETER_TO_REMOVE>"
```
- On IBM Z, execute the `zipl` command with no options to update the boot menu.

Additional resources

- Understanding kernel command-line parameters
- `grubby(8)` and `zipl(8)` manual pages
- `grubby` tool

### 5.5. Changing Kernel Command-Line Parameters for a Single Boot Entry

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

**Prerequisites**

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

**Procedure**

- To add a parameter:

```
# grubby --update-kernel=/boot/vmlinuz-$(uname -r) --args="<NEW_PARAMETER>"
```
- On IBM Z, execute the `zipl` command with no options to update the boot menu.

- To remove a parameter use the following:

```
# grubby --update-kernel=/boot/vmlinuz-$(uname -r) --remove-args="<PARAMETER_TO_REMOVE>"
```
- On IBM Z, execute the `zipl` command with no options to update the boot menu.

**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**

- Understanding kernel command-line parameters
- `grubby(8)` and `zipl(8)` manual pages
- `grubby` tool
5.6. CHANGING KERNEL COMMAND-LINE PARAMETERS TEMPORARILY AT BOOT TIME

The following procedure allows you to 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 \textit{e} key to edit the kernel parameters.

2. Find the kernel command line by moving the cursor down. The kernel command line starts with \texttt{linux} on 64-Bit IBM Power Series and x86-64 BIOS-based systems, or \texttt{linuxefi} on UEFI systems.

3. Move the cursor to the end of the line.

\textbf{NOTE}

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

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

\begin{verbatim}
\end{verbatim}

+ To enable the system messages, remove the \texttt{rhgb} and \texttt{quiet} parameters.

1. Press \texttt{Ctrl+x} to boot with the selected kernel and the modified command line parameters.

\textbf{IMPORTANT}

Press \texttt{Esc} key to leave command line editing and it will drop all the user made changes.

\textbf{NOTE}

This procedure applies only for a single boot and does not persistently make the changes.
CHAPTER 6. CONFIGURING KERNEL PARAMETERS AT RUNTIME

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

6.1. 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 6.1. Table of sysctl classes**

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

### 6.2. CONFIGURING KERNEL PARAMETERS TEMPORARILY WITH SYSCTL

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

**Prerequisites**

- Root permissions

**Procedure**

1. To list all parameters and their values, use the following:
   ```
   # sysctl -a
   ```

   **NOTE**

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

2. To configure a parameter temporarily, use the command as in the following example:
   ```
   # sysctl <TUNABLE_CLASS>.<PARAMETER>=<TARGET_VALUE>
   ```

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

   **NOTE**

   The changes return back to default after your system reboots.

**Additional resources**

- `sysctl(8)` manual page
- Configuring kernel parameters permanently with `sysctl`
- Using configuration files in `/etc/sysctl.d/` to adjust kernel parameters

### 6.3. CONFIGURING KERNEL PARAMETERS PERMANENTLY WITH SYSCTL
The following procedure describes how to use the `sysctl` command to permanently set kernel parameters.

**Prerequisites**
- Root permissions

**Procedure**

1. To list all parameters, use the following:
   ```
   # sysctl -a
   ```
   The command displays all kernel parameters that can be configured at runtime.

2. To configure a parameter permanently:
   ```
   # sysctl -w <TUNABLE_CLASS>.<PARAMETER>=<TARGET_VALUE> >> /etc/sysctl.conf
   ```
   The sample command changes the tunable value and writes it to the `/etc/sysctl.conf` file, which overrides the default values of kernel parameters. The changes take effect immediately and persistently, without a need for restart.

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

**Additional resources**
- `sysctl(8)` and `sysctl.conf(5)` manual pages
- Using configuration files in `/etc/sysctl.d/` to adjust kernel parameters

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

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

**Prerequisites**
- Root permissions

**Procedure**

1. Create a new configuration file in `/etc/sysctl.d/`:
   ```
   # vim /etc/sysctl.d/<some_file.conf>
   ```

2. Include kernel parameters, one per line, as follows:
3. Save the configuration file.

4. Reboot the machine for the changes to take effect.
   - Alternatively, to apply changes without rebooting, execute:

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

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

Additional resources

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

### 6.5. CONFIGURING KERNEL PARAMETERS TEMPORARILY THROUGH `/PROC/SYS/`

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

**Prerequisites**

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

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

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

### 7.1. 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.

### 7.2. 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.

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>0</td>
<td>kernel does not panic on soft lockup</td>
</tr>
<tr>
<td>Integer</td>
<td>1</td>
<td>kernel panics on soft lockup</td>
</tr>
</tbody>
</table>

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.
<table>
<thead>
<tr>
<th>Value</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>disables lockup detector</td>
</tr>
<tr>
<td>1</td>
<td>enables lockup detector</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Default threshold</th>
<th>Soft lockup threshold</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 seconds</td>
<td>2 * watchdog_thresh</td>
</tr>
</tbody>
</table>

Setting this parameter to zero disables lockup detection altogether.

**Additional resources**

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

### 7.3. 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 components and their interaction
• Virtual machine components and their interaction
• Virtual machine reports a "BUG: soft lockup"
CHAPTER 8. ADJUSTING KERNEL PARAMETERS FOR DATABASE SERVERS

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

8.1. 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

8.2. 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.shmmmax**

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

**kernel.shmni**
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 9. 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.

9.1. 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

9.2. 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 17 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 17 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 10. INSTALLING KDUMP

In many cases, the \texttt{kdump} service is installed and activated by default on the new Red Hat Enterprise Linux installations. This section includes information about \texttt{kdump}.

10.1. WHAT IS KDUMP

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

\textbf{IMPORTANT}

A kernel crash dump can be the only information available in the event of a system failure (a critical bug). Therefore, ensuring that \texttt{kdump} is operational is important in mission-critical environments. Red Hat advise that system administrators regularly update and test \texttt{kexec-tools} in your normal kernel update cycle. This is especially important when new kernel features are implemented.
CHAPTER 11. 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) 2 configuration file. The memory size depends on the `crashkernel=` value specified in the configuration file and the size of the system’s physical memory.

### 11.1. CONFIGURING KDUMP MEMORY USAGE

This procedure describes how to configure memory usage for *kdump*.

The `crashkernel` default mechanism contains the default `crashkernel` memory reservation values for the corresponding kernel build. *kdump* uses the default value to reserve the `crashkernel` memory for each kernel. You can also use the default value as the reference base value to estimate a required memory size when setting the `crashkernel=` value manually. The minimum size of the crash kernel might vary depending on the hardware and machine specifications.

The automatic memory allocation for *kdump* varies based on the system hardware architecture and available memory size. For example, on AMD64 and Intel 64, the `crashkernel` default parameters work only when the available memory is more than 1GB. Hence, by default, on AMD64 and Intel 64, the `crashkernel` mechanism configures the following memory reserve:

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

**NOTE**

The `crashkernel` default mechanism is supported on the *kdump* tool and does not require manual editing. It is recommended to not modify the default `crashkernel` parameters manually.

Red Hat recommends testing the *kdump* after you configure the `crashkernel=` value and then configure an appropriate memory size to set an acceptable value.

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

### Prerequisites

- Fulfilled *kdump* requirements for configurations and targets

### Procedure

1. To use the `crashkernel` default parameters, run the following command:

   ```bash
   kdumpctl reset-crashkernel
   ```

2. (Optional) To configure parameters other than the default, fetch an estimate of the `crashkernel=` value using the *kdumpctl* command and then set it to the kernel parameters:
   
   a. To estimate the `crashkernel=` value, run the *kdumpctl* command:

   ```bash
   kdumpctl estimate
   ```

   The *kdumpctl* sub-command allows you to fetch a rough estimate value without triggering a *kdump*. 
The command prints a summary report, which states the estimated memory consumption of each **kdump** component.

For example:

```
Encrypted kdump target requires extra memory, assuming using the keyslot with minimum memory requirement

Reserved crashkernel: 256M
Recommended crashkernel: 655M

Kernel image size: 47M
Kernel modules size: 12M
Initramfs size: 19M
Runtime reservation: 64M
LUKS required size: 512M
Large modules:
  xfs: 1892352
  nouveau: 2318336

WARNING: Current crashkernel size is lower than recommended size 655M.
```

Using the **kdumpctl** sub-command you can fetch a rough estimate value without triggering a **kdump**.

The estimated **crashkernel=** value may not be accurate and can serve as a reference to set an appropriate **crashkernel=** value.

b. Set the **crashkernel=** option to the required value.

   For example, to reserve 128 MB of memory, configure as follows:

   ```
   crashkernel=128M
   ```

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

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

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

   - Offset the reserved memory.

   Some systems need you to reserve memory with a specific fixed offset because the **crashkernel** reservation is very early, and might require some reservation for special usage. If the offset is set, the reserved memory begins there. To offset the reserved memory, use the following syntax:

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

   The command reserves 128 MB of memory starting at 16 MB (physical address 0x01000000). If the offset parameter is set to 0 or omitted entirely, **kdump** offsets the reserved memory automatically. This syntax can also be used when setting a variable memory reservation as described above. In this case, the offset is always specified last (for example, **crashkernel=512M-2G:64M,2G-:128M@16M**).
3. Run the following command to update the GRUB2 configuration file:

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

4. Reboot the system for changes to take effect.

   NOTE

   You can use the grubby utility to update kernel command line parameters of just one entry.

Additional resources

- Memory requirements for kdump
- Configuring kernel command line parameters
- How to manually modify the boot parameter in grub before the system boots
- How to install and boot custom kernels in Red Hat Enterprise Linux 8

11.2. CONFIGURING THE KDUMP TARGET

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

Prerequisites

- Fulfilled kdump requirements for configurations and targets.

Procedure

- To store the vmcore file in /var/crash/ directory of the local file system, edit the /etc/kdump.conf file and specify the path:

  path /var/crash

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

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

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

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

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

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

1. Remove the hash sign (“#”) from the beginning of the path /var/crash line.
2. Replace the value with the intended directory path. For example:

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

IMPORTANT

In 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. This behavior is different from earlier releases of RHEL, where the directory was being created automatically if it did not exist when starting the service.

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

1. Remove the hash sign (“#”) from the beginning of the ext4 line, depending on your choice.
   - device name (the ext4 /dev/vg/lv_kdump line)
   - file system label (the ext4 LABEL=/boot line)
   - UUID (the ext4 UUID=03138356-5e61-4ab3-b58e-27507ac41937 line)
2. Change the file system type as well as the device name, label or UUID to the desired values. For example:

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

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

To write the dump directly to a device:
1. Remove the hash sign ("#") from the beginning of the `#raw /dev/vg/lv_kdump` line.
2. Replace the value with the intended device name. For example:
   ```
   raw /dev/sdb1
   ```

To store the dump to a remote machine using the NFS protocol:
1. Remove the hash sign ("#") from the beginning of the `#nfs my.server.com:/export/tmp` line.
2. Replace the value with a valid hostname and directory path. For example:
   ```
   nfs penguin.example.com:/export/cores
   ```

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

### 11.3. Configuring the KDump Core Collector

The `kdump` service uses a `core_collector` program to capture the crash dump image. In RHEL, the `makedumpfile` utility is the default core collector. It helps shrink the dump file by:

- Compressing the size of a crash dump file and copying only necessary pages using various dump levels
- Excluding unnecessary crash dump pages
Filtering the page types to be included in the crash dump.

Syntax

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

Options

- `-c`, `-l`, or `-p`: specify compress dump file format by each page using either, `zlib` for `-c` option, `lzo` for `-l` option or `snappy` for `-p` option.
- `-d (dump_level)`: excludes pages so that they are not copied to the dump file.
- `--message-level`: specify the message types. You can restrict outputs printed by specifying `message_level` with this option. For example, specifying 7 as `message_level` prints common messages and error messages. The maximum value of `message_level` is 31.

Prerequisites

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

Procedure

1. As `root`, edit the `/etc/kdump.conf` configuration file and remove the hash sign ("#") from the beginning of the `#core_collector makedumpfile -l --message-level 1 -d 31`.

2. To enable crash dump file compression, execute:

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

The `-l` option specifies the `dump` compressed file format. The `-d` option specifies dump level as 31. The `--message-level` option specifies message level as 1.

Also, consider following examples with the `-c` and `-p` options:

- To compress a crash dump file using `-c`:

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

- To compress a crash dump file using `-p`:

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

Additional resources

- [makedumpfile(8)] manual page
- The `kdump` configuration file

11.4. 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**
- Root permissions.
- Fulfilled requirements for `kdump` configurations and targets. For details, see Supported `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
   ```

**Additional resources**
- Configuring the `kdump` target

### 11.5. 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 to append the `kdump` kernel command-line to obtain a detailed debugging output.

This section covers information on modifying the `KDUMP_COMMANDLINE_REMOVE` and `KDUMP_COMMANDLINE_APPEND` options for `kdump`. For information on 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

### 11.6. 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 \texttt{kptr\_restrict=1} as default. When \texttt{kptr\_restrict} is set to (1) as default, the \texttt{kdumpctl} service loads the crash kernel even if Kernel Address Space Layout (KASLR) is enabled or not enabled.

Troubleshooting step

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

To work around this problem, the \texttt{kexec-kdump-howto.txt} file displays a warning message, which specifies to keep the recommended setting as \texttt{kptr\_restrict=1}.

To ensure that \texttt{kdumpctl} service loads the crash kernel, verify that:

- Kernel \texttt{kptr\_restrict=1} in the \texttt{sysctl.conf} file.

Additional resources

- Configuring basic system settings

11.7. TESTING THE KDUMP CONFIGURATION

You can test that the crash dump process works and is valid before the machine enters production.

WARNING

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

Procedure

1. Reboot the system with \texttt{kdump} enabled.

2. Make sure that \texttt{kdump} is running:

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

3. Force the Linux kernel to crash:

   ```bash
   echo 1 > /proc/sys/kernel/sysrq
   echo c > /proc/sysrq-trigger
   ```
WARNING
The command above crashes the kernel, and a reboot is required.

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

**NOTE**
This action confirms the validity of the configuration. Also it is possible to use this action to record how long it takes for a crash dump to complete with a representative work-load.

Additional resources

- Configuring the kdump target

### 11.8. PREVENTING KERNEL DRIVERS FROM LOADING FOR KDUMP

This section explains how to prevent the capture kernel from loading certain kernel drivers using the **/etc/sysconfig/kdump** configuration file. You can prevent the kdump initramfs from loading the specified kernel module. To achieve this, you need to put the **KDUMP_COMMANDLINE_APPEND** variable in the **/etc/sysconfig/kdump** file. This helps to prevent the out-of-memory (oom) killer 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.

   ```bash
   $ 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.

   ```bash
   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` manual page
CHAPTER 12. CONFIGURING KDUMP IN THE WEB CONSOLE

Setup and test the kdump configuration in the RHEL 9 web console.

The web console is part of a default installation of RHEL 9 and enables or disables the kdump service at boot time. Further, the web console conveniently enables you to configure the reserved memory for kdump; or to select the vmcore saving location in an uncompressed or compressed format.

12.1. ADDITIONAL RESOURCES

- Getting started using the RHEL web console

12.2. CONFIGURING KDUMP MEMORY USAGE AND TARGET LOCATION IN WEB CONSOLE

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

Procedure

1. Open the Kernel Dump tab and start the kdump service.
2. Configure the kdump memory usage using the command line.
3. Click the link next to the Crash dump location option.
4. Select the Local Filesystem option from the drop-down and specify the directory you want to save the dump in.
Alternatively, select the **Remote over SSH** option from the drop-down to send the vmcore to a remote machine using the SSH protocol. Fill the **Server**, **ssh key**, and **Directory** fields with the remote machine address, ssh key location, and a target directory.

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

**NOTE**

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

5. Test your configuration by crashing the kernel.

**WARNING**

This step disrupts execution of the kernel and results in a system crash and loss of data.
Additional resources

- Supported kdump targets
- Using secure communications between two systems with OpenSSH
13.1. 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
```

Table 13.1 lists the minimum memory requirements to automatically reserve a memory size for `kdump` on the latest available versions. The size changes according to the system’s architecture and total available physical memory.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Available Memory</th>
<th>Minimum Reserved Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD64 and Intel 64 (<strong>x86_64</strong>)</td>
<td>1 GB to 4 GB</td>
<td>160 MB of RAM.</td>
</tr>
<tr>
<td></td>
<td>4 GB to 64 GB</td>
<td>192 MB of RAM.</td>
</tr>
<tr>
<td></td>
<td>64 GB to 1 TB</td>
<td>256 MB of RAM.</td>
</tr>
<tr>
<td></td>
<td>1 TB and more</td>
<td>512 MB of RAM.</td>
</tr>
<tr>
<td>64-bit ARM architecture (<strong>arm64</strong>)</td>
<td>2 GB and more</td>
<td>448 MB of RAM.</td>
</tr>
<tr>
<td>IBM Power Systems (<strong>ppc64le</strong>)</td>
<td>2 GB to 4 GB</td>
<td>384 MB of RAM.</td>
</tr>
<tr>
<td></td>
<td>4 GB to 16 GB</td>
<td>512 MB of RAM.</td>
</tr>
<tr>
<td></td>
<td>16 GB to 64 GB</td>
<td>1 GB of RAM.</td>
</tr>
<tr>
<td></td>
<td>64 GB to 128 GB</td>
<td>2 GB of RAM.</td>
</tr>
<tr>
<td></td>
<td>128 GB and more</td>
<td>4 GB of RAM.</td>
</tr>
<tr>
<td>IBM Z (<strong>s390x</strong>)</td>
<td>1 GB to 4 GB</td>
<td>160 MB of RAM.</td>
</tr>
<tr>
<td></td>
<td>4 GB to 64 GB</td>
<td>192 MB of RAM.</td>
</tr>
<tr>
<td></td>
<td>64 GB to 1 TB</td>
<td>256 MB of RAM.</td>
</tr>
</tbody>
</table>
On many systems, **kdump** is able to estimate the amount of required memory and reserve it automatically. This behavior is enabled by default, but only works on systems that have more than a certain amount of total available memory, which varies based on the system architecture.

### 13.2. Minimum Threshold for Automatic Memory Reservation

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

The table below lists the threshold values for automatic memory allocation. If the system has memory less than the specified threshold value, you must configure the memory manually.

#### Table 13.2. Minimum Amount of Memory Required for Automatic Memory Reservation

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Required Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD64 and Intel 64 (<strong>x86_64</strong>)</td>
<td>2 GB</td>
</tr>
<tr>
<td>IBM Power Systems (<strong>ppc64le</strong>)</td>
<td>2 GB</td>
</tr>
<tr>
<td>IBM Z (<strong>s390x</strong>)</td>
<td>4 GB</td>
</tr>
</tbody>
</table>

**IMPORTANT**

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

Additional resources

- How has the crashkernel parameter changed between RHEL8 minor releases?
- Technology capabilities and limits tables
- Configuring kdump memory usage
- Configuring kdump memory usage and target location in web console
- Minimum threshold for automatic memory reservation
13.3. SUPPORTED KDUMP TARGETS

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Supported Targets</th>
<th>Unsupported Targets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Raw device</td>
<td>All locally attached raw disks and partitions.</td>
<td>Any local file system not explicitly listed as supported in this table, including the <code>auto</code> type (automatic file system detection).</td>
</tr>
<tr>
<td>Local file system</td>
<td>ext2, ext3, ext4, and xfs file systems on directly attached disk drives, hardware RAID logical drives, LVM devices, and mdraid arrays.</td>
<td></td>
</tr>
<tr>
<td>Remote directories accessed using the iSCSI protocol over both hardware and software initiators.</td>
<td>Remote directories accessed using the iSCSI protocol on be2iscsi hardware.</td>
<td>Multipath-based storages.</td>
</tr>
<tr>
<td>Remote directories accessed using the SMB or CIFS protocol.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remote directories accessed using the FCoE (Fibre Channel over Ethernet) protocol.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remote directories accessed using wireless network interfaces.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
IMPORTANT

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

Additional resources

- Configuring kdump target
- Configuring kdump memory usage and target location in web console

13.4. SUPPORTED KDUMP FILTERING LEVELS

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

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Zero pages</td>
</tr>
<tr>
<td>2</td>
<td>Cache pages</td>
</tr>
<tr>
<td>4</td>
<td>Cache private</td>
</tr>
<tr>
<td>8</td>
<td>User pages</td>
</tr>
<tr>
<td>16</td>
<td>Free pages</td>
</tr>
</tbody>
</table>

NOTE

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

Additional resources

- Configuring the core collector

13.5. SUPPORTED DEFAULT FAILURE RESPONSES

By default, when kdump fails to create a core dump, the operating system reboots. You can, however, configure kdump to perform a different operation in case it fails to save the core dump to the primary target. The table below lists all default actions that are currently supported.
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>dump_to_rootfs</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>reboot</strong></td>
<td>Reboot the system, losing the core dump in the process.</td>
</tr>
<tr>
<td><strong>halt</strong></td>
<td>Halt the system, losing the core dump in the process.</td>
</tr>
<tr>
<td><strong>poweroff</strong></td>
<td>Power off the system, losing the core dump in the process.</td>
</tr>
<tr>
<td><strong>shell</strong></td>
<td>Run a shell session from within the initramfs, allowing the user to record the core dump manually.</td>
</tr>
<tr>
<td><strong>final_action</strong></td>
<td>Enable additional operations such as <strong>reboot</strong>, <strong>halt</strong>, and <strong>poweroff</strong> actions after a successful <strong>kdump</strong> or when an invoked <strong>failure_response</strong> mechanism using <strong>shell</strong> or <strong>dump_to_rootfs</strong> failure action completes. The default <strong>final_action</strong> option is <strong>reboot</strong>.</td>
</tr>
</tbody>
</table>

**Additional resources**
- Configuring the kdump default failure responses

**13.6. 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_response** mechanism using **shell** or **dump_to_rootfs** failure action completes. If the **final_action** option is not specified, it defaults to **reboot**.

**Procedure**

1. Edit the `/etc/kdump.conf` file and add the **final_action** parameter.

   ```
   final_action <reboot | halt | poweroff>
   ```

2. Restart the **kdump** service:

   ```
   kdumpctl restart
   ```
CHAPTER 14. 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.

14.1. INSTALLING THE CRASH UTILITY

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

Procedure

1. Enable the relevant repositories:
   # subscription-manager repos --enable baseos repository
   # subscription-manager repos --enable appstream repository
   # subscription-manager repos --enable rhel-8-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 your running kernel and provides the data necessary for the dump analysis.

Additional resources

- Configuring basic system settings

14.2. RUNNING AND EXITING THE CRASH UTILITY

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

Prerequisites

- Identify the currently running kernel (for example 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 14.1. 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)
```

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

Example 14.2. 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
# 14.3. Displaying Various Indicators in the Crash Utility

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

## Displaying the Message Buffer

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

```
crash> log
... several lines omitted ...
EIP: 0060:[<c068124f>] EFLAGS: 00010096 CPU: 2
EIP is at sysrq_handle_crash+0xf/0x20
EAX: 00000063 EBX: 00000063 ECX: c09e1c8c EDX: 00000000
ESI: c0a09ca0 EDI: 00000286 EBP: 00000000 ESP: ef4dbf24
DS: 007b ES: 007b FS: 00d8 GS: 00e0 SS: 0068
Process bash (pid: 5591, ti=ef4da000 task=f196d560 task.ti=ef4da000)
Stack:
c068146b c0960891 c0968653 00000003 00000000 00000002 efafe5c0 c06814d0
<0> ffffffb c068150f b7776000 f2600c40 c0569ec4 ef4dbf9c 00000002 b7776000
<0> efafe5c0 00000002 b7776000 c0569e60 c051de50 ef4dbf9c f196d560 ef4dbfb4
Call Trace:
[<c068146b>] ? __handle_sysrq+0xfb/0x160
[<c06814d0>] ? write_sysrq_trigger+0x0/0x50
[<c068150f>] ? write_sysrq_trigger+0x3f/0x50
[<c0569ec4>] ? proc_reg_write+0x64/0xa0
[<c0569e60>] ? proc_reg_write+0x0/0xa0
[<c051de50>] ? vfs_write+0xa0/0x190
[<c051e8d1>] ? sys_write+0x41/0x70
[<c0409adc>] ? syscall_call+0x7/0xb
Code: a0 c0 01 0f b6 41 03 19 d2 f7 d2 83 e2 03 83 e0 cf c1 e2 04 09 d0 88 41 03 f3 c3 90 c7 05
c8 1b 9e c0 01 00 00 00 0f ae f8 89 f6 <c6> 05 00 00 00 00 01 c3 89 f6 8d bc 27 00 00 00 00 8d 50
d0 83
EIP: [<c068124f>] sysrq_handle_crash+0xf/0x20 SS:ESP 0068:ef4dbf24
CR2: 0000000000000000
```

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

## Displaying a Backtrace

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

```
crash> bt
PID: 5591 TASK: f196d560 CPU: 2 COMMAND: "bash"
#0 [ef4dbd8c] crash_kexec at c0494922
#1 [ef4d9920] oops_end at c0494922
```

**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.
#2 [ef4dbe34] no_context at c043089d
#3 [ef4dbe58] bad_area at c0430b26
#4 [ef4dbe6c] do_page_fault at c080fb9b
#5 [ef4dbeec] error_code (via page_fault) at c080d809
  EAX: 00000063  EBX: 00000063  ECX: c09e1c8c  EDX: 00000000  EBP: 00000000
  DS:  007b      ESI: c0a09ca0  ES:  007b      EDI: 00000286  GS:  00e0
  CS:  0060      EIP: c068124f ERR: ffffffff  EFLAGS: 00010096
#6 [ef4dbf18] sysrq_handle_crash at c068124f
#7 [ef4dbf24] __handle_sysrq at c0681469
#8 [ef4dbf48] write_sysrq_trigger at c068150a
#9 [ef4dbf54] proc_reg_write at c0569ec2
#10 [ef4dbf74] vfs_write at c051de4e
#11 [ef4dbf94] sys_write at c051e8cc
#12 [ef4dbfb0] system_call at c0409ad5
  EAX: ffffffda  EBX: 00000001  ECX: b7776000  EDX: 00000002
  DS:  007b      ESI: 00000002  ES:  007b      EDI: b7776000
  SS:  007b      ESP: bfcb2088  EBP: bfcb20b4  GS:  0033
  CS:  0073      EIP: 00edc416  ERR: 00000004  EFLAGS: 00000246

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

Displaying a process status

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

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

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

Displaying virtual memory information

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

```
crash> vm
   PID: 5591  TASK: f196d560  CPU: 2  COMMAND: "bash"
   MM  PGD  RSS  TOTAL_VM
f19b5900 ef9c6000  1648k  5084k
VMA  START  END  FLAGS  FILE
f1bb0310   242000   260000  8000875  /lib/ld-2.12.so
f26af0b8   260000   261000  8100871  /lib/ld-2.12.so
efbc275c   261000   262000  8100873  /lib/ld-2.12.so
```
Use `vm <pid>` to display information on a single specific process, or use `help vm` for more information on `vm` usage.

Displaying open files

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

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

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

14.4. 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

1. Access the Kernel Oops Analyzer tool.

2. 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.

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

**14.5. 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
CHAPTER 15. 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.

15.1. 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.

15.2. 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?

15.3. 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.

15.4. 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.

15.5. HOW KERNEL LIVE PATCHING WORKS

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

1. The kernel patch module is copied to the /var/lib/kpatch/ directory and registered for re-application to the kernel by systemd on next boot.

2. The kpatch module is loaded into the running kernel and the new functions are registered to the ftrace mechanism with a pointer to the location in memory of the new code.
3. When the kernel accesses the patched function, it is redirected by the `ftrace` mechanism which bypasses the original functions and redirects the kernel to patched version of the function.

**Figure 15.1. How kernel live patching works**

![Diagram showing how kernel live patching works](image)

15.6. 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**

1. Optionally, check your kernel version:

```bash
# 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.

4. Optionally, verify that the kernel is 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, which is now patched with the latest fixes from the `kpatch-patch-5_14_0-1-0-1.el9.x86_64.rpm` package.

   **Additional resources**

   - `kpatch(1)` manual page
   - *Configuring basic system settings* in RHEL
   - *Configuring basic system settings* in RHEL
   - *Configuring basic system settings* in RHEL

**15.7. 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**

- 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.

4. Optionally, verify that all installed kernels were patched:

```bash
# 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_0-1-0-0.el9.x86_64.rpm` and `kpatch-patch-5_14_0-2-0-1.el9.x86_64.rpm` packages respectively.

Additional resources

- `kpatch(1)` and `dnf-kpatch(8) manual pages`
- `Configuring basic system settings in RHEL`
- `Configuring basic system settings in RHEL`
- `Configuring basic system settings in RHEL`

### 15.8. 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:

  ```bash
  # 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:

  ```bash
  # 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
- *Configuring basic system settings* in RHEL
- *Configuring basic system settings* in RHEL

15.9. REMOVING THE LIVE PATCHING PACKAGE

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

**Prerequisites**

- Root permissions
- The live patching package is installed.

**Procedure**

1. Select the live patching package:
   ```
   # 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

- **kpatch(1)** manual page
- *Configuring basic system settings* in RHEL
- *Configuring basic system settings* in RHEL
- *Configuring basic system settings* in RHEL

15.10. UNINSTALLING THE KERNEL PATCH MODULE

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

Prerequisites

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

Procedure

1. Select a kernel patch module:

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

- `kpatch(1)` manual page

**15.11. DISABLING KPATCH.SERVICE**

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

**Prerequisites**

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

**Procedure**

1. Verify `kpatch.service` is enabled:

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

2. Disable `kpatch.service`:

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

- Note that the applied kernel patch module is still loaded:
# kpatch list
Loaded patch modules:
kpatch_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

- `kpatch(1)` manual page
- `Configuring basic system settings` in RHEL
- `Configuring basic system settings` in RHEL
- `Configuring basic system settings` in RHEL
CHAPTER 16. SETTING LIMITS FOR APPLICATIONS

You can use the control groups (cgroups) kernel functionality to set limits, prioritize or isolate the hardware resources of processes. This allows you to granularly control resource usage of applications to utilize them more efficiently.

16.1. UNDERSTANDING CONTROL GROUPS

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

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

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

Control groups version 1

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

The cgroups-v1 controllers were developed across a large time span and as a result, the behavior and naming of their control files is not uniform.

Control groups version 2

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

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

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

Additional resources

- What kernel resource controllers are
- cgroups(7) manual page
- Role of systemd in control groups

16.2. WHAT KERNEL RESOURCE CONTROLLERS ARE

The functionality of control groups is enabled by kernel resource controllers. 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 system and service manager. Find a list of currently mounted resource controllers in the /proc/cgroups file.

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

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

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

- **io** - A follow-up to blkio of cgroups-v1.
- **memory** - A follow-up to memory of cgroups-v1.
- **pids** - Same as pids in cgroups-v1.
- **rdma** - Same as rdma in cgroups-v1.
- **cpu** - A follow-up to cpu and cpusacct of cgroups-v1.
• **cpuset** - Supports only the core functionality (`cpus{,.effective}`, `mems{,.effective}`) with a new partition feature.

• **perf_event** - Support is inherent, no explicit control file. You can specify a v2 cgroup as a parameter to the `perf` command that will profile all the tasks within that cgroup.

**IMPORTANT**

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

Additional resources

• [cgroups](7) manual page

• Documentation in `/usr/share/doc/kernel-doc-<kernel_version>/Documentation/cgroups-v1/` directory

### 16.3. WHAT NAMESPACES ARE

Namespaces are one of the most important methods for organizing and identifying software objects.

A namespace wraps a global system resource (for example a mount point, a network device, or a hostname) in an abstraction that makes it appear to processes within the namespace that they have their own isolated instance of the global resource. One of the most common technologies that utilize namespaces are containers.

Changes to a particular global resource are visible only to processes in that namespace and do not affect the rest of the system or other namespaces.

To inspect which namespaces a process is a member of, you can check the symbolic links in the `/proc/<PID>/ns/` directory.

The following table shows supported namespaces and resources which they isolate:

<table>
<thead>
<tr>
<th>Namespace</th>
<th>Isolates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mount</td>
<td>Mount points</td>
</tr>
<tr>
<td>UTS</td>
<td>Hostname and NIS domain name</td>
</tr>
<tr>
<td>IPC</td>
<td>System V IPC, POSIX message queues</td>
</tr>
<tr>
<td>PID</td>
<td>Process IDs</td>
</tr>
<tr>
<td>Network</td>
<td>Network devices, stacks, ports, etc</td>
</tr>
<tr>
<td>User</td>
<td>User and group IDs</td>
</tr>
<tr>
<td>Control groups</td>
<td>Control group root directory</td>
</tr>
</tbody>
</table>
16.4. SETTING CPU LIMITS TO APPLICATIONS USING CGROUPS-V1

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

Prerequisites

- 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 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   3.7  11.0   0:01.87 systemd
   505 root      20   0       0      0      0 I   0.3   0.0   0:00.75 kworker/u4:4-events_unbound
   ... 
   ```

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

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

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

3. Optionally, inspect the newly created control group:

```bash
# 1l /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-r-- 1 root root 0 Mar 11 11:42 cpuacct.stat
-r-r-r-r-- 1 root root 0 Mar 11 11:42 cpuacct.usage
-r-r-r-r-- 1 root root 0 Mar 11 11:42 cpuacct.usage_all
-r-r-r-r-- 1 root root 0 Mar 11 11:42 cpuacct.usage_percpu
-r-r-r-r-- 1 root root 0 Mar 11 11:42 cpuacct.usage_percpu_sys
-r-r-r-r-- 1 root root 0 Mar 11 11:42 cpuacct.usage_percpu_user
-r-r-r-r-- 1 root root 0 Mar 11 11:42 cpuacct.usage_sys
-r-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
-r-w-r-r-- 1 root root 0 Mar 11 11:42 cpushares
-r-r-r-r-- 1 root root 0 Mar 11 11:42 cpu.stat
-rw-r-r-- 1 root root 0 Mar 11 11:42 notify_on_release
-rw-r-r-- 1 root root 0 Mar 11 11:42 tasks
```

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

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

4. Configure CPU limits for the control group:

```bash
# 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 (\(\mu\text{s}\), represented here as "us") for how frequently a control group’s access to CPU resources should be reallocated. The upper limit is 1 second and the lower limit is 1000 microseconds.

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

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

5. Optionally, verify the limits:

```bash
# cat /sys/fs/cgroup/cpu/Example/cpu.cfs_period_us
```
6. Add the application’s PID to the Example control group:

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

or

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

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

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

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

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

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

```bash
# top
```

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

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

- Understanding control groups
- What kernel resource controllers are
- `cgroups(7), sysfs(5)` manual pages

[1] Linux Control Group v2 - An Introduction, Devconf.cz 2019 presentation by Waiman Long
CHAPTER 17. USING CGROUPS-V2 TO CONTROL DISTRIBUTION OF CPU TIME FOR APPLICATIONS

Some applications use too much CPU time, which can negatively impact the overall health of your environment. You can put your applications into control groups version 2 (cgroups-v2) and configure CPU limits for those control groups. As a result, you can regulate your applications in CPU consumption.

The user has two methods how to regulate distribution of CPU time allocated to a control group:

- Setting **CPU bandwidth** (editing the `cpu.max` controller file)
- Setting **CPU weight** (editing the `cpu.weight` controller file)

17.1. MOUNTING CGROUPS-V2

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.

**Prerequisites**

- You have root permissions.

**Verification steps**

1. Optionally, verify that the `cgroups-v2` filesystem was mounted:

   ```bash
   # mount -l | grep cgroup
   cgroup2 on /sys/fs/cgroup type cgroup2
   (rw,nosuid,nodev,noexec,relatime,seclabel,nsdelegate)
   
   The `cgroups-v2` filesystem was successfully mounted on the `/sys/fs/cgroup/` directory.
   
2. Optionally, inspect the contents of the `/sys/fs/cgroup/` directory:

   ```bash
   # ls /sys/fs/cgroup/
   -r--r--r-- 1 root root 0 Apr 29 12:03 cgroup.controllers
   -rw-r--r-- 1 root root 0 Apr 29 12:03 cgroup.max.depth
   -rw-r--r-- 1 root root 0 Apr 29 12:03 cgroup.max.descendants
   -rw-r--r-- 1 root root 0 Apr 29 12:03 cgroup.procs
   -r--r--r-- 1 root root 0 Apr 29 12:03 cgroup.stat
   -rw-r--r-- 1 root root 0 Apr 29 12:03 cgroup.subtree_control
   -rw-r--r-- 1 root root 0 Apr 29 12:03 cpuset.cpu.pressure
   -r--r--r-- 1 root root 0 Apr 29 12:03 cpuset.mems.effective
   -r--r--r-- 1 root root 0 Apr 29 12:03 cpuset.cpus.effective
   drwxr-xr-x 69 root root 0 Apr 29 12:03 system.slice
   drwxr-xr-x 3 root root 0 Apr 29 12:18 user.slice
   ```
The /sys/fs/cgroup/ directory, also called the *root control group*, by default, contains interface files (starting with *cgroup*) and controller-specific files such as *cpuset.cpus.effective*. In addition, there are some directories related to *systemd*, such as, /sys/fs/cgroup/init.scope, /sys/fs/cgroup/system.slice, and /sys/fs/cgroup/user.slice.

### Additional resources
- Understanding control groups
- What kernel resource controllers are
- cgroups(7), sysfs(5) manual pages

### 17.2. PREPARING THE CGROUP FOR DISTRIBUTION OF CPU TIME

To control CPU consumption of your applications, you need to enable specific CPU controllers and create two levels of child control groups inside the /sys/fs/cgroup/ root control group. The root control group already contains some of the resource controllers by default. Therefore two levels of child control groups are advisable to ensure organizational clarity of cgroup files.

#### Prerequisites
- You have at least two applications whose CPU consumption you want to regulate.
- You have root permissions.
- You have mounted *cgroups-v2* filesystem.

#### Procedure

1. Identify the process IDs (PIDs) of applications whose CPU consumption you want to constrict:

   ```
   # top
   Tasks: 104 total,  3 running, 101 sleeping,  0 stopped, 0 zombie
   %Cpu(s): 17.6 us, 81.6 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.8 hi,  0.0 si,  0.0 st
   MiB Mem :  3737.4 total,  3312.7 free,  133.3 used,  291.4 buff/cache
   MiB Swap:  4060.0 total,  4060.0 free,  0.0 used.  3376.1 avail Mem
   
   PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   34578 root      20   0   18720   1756   1468 R 99.0 0.0   0:31.09 sha1sum
   34579 root      20   0   18720   1772   1480 R 99.0 0.0   0:30.54 sha1sum
   1 root      20   0  186192  13940  9500 S   0.0   0.4   0:01.60 systemd
   2 root      20   0    0    0    0 S   0.0   0.0   0:00.01 kthreadd
   3 root      20   0    0    0    0 I   0.0   0.0   0:00.00 rcu_gp
   4 root      20   0    0    0    0 I   0.0   0.0   0:00.00 rcu_par_gp
   ...
   
   The example output reveals that PID 34578 and 34579 (two illustrative applications of sha1sum) consume a lot of resources, namely CPU. Both are example applications used to demonstrate managing the *cgroups-v2* functionality.
   
   2. Verify that the *cpu* and *cpuset* controllers are available in the /sys/fs/cgroup/cgroup.controllers file:
# cat /sys/fs/cgroup/cgroup.controllers
cpuset cpu io memory hugetlb pids rdma

3. Enable CPU-related controllers:

```bash
# 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 children groups of the `/sys/fs/cgroup/` root 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.

4. Create the `/sys/fs/cgroup/Example/` directory:

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

The `/sys/fs/cgroup/Example/` directory defines a child group. Also, the previous step enabled the `cpu` and `cpuset` controllers for this child group.

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

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

```bash
# ll /sys/fs/cgroup/Example/
```

```
- rw-r--r-- 1 root 0 Jun 01 10:33 cgroup.controllers
- rw-r--r-- 1 root 0 Jun 01 10:33 cgroup.events
- rw-r--r-- 1 root 0 Jun 01 10:33 cgroup.freeze
- rw-r--r-- 1 root 0 Jun 01 10:33 cgroup.max.depth
- rw-r--r-- 1 root 0 Jun 01 10:33 cgroup.max.descendants
- rw-r--r-- 1 root 0 Jun 01 10:33 cgroup.procs
- rw-r--r-- 1 root 0 Jun 01 10:33 cgroup.stat
- rw-r--r-- 1 root 0 Jun 01 10:33 cgroup.subtree_control
...
- rw-r--r-- 1 root 0 Jun 01 10:33 cpuset.cpus
- rw-r--r-- 1 root 0 Jun 01 10:33 cpuset.cpus.effective
- rw-r--r-- 1 root 0 Jun 01 10:33 cpuset.cpus.partition
- rw-r--r-- 1 root 0 Jun 01 10:33 cpuset.mems
- rw-r--r-- 1 root 0 Jun 01 10:33 cpuset.mems.effective
- rw-r--r-- 1 root 0 Jun 01 10:33 cpu.stat
- rw-r--r-- 1 root 0 Jun 01 10:33 cpu.weight
- rw-r--r-- 1 root 0 Jun 01 10:33 cpu.weight.nice
...
- r----r-- 1 root 0 Jun 01 10:33 memory.events.local
```
The example output shows files such as `cpuset.cpus` and `cpu.max`. These files are specific to the `cpuset` and `cpu` controllers. The `cpuset` and `cpu` controllers are manually enabled for the root’s (`/sys/fs/cgroup/`) direct child control groups using the `/sys/fs/cgroup/cgroup.subtree_control` file.

The directory also includes general `cgroup` control interface files such as `cgroup.procs` or `cgroup.controllers`, which 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 always enabled by default.

By default, the newly created child group inherits access to all of the system’s CPU and memory resources, without any limits.

6. Enable the CPU-related controllers in `/sys/fs/cgroup/Example/` to obtain controllers that are relevant only to CPU:

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

These commands ensure that the immediate child control group will only have controllers relevant to regulate the CPU time distribution - not to `memory` or `pids` controllers.

7. Create the `/sys/fs/cgroup/Example/tasks/` directory:

```bash
# 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.

8. Optionally, inspect another child control group:

```bash
# ll /sys/fs/cgroup/Example/tasks
-r--r--r--. 1 root root 0 Jun 11:45 cgroup.controllers
-r--r--r--. 1 root root 0 Jun 11:45 cgroup.events
-r--r--r--. 1 root root 0 Jun 11:45 cgroup.freeze
-r--r--r--. 1 root root 0 Jun 11:45 cgroup.max.depth
-r--r--r--. 1 root root 0 Jun 11:45 cgroup.max.descendants
-r--r--r--. 1 root root 0 Jun 11:45 cgroup.procs
-r--r--r--. 1 root root 0 Jun 11:45 cgroup.stat
-r--r--r--. 1 root root 0 Jun 11:45 cgroup.subtree_control
-r--r--r--. 1 root root 0 Jun 11:45 cgroup.type
-r--r--r--. 1 root root 0 Jun 11:45 cpu.max
-r--r--r--. 1 root root 0 Jun 11:45 cpu.pressure
-r--r--r--. 1 root root 0 Jun 11:45 cpuset.cpus
-r--r--r--. 1 root root 0 Jun 11:45 cpuset.cpus.effective
-r--r--r--. 1 root root 0 Jun 11:45 cpuset.cpus.partition
-r--r--r--. 1 root root 0 Jun 11:45 cpuset.mems
```
9. Ensure the processes that you want to control for CPU time compete on the same CPU:

```
# echo "1" > /sys/fs/cgroup/Example/tasks/cpuset.cpus
```

The previous command ensures that the processes you will place in the `Example/tasks` child control group, compete on the same CPU. This setting is important for the `cpu` controller to activate.

**IMPORTANT**

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

**Verification steps**

1. Optional: ensure that the CPU-related controllers are enabled for the immediate children cgroups:

```
# cat /sys/fs/cgroup/cgroup.subtree_control
/sys/fs/cgroup/Example/cgroup.subtree_control
  cpuset cpu memory pids
  cpuset cpu
```

2. Optional: ensure the processes that you want to control for CPU time compete on the same CPU:

```
# cat /sys/fs/cgroup/Example/tasks/cpuset.cpus
1
```

**Additional resources**

- [Understanding control groups](#)
- [What kernel resource controllers are](#)
- [Mounting cgroups-v2](#)
- [cgroups(7), sysfs(5) manual pages](#)

### 17.3. CONTROLLING DISTRIBUTION OF CPU TIME FOR APPLICATIONS BY ADJUSTING CPU BANDWIDTH

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 at least two applications for which you want to control distribution of CPU time.

You ensured the relevant applications compete for CPU time on the same CPU as described in Preparing the cgroup for distribution of CPU time.

You mounted cgroups-v2 filesystem as described in Mounting cgroups-v2.

You enabled cpu and cpuset controllers both in the parent control group and in child control group similarly as described in Preparing the cgroup for distribution of CPU time.

You created two levels of child control groups inside the /sys/fs/cgroup/ root control group as in the example below:

```
...   Example
      └── tasks
...
```

### Procedure

1. Configure CPU bandwidth to achieve resource restrictions within the control group:

   ```
   # echo "200000 1000000" > /sys/fs/cgroup/Example/tasks/cpu.max
   ```

   The first value is the allowed time quota in microseconds for which all processes collectively in a child group can run during one period. The second value specifies the length of the period.

   During a single period, when processes in a control group collectively exhaust the time specified by this quota, they are throttled for the remainder of the period and not allowed to run until the next period.

   This command sets CPU time distribution controls so that all processes collectively in the /sys/fs/cgroup/Example/tasks child group can run on the CPU for only 0.2 seconds of every 1 second. That is, one fifth of each second.

2. Optionally, verify the time quotas:

   ```
   # cat /sys/fs/cgroup/Example/tasks/cpu.max
   200000 1000000
   ```

3. Add the applications' PIDs to the Example/tasks child group:

   ```
   # echo "34578" > /sys/fs/cgroup/Example/tasks/cgroup.procs
   # echo "34579" > /sys/fs/cgroup/Example/tasks/cgroup.procs
   ```

   The example commands ensure that desired applications become members of the Example/tasks child group and do not exceed the CPU time distribution configured for this child group.

### Verification steps

1. Verify that the applications run in the specified control group:
The output above shows the processes of the specified applications that run in the Example/tasks child group.

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

```
# top
```

Notice that the CPU consumption for the PID 34578 and PID 34579 has decreased to 10%. The Example/tasks child group regulates its processes to 20% of the CPU time collectively. Since there are 2 processes in the control group, each can utilize 10% of the CPU time.

Additional resources

- Understanding control groups
- What kernel resource controllers are
- Mounting cgroups-v2
- Preparing the cgroup for distribution of CPU time
- cgroups(7), sysfs(5) manual pages

17.4. 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 ensured the relevant applications compete for CPU time on the same CPU as described in Preparing the cgroup for distribution of CPU time.
You mounted `cgroups-v2` filesystem as described in Mounting cgroups-v2.

You created two levels of child control groups inside the `/sys/fs/cgroup/` root control group as in the following example:

```
  ...  
  │   Example  
  │   ├── g1  
  │   │   ├── g2  
  │   │   └── g3  
  │   ...  
  ...  
```

You enabled `cpu` and `cpuset` controllers in the parent control group and in child control groups similarly as described in Preparing the cgroup for distribution of CPU time.

Procedure

1. Configure 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 do not exceed the distribution control configured for those child 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:

<table>
<thead>
<tr>
<th>Child cgroup</th>
<th><code>cpu.weight</code> file</th>
<th>CPU time allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>g1</td>
<td>150</td>
<td>~50% (150/300)</td>
</tr>
<tr>
<td>g2</td>
<td>100</td>
<td>~33% (100/300)</td>
</tr>
<tr>
<td>g3</td>
<td>50</td>
<td>~16% (50/300)</td>
</tr>
</tbody>
</table>

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`: 

---

100
### Child cgroup

<table>
<thead>
<tr>
<th>Child cgroup</th>
<th>cpu.weight</th>
<th>CPU time allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>g1</td>
<td>150</td>
<td>~75% (150/200)</td>
</tr>
<tr>
<td>g3</td>
<td>50</td>
<td>~25% (50/200)</td>
</tr>
</tbody>
</table>

**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
   ```

   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 kernel resource controllers are
- Mounting cgroups-v2
• Preparing the cgroup for distribution of CPU time
• Resource Distribution Models
• cgroups(7), sysfs(5) manual pages
CHAPTER 18. USING CONTROL GROUPS VERSION 1 WITH SYSTEMD

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

18.1. ROLE OF SYSTEMD IN CONTROL GROUPS VERSION 1

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.

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

Three systemd unit types are used for resource control:

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

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

  `<name>.scope`

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

  `<parent-name>.slice`

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

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

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

```
Control group /:
  -.slice
  ├─ user.slice
  │  ├─ user-42.slice
  │  └─ session-c1.scope
```
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
- Configuring basic system settings in Red Hat Enterprise Linux
- Configuring basic system settings in Red Hat Enterprise Linux
- What are kernel resource controllers
- systemd.resource-control(5), cgroups(7), fork(), fork(2) manual pages

18.2. CREATING TRANSIENT CONTROL GROUPS

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

Procedure

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

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

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

- Understanding control groups
- Role of systemd in control groups
- Configuring basic system settings in RHEL
- Configuring basic system settings in RHEL
- Configuring basic system settings in RHEL
- `systemd-run(1)` manual page

### 18.3. CREATING PERSISTENT CONTROL GROUPS

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

**Procedure**

- To create a persistent control group, execute:

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

Additional resources

- Understanding control groups
- Role of systemd in control groups
- Configuring basic system settings in RHEL
- Configuring basic system settings in RHEL
- Configuring basic system settings in RHEL
- `systemd-run(1)` manual page

## 18.4. CONFIGURING MEMORY RESOURCE CONTROL SETTINGS ON THE COMMAND-LINE

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

### Procedure

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

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

  The command instantly assigns the memory limit of 1,500 kilobytes to processes executed in a control group the `example.service` service belongs to. The `MemoryLimit` parameter, in this configuration variant, is defined in the `/etc/systemd/system.control/example.service.d/50-MemoryLimit.conf` file and controls the value of the `/sys/fs/cgroup/memory/system.slice/example.service/memory.limit_in_bytes` file.

- Optionally, to temporarily limit the memory usage of a service, run:

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

  The command instantly assigns the memory limit to the `example.service` service. The `MemoryLimit` parameter is defined until the next reboot in the `/run/systemd/system.control/example.service.d/50-MemoryLimit.conf` file. With a reboot, the whole `/run/systemd/system.control/` directory and `MemoryLimit` are removed.

### NOTE

The `50-MemoryLimit.conf` file stores the memory limit as a multiple of 4096 bytes - one kernel page size specific for AMD64 and Intel 64. The actual number of bytes depends on a CPU architecture.

Additional resources

- Understanding control groups
What kernel resource controllers are

- systemd.resource-control(5) and cgroups(7) manual pages
- Role of systemd in control groups

18.5. CONFIGURING MEMORY RESOURCE CONTROL SETTINGS WITH UNIT FILES

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

Manually modifying unit files is one of the ways how to set limits, prioritize, or control access to hardware resources for groups of processes.

Procedure

1. To limit the memory usage of a service, modify the /usr/lib/systemd/system/example.service file as follows:

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

The configuration above places a limit on maximum memory consumption of processes executed in a control group, which example.service is part of.

**NOTE**

Use suffixes K, M, G, or T to identify Kilobyte, Megabyte, Gigabyte, or Terabyte as a unit of measurement.

2. Reload all unit configuration files:

```
# systemctl daemon-reload
```

3. Restart the service:

```
# systemctl restart example.service
```

4. Reboot the system.

5. Optionally, check that the changes took effect:

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

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

The `memory.limit_in_bytes` file stores the memory limit as a multiple of 4096 bytes - one kernel page size specific for AMD64 and Intel 64. The actual number of bytes depends on a CPU architecture.

Additional resources

- Understanding control groups
- What kernel resource controllers are
- `systemd.resource-control(5), cgroups(7)` manual pages
- `Configuring basic system settings` in RHEL
- `Configuring basic system settings` in RHEL
- `Configuring basic system settings` in RHEL
- Role of systemd in control groups

18.6. 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

- Understanding control groups
- What kernel resource controllers are
- `systemd.resource-control(5), cgroups(7)` manual pages
- Role of systemd in control groups
18.7. REMOVING PERSISTENT CONTROL GROUPS

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

Persistent cgroups are released when a service or a scope unit is stopped or disabled and its configuration file is deleted.

**Procedure**

1. Stop the service unit:

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

2. Disable the service unit:

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

3. Remove the relevant unit configuration file:

   ```
   # rm /usr/lib/systemd/system/<name>.service
   ```

4. Reload all unit configuration files so that changes take effect:

   ```
   # systemctl daemon-reload
   ```

**Additional resources**

- Understanding control groups
- What kernel resource controllers are
- `systemd.resource-control(5)`, `cgroups(7)`, and `systemd.kill(5)` manual pages
- Role of systemd in control groups
- `Configuring basic system settings` in RHEL

18.8. LISTING SYSTEMD UNITS

The following procedure describes how to use the `systemd` system and service manager to list its units.

**Procedure**
To list all active units on the system, execute the `# systemctl` command and the terminal will return an output similar to the following example:

```
# 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 core dump 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  loaded active active
system-getty.slice        loaded active active  loaded active active
system-lvm2x2dpvscan.slice loaded active active system-
lvm2x2dpvscan.slice       loaded active active system-
system-sshdx2dkeygen.slice loaded active active system-
sshdx2dkeygen.slice       loaded active active system-
system-systemd\x2dhibernate\x2dresume.slice loaded active active system-
systemd\x2dhibernate\x2dresume> system-
system-user\x2druntime\x2ddir.slice loaded active active system-
user\x2druntime\x2ddir.slice loaded active active system-
system.slice              loaded active active  System Slice
user-1000.slice           loaded active active  User Slice of UID 1000
user-42.slice             loaded active active  User Slice of UID 42
user.slice                loaded active active  User and Session Slice
...                        ...
```

- **UNIT** - a name of a unit that also reflects the unit position in a control group hierarchy. The units relevant for resource control are a *slice*, a *scope*, and a *service*.

- **LOAD** - indicates whether the unit configuration file was properly loaded. If the unit file failed to load, the field contains the state *error* instead of *loaded*. Other unit load states are: *stub*, *merged*, and *masked*.

- **ACTIVE** - the high-level unit activation state, which is a generalization of *SUB*.

- **SUB** - the low-level unit activation state. The range of possible values depends on the unit type.

- **DESCRIPTION** - the description of the unit content and functionality.

To list inactive units, execute:

```
# systemctl --all
```

To limit the amount of information in the output, execute:

```
# systemctl --type service,masked
```

The `--type` option requires a comma-separated list of unit types such as a *service* and a *slice*, or unit load states such as *loaded* and *masked*. 
18.9. VIEWING A CONTROL GROUP VERSION 1 HIERARCHY

The following procedure describes how to display control groups (cgroups) hierarchy and processes running in specific cgroups.

Procedure

- To display the whole cgroups hierarchy on your system, execute 
  `# systemd-cgls`:

```
# systemd-cgls
Control group /:
  -.slice
   |  
   |  user.slice
   |  |  user-42.slice
   |  |  session-c1.scope
   |  |  965 gdm-session-worker [pam/gdm-launch-environment]
   |  |  1040 /usr/libexec/gdm-x-session gnome-session --autostart
   |  |  /usr/share/gdm/greeter/autostart
   |  ...
   |  init.scope
   |  |  1 /usr/lib/systemd/systemd --switched-root --system --deserialize 18
   |  |  system.slice
   |  ...
   |  example.service
   |  |  6882 /bin/bash /home/jdoe/example.sh
   |  |  6902 sleep 1
   |  |  systemd-journald.service
   |  |  629 /usr/lib/systemd/systemd-journald
   |  ...
```

The example output returns the entire cgroups hierarchy, where the highest level is formed by slices.

- To display the cgroups hierarchy filtered by a resource controller, execute 
  `# systemd-cgls <resource_controller>`:

```
# systemd-cgls memory
Controller memory; Control group /:
  1 /usr/lib/systemd/systemd --switched-root --system --deserialize 18
  user.slice
  |  user-42.slice
  |  |  session-c1.scope
  |  |  965 gdm-session-worker [pam/gdm-launch-environment]
  ...
  system.slice
  ...
```

Additional resources

- Configuring basic system settings in RHEL
- Configuring basic system settings in RHEL
- Configuring basic system settings in RHEL
The example output of the above command lists the services that interact with the selected controller.

- To display detailed information about a certain unit and its part of the cgroups hierarchy, execute `# systemctl status <system_unit>`:

```
# systemctl status example.service
```

```
● example.service - My example service
Loaded: loaded (/usr/lib/systemd/system/example.service; enabled; vendor preset: disabled)
Active: active (running) since Tue 2019-04-16 12:12:39 CEST; 3s ago
Main PID: 17737 (bash)
Tasks: 2 (limit: 11522)
Memory: 496.0K (limit: 1.5M)
CGroup: /system.slice/example.service
    17737 /bin/bash /home/jdoe/example.sh
    17743 sleep 1
Apr 16 12:12:39 redhat systemd[1]: Started My example service.
Apr 16 12:12:39 redhat bash[17737]: The current time is Tue Apr 16 12:12:39 CEST 2019
Apr 16 12:12:40 redhat bash[17737]: The current time is Tue Apr 16 12:12:40 CEST 2019
```

Additional resources

- What kernel resource controllers are
- `systemd.resource-control(5), cgroups(7)` manual pages

### 18.10. VIEWING RESOURCE CONTROLLERS

The following procedure describes how to learn which processes use which resource controllers.

**Procedure**

1. To view which resource controllers a process interacts with, execute the `# cat /proc/<PID>/cgroup` command:

```
# cat /proc/11269/cgroup
12:freezer:
11:cpuset:
10:devices:/system.slice
9:memory:/system.slice/example.service
8:pids:/system.slice/example.service
7:hugetlb:
6:rdma:
5:perf_event:
4:cpu,cpuacct:
```

Red Hat Enterprise Linux 9.0 Beta Managing, monitoring, and updating the kernel
The example output relates to a process of interest. In this case, it is a process identified by PID 11269, which belongs to the example.service unit. You can determine whether the process was placed in a correct control group as defined by the systemd unit file specifications.

**NOTE**

By default, the items and their ordering in the list of resource controllers is the same for all units started by systemd, since it automatically mounts all the default resource controllers.

Additional resources

- [cgroups(7) manual page](#)
- Documentation in the `/usr/share/doc/kernel-doc-<kernel_version>/Documentation/cgroups-v1/` directory

### 18.11. MONITORING RESOURCE CONSUMPTION

The following procedure describes how to view a list of currently running control groups (cgroups) and their resource consumption in real-time.

**Procedure**

1. To see a dynamic account of currently running cgroups, execute 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-rhel7x2dswap.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

- systemctl-cgtop(1) manual page

18.12. CONFIGURING CPUSET CONTROLLER USING SYSTEMD

The **systemd** resource management API allows the user to configure limits on a set of CPUs and NUMA nodes that a service can use. This limit restricts access to system resources utilized by the processes. The requested configuration is written in `cpuset.cpus` and `cpuset.mems`. However, the requested configuration may not be used, as the parent `cgroup` limits either `cpus` or `mems`. To access the current configuration, the `cpuset.cpus.effective` and `cpuset.mems.effective` files are exported to the users.

Procedure

- To set **AllowedCPUs**:
  
  ```
  # systemctl set-property service_name.service AllowedCPUs=value
  
  For example:
  
  systemctl set-property service_name.service AllowedCPUs=0-5
  ```

- To set **AllowedMemoryNodes**:

  ```
  # systemctl set-property service_name.service AllowedMemoryNodes=value
  
  For example:
  
  systemctl set-property service_name.service AllowedMemoryNodes=0
  ```
CHAPTER 19. CONFIGURING RESOURCE MANAGEMENT USING CGROUPS VERSION 2 WITH SYSTEMD

The core of systemd is service management and supervision. systemd ensures that the right services start at the right time and in the correct order during the boot process. When the services are running, they 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.

19.1. PREREQUISITES

- Basic knowledge of the Linux cgroup subsystem.

19.2. INTRODUCTION TO RESOURCE DISTRIBUTION MODELS

For resource management, systemd uses the cgroups v2 interface.

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

**Weights**

The resource is distributed 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 and each cgroup receives one tenth of the resource.

Weight is usually used to distribute stateless resources. The \texttt{CPUWeight=} option is an implementation of this resource distribution model.

**Limits**

A cgroup can consume up to the configured amount of the resource, but you can also overcommit resources. Therefore, the sum of sub-group limits can exceed the limit of the parent cgroup. The \texttt{MemoryMax=} option is an implementation of this resource distribution model.

**Protections**

A protected amount of a resource can be set up 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 allowed. The \texttt{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 allowed. An example of this resource type in Linux is the real-time budget.

**Additional resources**

- Managing CPU with systemd
- Allocating memory resources using systemd
- Configuring I/O bandwidth using systemd
19.3. ALLOCATING CPU RESOURCES USING SYSTEMD

On a system managed by systemd, each system service is started in its cgroup. By enabling the support for the CPU cgroup controller, the system uses the service-aware distribution of CPU resources instead of the per-process distribution. In the service-aware distribution, each service receives approximately the same amount of CPU time relative to all other services running on the system, regardless of the number of processes that comprise the service.

If a specific service requires more CPU resources, you can grant them by changing the CPU time allocation policy for the service.

Procedure

To set a CPU time allocation policy option when using systemd:

1. Check the assigned values of the CPU time allocation policy option in the service of your choice:

   $ systemctl show --property <CPU time allocation policy option> <service name>

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

   # systemctl set-property <service name> <CPU time allocation policy option>=<value>

   **NOTE**

   The cgroup properties are applied immediately after they are set. Therefore, the service does not need to be restarted.

Verification steps

- To verify whether you successfully changed the required value of the CPU time allocation policy option for your service, run the following command:

  $ systemctl show --property <CPU time allocation policy option> <service name>

Additional resources

- CPU time allocation policy options for systemd
- Introduction to resource distribution models

19.4. CPU TIME ALLOCATION POLICY OPTIONS FOR SYSTEMD

The most frequently used CPU time allocation policy options include:

**CPUWeight=**

Assigns higher priority to a particular service over all other services. You can select a value from the interval 1 – 10,000. The default value is 100.

For example, to give `httpd.service` twice as much CPU as to all other services, set the value to `CPUWeight=200`. 
Note that CPUWeight= is applied only in cases when the operating system is overloaded.

CPUQuota=
Assigns the absolute CPU time quota to a service. The value of this option specifies the maximum percentage of CPU time that a service will receive relative to the total CPU time available, for example CPUQuota=30%.
Note that CPUQuota= represents the limit value for particular resource distribution models described in Introduction to resource distribution models.

For more information on CPUQuota=, see the systemd.resource-control(5) man page.

Additional resources
- Introduction to resource distribution models
- Allocating CPU resources using systemd

19.5. ALLOCATING MEMORY RESOURCES USING SYSTEMD

This section describes how to use any of the memory configuration options (MemoryMin, MemoryLow, MemoryHigh, MemoryMax, MemorySwapMax) to allocate memory resources using systemd.

Procedure
To set a memory allocation configuration option when using systemd:

1. Check the assigned values of the memory allocation configuration option in the service of your choice:

   $ systemctl show --property <memory allocation configuration option> <service name>

2. Set the required value of the memory allocation configuration option as a root:

   # systemctl set-property <service name> <memory allocation configuration option>=<value>

   NOTE
   The cgroup properties are applied immediately after they are set. Therefore, the service does not need to be restarted.

Verification steps
- To verify whether you successfully changed the required value of the memory allocation configuration option for your service, run the following command:

   $ systemctl show --property <memory allocation configuration option> <service name>

Additional resources
- Memory allocation configuration options for systemd
- Introduction to resource distribution models
19.6. MEMORY ALLOCATION CONFIGURATION OPTIONS FOR SYSTEMD

You can use the following options when using systemd to configure system memory allocation:

**MemoryMin**

Hard memory protection. If the memory usage is below the limit, the cgroup memory will not be reclaimed.

**MemoryLow**

Soft memory protection. If the memory usage is below the limit, the cgroup memory can be reclaimed only if no memory is reclaimed from unprotected cgroups.

**MemoryHigh**

Memory throttle limit. If the memory usage goes above the limit, the processes in the cgroup are throttled and put under a heavy reclaim pressure.

**MemoryMax**

Absolute limit for the memory usage. You can use the kilo (K), mega (M), giga (G), tera (T) suffixes, for example `MemoryMax=1G`.

**MemorySwapMax**

Hard limit on the swap usage.

**NOTE**

When you exhaust your memory limit, the Out-of-memory (OOM) killer will stop the running service. To prevent this, lower the `OOMScoreAdjust=` value to increase the memory tolerance.

Additional resources

- Allocating memory resources using systemd
- Introduction to resource distribution models

19.7. CONFIGURING I/O BANDWIDTH USING SYSTEMD

To improve the performance of a specific service in RHEL 9, you can allocate I/O bandwidth resources to that service using systemd.

To do so, you can use the following I/O configuration options:

- **IOWeight**
- **IODeviceWeight**
- **IOReadBandwidthMax**
- **IOWriteBandwidthMax**
- **IOReadIOPSMax**
- **IOWriteIOPSMax**

Procedure
To set an I/O bandwidth configuration option using systemd:

1. Check the assigned values of the I/O bandwidth configuration option in the service of your choice:

   ```
   $ systemctl show --property <I/O bandwidth configuration option> <service name>
   ```

2. Set the required value of the I/O bandwidth configuration option as a root:

   ```
   # systemctl set-property <service name> <I/O bandwidth configuration option>=<value>
   ```

   The cgroup properties are applied immediately after they are set. Therefore, the service does not need to be restarted.

Verification steps

- To verify whether you successfully changed the required value of the I/O bandwidth configuration option for your service, run the following command:

  ```
  $ systemctl show --property <I/O bandwidth configuration option> <service name>
  ```

Additional resources

- I/O bandwidth configuration options for systemd
- Introduction to resource distribution models

### 19.8. I/O BANDWIDTH CONFIGURATION OPTIONS FOR SYSTEMD

To manage the block layer I/O policies with systemd, the following configuration options are available:

**IOWeight**

Sets the default I/O weight. The weight value is used as a basis for the calculation of how much of the real I/O bandwidth the service receives in relation to the other services.

**IODeviceWeight**

Sets the I/O weight for a specific block device.

For example, `IODeviceWeight=/dev/disk/by-id/dm-name-rhel-root 200`.

**IOReadBandwidthMax, IOWriteBandwidthMax**

Sets the absolute bandwidth per device or a mount point.

For example, `IOWriteBandwidth=/var/log 5M`.

**NOTE**

Systemd handles the file-system-to-device translation automatically.

**IOTreadIOPSMax, IOWriteIOPSMax**

A similar option to the previous one: sets the absolute bandwidth in Input/Output Operations Per Second (IOPS).
NOTE

Weight-based options are supported only if the block device is using the CFQ I/O scheduler. No option is supported if the device uses the Multi-Queue Block I/O queuing mechanism.

Additional resources

- Configuring I/O bandwidth using systemd
- Introduction to resource distribution models
CHAPTER 20. CONFIGURING CPU AFFINITY AND NUMA POLICIES USING SYSTEMD

The CPU management, memory management, and I/O bandwidth options deal with partitioning available resources.

20.1. CONFIGURING CPU AFFINITY USING SYSTEMD

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. 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.

NOTE

After configuring CPU affinity mask for a particular systemd service, you must restart the system 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>
   ```

To set CPU affinity mask for a particular systemd service using the manager configuration option:

1. Edit the `/etc/systemd/system.conf` file:

   ```
   # vi /etc/systemd/system.conf
   ```

2. Search for the `CPUAffinity=` option and set the CPU numbers

3. Save the edited file and restart the server to apply the changes.
20.2. CONFIGURING NUMA USING SYSTEMD

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

To configure NUMA, systemd provides the unit file option for `NUMAPolicy` and `NUMAMask`, and the manager configuration option in the `/etc/systemd/system.conf` file.

**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 the `NUMAPolicy` through the manager configuration option:

1. Edit the `/etc/systemd/system.conf` file:
   
   ```
   # vi /etc/systemd/system.conf
   ```

2. Search for the `NUMAPolicy` option and set the policy type.

3. Save the edited file and restart the server to apply the changes.

**Additional resources**

- NUMA policy configuration options for systemd

20.3. 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.resource-control(5)), [systemd.exec(5)](systemd.exec(5)), and [set_mempolicy(2)](set_mempolicy(2)) manual pages
- [Configuring NUMA using systemd](#)
CHAPTER 21. 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.

21.1. AN INTRODUCTION TO BCC

BPF Compiler Collection (BCC) is a library, which facilitates the creation of the extended Berkeley Packet Filter (eBPF) programs. The main utility of eBPF programs is analyzing OS performance and network performance without experiencing overhead or security issues.

BCC removes the need for users to know deep technical details of eBPF, and provides many out-of-the-box starting points, such as the bcc-tools package with pre-created eBPF programs.

NOTE

The eBPF programs are triggered on events, such as disk I/O, TCP connections, and process creations. It is unlikely that the programs should cause the kernel to crash, loop or become unresponsive because they run in a safe virtual machine in the kernel.

21.2. INSTALLING THE BCC-TOOLS PACKAGE

This section describes how to install the bcc-tools package, which also installs the BPF Compiler Collection (BCC) library as a dependency.

Prerequisites

- An active Red Hat Enterprise Linux subscription
- An enabled repository containing the bcc-tools package
- Updated kernel
- Root permissions

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_deelem.c
21.3. USING SELECTED BCC-TOOLS FOR PERFORMANCE ANALYSES

This section describes how to use certain pre-created programs from the BPF Compiler Collection (BCC) library to efficiently and securely analyze the system performance on the per-event basis. The set of pre-created programs in the BCC library can serve as examples for creation of additional programs.

Prerequisites

- Installed bcc-tools package
- Root permissions

Using execsnoop to examine the system processes

1. Execute the `execsnoop` program in one terminal:

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

2. In another terminal execute for example:

   ```
   $ ls /usr/share/bcc/tools/doc/
   ```

   The above creates a short-lived process of the `ls` command.

3. The terminal running `execsnoop` shows the output similar to the following:

   ```
   PCOMM PID PPID RET ARGS
   ls  8382 8287 0 /usr/bin/ls --color=auto /usr/share/bcc/tools/doc/ ...
   ```

   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. Execute the **opensnoop** program in one terminal:

   ```
   # /usr/share/bcc/tools/opensnoop -n uname
   ```

   The above prints output for files, which are opened only by the process of the **uname** command.

2. In another terminal execute:

   ```
   $ uname
   ```

   The command above opens certain files, which are captured in the next step.

3. The terminal running **opensnoop** shows the output similar to the following:

   ```
   PID   COMM  FD  ERR   PATH
   8596  uname  3  0   /etc/ld.so.cache
   8596  uname  3  0   /lib64/libc.so.6
   8596  uname  3  0   /usr/lib/locale/locale-archive
   ... 
   ```

   The **opensnoop** program watches the **open()** system call across the whole system, and prints a line of output for each file that **uname** tried to open along the way.

   The **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. Execute the **biotop** program in one terminal:

   ```
   # /usr/share/bcc/tools/biotop 30
   ```
The command enables you to monitor the top processes, which perform I/O operations on the disk. The argument ensures that the command will produce a 30 second summary.

**NOTE**

When no argument provided, the output screen by default refreshes every 1 second.

2. In another terminal execute for example:

```bash
# 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:

<table>
<thead>
<tr>
<th>PID</th>
<th>COMM</th>
<th>D</th>
<th>MAJ</th>
<th>MIN</th>
<th>DISK</th>
<th>I/O</th>
<th>Kbytes</th>
<th>AVGms</th>
</tr>
</thead>
<tbody>
<tr>
<td>9568</td>
<td>dd</td>
<td>R</td>
<td>252</td>
<td>0</td>
<td>vda</td>
<td>1629</td>
<td>14440636.0</td>
<td>3.69</td>
</tr>
<tr>
<td>48</td>
<td>kswapd0</td>
<td>W</td>
<td>252</td>
<td>0</td>
<td>vda</td>
<td>1763</td>
<td>120696.0</td>
<td>1.65</td>
</tr>
<tr>
<td>7571</td>
<td>gnome-shell</td>
<td>R</td>
<td>252</td>
<td>0</td>
<td>vda</td>
<td>834</td>
<td>83612.0</td>
<td>0.33</td>
</tr>
<tr>
<td>1891</td>
<td>gnome-shell</td>
<td>R</td>
<td>252</td>
<td>0</td>
<td>vda</td>
<td>1379</td>
<td>19792.0</td>
<td>0.15</td>
</tr>
<tr>
<td>7515</td>
<td>Xorg</td>
<td>R</td>
<td>252</td>
<td>0</td>
<td>vda</td>
<td>280</td>
<td>9940.0</td>
<td>0.28</td>
</tr>
<tr>
<td>7579</td>
<td>llvmpipe-1</td>
<td>R</td>
<td>252</td>
<td>0</td>
<td>vda</td>
<td>228</td>
<td>6928.0</td>
<td>0.19</td>
</tr>
<tr>
<td>9515</td>
<td>gnome-control-c</td>
<td>R</td>
<td>252</td>
<td>0</td>
<td>vda</td>
<td>62</td>
<td>6444.0</td>
<td>0.43</td>
</tr>
<tr>
<td>8112</td>
<td>gnome-terminal-</td>
<td>R</td>
<td>252</td>
<td>0</td>
<td>vda</td>
<td>67</td>
<td>2572.0</td>
<td>1.54</td>
</tr>
<tr>
<td>7807</td>
<td>gnome-software</td>
<td>R</td>
<td>252</td>
<td>0</td>
<td>vda</td>
<td>31</td>
<td>2336.0</td>
<td>0.73</td>
</tr>
<tr>
<td>9578</td>
<td>awk</td>
<td>R</td>
<td>252</td>
<td>0</td>
<td>vda</td>
<td>17</td>
<td>2228.0</td>
<td>0.66</td>
</tr>
<tr>
<td>7578</td>
<td>llvmpipe-0</td>
<td>R</td>
<td>252</td>
<td>0</td>
<td>vda</td>
<td>156</td>
<td>2204.0</td>
<td>0.07</td>
</tr>
<tr>
<td>9581</td>
<td>pgrep</td>
<td>R</td>
<td>252</td>
<td>0</td>
<td>vda</td>
<td>58</td>
<td>1748.0</td>
<td>0.42</td>
</tr>
<tr>
<td>7531</td>
<td>InputThread</td>
<td>R</td>
<td>252</td>
<td>0</td>
<td>vda</td>
<td>30</td>
<td>1200.0</td>
<td>0.48</td>
</tr>
<tr>
<td>7504</td>
<td>gdbus</td>
<td>R</td>
<td>252</td>
<td>0</td>
<td>vda</td>
<td>3</td>
<td>1164.0</td>
<td>0.30</td>
</tr>
<tr>
<td>1983</td>
<td>llvmpipe-1</td>
<td>R</td>
<td>252</td>
<td>0</td>
<td>vda</td>
<td>39</td>
<td>724.0</td>
<td>0.08</td>
</tr>
<tr>
<td>1982</td>
<td>llvmpipe-0</td>
<td>R</td>
<td>252</td>
<td>0</td>
<td>vda</td>
<td>36</td>
<td>652.0</td>
<td>0.06</td>
</tr>
</tbody>
</table>

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. Execute the **xfsslower** program in one terminal:

```
# /usr/share/bcc/tools/xfsslower 1
```

The command above measures the time the XFS file system spends in performing read, write, open or sync (**fsync**) operations. The **1** argument ensures that the program shows only the operations that are slower than 1 ms.

**NOTE**

When no arguments provided, **xfsslower** by default displays operations slower than 10 ms.

2. In another terminal execute, for example, the following:

```
$ vim text
```

The command above creates a text file in the **vim** editor to initiate certain interaction with the XFS file system.

3. The terminal running **xfsslower** shows something similar upon saving the file from the previous step:

```
TIME     COMM           PID    T BYTES   OFF_KB   LAT(ms) FILENAME
13:07:14 b'bash'        4754   R 256     0           7.11 b'vim'
13:07:14 b'vim'         4754   R 832     0           4.03 b'libgpm.so.2.1.0'
13:07:14 b'vim'         4754   R 32      20          1.04 b'libgpm.so.2.1.0'
13:07:14 b'vim'         4754   R 1982    0           2.30 b'vimrc'
13:07:14 b'vim'         4754   R 1393    0           2.52 b'getscriptPlugin.vim'
13:07:45 b'vim'         4754   S 0       0           6.71 b'text'
13:07:45 b'pool'        2588   R 16      0           5.58 b'text'
...```

Each line above represents an operation in the file system, which took more time than a certain threshold. **xfsslower** is good at exposing possible file system problems, which can take form of unexpectedly slow operations.

The **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 \texttt{xfsslower}, refer to the 
/\texttt{usr/share/bcc/tools/doc/xfsslower_example.txt} file.

For more information about \texttt{fsync}, see \texttt{fsync(2)} manual pages.
CHAPTER 22. ENHANCING SECURITY WITH THE KERNEL INTEGRITY SUBSYSTEM

You can increase the protection of your system by utilizing components of the kernel integrity subsystem. The following sections introduce the relevant components and provide guidance on their configuration.

22.1. THE KERNEL INTEGRITY SUBSYSTEM

The integrity subsystem is a part of the kernel which is responsible for maintaining the overall system's data integrity. This subsystem helps to keep the state of a certain system the same from the time it was built thereby it prevents undesired modification on specific system files from users.

The kernel integrity subsystem consists of two major components:

Integrity Measurement Architecture (IMA)

- Measures files' content whenever it is executed or opened. Users can change this behavior by applying custom policies.
- Places the measured values within the kernel's memory space thereby it prevents any modification from the users of the system.
- Allows local and remote parties to verify the measured values.

Extended Verification Module (EVM)

- Protects files' extended attributes (also known as xattr) that are related to the system's security, like IMA measurements and SELinux attributes, by cryptographically hashing their corresponding values.

Both IMA and EVM also contain numerous feature extensions that bring additional functionality. For example:

IMA-Appraisal

- Provides local validation of the current file's content against the values previously stored in the measurement file within the kernel memory. This extension forbids any operation to be performed over a specific file in case the current and the previous measure do not match.

EVM Digital Signatures

- Allows digital signatures to be used through cryptographic keys stored into the kernel’s keyring.

NOTE

The feature extensions complement each other, but you can configure and use them independently of one another.

The kernel integrity subsystem can harness the Trusted Platform Module (TPM) to harden the system security even more. TPM is a specification by the Trusted Computing Group (TCG) for important cryptographic functions. TPMs are usually built as dedicated hardware that is attached to the platform's

130
motherboard and prevents software-based attacks by providing cryptographic functions from a protected and tamper-proof area of the hardware chip. Some of the TPM features are:

- Random-number generator
- Generator and secure storage for cryptographic keys
- Hashing generator
- Remote attestation

Additional resources

- Integrity Measurement Architecture (IMA)
- Trusted Computing Group resources
- Security hardening
- Basic and advanced configuration of Security-Enhanced Linux (SELinux)

22.2. INTEGRITY MEASUREMENT ARCHITECTURE

Integrity Measurement Architecture (IMA) is a component of the kernel integrity subsystem. IMA aims to maintain the contents of local files. Specifically, IMA measures, stores, and appraises files’ hashes before they are accessed, which prevents the reading and execution of unreliable data. Thereby, IMA enhances the security of the system.

Additional resources

- Integrity Measurement Architecture (IMA)
- Security hardening
- Basic and advanced configuration of Security-Enhanced Linux (SELinux)

22.3. EXTENDED VERIFICATION MODULE

Extended Verification Module (EVM) is a component of the kernel integrity subsystem, which monitors changes in files’ extended attributes (xattr). Many security-oriented technologies, including Integrity Measurement Architecture (IMA), store sensitive file information, such as content hashes, in the extended attributes. EVM creates another hash from these extended attributes and from a special key, which is loaded at boot time. The resulting hash is validated every time the extended attribute is used. For example, when IMA appraises the file.

RHEL 9 accepts the special encrypted key under the evm-key keyring. The key was created by a master key held in the kernel keyrings.

Additional resources

- Integrity Measurement Architecture (IMA)
- Security hardening
- Basic and advanced configuration of Security-Enhanced Linux (SELinux)
22.4. TRUSTED AND ENCRYPTED KEYS

The following section introduces trusted and encrypted keys as an important part of enhancing system security.

**Trusted and encrypted keys** are variable-length symmetric keys generated by the kernel that utilize the kernel keyring service. The fact that this type of keys never appear in the user space in an unencrypted form means that their integrity can be verified, which in turn means that they can be used, for example, by the extended verification module (EVM) to verify and confirm the integrity of a running system. User-level programs can only access the keys in the form of encrypted *blobs*.

Trusted keys need a hardware component: the Trusted Platform Module (TPM) chip, which is used to both create and encrypt (seal) the keys. The TPM seals the keys using a 2048-bit RSA key called the *storage root key* (SRK).

**NOTE**

To use a TPM 1.2 specification, enable and activate it through a setting in the machine firmware or by using the `tpm_setactive` command from the `tpm-tools` package of utilities. Also, the *TrouSers* software stack needs to be installed and the `tcsd` daemon needs to be running to communicate with the TPM (dedicated hardware). The `tcsd` daemon is part of the *TrouSers* suite, which is available through the `trousers` package. The more recent and backward incompatible TPM 2.0 uses a different software stack, where the `tpm2-tools` or `ibm-tss` utilities provide access to the dedicated hardware.

In addition to that, the user can seal the trusted keys with a specific set of the TPM's *platform configuration register* (PCR) values. PCR contains a set of integrity-management values that reflect the firmware, boot loader, and operating system. This means that PCR-sealed keys can only be decrypted by the TPM on the same system on which they were encrypted. However, once a PCR-sealed trusted key is loaded (added to a keyring), and thus its associated PCR values are verified, it can be updated with new (or future) PCR values, so that a new kernel, for example, can be booted. A single key can also be saved as multiple blobs, each with different PCR values.

Encrypted keys do not require a TPM, as they use the kernel Advanced Encryption Standard (AES), which makes them faster than trusted keys. Encrypted keys are created using kernel-generated random numbers and encrypted by a *master key* when they are exported into user-space blobs. The master key is either a trusted key or a user key. If the master key is not trusted, the encrypted key is only as secure as the user key used to encrypt it.

22.5. WORKING WITH TRUSTED KEYS

The following section describes how to create, export, load or update trusted keys with the `keyctl` utility to improve the system security.

**Prerequisites**

- For the 64-bit ARM architecture and IBM Z, the `trusted` kernel module needs to be loaded. For more information on how to load kernel modules, see Managing kernel modules.

- Trusted Platform Module (TPM) needs to be enabled and active. For more information about TPM see, The kernel integrity subsystem and Trusted and encrypted keys.

**Procedure**
1. To create a trusted key using a TPM, execute:

```bash
# keyctl add trusted <name> "new <key_length> [options]" <key_ring>
```

- Based on the syntax, construct an example command as follows:

```bash
# keyctl add trusted kmk "new 32" @u 642500861
```

The command creates a trusted key called `kmk` with the length of 32 bytes (256 bits) and places it in the user keyring (`@u`). The keys may have a length of 32 to 128 bytes (256 to 1024 bits).

2. To list the current structure of the kernel keyrings:

```bash
# keyctl show
Session Keyring
-3 --alswrv 500 500 keyring: _ses
9783714 --alswrv 500 -1 _ keyring: _uid.1000
642500861 --alswrv 500 500 _ trusted: kmk
```

3. To export the key to a user-space blob, execute:

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

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

4. To load the trusted key from the user-space blob, use the `add` subcommand with the blob as an argument:

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

5. Create secure encrypted keys based on the TPM-sealed trusted key:

```bash
# keyctl add encrypted <pass:quotes[name]> "new [format] <pass:quotes[key_type]>:
<pass:quotes[primary_key_name]> <pass:quotes[keylength]>"
<pass:quotes[key_ring]>
```

- Based on the syntax, generate an encrypted key using the already created trusted key:

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

The command uses the TPM-sealed trusted key (`kmk`), produced in the previous step, as a primary key for generating encrypted keys.

**Additional resources**

- `keyctl(1)` manual page
- Trusted and encrypted keys
- Kernel Key Retention Service
The kernel integrity subsystem

22.6. WORKING WITH ENCRYPTED KEYS

The following section describes managing encrypted keys to improve the system security on systems where a Trusted Platform Module (TPM) is not available.

Prerequisites

- For the 64-bit ARM architecture and IBM Z, the encrypted-keys kernel module needs to be loaded. For more information on how to load kernel modules, see Managing kernel modules.

Procedure

1. Use a random sequence of numbers to generate a user key:

```
# keyctl add user kmk-user "$(dd if=/dev/urandom bs=1 count=32 2>/dev/null)" @u
427069434
```

The command generates a user key called kmk-user which acts as a primary key and is used to seal the actual encrypted keys.

2. Generate an encrypted key using the primary key from the previous step:

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

3. Optionally, list all keys in the specified user keyring:

```
# keyctl list @u
2 keys in keyring:
427069434: --alswrv 1000 1000 user: kmk-user
1012412758: --alswrv 1000 1000 encrypted: encr-key
```

**IMPORTANT**

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

Additional resources

- keyctl(1) manual page
- Kernel Key Retention Service

22.7. ENABLING INTEGRITY MEASUREMENT ARCHITECTURE AND EXTENDED VERIFICATION MODULE

Integrity measurement architecture (IMA) and extended verification module (EVM) belong to the kernel integrity subsystem and enhance the system security in various ways. The following section describes how to enable and configure IMA and EVM to improve the security of the operating system.
**Prerequisites**

- Verify that the `securityfs` filesystem is mounted on the `/sys/kernel/security/` directory and the `/sys/kernel/security/integrity/ima/` directory exists.

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

- Verify that the `systemd` service manager is already patched to support IMA and EVM on boot time:

```bash
# dmesg | grep -i -e EVM -e IMA
[    0.000000] Command line: BOOT_IMAGE=(hd0,msdos1)/vmlinuz-5.14.0-1.el9.x86_64oot=/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.
[    0.911527] ima: No TPM chip found, activating TPM-bypass!
[    0.911538] ima: Allocated hash algorithm: sha1
[    0.911580] evm: Initialising EVM extended attributes:
[    0.911581] evm: security.selinux
[    0.911581] evm: security.ima
[    0.911582] evm: security.capability
[    0.911582] evm: HMAC attrs: 0x1
[    1.715151] systemd[1]: systemd 239 running in system mode. (+PAM +AUDIT +SELINUX +IMA -APPARMOR +SMACK +SYSVINIT +UTMP +LIBCRYPTSETUP +GCRYPT +GNUTLS +ACL +XZ +LZ4 +SECCOMP +BLKID +ELFUTILS +KMOD +IDN2 -IDN +PCRE2 default-hierarchy=legacy)
[    3.824198] fbcon: qxldrmfb (fb0) is primary device
[    4.673457] PM: Image not found (code -22)
[    6.549966] systemd[1]: systemd 239 running in system mode. (+PAM +AUDIT +SELINUX +IMA -APPARMOR +SMACK +SYSVINIT +UTMP +LIBCRYPTSETUP +GCRYPT +GNUTLS +ACL +XZ +LZ4 +SECCOMP +BLKID +ELFUTILS +KMOD +IDN2 -IDN +PCRE2 default-hierarchy=legacy)
```

**Procedure**

1. Add the following kernel command line parameters:

```bash
# grubby --update-kernel=/boot/vmlinuz-$(uname -r) --args="ima_policy=appraise_tcb ima_appraise=fix evm=fix"
```

The command enables IMA and EVM in the `fix` mode for the current boot entry and allows users to gather and update the IMA measurements.

The `ima_policy=appraise_tcb` kernel command line parameter ensures that the kernel uses the default Trusted Computing Base (TCB) measurement policy and the appraisal step. The appraisal part forbids access to files, whose prior and current measures do not match.
2. Reboot to make the changes come into effect.

3. Optionally, verify that the parameters have been added to the kernel command line:

```bash
# cat /proc/cmdline
BOOT_IMAGE=(hd0,msdos1)/vmlinuz-5.14.0-1.el9.x86_64 root=/dev/mapper/rhel-root ro
options=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:

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

The kernel master key (kmk) is kept entirely in the kernel space memory. The 32-byte long value of the kernel master key kmk is generated from random bytes from the /dev/urandom file and placed in the user (@u) keyring. The key serial number is on the second line of the previous output.

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

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

The command uses kmk to generate and encrypt a 64-byte long user key (named evm-key) and places it in the user (@u) keyring. The key serial number is on the second line of the previous output.

**IMPORTANT**

It is necessary to name the user key as evm-key because that is the name the EVM subsystem is expecting and is working with.

6. Create a directory for exported keys:

```bash
# mkdir -p /etc/keys/
```

7. Search for the kmk key and export its value into a file:

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

The command places the unencrypted value of the kernel master key (kmk) into a file of previously defined location (/etc/keys/).

8. Search for the evm-key user key and export its value into a file:

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

The command places the encrypted value of the user evm-key key into a file of arbitrary location. The evm-key has been encrypted by the kernel master key earlier.

9. Optionally, view the newly created keys:

```bash
```

Red Hat Enterprise Linux 9.0 Beta Managing, monitoring, and updating the kernel
# keyctl show
Session Keyring
974575405 --alswrv 0 0 keyring: _ses
299489774 --alswrv 0 65534 keyring: _uid.0
748544121 --alswrv 0 0 user: kmk
641780271 --alswrv 0 0 encrypted: evm-key

You should be able to see a similar output.

10. Activate EVM:

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

11. Optionally, verify that EVM has been initialized:

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

Additional resources

- Extended verification module
- Integrity measurement architecture
- The kernel integrity subsystem
- Trusted and encrypted keys.

### 22.8. COLLECTING FILE HASHES WITH INTEGRITY MEASUREMENT ARCHITECTURE

The first level of operation of integrity measurement architecture (IMA) is the measurement phase, which allows to create file hashes and store them as extended attributes (xattrs) of those files. The following section describes how to create and inspect the files' hashes.

Prerequisites

- Enable integrity measurement architecture (IMA) and extended verification module (EVM) as described in Enabling integrity measurement architecture and extended verification module.
- Verify that the `ima-evm-utils`, `attr`, and `keyutils` packages are already installed:

```bash
# dnf install ima-evm-utils attr keyutils
```

Updating Subscription Management repositories.
This system is registered to Red Hat Subscription Management, but is not receiving updates.
You can use subscription-manager to assign subscriptions.
Last metadata expiration check: 0:58:22 ago on Fri 14 Feb 2020 09:58:23 AM CET.
Package ima-evm-utils-1.1-5.el8.x86_64 is already installed.
Package attr-2.4.48-3.el8.x86_64 is already installed.
Package keyutils-1.5.10-7.el8.x86_64 is already installed.
Dependencies resolved.
Nothing to do.
Complete!
Procedure

1. Create a test file:

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

   IMA and EVM ensure that the example file `test_file` is assigned hash values, which are stored as its extended attributes.

2. Inspect extended attributes of the file:

   ```bash
   # getfattr -m . -d test_file
   # file: test_file
   security.evm=0sAnDiy4VPA0HArpPO/EqiutnNyBql
   security.ima=0sAQODEeuUnWzwwKYk+n66h/vby3eD
   security.selinux="unconfined_u:object_r:admin_home_t:s0"
   ```

   The previous example output shows extended attributes related to SELinux and the IMA and EVM hash values. EVM actively adds a `security.evm` extended attribute and detects any offline tampering to xattrs of other files such as `security.ima` that are directly related to content integrity of files. The value of the `security.evm` field is in Hash-based Message Authentication Code (HMAC-SHA1), which was generated with the `evm-key` user key.

Additional resources

- Security hardening
- Extended verification module
- Integrity measurement architecture
CHAPTER 23. USING ANSIBLE ROLES TO PERMANENTLY CONFIGURE KERNEL PARAMETERS

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

23.1. 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
23.2. 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 on 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:

```bash
# 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:
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:

   ```yaml
   ---
   - 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.

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

This example shows the successful verification of a playbook.

5. Execute your playbook.

```bash
# ansible-playbook kernel-roles.yml
...
BECOME password:
PLAY [Configure kernel settings]
********************************************************************************************************
```

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**

- Getting started with 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 24. USING ANSIBLE AUTOMATION PLATFORM AND ANSIBLE EXECUTION ENVIRONMENTS TO AUTOMATE WORKLOAD

Previously, customers used to develop their Ansible content such as playbooks, roles, or collections locally. They had to ensure that all the necessary dependencies were readily available directly on their computer for their Ansible content to work correctly. The dependencies could be:

- Python libraries
- Dependencies for Ansible Collections
- Dependencies for system packages

Afterwards, customers had to install the content on their production environment. This could cause a problem with restrictions when putting such content on the lock-down production systems.

Next, verifying that everything was installed correctly could be another obstacle. Especially when some dependencies were omitted, or when additional content which required dependencies was added. Since at each such time the whole process had to be repeated.

The following sections introduce Ansible execution environment and explain how to use it to avoid the obstacles outlined above.

IMPORTANT

For customers to profit from content consumption of this chapter, it is necessary to have the Ansible Automation Platform subscription.
24.1. ANSIBLE EXECUTION ENVIRONMENT

Ansible execution environments (AEE) are consistent and shareable container images based on Universal Base Image (UBI). These container images serve as Ansible control nodes and provide predictable and reproducible run-time environments. These environments contain everything needed to run Ansible Automation Platform, and resolve challenges with Ansible content that requires non-default dependencies. With AEE, you can have a customized container, which has only as much data as you need to run jobs. This streamlines and simplifies the development process and helps ensure predictable, reproducible results.

The execution environment is where your Ansible playbook actually runs. To run a playbook, you normally use a command such as `ansible-playbook`, or `ansible-navigator`. However, the playbook runs inside the container rather than directly on your system.

Figure 24.2. Ansible execution environment

AEE container image typically contains:

- Ansible Core
- Ansible Runner
- Python and/or system dependencies
- Ansible Collections
  - modules
  - playbooks
  - roles
  - plugins (Inventory, Connection, Lookup, ...)

The Ansible Builder utility prepares execution environments, which you can distribute from any container registry.

Red Hat provides three pre-built AEEs:
## Short name | Full name | Description
---|---|---
Minimal | ansible-automation-platform-21-ee-minimal-rhel8 | A minimal execution environment based on Ansible Core 2.12.
Supported | ansible-automation-platform-21-ee-supported-rhel8 | An execution environment based on Ansible Core 2.12 that includes Red Hat-supported certified content collections and their software dependencies.
Ansible 2.9 | ansible-automation-platform-21-ee-29-rhel8 | An execution environment based on Ansible 2.9.

You can use any of these execution environments, or you can create your own.

### Additional resources

- Using collections
- Getting Started With Ansible Content Collections
- Using Runner with Execution Environments
- Universal Base Images (UBI): Images, repositories, packages, and source code
- About ansible-core

## 24.2. PREPARING A SYSTEM FOR CUSTOM ANSIBLE EXECUTION ENVIRONMENTS

If your roles do not succeed using the Red Hat pre-built execution environments, then you can prepare your system to be able to produce your own execution environment.

### Prerequisites

- You have root permissions.
- You obtained an API token for client authentication from the [Connect to Hub](#) website.

### Procedure

1. Enable the repository containing the necessary utilities for compiling custom Ansible execution environments:

   ```
   # subscription-manager repos --enable ansible-automation-platform-2.1-for-rhel-8-x86_64-rpms
   ```

   The repository makes available utilities such as ansible-builder, or ansible-navigator and their dependencies. These utilities are useful for creating and manipulating Ansible execution environments.

2. Install the ansible-core, ansible-builder, and ansible-navigator utilities:
# dnf install ansible-core ansible-builder ansible-navigator

Ansible core is an automation utility for system configuration and deployment, orchestration of tasks and other advanced IT challenges.

Ansible Builder is a utility that you can use to customize and build your own Ansible execution environments with the collections and dependencies you need. The utility uses metadata defined in various Ansible collections, as well as by the user.

Ansible Navigator is a utility that you can use to debug automation, inspect multiple aspects of the automation environment, including execution environments, inventories, collections, modules and more.

3. Install the container-tools module:

    # dnf module install container-tools

The container-tools module includes software packages that install several utilities such as podman, or skopeo. You need these utilities to compile your execution environment with ansible-builder.

4. Login to the Red Hat Registry:

    # podman login registry.redhat.io
    Username: 
    Password: 
    Login Succeeded!

Use your Customer Portal credentials to access a base container image when you compile your custom execution environment.

5. Create a new directory to store the files used to create the new execution environment:

    # mkdir ansible-execution-environment

6. In the new directory, create the execution-environment.yml file with this content:

```
---
version: 1
ansible_config: ansible.cfg
build_arg_defaults:
    EE_BASE_IMAGE: registry.redhat.io/ansible-automation-platform-21/ee-supported-rhel8:latest
dependencies:
    galaxy: requirements.yml
    system: bindep.txt
```

The execution-environment.yml file is a definition file that outlines the execution environment’s collection-level dependencies, base image source, and overrides for specific items within the execution environment.

1 ansible.cfg is required if the new execution environment pulls collections from a location that requires authentication, such as Ansible Automation Hub.
Initial base image to use as the starting point, in case you want to further customize your execution environment.

Specifies Ansible collections to be used by your execution environment. You can obtain such collections from Automation Hub on Red Hat Hybrid Cloud.

Specifies system-level dependencies and it can be used to specify cross-platform requirements.

7. Create the `requirements.yml` file with similar content according to your needs:

```yaml
---
collections:
  - name: redhat.rhel_system_roles
    version: 1.14.0
...
```

1. Ansible collection name
2. Specific version to be obtained. If not defined, the latest version is used.

8. Create the `ansible.cfg` file with similar content according to your needs:

```yaml
[galaxy]
server_list = automation_hub

[galaxy_server.automation_hub]
url=https://cloud.redhat.com/api/automation-hub/
token=eyJh...
```

1. List of all repositories ansible-builder can use to download collections. For each name you define, add a [galaxy_server.name] section to provide the connection parameters.
2. URL to access the repository
3. URL for authentication
4. Authentication token associated with your Ansible Automation Hub account.

Since the example collection redhat.rhel_system_roles is located in Ansible Automation Hub, you need to set in the ansible.cfg configuration file the corresponding directives for this source.

By default, the ansible-builder command uses Ansible Galaxy source to download collections.

9. Optionally, create the `bindep.txt` file with similar content according to your needs:

```text
libxml2-devel [platform:rpm]
grep >=3.1
...
The `bindep.txt` file lists additional dependencies for any system-level application or library.

10. The final directory structure of your `ansible-execution-environment` could look like this:

```
├── ansible.cfg
├── bindep.txt
├── execution-environment.yml
├── my_token
└── requirements.yml
```

Additional resources

- About ansible-core
- `podman(1)` manual pages
- Introduction to Ansible Builder
- Red Hat Container Registry Authentication
- bindep
- Configuring Ansible
- Working With Playbooks
- Using collections

24.3. CREATING CUSTOM ANSIBLE EXECUTION ENVIRONMENTS

To use RHEL System Roles without having to resolve various platform dependencies (OS dependencies, Python dependencies, Ansible collection dependencies), you need to build and use an Ansible execution environment. This environment is a container image with the Ansible runtime, RHEL System Roles, and any OS, Python, and Ansible dependencies built-in.

Prerequisites

- You have arranged your system as described in [Preparing a system for custom Ansible execution environments](#).
- You have root permissions.

Procedure

1. Build a container image with the execution environment.

```
# ansible-builder build --tag my_first_image:v1.0
```

Running command:
```
podman build -f context/Containerfile -t my_first_image:v1.0 context
```

Complete! The build context can be found at: /root/ansible-execution-environment/context

The `--tag` option provides a name to the container image. The name has two parts: a name and an optional tag. For example, you can use `--tag my_first_image:v1.0` to name the container `my_first_image` and give it the `v1.0` tag. The tag defaults to `latest` if not specified.
After compilation, your project directory will have a similar structure as in the following example:

```
├── ansible.cfg
├── bindep.txt
├── context
│   ├── _build
│   │   ├── ansible.cfg
│   │   └── bindep.txt
│   └── requirements.yml
└── Containerfile
    ├── execution-environment.yml
    └── requirements.yml
```

Each run of the `ansible-builder build` command recreates the `context/Containerfile` file, which removes any manual changes made to that file.

2. Verify the newly created image:

```
# podman images
REPOSITORY                                                                 TAG         IMAGE ID
CREATED             SIZE
localhost/my_first_image                                          latest      4acd74808d51    33 seconds ago      1.05 GB
<none>                                                                                <none>      718a4ac55dab    About a minute ago 796 MB
<none>                                                                                <none>      6b8d70da5486    2 minutes ago 889 MB
quay.io/ansible/ansible-runner                                            latest      f420eabd59f3    11 hours ago        885 MB
registry.redhat.io/ansible-automation-platform-20-early-access/ansible-builder-rhel8 2.0.0 df5d9fdcdbd4 6 months ago 359 MB
```

Additional resources

- Ansible execution environment
- `podman(1)` manual page
- Introduction to Ansible Builder

### 24.4. DEPLOYING RHEL SYSTEM ROLES COLLECTION USING CUSTOM ANSIBLE EXECUTION ENVIRONMENT

As an Ansible Automation Platform (AAP) customer, you can benefit from Ansible execution environments when running your playbooks. Such environments already contain the system dependencies, the required Ansible content, and other required packages. This environment setup enables you to automate your workload without having to deal with any missing dependencies.

**IMPORTANT**

You must have the Ansible Automation Platform (AAP) subscription.
Prerequisites

- You have root permissions.
- You configured your system as described in *Preparing a system for custom Ansible execution environments*.
- You created a custom Ansible execution environment as described in *Creating custom Ansible execution environments*.
- An inventory of managed hosts is present on the control machine and Ansible is able to connect to them.

Procedure

1. Optionally, review the `inventory` file:

   ```
   # cat /home/devops/<ansible_project_name>/inventory
   [production]
   server1.example.com

   [test]
   server2.example.com

   [webservers:children]
   production
   test
   ```

   The file defines the `[production]` group and other groups and sub-groups such as `[webservers]`. It allows you to run Ansible more effectively against a specific set of systems.

2. Create the `ansible.cfg` file with similar content according to your needs:

   ```
   [defaults]
   inventory = ./inventory
   remote_user = devops

   [privilege_escalation]
   become = True

   [galaxy]
   server_list = automation_hub

   [galaxy_server.automation_hub]
   url=https://cloud.redhat.com/api/automation-hub/
   token=eyJh...
   ```

   - The `[defaults]` section specifies a path to the inventory file of managed hosts or a user account to be used for connecting to managed hosts. You must be able to establish an SSH connection to the managed hosts with that user account.
   - The `[privilege_escalation]` section defines that user privileges are shifted to `root` once connected to the specified managed hosts.
3. Settings for Ansible Automation Hub you used when creating the execution environment.

3. Add your SSH private keys to the `ssh-agent` utility:

```bash
[devops@controlnode ansible-execution-environment]$ eval $(ssh-agent)
Agent pid 5777
[devops@controlnode ansible-execution-environment]$ ssh-add
Identity added: /home/devops/.ssh/id_rsa (devops@controlnode.example.com)
```

With that you ensure the correct usage of your user account SSH keys during interaction with the Ansible execution environment. Also, you prevent any potential problems with SSH when using the `ansible-navigator` utility with Ansible execution environments.

4. Create an Ansible playbook that uses the `redhat.rhel_system_roles` collection.

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

   ```
   # vi /home/devops/<ansible_project_name>/test.yml
   ```

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

   b. Insert the following content into the file:

   ```yaml
   ---
   - name: A playbook to verify that execution environment is working
     hosts: server2.example.com
     collections:
     - redhat.rhel_system_roles
     tasks:
     - name: Include kernel_settings role
       include_role:
       name: redhat.rhel_system_roles.kernel_settings
       vars:
       kernel_settings_sysctl:
       - name: fs.file-max
         value: 400000
       - name: kernel.threads-max
         value: 65536
   ```

   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 `collections` keyword specifies what Ansible content your playbook should use.

   The first task dynamically includes and executes the `kernel_settings` RHEL System Role from the `redhat.rhel_system_roles` Ansible collection.
The `vars` section represents a list of variables containing selected kernel parameter names and values to which they have to be set.

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

```bash
# ansible-playbook --syntax-check test.yml
```

playbook: test.yml

This example shows the successful verification of a playbook.

6. Execute your playbook using the `ansible-navigator` utility:

```bash
# ansible-navigator run test.yml --ee true --eei my_first_image --pp never
```

The `--ee true` option enables the execution environment.

The `--eei my_first_image` option specifies a container image for the execution environment.

The `--pp never` option disables pulling a container image to be used from a remote registry. The image you need is available locally on your control node.

For more information, refer to the output of the `ansible-navigator --help` command.

The interface of `ansible-navigator` looks as follows:

**Figure 24.3. Ansible navigator output**

To see more details of your playbook, hit the 0 key.

**Figure 24.4. Ansible navigator playbook details**

To see task-level details, hit the respective numeral key corresponding to the task you are interested in.

**Verification**

1. Log into the managed host, which was affected by your playbook and check the targeted kernel parameters:

```bash
# sysctl -a | grep kernel.threads-max
kernel.threads-max = 65536
```

```bash
# sysctl -a | grep fs.file-max
fs.file-max = 400000
```
2. Optionally, verify that the Ansible execution environment was deployed correctly:

```bash
# ansible-galaxy collection list
<NO_OUTPUT>

# ansible-galaxy role list
/usr/share/ansible/roles
/etc/ansible/roles
[WARNING]: - the configured path /home/devops/.ansible/roles does not exist.
```

The output confirms that there were not any roles or collections installed on the control node. All Ansible content, system dependencies and other necessities for correct execution of your tasks was provided by the Ansible execution environment.

**Additional resources**

- Getting started with RHEL System Roles
- Build Your Inventory
- Configuring Ansible
- Working With Playbooks
- Using collections in a Playbook
- Using Variables
- Using Ansible Navigator
- Upgrading to automation execution environments