Red Hat Enterprise Linux 7
Kernel Administration Guide

Examples of Tasks for Managing the Kernel

Mark Flitter
Jaromir Hradilek
Robert Krátký
Douglas Silas
Marie Dolezelova
Stephen Wadeley
Eliska Slobodova
Maxim Svistunov
Florian Nadge

Red Hat Enterprise Linux 7 Kernel Administration Guide

Examples of Tasks for Managing the Kernel

Mark Flitter
Red Hat Customer Content Services
mflitter@redhat.com

Douglas Silas
Red Hat Customer Content Services

Eliska Slobodova
Red Hat Customer Content Services

Jaromir Hradilek
Red Hat Customer Content Services

Marie Dolezelova
Red Hat Customer Content Services

Maxim Svistunov
Red Hat Customer Content Services

Robert Krátký
Red Hat Customer Content Services

Stephen Wadeley
Red Hat Customer Content Services

Florian Nadge
Red Hat Customer Content Services

Legal Notice

Abstract

The Kernel Administration Guide documents tasks for maintaining the Red Hat Enterprise Linux 7 kernel. This first release, includes information on using kpatch, managing kernel modules, and manually updating the kernel.

Table of Contents

Chapter 1. Working with kpatch

This chapter explains how **kpatch** works and how to use it.

1.1. What Is kpatch?

kpatch is a live kernel patching solution that allows you to patch a running kernel without rebooting or restarting any processes. It enables system administrators to apply critical security patches to the kernel immediately, without having to wait for long-running tasks to complete, for users to log off, or for scheduled downtime. It gives more control over uptime without sacrificing security or stability.

Warning

Some incompatibilities exist between **kpatch** and other kernel subcomponents. Read the Section 1.4, “Limitations” section carefully before using **kpatch**.

1.2. What Is the Scope of Support for kpatch?

- Live kernel patching with **kpatch** is supported from Red Hat Enterprise Linux 7.2 onwards.
- Live kernel patching is supported for customers with Premium SLA subscriptions.
- Live kernel patching is *only* supported on the active Red Hat Enterprise Linux 7 maintenance stream that is within the current async errata phase. See Red Hat Enterprise Linux Life Cycle for information about current support phases.
- Live kernel patching is *not* available on Extended Update Support releases.
- **kpatch** is *not* supported on the Red Hat Enterprise Linux Realtime (RT) kernel.
- **kpatch** is *not* supported on Red Hat Enterprise Linux 5 or Red Hat Enterprise Linux 6.
- Only *one* live kernel patch may be installed on the kernel at any given time.
- Not all issues may be covered under live kernel patching, including hardware enablement.

1.3. Access and Delivery

Live kernel patching capability is implemented via a kernel module (kmod) that is delivered as an RPM package. The **kpatch** utility is used to install and remove the kernel modules for live kernel patching.

Customers with Premium subscriptions are eligible to request a live kernel patch as part of an accelerated fix solution from Red Hat Support.

Customers with Premium subscriptions who typically used **hotfix** kernels which required a reboot can now request a **kpatch** kmod that requires no down time. The **kpatch** patch will be supported for 30 days after the errata that contains the fix is released, in the same manner as **hotfix** kernels.

Customers should open a support case directly in the Red Hat Customer Portal and discuss appropriate accelerated fix options.

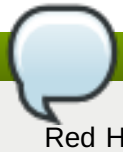
1.4. Limitations

kpatch 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 the probe has been removed.

Do *not* directly access the **ftrace** output file, for example by running **cat /sys/kernel/debug/tracing/trace**. The **trace-cmd** utility is supported instead.

Do *not* suspend or hibernate the system when using **kpatch**. This can result in a patch being temporarily disabled for a small amount of time.



Note

Red Hat is actively working to remove many of the limitations of **kpatch** for future releases.

1.5. How Do I Enable and Use kpatch?

The components of **kpatch** are as follows:

Components of kpatch

A systemd Integration Point

A **systemd** service called **kpatch.service** that is required by **multiuser.target** which loads the kpatch modules at boot time.

A Patch Module

- The delivery mechanism for new kernel code.
- This is another kernel module that is named to match the **kpatch** being applied.
- The patch module contains the compiled code from the latest hot fixes introduced to the kernel.
- The patch modules register with the core module, **kpatch.ko** and provide information about which original functions will be replaced, with corresponding pointers to the replacement functions.

The kpatch Utility

A command-line tool which allows you to manage patch modules.

1.5.1. Installing the kpatch Tools

Before you can install a **kpatch** module, you must install the **kpatch** tools package. To do so, type the following at a shell prompt as **root**

```
# yum install kpatch
```

1.5.2. Installing a kpatch Hot Fix

To install a **kpatch** hot fix, install the supplied **kpatch-patch** RPM package with **yum**. For example, to install **kpatch-patch-7.0-1.el7.x86_64.rpm**, issue the following command as **root**

```
# yum install kpatch-patch-7.0-1.el7.x86_64.rpm
```

1.5.3. Listing Installed kpatch Hot Fixes

To verify a patch is loaded and installed, run the **kpatch list** command as **root**:

```
# kpatch list
Loaded patch modules:
kpatch_7_0_1_el7
Installed patch modules:
kpatch-7-0-1-el7.ko (3.10.0-121.el7.x86_64)
kpatch-7-0-1-el7.ko (3.10.0-123.el7.x86_64)
```

The example output above shows that the module has been loaded into the kernel, meaning the kernel is now patched with the latest hot fixes contained in the **kpatch-patch-7.0-1.el7.x86_64.rpm** package. It also shows that it has been saved to **/var/lib/kpatch** to be loaded by **systemd** during future reboots for kernel versions 3.10.0-121 and 3.10.0-123.

1.5.4. Updating a kpatch Hot Fix

If a new version of the **kpatch-patch** RPM package is later released, upgrade the applied patch with **yum** For example, to upgrade to **kpatch-patch-7.0-2.el7.x86_64.rpm** run as **root**

```
# yum update kpatch-patch-7.0-2.el7.x86_64.rpm
```

Upgrading the RPM package atomically replaces the patch module in the running kernel and updates the **/var/lib/kpatch** structures used by **systemd** on reboot.

Note

The patch modules in the **kpatch-patch** RPM packages are cumulative. Consequently, you could skip installing kpatch-patch-7.0-1 and instead start with installing kpatch-patch-7.0-2 if it were available.

Loading a patch module sets the **TAINT_LIVEPATCH** kernel taint flag (which corresponds to bit 15), **TAINT_OOT_MODULE** (which corresponds to bit 12), and **TAINT_TECH_PREVIEW** (which corresponds to bit 29). To determine whether the kernel has been patched, use the **cat** command to display the contents of **/proc/sys/kernel/tainted** and check the value in the file. Unless you have other taint flags set, the value should be **536907776 (0x20009000)** when the kernel is patched.

1.5.5. Removing a kpatch Hot Fix

To unload a **kpatch** module from the running kernel, use the **kpatch unload** command, specifying the name of the patch module. For example, to unload **kpatch_7_0_2_el7** type the following at a shell prompt as **root**

```
# kpatch unload kpatch_7_0_2_el7
```

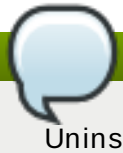
The patch module must also be uninstalled from **/var/lib/kpatch** with the **kpatch** uninstall command as follows:

```
# kpatch uninstall kpatch_7_0_2_el7
```

The default behavior of this command is to uninstall the **kpatch** from the kernel corresponding to the current kernel version, but you can specify a different kernel version by using the **kernel-version** option:

```
# kpatch uninstall --kernel-version 3.10.0-121.el7.x86_64
kpatch_7_0_2_el7
```

Alternatively, you can uninstall the **kpatch-patch** RPM package, which also removes the patch module from **/var/lib/kpatch**



Note

Uninstalling the RPM package does not unload the **kpatch** module from the kernel. An explicit call to **kpatch unload** as described above is required.

1.6. How Does kpatch Work?

The **kpatch** utilities use **ftrace** for arbitrary remapping of pointers to kernel functions. When a live kernel patch is applied to a system, the following things happen:

1. The new compiled code in the module is copied to **/var/lib/kpatch** and registered for re-application to the kernel via **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 to the **ftrace** mechanism which bypasses the original functions and redirects the kernel to patched version of the function.

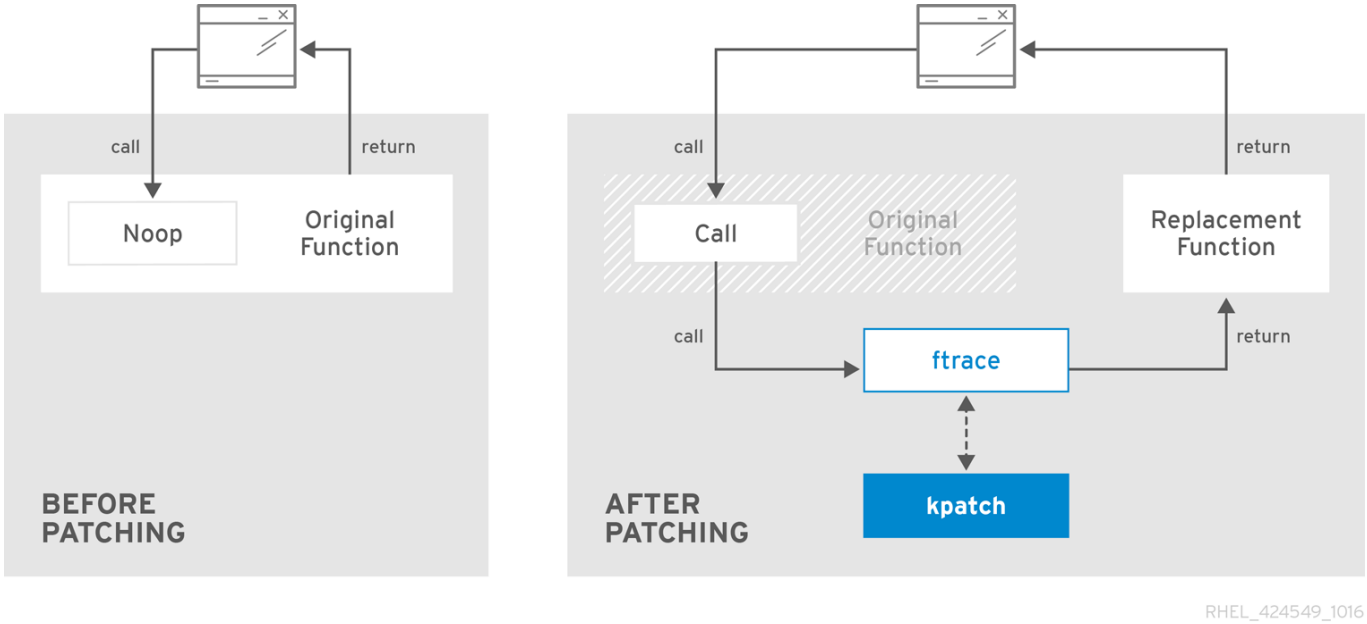


Figure 1.1. How kpatch Works

1.7. Are Any Third-party Live Patching Solutions Supported?

Although several third-party and proprietary tools that provide live kernel patching are available, Red Hat only supports **kpatch** and the RPM modules supplied through your Red Hat support contract. Red Hat cannot support third-party live-patches, however requests for Engineering and an official Red Hat **kpatch** can be opened at any time.

For any Red Hat review of a third-party live-patch, the source code would need to be supplied to determine if they meet the following criteria:

1. Impact the same subsystems and codepaths as the kernel encountered during a failure.
2. Applying the same patches using supported means, within the applicable streams, result in no failure being encountered.

Red Hat recommends that you open a case with the live kernel patching vendor at the outset of any investigation in which a root cause determination is necessary. This will allow 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 kernel 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 will 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? in the knowledgebase at https://access.redhat.com/articles/1067.

Chapter 2. Working with Kernel Modules

The Linux kernel is modular, which means it can extend its capabilities through the use of dynamically-loaded *kernel modules*. A kernel module can provide:

- a device driver which adds support for new hardware; or,
- support for a file system such as **GFS2** or **NFS**.

Like the kernel itself, modules can take parameters that customize their behavior, though the default parameters work well in most cases. User-space tools can list the modules currently loaded into a running kernel; query all available modules for available parameters and module-specific information; and load or unload (remove) modules dynamically into or from a running kernel. Many of these utilities, which are provided by the *kmod* package, take module dependencies into account when performing operations so that manual dependency-tracking is rarely necessary.

On modern systems, kernel modules are automatically loaded by various mechanisms when the conditions call for it. However, there are occasions when it is necessary to load or unload modules manually, such as when one module is preferred over another although either could provide basic functionality, or when a module is misbehaving.

This chapter explains how to:

- use the user-space **kmod** utilities to display, query, load and unload kernel modules and their dependencies;
- set module parameters both dynamically on the command line and permanently so that you can customize the behavior of your kernel modules; and,
- load modules at boot time.

Note

In order to use the kernel module utilities described in this chapter, first ensure the *kmod* package is installed on your system by running, as root:

```
# yum install kmod
```

2.1. Listing Currently-Loaded Modules

You can list all kernel modules that are currently loaded into the kernel by running the **lsmod** command, for example:

```
# lsmod
Module                  Size  Used by
tcp_lp                 12663  0
bnep                   19704  2
bluetooth             372662  7 bnep
rfkill                 26536  3 bluetooth
fuse                   87661  3
ebtable_broute         12731  0
bridge                110196  1 ebtable_broute
```

```
stp                    12976  1 bridge
llc                    14552  2 stp,bridge
ebtable_filter         12827  0
ebtables               30913  3
ebtable_broute,ebtable_nat,ebtable_filter
ip6table_nat           13015  1
nf_nat_ipv6            13279  1 ip6table_nat
iptable_nat            13011  1
nf_conntrack_ipv4      14862  4
nf_defrag_ipv4         12729  1 nf_conntrack_ipv4
nf_nat_ipv4            13263  1 iptable_nat
nf_nat                 21798  4
nf_nat_ipv4,nf_nat_ipv6,ip6table_nat,iptable_nat
[output truncated]
```

Each row of **lsmod** output specifies:

- the name of a kernel module currently loaded in memory;
- the amount of memory it uses; and,
- the sum total of processes that are using the module and other modules which depend on it, followed by a list of the names of those modules, if there are any. Using this list, you can first unload all the modules depending the module you want to unload. For more information, see Section 2.4, “Unloading a Module”.

Finally, note that **lsmod** output is less verbose and considerably easier to read than the content of the **/proc/modules** pseudo-file.

2.2. Displaying Information About a Module

You can display detailed information about a kernel module using the **modinfo** ***module_name*** command.

Note

When entering the name of a kernel module as an argument to one of the **kmod** utilities, do not append a **.ko** extension to the end of the name. Kernel module names do not have extensions; their corresponding files do.

Example 2.1. Listing information about a kernel module with lsmod

To display information about the **e1000e** module, which is the Intel PRO/1000 network driver, enter the following command as **root**:

```
# modinfo e1000e
filename:       /lib/modules/3.10.0-
121.el7.x86_64/kernel/drivers/net/ethernet/intel/e1000e/e1000e.ko
version:        2.3.2-k
license:        GPL
description:    Intel(R) PRO/1000 Network Driver
author:         Intel Corporation, <linux.nics@intel.com>
srcversion:     E9F7E754F6F3A1AD906634C
```

```
alias:          pci:v00008086d000015A3sv*sd*bc*sc*i*
alias:          pci:v00008086d000015A2sv*sd*bc*sc*i*
[some alias lines omitted]
alias:          pci:v00008086d0000105Esv*sd*bc*sc*i*
depends:        ptp
intree:         Y
vermagic:       3.10.0-121.el7.x86_64 SMP mod_unload modversions
signer:         Red Hat Enterprise Linux kernel signing key
sig_key:
42:49:68:9E:EF:C7:7E:95:88:0B:13:DF:E4:67:EB:1B:7A:91:D1:08
sig_hashalgo:   sha256
parm:           debug:Debug level (0=none,...,16=all) (int)
parm:           copybreak:Maximum size of packet that is copied to a
new buffer on receive (uint)
parm:           TxIntDelay:Transmit Interrupt Delay (array of int)
parm:           TxAbsIntDelay:Transmit Absolute Interrupt Delay (array
of int)
parm:           RxIntDelay:Receive Interrupt Delay (array of int)
parm:           RxAbsIntDelay:Receive Absolute Interrupt Delay (array
of int)
parm:           InterruptThrottleRate:Interrupt Throttling Rate (array
of int)
parm:           IntMode:Interrupt Mode (array of int)
parm:           SmartPowerDownEnable:Enable PHY smart power down
(array of int)
parm:           KumeranLockLoss:Enable Kumeran lock loss workaround
(array of int)
parm:           WriteProtectNVM:Write-protect NVM [WARNING: disabling
this can lead to corrupted NVM] (array of int)
parm:           CrcStripping:Enable CRC Stripping, disable if your BMC
needs the CRC (array of int)
```

Here are descriptions of a few of the fields in **modinfo** output:

filename

The absolute path to the **.ko** kernel object file. You can use **modinfo -n** as a shortcut command for printing only the **filename** field.

description

A short description of the module. You can use **modinfo -d** as a shortcut command for printing only the description field.

alias

The **alias** field appears as many times as there are aliases for a module, or is omitted entirely if there are none.

depends

This field contains a comma-separated list of all the modules this module depends on.

Note

If a module has no dependencies, the **depends** field may be omitted from the output.

parm

Each **parm** field presents one module parameter in the form ***parameter_name*:*description***, where:

- *parameter_name* is the exact syntax you should use when using it as a module parameter on the command line, or in an option line in a **.conf** file in the **/etc/modprobe.d/** directory; and,
- *description* is a brief explanation of what the parameter does, along with an expectation for the type of value the parameter accepts (such as int, unit or array of int) in parentheses.

Example 2.2. Listing module parameters

You can list all parameters that the module supports by using the **-p** option. However, because useful value type information is omitted from **modinfo -p** output, it is more useful to run:

```
# modinfo e1000e | grep "^parm" | sort
parm:           copybreak:Maximum size of packet that is copied
to a new buffer on receive (uint)
parm:           CrcStripping:Enable CRC Stripping, disable if
your BMC needs the CRC (array of int)
parm:           debug:Debug level (0=none,...,16=all) (int)
parm:           InterruptThrottleRate:Interrupt Throttling Rate
(array of int)
parm:           IntMode:Interrupt Mode (array of int)
parm:           KumeranLockLoss:Enable Kumeran lock loss
workaround (array of int)
parm:           RxAbsIntDelay:Receive Absolute Interrupt Delay
(array of int)
parm:           RxIntDelay:Receive Interrupt Delay (array of
int)
parm:           SmartPowerDownEnable:Enable PHY smart power
down (array of int)
parm:           TxAbsIntDelay:Transmit Absolute Interrupt Delay
(array of int)
parm:           TxIntDelay:Transmit Interrupt Delay (array of
int)
parm:           WriteProtectNVM:Write-protect NVM [WARNING:
disabling this can lead to corrupted NVM] (array of int)
```

2.3. Loading a Module

To load a kernel module, run **modprobe *module_name*** as **root**. For example, to load the **wacom** module, run:

```
# modprobe wacom
```

By default, **modprobe** attempts to load the module from **/lib/modules/*kernel_version*/kernel/drivers/**. In this directory, each type of module has its own subdirectory, such as **net/** and **scsi/**, for network and SCSI interface drivers respectively.

Some modules have dependencies, which are other kernel modules that must be loaded before the module in question can be loaded. The **modprobe** command always takes dependencies into account when performing operations. When you ask **modprobe** to load a specific kernel module, it first examines the dependencies of that module, if there are any, and loads them if they are not already loaded into the kernel. **modprobe** resolves dependencies recursively: it will load all dependencies of dependencies, and so on, if necessary, thus ensuring that all dependencies are always met. ###Because of this, there is no need to determine and resolve module dependencies manually.

You can use the **-v** (or **-#-verbose**) option to cause **modprobe** to display detailed information about what it is doing, which can include loading module dependencies.

Example 2.3. showing dependecies

You can load the **Fibre Channel over Ethernet** module verbosely by typing the following at a shell prompt:

```
# modprobe -v fcoe
insmod /lib/modules/3.10.0-
121.el7.x86_64/kernel/drivers/scsi/scsi_tgt.ko
insmod /lib/modules/3.10.0-
121.el7.x86_64/kernel/drivers/scsi/scsi_transport_fc.ko
insmod /lib/modules/3.10.0-
121.el7.x86_64/kernel/drivers/scsi/libfc/libfc.ko
insmod /lib/modules/3.10.0-
121.el7.x86_64/kernel/drivers/scsi/fcoe/libfcoe.ko
insmod /lib/modules/3.10.0-
121.el7.x86_64/kernel/drivers/scsi/fcoe/fcoe.ko
```

In this example, you can see that **modprobe** loaded the **scsi_tgt**, **scsi_transport_fc**, **libfc** and **libfcoe** modules as dependencies before finally loading **fcoe**. Also note that **modprobe** used the more primitive **insmod** command to insert the modules into the running kernel.

Important

Although the **insmod** command can also be used to load kernel modules, it does not resolve dependencies. Because of this, you should *always* load modules using **modprobe** instead.

2.4. Unloading a Module

You can unload a kernel module by running **modprobe -r *module_name*** as **root**. For example, assuming that the **wacom** module is already loaded into the kernel, you can unload it by running:

```
# modprobe -r wacom
```

However, this command will fail if a process is using:

- the **wacom** module;
- a module that **wacom** directly depends on, or;
- any module that **wacom**, through the dependency tree, depends on indirectly.

See Section 2.1, “Listing Currently-Loaded Modules” for more information about using **lsmod** to obtain the names of the modules which are preventing you from unloading a certain module.

Example 2.4. Unloading a kernel module

For example, if you want to unload the **firewire_ohci** module, your terminal session might look similar to this:

```
# modinfo -F depends firewire_ohci
firewire-core
# modinfo -F depends firewire_core
crc-itu-t
# modinfo -F depends crc-itu-t
```

You have figured out the dependency tree (which does not branch in this example) for the loaded Firewire modules: **firewire_ohci** depends on **firewire_core**, which itself depends on **crc-itu-t**.

You can unload **firewire_ohci** using the **modprobe -v -r *module_name*** command, where **-r** is short for **-#-remove** and **-v** for **-#-verbose**:

```
# modprobe -r -v firewire_ohci
rmmod firewire_ohci
rmmod firewire_core
rmmod crc_itu_t
```

The output shows that modules are unloaded in the reverse order that they are loaded, given that no processes depend on any of the modules being unloaded.

Important

Although the **rmmod** command can be used to unload kernel modules, it is recommended to use **modprobe -r** instead.

2.5. Setting Module Parameters

Like the kernel itself, modules can also take parameters that change their behavior. Most of the time, the default ones work well, but occasionally it is necessary or desirable to set custom parameters for a module. Because parameters cannot be dynamically set for a module that is already loaded into a running kernel, there are two different methods for setting them.

1. You can unload all dependencies of the module you want to set parameters for, unload the module using **modprobe -r**, and then load it with **modprobe** along with a list of customized parameters. This method is often used when the module does not have many dependencies,

or to test different combinations of parameters without making them persistent, and is the method covered in this section.

2. Alternatively, you can list the new parameters in an existing or newly created file in the **/etc/modprobe.d/** directory. This method makes the module parameters persistent by ensuring that they are set each time the module is loaded, such as after every reboot or **modprobe** command. This method is covered in Section 2.6, “Persistent Module Loading”, though the following information is a prerequisite.

Example 2.5. Supplying_optional_parameters_when_loading_a_kernel_module

You can use **modprobe** to load a kernel module with custom parameters using the following command line format:

```
# modprobe module_name [parameter=value]
```

When loading a module with custom parameters on the command line, be aware of the following:

- You can enter multiple parameters and values by separating them with spaces.
- Some module parameters expect a list of comma-separated values as their argument. When entering the list of values, do *not* insert a space after each comma, or **modprobe** will incorrectly interpret the values following spaces as additional parameters.
- The **modprobe** command silently succeeds with an exit status of **0** if:
  - it successfully loads the module, *or*
  - the module is *already* loaded into the kernel.

  Thus, you must ensure that the module is not already loaded before attempting to load it with custom parameters. The **modprobe** command does not automatically reload the module, or alert you that it is already loaded.

Here are the recommended steps for setting custom parameters and then loading a kernel module. This procedure illustrates the steps using the **e1000e** module, which is the network driver for Intel PRO/1000 network adapters, as an example:

Procedure 2.1. Loading a Kernel Module with Custom Parameters

1. First, ensure the module is not already loaded into the kernel:

   ```
   # lsmod |grep e1000e
   #
   ```

   Output would indicate that the module is already loaded into the kernel, in which case you must first unload it before proceeding. See Section 2.4, “Unloading a Module” for instructions on safely unloading it.

2.

Load the module and list all custom parameters after the module name. For example, if you wanted to load the Intel PRO/1000 network driver with the interrupt throttle rate set to 3000 interrupts per second for the first, second, and third instances of the driver, and turn on debug, you would run, as **root**:

```
# modprobe e1000e InterruptThrottleRate=3000,3000,3000 debug=1
```

This example illustrates passing multiple values to a single parameter by separating them with commas and omitting any spaces between them.

2.6. Persistent Module Loading

As shown in Example 2.1, “Listing information about a kernel module with lsmod”, many kernel modules are loaded automatically at boot time. You can specify additional modules to be loaded by the **systemd-modules-load.service** daemon by creating a ***program*.conf** file in the **/etc/modules-load.d/** directory, where *program* is any descriptive name of your choice. The files in **/etc/modules-load.d/** are text files that list the modules to be loaded, one per line.

Example 2.6. A Text File to Load a Module

To create a file to load the **virtio-net.ko** module, create a file **/etc/modules-load.d/virtio-net.conf** with the following content:

```
# Load virtio-net.ko at boot
virtio-net
```

See the **modules-load.d(5)** and **systemd-modules-load.service(8)** man pages for more information.

2.7. Installing Modules from a Driver Update Disk

Driver modules for hardware can be provided in the form of a *driver update disk* (DUD). The driver update disk, or an ISO image, is normally used at installation time to load and install any modules required for the hardware in use, and this process is described in the Red Hat Enterprise Linux 7 Installation Guide. However, if new driver modules are required after installation, use the following procedure. If you already have RPM files, go directly to step **5**.

Procedure 2.2. Installing New Modules from a Driver Update Disk

Follow this post-installation procedure to install new driver modules from a driver update disk (DUD).

1. Install the driver update disk.
2. Create a mount point and mount the DUD. For example, as **root**:

```
# mkdir /run/OEMDRV
# mount -r -t iso9660 /dev/sr0 /run/OEMDRV
```

3. View the contents of the DUD. For example:

```
# ls /run/OEMDRV/
rhdd3  rpms  src
```

4. Change into the directory relevant to the architecture of your system, contained within the **rpms/** directory, and list the contents. For example:

```
# cd /run/OEMDRV/rpms/x86_64/
# ls
kmod-bnx2x-1.710.51-3.el7_0.x86_64.rpm  kmod-bnx2x-firmware-
1.710.51-3.el7_0.x86_64.rpm  repodata
```

   In the above output the package version is **1.710.51** and the release is **3.el7_0**.

5. Install the RPM files simultaneously. For example:

```
# yum install kmod-bnx2x-1.710.51-3.el7_0.x86_64.rpm kmod-bnx2x-
firmware-1.710.51-3.el7_0.x86_64.rpm
Loaded plugins: product-id, subscription-manager
Examining kmod-bnx2x-1.710.51-3.el7_0.x86_64.rpm: kmod-bnx2x-
1.710.51-3.el7_0.x86_64
Marking kmod-bnx2x-1.710.51-3.el7_0.x86_64.rpm to be installed
Examining kmod-bnx2x-firmware-1.710.51-3.el7_0.x86_64.rpm: kmod-
bnx2x-firmware-1.710.51-3.el7_0.x86_64
Marking kmod-bnx2x-firmware-1.710.51-3.el7_0.x86_64.rpm to be
installed
Resolving Dependencies
-#-> Running transaction check
-#-#-> Package kmod-bnx2x.x86_64 0:1.710.51-3.el7_0 will be
installed
-#-#-> Package kmod-bnx2x-firmware.x86_64 0:1.710.51-3.el7_0 will
be installed
-#-> Finished Dependency Resolution

Dependencies Resolved

================================================================
=============
Package              Arch    Version           Repository
================================================================
=============
Installing:
kmod-bnx2x           x86_64  1.710.51-3.el7_0  /kmod-bnx2x-1.710.51-
3.el7_0.x8
kmod-bnx2x-firmware  x86_64  1.710.51-3.el7_0  /kmod-bnx2x-firmware-
1.710.51-3

Transaction Summary
================================================================
=============
Install  2 Packages

Total size: 1.6 M
Installed size: 1.6 M
Is this ok [y/d/N]:
```

6. Enter the following command to make **depmod** probe all modules and update the list of dependencies:

```
# depmod -a
```

7. Make a backup copy of the *initial RAM file system*, by entering the following command:

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

8. Rebuild the initial RAM file system:

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

9. To list the file contents of an initial RAM file system image created by dracut, enter a command as follows:

```
# lsinitrd /boot/initramfs-3.10.0-229.el7.x86_64.img
```

The output is very long, pipe the output through **less** or **grep** to find the module you are updating. For example:

```
# lsinitrd /boot/initramfs-3.10.0-229.el7.x86_64.img | grep bnx
drwxr-xr-x 2 root  root        0 Jun  9 11:25
usr/lib/firmware/bnx2x
-rw-r-#-r-#- 1 root  root   164392 Nov 25  2014
usr/lib/firmware/bnx2x/bnx2x-e1-7.10.51.0.fw
-rw-r-#-r-#- 1 root  root   173016 Nov 25  2014
usr/lib/firmware/bnx2x/bnx2x-e1h-7.10.51.0.fw
-rw-r-#-r-#- 1 root  root   321456 Nov 25  2014
usr/lib/firmware/bnx2x/bnx2x-e2-7.10.51.0.fw
drwxr-xr-x 2 root  root        0 Jun  9 11:25
usr/lib/modules/3.10.0-
229.el7.x86_64/kernel/drivers/net/ethernet/broadcom/bnx2x
-rw-r-#-r-#- 1 root  root  1034553 Jan 29 19:11
usr/lib/modules/3.10.0-
229.el7.x86_64/kernel/drivers/net/ethernet/broadcom/bnx2x/bnx2x.ko
```

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

If required, to view the current in-kernel driver, use the **modinfo *driver_name*** command as follows:

```
# modinfo bnx2x
filename:       /lib/modules/3.10.0-
229.el7.x86_64/kernel/drivers/net/ethernet/broadcom/bnx2x/bnx2x.ko
firmware:       bnx2x/bnx2x-e2-7.10.51.0.fw
firmware:       bnx2x/bnx2x-e1h-7.10.51.0.fw
firmware:       bnx2x/bnx2x-e1-7.10.51.0.fw
version:        1.710.51-0
license:        GPL
description:    Broadcom NetXtreme II
BCM57710/57711/57711E/57712/57712_MF/57800/57800_MF/57810/57810_MF/57840/578
40_MF Driver
author:         Eliezer Tamir
rhelversion:    7.1
```

2.8. Signing Kernel Modules for Secure Boot

Red Hat Enterprise Linux 7 includes support for the UEFI Secure Boot feature, which means that Red Hat Enterprise Linux 7 can be installed and run on systems where UEFI Secure Boot is enabled. Note that Red Hat Enterprise Linux 7 does not require the use of Secure Boot on UEFI systems.

When Secure Boot is enabled, the EFI operating system boot loaders, the Red Hat Enterprise Linux kernel, and all kernel modules must be signed with a private key and authenticated with the corresponding public key. The Red Hat Enterprise Linux 7 distribution includes signed boot loaders, signed kernels, and signed kernel modules. In addition, the signed first-stage boot loader and the signed kernel include embedded Red Hat public keys. These signed executable binaries and embedded keys enable Red Hat Enterprise Linux 7 to install, boot, and run with the Microsoft UEFI Secure Boot CA 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.

The information provided in the following sections describes steps necessary to enable you to self-sign privately built kernel modules for use with Red Hat Enterprise Linux 7 on UEFI-based systems where Secure Boot is enabled. These sections also provide an overview of available options for getting your public key onto the target system where you want to deploy your kernel module.

2.8.1. Prerequisites

In order to enable signing of externally built modules, the tools listed in the following table are required to be installed on the system.

Table 2.1. Required Tools

Tool	Provided by Package	Used on	Purpose
openssl	*openssl*	Build system	Generates public and private X.509 key pair
sign-file	*kernel-devel*	Build system	Perl script used to sign kernel modules
perl	*perl*	Build system	Perl interpreter used to run the signing script
mokutil	*mokutil*	Target system	Optional tool used to manually enroll the public key
keyctl	*keyutils*	Target system	Optional tool used to display public keys in the system key ring

Note

Note that 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.

2.8.2. Kernel Module Authentication

In Red Hat Enterprise Linux 7, when a kernel module is loaded, the module's signature is checked using the public X.509 keys on the kernel's system key ring, excluding those keys that are on the kernel's system black-list key ring.

2.8.2.1. Sources For Public Keys Used To Authenticate Kernel Modules

During boot, the kernel loads X.509 keys into the system key ring or the system black-list key ring from a set of persistent key stores as shown in Table 2.2, “Sources For System Key Rings”

Table 2.2. Sources For System Key Rings

Source of X.509 Keys	User Ability to Add Keys	UEFI Secure Boot State	Keys Loaded During Boot
Embedded in kernel	No	-	**.system_keyring**
UEFI Secure Boot "db"	Limited	Not enabled	No
		Enabled	**.system_keyring**
UEFI Secure Boot "dbx"	Limited	Not enabled	No
		Enabled	**.system_keyring**
Embedded in **shim.efi** boot loader	No	Not enabled	No
		Enabled	**.system_keyring**
Machine Owner Key (MOK) list	Yes	Not enabled	No
		Enabled	**.system_keyring**

Note that if the system is not UEFI-based or if UEFI Secure Boot is not enabled, then only the keys that are embedded in the kernel are loaded onto the system key ring and you have no ability to augment that set of keys without rebuilding the kernel. The system black list key ring is a list of X.509 keys which have been revoked. If your module is signed by a key on the black list then it will fail authentication even if your public key is in the system key ring.

You can display information about the keys on the system key rings using the **keyctl** utility. The following is abbreviated example output from a Red Hat Enterprise Linux 7 system where UEFI Secure Boot is not enabled.

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

The following is abbreviated example output from a Red Hat Enterprise Linux 7 system where UEFI Secure Boot is enabled.

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

```
...asymmetric: Red Hat Enterprise Linux kernel signing key:
4249689eefc77e95880b...
...asymmetric: Red Hat Enterprise Linux kpatch signing key:
4d38fd864ebe18c5f0b7...
```

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

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

2.8.2.2. Kernel Module Authentication Requirements

If UEFI Secure Boot is enabled or if the **module.sig_enforce** kernel parameter has been specified, then only signed kernel modules that are authenticated using a key on the system key ring can be successfully loaded, provided that the public key is not on the system black list key ring. If UEFI Secure Boot is disabled and if the **module.sig_enforce** kernel parameter has not been specified, then unsigned kernel modules and signed kernel modules without a public key can be successfully loaded. This is summarized in Table 2.3, “Kernel Module Authentication Requirements for Loading”.

Table 2.3. Kernel Module Authentication Requirements for Loading

Module Signed	Public Key Found and Signature Valid	UEFI Secure Boot State	module.sig_enforce	Module Load	Kernel Tainted
Unsigned	-	Not enabled	Not enabled	Succeeds	Yes
		Not enabled	Enabled	Fails	
		Enabled	-	Fails	-
Signed	No	Not enabled	Not enabled	Succeeds	Yes
		Not enabled	Enabled	Fails	-
		Enabled	-	Fails	-
Signed	Yes	Not enabled	Not enabled	Succeeds	No
		Not enabled	Enabled	Succeeds	No
		Enabled	-	Succeeds	No

Subsequent sections will describe how to generate a public and private X.509 key pair, how to use the private key to sign a kernel module, and how to enroll the public key into a source for the system key ring.

2.8.3. Generating a Public and Private X.509 Key Pair

You need to generate a public and private X.509 key pair that will be used to sign a kernel module after it has been built. The corresponding public key will be used to authenticate the kernel module when it is loaded.

1. The **openssl** tool can be used to generate a key pair that satisfies the requirements for kernel module signing in Red Hat Enterprise Linux 7. Some of the parameters for this key generation request are best specified with a configuration file; follow the example below to create your own configuration file.

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

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

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

2. After you have created the configuration file, you can create an X.509 public and private key pair. The public key will be written to the ***public_key*.der** file and the private key will be written to the ***private_key*.priv** file.

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

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

Warning

Take proper care to guard the contents of your private key. In the wrong hands, the key could be used to compromise any system which has your public key.

2.8.4. Enrolling Public Key on Target System

When Red Hat Enterprise Linux 7 boots on a UEFI-based system with Secure Boot enabled, all keys that are in the Secure Boot db key database, but not in the dbx database of revoked keys, are loaded onto the system keyring by the kernel. The system keyring is used to authenticate kernel modules.

2.8.4.1. Factory Firmware Image Including Public Key

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

2.8.4.2. Executable Key Enrollment Image Adding Public Key

It is possible to add a key to an existing populated and active Secure Boot key database. This can be done by writing and providing an EFI executable *enrollment* image. Such an enrollment image contains a properly formed request to append a key to the Secure Boot key database. This request must include data that is properly signed by the private key that corresponds to a public key that is already in the system's Secure Boot Key Exchange Key (KEK) database. Additionally, this EFI image must be signed by a private key that corresponds to a public key that is already in the key database.

It is also possible to write an enrollment image that runs under Red Hat Enterprise Linux 7. However, the Red Hat Enterprise Linux 7 image must be properly signed by a private key that corresponds to a public key that is already in the KEK database.

The construction of either type of key enrollment images requires assistance from the platform vendor.

2.8.4.3. System Administrator Manually Adding Public Key to the MOK List

The Machine Owner Key (MOK) facility is a feature that is supported by Red Hat Enterprise Linux 7 and can be used to augment the UEFI Secure Boot key database. When Red Hat Enterprise Linux 7 boots on a UEFI-enabled system with Secure Boot enabled, the keys on the MOK list are also added to the system keyring in addition to the keys from the key database. The MOK list keys are also stored persistently and securely in the same fashion as the Secure Boot key database keys, but these are two separate facilities. The MOK facility is supported by shim.efi, MokManager.efi, grubx64.efi, and the Red Hat Enterprise Linux 7 **mokutil** utility.

The major capability provided by the MOK facility is the ability to add public keys to the MOK list without needing to have the key chain back to another key that is already in the KEK database. However, enrolling a MOK key requires manual interaction by a *physically present* user at the UEFI system console on each target system. Nevertheless, the MOK facility provides an excellent method for testing newly generated key pairs and testing kernel modules signed with them.

Follow these steps to add your public key to the MOK list:

1. Request addition of your public key to the MOK list using a Red Hat Enterprise Linux 7 userspace utility:

   ```
   # 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.

3. 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. You will need to enter the password you previously associated with this request and confirm the enrollment. Your public key is added to the MOK list, which is persistent.

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

2.8.5. Signing Kernel Module with the Private Key

There are no extra steps required to prepare your kernel module for signing. You build your kernel module normally. Assuming an appropriate Makefile and corresponding sources, follow these steps to build your module and sign it:

1. Build your **my_module.ko** module the standard way:

```
# make -C /usr/src/kernels/$(uname -r) M=$PWD modules
```

2. Sign your kernel module with your private key. This is done with a Perl script. Note that the script requires that you provide both the files that contain your private and the public key as well as the kernel module file that you want to sign.

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

Your kernel module is in ELF image format and this script computes and appends the signature directly to the ELF image in your **my_module.ko** file. The **modinfo** utility can be used to display information about the kernel module's signature, if it is present. For information on using the utility, see Section 2.2, “Displaying Information About a Module”.

Note that this appended signature is not contained in an ELF image section and is not a formal part of the ELF image. Therefore, tools 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.

2.8.6. Loading Signed Kernel Module

Once your public key is enrolled and is in the system keyring, the normal kernel module loading mechanisms will work transparently. In the following example, you will use **mokutil** to add your public key to the MOK list and you will manually load your kernel module with **modprobe**.

1. Optionally, you can verify that your kernel module will not load before you have enrolled your public key. First, verify what keys have been added to the system key ring on the current boot by running the **keyctl list %:.system_keyring** as root. Since your public key has not been enrolled yet, it should not be displayed in the output of the command.

2. Request enrollment of your public key.

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

3. Reboot, and complete the enrollment at the UEFI console.

```
# reboot
```

4. After the system reboots, verify the keys on the system key ring again.

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

5. You should now be able to load your kernel module successfully.

```
# modprobe -v my_module
insmod /lib/modules/3.10.0-123.el7.x86_64/extra/my_module.ko
# lsmod | grep my_module
my_module 12425 0
```

Chapter 3. Manually Upgrading the Kernel

The Red Hat Enterprise Linux kernel is custom-built by the Red Hat Enterprise Linux kernel team to ensure its integrity and compatibility with supported hardware. Before Red Hat releases a kernel, it must first pass a rigorous set of quality assurance tests.

Red Hat Enterprise Linux kernels are packaged in the RPM format so that they are easy to upgrade and verify using the **Yum** or **PackageKit** package managers. **PackageKit** automatically queries the Red Hat Content Delivery Network servers and informs you of packages with available updates, including kernel packages.

This chapter is therefore *only* useful for users who need to manually update a kernel package using the **rpm** command instead of **yum**.

Warning

Whenever possible, use either the **Yum** or **PackageKit** package manager to install a new kernel because they always *install* a new kernel instead of replacing the current one, which could potentially leave your system unable to boot.

Warning

Custom kernels are *not* supported by Red Hat. However, guidance can be obtained from the knowledgebase document https://access.redhat.com/solutions/25039.

For more information on installing kernel packages with **yum**, see https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/System_Administrators_Guide/ch-yum.html#sec-Updating_Packages.

For information on Red Hat Content Delivery Network, see https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/System_Administrators_Guide/chap-Subscription_and_Support-Registering_a_System_and_Managing_Subscriptions.html.

3.1. Overview of Kernel Packages

Red Hat Enterprise Linux contains the following kernel packages:

- *kernel* — Contains the kernel for single-core, multi-core, and multi-processor systems.
- *kernel-debug* — Contains a kernel with numerous debugging options enabled for kernel diagnosis, at the expense of reduced performance.
- *kernel-devel* — Contains the kernel headers and makefiles sufficient to build modules against the *kernel* package.
- *kernel-debug-devel* — Contains the development version of the kernel with numerous debugging options enabled for kernel diagnosis, at the expense of reduced performance.

- *kernel-doc* — Documentation files from the kernel source. Various portions of the Linux kernel and the device drivers shipped with it are documented in these files. Installation of this package provides a reference to the options that can be passed to Linux kernel modules at load time.

  By default, these files are placed in the **/usr/share/doc/kernel-doc-*kernel_version*/** directory.

- *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.

- *linux-firmware* — Contains all of the firmware files that are required by various devices to operate.

- *perf* — This package contains the **perf** tool, which enables performance monitoring of the Linux kernel.

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

- *kernel-tools* — Contains tools for manipulating the Linux kernel and supporting documentation.

3.2. Preparing to Upgrade

Before upgrading the kernel, it is recommended that you take some precautionary steps.

First, ensure that working boot media exists for the system in case a problem occurs. If the boot loader is not configured properly to boot the new kernel, you can use this media to boot into Red Hat Enterprise Linux.

USB media often comes in the form of flash devices sometimes called *pen drives*, *thumb disks*, or *keys*, or as an externally-connected hard disk device. Almost all media of this type is formatted as a **VFAT** file system. You can create bootable USB media on media formatted as **ext2**, **ext3**, **ext4**, or **VFAT**.

You can transfer a distribution image file or a minimal boot media image file to USB media. Make sure that sufficient free space is available on the device. Around **4 GB** is required for a distribution DVD image, around **700 MB** for a distribution CD image, or around **10 MB** for a minimal boot media image.

You must have a copy of the **boot.iso** file from a Red Hat Enterprise Linux installation DVD, or installation CD-ROM #1, and you need a USB storage device formatted with the **VFAT** file system and around **16 MB** of free space.

For more information on using USB storage devices please review https://access.redhat.com/solutions/624423 How to format a USB key and https://access.redhat.com/solutions/39373 How to manually mount a USB flash drive in a non-graphical environment.

The following procedure will not affect existing files on the USB storage device unless they have the same path names as the files that you copy onto it. To create USB boot media, perform the following commands as the **root** user:

1. Install the *syslinux* package if it is not installed on your system. To do so, as root, run the **yum install *syslinux*** command.

2. Install the **SYSLINUX** bootloader on the USB storage device:

```
# syslinux /dev/sdX1
```

...where *sdX* is the device name.

3. Create mount points for **boot.iso** and the USB storage device:

```
# mkdir /mnt/isoboot /mnt/diskboot
```

4. Mount **boot.iso**:

```
# mount -o loop boot.iso /mnt/isoboot
```

5. Mount the USB storage device:

```
# mount /dev/sdX1 /mnt/diskboot
```

6. Copy the **ISOLINUX** files from the **boot.iso** to the USB storage device:

```
# cp /mnt/isoboot/isolinux/* /mnt/diskboot
```

7. Use the **isolinux.cfg** file from **boot.iso** as the **syslinux.cfg** file for the USB device:

```
# grep -v local /mnt/isoboot/isolinux/isolinux.cfg >
/mnt/diskboot/syslinux.cfg
```

8. Unmount **boot.iso** and the USB storage device:

```
# umount /mnt/isoboot /mnt/diskboot
```

9. You should reboot the machine with the boot media and verify that you are able to boot with it before continuing.

Alternatively, on systems with a floppy drive, you can create a boot diskette by installing the *mkbootdisk* package and running the **mkbootdisk** command as **root**. See **man mkbootdisk** man page after installing the package for usage information.

To determine which kernel packages are installed, execute the command **yum list installed "kernel-*"** at a shell prompt. The output will comprise some or all of the following packages, depending on the system's architecture, and the version numbers might differ:

```
# yum list installed "kernel-*"
kernel.x86_64                 3.10.0-54.0.1.el7          @rhel7/7.0
kernel-devel.x86_64           3.10.0-54.0.1.el7          @rhel7
kernel-headers.x86_64         3.10.0-54.0.1.el7          @rhel7/7.0
```

From the output, determine which packages need to be downloaded for the kernel upgrade. For a single processor system, the only required package is the *kernel* package. See Section 3.1, “Overview of Kernel Packages” for descriptions of the different packages.

3.3. Downloading the Upgraded Kernel

There are several ways to determine if an updated kernel is available for the system.

- Security Errata — See https://access.redhat.com/site/security/updates/active/ for information on security errata, including kernel upgrades that fix security issues.

- The Red Hat Content Delivery Network — For a system subscribed to the Red Hat Content Delivery Network, the **yum** package manager can download the latest kernel and upgrade the kernel on the system. The **Dracut** utility will create an initial RAM disk image if needed, and configure the boot loader to boot the new kernel. For more information on installing packages from the Red Hat Content Delivery Network, see https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/System_Administrators_Guide/ch-yum.html. For more information on subscribing a system to the Red Hat Content Delivery Network, see https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/System_Administrators_Guide/chap-Subscription_and_Support-Registering_a_System_and_Managing_Subscriptions.html.

If **yum** was used to download and install the updated kernel from the Red Hat Network, follow the instructions in Section 3.5, “Verifying the Initial RAM Disk Image” and Section 3.6, “Verifying the Boot Loader” only, *do not* change the kernel to boot by default. Red Hat Network automatically changes the default kernel to the latest version. To install the kernel manually, continue to Section 3.4, “Performing the Upgrade”.

3.4. Performing the Upgrade

After retrieving all of the necessary packages, it is time to upgrade the existing kernel.

Important

It is strongly recommended that you keep the old kernel in case there are problems with the new kernel.

At a shell prompt, change to the directory that contains the kernel RPM packages. Use **-i** argument with the **rpm** command to keep the old kernel. Do *not* use the **-U** option, since it overwrites the currently installed kernel, which creates boot loader problems. For example:

```
# rpm -ivh kernel-kernel_version.arch.rpm
```

The next step is to verify that the initial RAM disk image has been created. See Section 3.5, “Verifying the Initial RAM Disk Image” for details.

3.5. Verifying the Initial RAM Disk Image

The job of the initial RAM disk image is to preload the block device modules, such as for IDE, SCSI or RAID, so that the root file system, on which those modules normally reside, can then be accessed and mounted. On Red Hat Enterprise Linux 7 systems, whenever a new kernel is installed using either the **Yum**, **PackageKit**, or **RPM** package manager, the **Dracut** utility is always called by the installation scripts to create an *initramfs* (initial RAM disk image).

If you make changes to the kernel attributes by modifying the **/etc/sysctl.conf** file or another **sysctl** configuration file, and if the changed settings are used early in the boot process, then rebuilding the Initial RAM Disk Image by running the **dracut -f** command might be necessary. An example is if you have made changes related to networking and are booting from network-attached storage.

On all architectures other than IBM eServer System i (see Section 3.5, “Verifying the Initial RAM Disk Image and Kernel on IBM eServer System i”), you can create an **initramfs** by running the **dracut** command. However, you usually don't need to create an **initramfs** manually: this step is

automatically performed if the kernel and its associated packages are installed or upgraded from RPM packages distributed by Red Hat.

You can verify that an **initramfs** corresponding to your current kernel version exists and is specified correctly in the **grub.cfg** configuration file by following this procedure:

Procedure 3.1. Verifying the Initial RAM Disk Image

1. As **root**, list the contents in the **/boot** directory and find the kernel (**vmlinuz-*kernel_version***) and **initramfs-*kernel_version*** with the latest (most recent) version number:

   Example 3.1. Ensuring that the kernel and initramfs versions match

   ```
   # ls /boot
   config-3.10.0-67.el7.x86_64
   config-3.10.0-78.el7.x86_64
   efi
   grub
   grub2
   initramfs-0-rescue-07f43f20a54c4ce8ada8b70d33fd001c.img
   initramfs-3.10.0-67.el7.x86_64.img
   initramfs-3.10.0-67.el7.x86_64kdump.img
   initramfs-3.10.0-78.el7.x86_64.img
   initramfs-3.10.0-78.el7.x86_64kdump.img
   initrd-plymouth.img
   symvers-3.10.0-67.el7.x86_64.gz
   symvers-3.10.0-78.el7.x86_64.gz
   System.map-3.10.0-67.el7.x86_64
   System.map-3.10.0-78.el7.x86_64
   vmlinuz-0-rescue-07f43f20a54c4ce8ada8b70d33fd001c
   vmlinuz-3.10.0-67.el7.x86_64
   vmlinuz-3.10.0-78.el7.x86_64
   ```

   Example 3.1, “Ensuring that the kernel and initramfs versions match” shows that:

   - we have three kernels installed (or, more correctly, three kernel files are present in the **/boot** directory),
   - the latest kernel is **vmlinuz-3.10.0-78.el7.x86_64**, and
   - an **initramfs** file matching our kernel version, **initramfs-3.10.0-78.el7.x86_64kdump.img**, also exists.

   Important

   In the **/boot** directory you might find several **initramfs-*kernel_version*kdump.img** files. These are special files created by the **Kdump** mechanism for kernel debugging purposes, are not used to boot the system, and can safely be ignored. For more information on **kdump**, see the Red Hat Enterprise Linux 7 Kernel Crash Dump Guide.

2. If your **initramfs-*kernel_version*** file does not match the version of the latest kernel in

the **/boot** directory, or, in certain other situations, you might need to generate an **initramfs** file with the **Dracut** utility. Simply invoking **dracut** as **root** without options causes it to generate an **initramfs** file in **/boot** for the latest kernel present in that directory:

```
# dracut
```

You must use the **-f**, **--force** option if you want **dracut** to overwrite an existing **initramfs** (for example, if your **initramfs** has become corrupt). Otherwise **dracut** will refuse to overwrite the existing **initramfs** file:

```
# dracut
          Will not override existing initramfs (/boot/initramfs-
3.10.0-78.el7.x86_64.img) without --force
```

You can create an initramfs in the current directory by calling **dracut *initramfs_name kernel_version*** :

```
# dracut "initramfs-$(uname -r).img" $(uname -r)
```

If you need to specify specific kernel modules to be preloaded, add the names of those modules (minus any file name suffixes such as **.ko**) inside the parentheses of the **add_dracutmodules+="*module* [*more_modules*]"** directive of the **/etc/dracut.conf** configuration file. You can list the file contents of an **initramfs** image file created by dracut by using the **lsinitrd *initramfs_file*** command:

```
# lsinitrd /boot/initramfs-3.10.0-78.el7.x86_64.img
Image: /boot/initramfs-3.10.0-78.el7.x86_64.img: 11M
================================================================
======
dracut-033-68.el7
================================================================
======

drwxr-xr-x  12 root     root            0 Feb  5 06:35 .
drwxr-xr-x   2 root     root            0 Feb  5 06:35 proc
lrwxrwxrwx   1 root     root           24 Feb  5 06:35 init ->
/usr/lib/systemd/systemd
drwxr-xr-x  10 root     root            0 Feb  5 06:35 etc
drwxr-xr-x   2 root     root            0 Feb  5 06:35
usr/lib/modprobe.d
[output truncated]
```

See **man dracut** and **man dracut.conf** for more information on options and usage.

3. Examine the **/boot/grub2/grub.cfg** configuration file to ensure that an **initramfs-*kernel_version*.img** file exists for the kernel version you are booting. For example:

```
# grep initramfs /boot/grub2/grub.cfg
initrd16 /initramfs-3.10.0-123.el7.x86_64.img
initrd16 /initramfs-0-rescue-6d547dbfd01c46f6a4c1baa8c4743f57.img
```

See Section 3.6, “Verifying the Boot Loader” for more information.

Verifying the Initial RAM Disk Image and Kernel on IBM eServer System i

On IBM eServer System i machines, the initial RAM disk and kernel files are combined into a single file, which is created with the **addRamDisk** command. This step is performed automatically if the kernel and its associated packages are installed or upgraded from the RPM packages distributed by Red Hat; thus, it does not need to be executed manually. To verify that it was created, run the following command as **root** to make sure the **/boot/vmlinitrd-*kernel_version*** file already exists:

```
# ls -l /boot/
```

The *kernel_version* should match the version of the kernel just installed.

Reversing the Changes Made to the Initial RAM Disk Image

In some cases, for example, if you misconfigure the system and it no longer boots, you may need to reverse the changes made to the Initial RAM Disk Image by following this procedure:

Procedure 3.2. Reversing Changes Made to the Initial RAM Disk Image

1. Reboot the system choosing the rescue kernel in the GRUB menu.
2. Change the incorrect setting that caused the **initramfs** to malfunction.
3. Recreate the **initramfs** with the correct settings by running the following command as root:

```
# dracut --kver kernel_version --force
```

The above procedure might be useful if, for example, you incorrectly set the **vm.nr_hugepages** in the **sysctl.conf** file. Because the **sysctl.conf** file is included in **initramfs**, the new **vm.nr_hugepages** setting gets applied in **initramfs** and causes rebuilding of the **initramfs**. However, because the setting is incorrect, the new **initramfs** is broken and the newly built kernel does not boot, which necessitates correcting the setting using the above procedure.

Listing the Contents of the Initial RAM Disk Image

To list the files that are included in the **initramfs**, run the following command as root:

```
# lsinitrd
```

To only list files in the **/etc** directory, use the following command:

```
# lsinitrd | grep etc/
```

To output the contents of a specific file stored in the **initramfs** for the current kernel, use the **-f** option:

```
# lsinitrd -f filename
```

For example, to output the contents of **sysctl.conf**, use the following command:

```
# lsinitrd -f /etc/sysctl.conf
```

To specify a kernel version, use the **--kver** option:

```
# lsinitrd --kver kernel_version -f /etc/sysctl.conf
```

For example, to list the information about kernel version 3.10.0-327.10.1.el7.x86_64, use the following command:

```
# lsinitrd --kver 3.10.0-327.10.1.el7.x86_64 -f /etc/sysctl.conf
```

3.6. Verifying the Boot Loader

When you install a kernel using **rpm**, the kernel package creates an entry in the boot loader configuration file for that new kernel. However, **rpm** does *not* configure the new kernel to boot as the default kernel. You must do this manually when installing a new kernel with **rpm**.

It is always recommended to double-check the boot loader configuration file after installing a new kernel with **rpm** to ensure that the configuration is correct. Otherwise, the system might not be able to boot into Red Hat Enterprise Linux properly. If this happens, boot the system with the boot media created earlier and re-configure the boot loader.

Appendix A. Revision History

Revision 0.1-2	**Mon Jul 31 2017**	**Mark Flitter**
Document version for 7.4 GA publication.		
Revision 0.1-0	**Thu Apr 20 2017**	**Mark Flitter**
Initial build for review		