- Fastest way, which provides lower quality random data:

```
badblocks -c 10240 -s -w -t random -v <device>
```

C.4.3. Format the Device as a dm-crypt/LUKS Encrypted Device

WARNING

The command below will destroy any existing data on the device.

```
cryptsetup luksFormat <device>
```

NOTE

For more information, read the **cryptsetup(8)** man page.

After supplying the passphrase twice the device will be formatted for use. To verify, use the following command:

```
cryptsetup isLuks <device> && echo Success
```

To see a summary of the encryption information for the device, use the following command:

```
cryptsetup luksDump <device>
```

C.4.4. Create a Mapping to Allow Access to the Device's Decrypted Contents

To access the device's decrypted contents, a mapping must be established using the kernel **device-mapper**.

It is useful to choose a meaningful name for this mapping. LUKS provides a UUID (Universally Unique Identifier) for each device. This, unlike the device name (eg: **/dev/sda3**), is guaranteed to remain constant as long as the LUKS header remains intact. To find a LUKS device's UUID, run the following command:

```
cryptsetup luksUUID <device>
```

An example of a reliable, informative and unique mapping name would be **luks-<uuid>**, where <uuid> is replaced with the device's LUKS UUID (eg: **luks-50ec957a-5b5a-47ee-85e6-f8085bbc97a8**). This naming convention might seem unwieldy but is it not necessary to type it often.

```
cryptsetup luksOpen <device> <name>
```

There should now be a device node, **/dev/mapper/<name>**, which represents the decrypted device. This block device can be read from and written to like any other unencrypted block device.

To see some information about the mapped device, use the following command:

```
dmsetup info <name>
```

NOTE

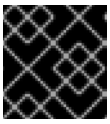
For more information, read the **dmsetup(8)** man page.

C.4.5. Create File Systems on the Mapped Device or Continue to Build Complex Storage Structures Using the Mapped Device

Use the mapped device node (**/dev/mapper/<name>**) as any other block device. To create an **ext2** filesystem on the mapped device, use the following command:

```
mke2fs /dev/mapper/<name>
```

To mount this filesystem on **/mnt/test**, use the following command:



IMPORTANT

The directory **/mnt/test** must exist before executing this command.

```
mount /dev/mapper/<name> /mnt/test
```

C.4.6. Add the Mapping Information to /etc/crypttab

In order for the system to set up a mapping for the device, an entry must be present in the **/etc/crypttab** file. If the file doesn't exist, create it and change the owner and group to root (**root:root**) and change the mode to **0744**. Add a line to the file with the following format:

```
<name>  <device>  none
```

The <device> field should be given in the form "UUID=<luks_uuid>", where <luks_uuid> is the LUKS uuid as given by the command **cryptsetup luksUUID <device>**. This ensures the correct device will be identified and used even if the device node (eg: **/dev/sda5**) changes.

NOTE

For details on the format of the **/etc/crypttab** file, read the **crypttab(5)** man page.

C.4.7. Add an Entry to /etc/fstab

Add an entry to /etc/fstab. This is only necessary if you want to establish a persistent association between the device and a mountpoint. Use the decrypted device, **/dev/mapper/<name>** in the **/etc/fstab** file.

In many cases it is desirable to list devices in **/etc/fstab** by UUID or by a filesystem label. The main purpose of this is to provide a constant identifier in the event that the device name (eg: **/dev/sda4**) changes. LUKS device names in the form of **/dev/mapper/luks-<luks_uuid>** are based only on the

device's LUKS UUID, and are therefore guaranteed to remain constant. This fact makes them suitable for use in **/etc/fstab**.

NOTE

For details on the format of the **/etc/fstab** file, read the **fstab(5)** man page.

C.5. COMMON POST-INSTALLATION TASKS

The following sections are about common post-installation tasks.

C.5.1. Set a Randomly Generated Key as an Additional Way to Access an Encrypted Block Device

The following sections are about generating keys and adding keys.

C.5.1.1. Generate a Key

This will generate a 256-bit key in the file **$HOME/keyfile**.

```
dd if=/dev/urandom of=$HOME/keyfile bs=32 count=1
chmod 600 $HOME/keyfile
```

C.5.1.2. Add the Key to an Available Keyslot on the Encrypted Device

```
cryptsetup luksAddKey <device> ~/keyfile
```

C.5.2. Add a New Passphrase to an Existing Device

```
cryptsetup luksAddKey <device>
```

After being prompted for any one of the existing passphrases for authentication, you will be prompted to enter the new passphrase.

C.5.3. Remove a Passphrase or Key from a Device

```
cryptsetup luksRemoveKey <device>
```

You will be prompted for the passphrase you wish to remove and then for any one of the remaining passphrases for authentication.

APPENDIX D. UNDERSTANDING LVM

LVM (Logical Volume Management) partitions provide a number of advantages over standard partitions. LVM partitions are formatted as *physical volumes*. One or more physical volumes are combined to form a *volume group*. Each volume group's total storage is then divided into one or more *logical volumes*. The logical volumes function much like standard partitions. They have a file system type, such as **ext4**, and a mount point.

NOTE

On most architectures, the boot loader cannot read LVM volumes. You must make a standard, non-LVM disk partition for your **/boot** partition.

However, on System z, the **zipl** boot loader supports **/boot** on LVM logical volumes with linear mapping.

To understand LVM better, imagine the physical volume as a pile of *blocks*. A block is simply a storage unit used to store data. Several piles of blocks can be combined to make a much larger pile, just as physical volumes are combined to make a volume group. The resulting pile can be subdivided into several smaller piles of arbitrary size, just as a volume group is allocated to several logical volumes.

An administrator may grow or shrink logical volumes without destroying data, unlike standard disk partitions. If the physical volumes in a volume group are on separate drives or RAID arrays then administrators may also spread a logical volume across the storage devices.

You may lose data if you shrink a logical volume to a smaller capacity than the data on the volume requires. To ensure maximum flexibility, create logical volumes to meet your current needs, and leave excess storage capacity unallocated. You may safely grow logical volumes to use unallocated space, as your needs dictate.

NOTE

By default, the installation process creates / and swap partitions within LVM volumes, with a separate **/boot** partition.

APPENDIX E. THE GRUB BOOT LOADER

When a computer running Linux is turned on, the operating system is loaded into memory by a special program called a *boot loader*. A boot loader usually exists on the system's primary hard drive (or other media device) and has the sole responsibility of loading the Linux kernel with its required files or (in some cases) other operating systems into memory.

E.1. BOOT LOADERS AND SYSTEM ARCHITECTURE

Each architecture capable of running Red Hat Enterprise Linux uses a different boot loader. The following table lists the boot loaders available for each architecture:

Table E.1. Boot Loaders by Architecture

Architecture	Boot Loaders
AMD AMD64	GRUB
IBM Power Systems	yaboot
IBM System z	z/IPL
x86	GRUB

This appendix discusses commands and configuration options for the GRUB boot loader included with Red Hat Enterprise Linux for the x86 architecture.

IMPORTANT

The /**boot** and / (root) partition in Red Hat Enterprise Linux 6.9 can only use the ext2, ext3, and ext4 (recommended) file systems. You cannot use any other file system for this partition, such as Btrfs, XFS, or VFAT. Other partitions, such as /**home**, can use any supported file system, including Btrfs and XFS (if available). See the following article on the Red Hat Customer Portal for additional information: https://access.redhat.com/solutions/667273.

E.2. GRUB

The *GNU GRand Unified Boot loader* (GRUB) is a program which enables the selection of the installed operating system or kernel to be loaded at system boot time. It also allows the user to pass arguments to the kernel.

E.2.1. GRUB and the Boot Process on BIOS-based x86 Systems

This section describes the specific role GRUB plays when booting a BIOS-based x86 system. For a look at the overall boot process, refer to Section F.2, “A Detailed Look at the Boot Process” .

GRUB loads itself into memory in the following stages:

1. *The Stage 1 or primary boot loader is read into memory by the BIOS from the MBR* [16]. The primary boot loader exists on less than 512 bytes of disk space within the MBR and is capable of loading either the Stage 1.5 or Stage 2 boot loader.

   BIOS cannot read partition tables or file systems. It initializes the hardware, reads the MBR, then depends entirely on the stage 1 bootloader to continue the boot process.

2. *The Stage 1.5 boot loader is read into memory by the Stage 1 boot loader, if necessary.* Some hardware requires an intermediate step to get to the Stage 2 boot loader. This is sometimes true when the **/boot/** partition is above the 1024 cylinder head of the hard drive or when using LBA mode. The Stage 1.5 boot loader is found either on the **/boot/** partition or on a small part of the MBR and the **/boot/** partition.

3. *The Stage 2 or secondary boot loader is read into memory.* The secondary boot loader displays the GRUB menu and command environment. This interface allows the user to select which kernel or operating system to boot, pass arguments to the kernel, or look at system parameters.

4. *The secondary boot loader reads the operating system or kernel as well as the contents of **/boot/sysroot/** into memory.* Once GRUB determines which operating system or kernel to start, it loads it into memory and transfers control of the machine to that operating system.

The method used to boot Linux is called *direct loading* because the boot loader loads the operating system directly. There is no intermediary between the boot loader and the kernel.

The boot process used by other operating systems may differ. For example, the Microsoft Windows operating system, as well as other operating systems, are loaded using *chain loading*. Under this method, the MBR points to the first sector of the partition holding the operating system, where it finds the files necessary to actually boot that operating system.

GRUB supports both direct and chain loading boot methods, allowing it to boot almost any operating system.

WARNING

During installation, Microsoft's DOS and Windows installation programs completely overwrite the MBR, destroying any existing boot loaders. If creating a dual-boot system, it is best to install the Microsoft operating system first.

E.2.2. GRUB and the Boot Process on UEFI-based x86 Systems

This section describes the specific role GRUB plays when booting a UEFI-based x86 system. For a look at the overall boot process, refer to Section F.2, “A Detailed Look at the Boot Process”.

GRUB loads itself into memory in the following stages:

1. The UEFI-based platform reads the partition table on the system storage and mounts the *EFI System Partition* (ESP), a VFAT partition labeled with a particular *globally unique identifier* (GUID). The ESP contains EFI applications such as bootloaders and utility software, stored in directories specific to software vendors. Viewed from within the Red Hat Enterprise Linux 6.9 file system, the ESP is **/boot/efi/**, and EFI software provided by Red Hat is stored in **/boot/efi/EFI/redhat/**.

2. The **/boot/efi/EFI/redhat/** directory contains **grub.efi**, a version of GRUB compiled for the EFI firmware architecture as an EFI application. In the simplest case, the EFI boot manager selects **grub.efi** as the default bootloader and reads it into memory.

   If the ESP contains other EFI applications, the EFI boot manager might prompt you to select an application to run, rather than load **grub.efi** automatically.

3. GRUB determines which operating system or kernel to start, loads it into memory, and transfers control of the machine to that operating system.

Because each vendor maintains its own directory of applications in the ESP, chain loading is not normally necessary on UEFI-based systems. The EFI boot manager can load any of the operating system bootloaders that are present in the ESP.

E.2.3. Features of GRUB

GRUB contains several features that make it preferable to other boot loaders available for the x86 architecture. Below is a partial list of some of the more important features:

- *GRUB provides a true command-based, pre-OS environment on x86 machines.* This feature affords the user maximum flexibility in loading operating systems with specified options or gathering information about the system. For years, many non-x86 architectures have employed pre-OS environments that allow system booting from a command line.

- *GRUB supports Logical Block Addressing (LBA) mode.* LBA places the addressing conversion used to find files in the hard drive's firmware, and is used on many IDE and all SCSI hard devices. Before LBA, boot loaders could encounter the 1024-cylinder BIOS limitation, where the BIOS could not find a file after the 1024 cylinder head of the disk. LBA support allows GRUB to boot operating systems from partitions beyond the 1024-cylinder limit, so long as the system BIOS supports LBA mode. Most modern BIOS revisions support LBA mode.

- *GRUB can read ext2 partitions.* This functionality allows GRUB to access its configuration file, **/boot/grub/grub.conf**, every time the system boots, eliminating the need for the user to write a new version of the first stage boot loader to the MBR when configuration changes are made. The only time a user needs to reinstall GRUB on the MBR is if the physical location of the **/boot/** partition is moved on the disk.

E.3. INSTALLING GRUB

In a vast majority of cases, **GRUB** is installed and configured by default during the installation of Red Hat Enterprise Linux. However, if for some reason **GRUB** is not installed, or if you need to install it again, it is possible to install grub manually.

On systems without UEFI firmware, a valid GRUB configuration file must be present at **/boot/grub/grub.conf**. You can use the **grub-install** script (part of the grub package) to install GRUB. For example:

```
# grub-install disk
```

Replace *disk* with the device name of your system's boot drive such as **/dev/sda**.

On systems with UEFI firmware, a valid GRUB configuration file must be present at **/boot/efi/EFI/redhat/grub.conf**. An image of GRUB's first-stage boot loader is available on the EFI System Partitition in the directory **EFI/redhat/** with the filename **grubx64.efi**, and you can use the **efibootmgr** command to install this image into your system's EFI System Partition. For example:

```
# efibootmgr -c -d disk -p partition_number -l /EFI/redhat/grubx64.efi -L "grub_uefi"
```

Replace *disk* with the name of the device containing the EFI System Partition (such as **/dev/sda**) and *partition_number* with the partition number of your EFI System Partition (the default value is 1, meaning the first partition on the disk).

IMPORTANT

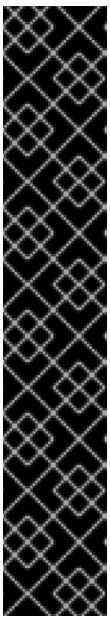
The grub package does not automatically update the system boot loader when the package is updated using **Yum** or **RPM**. Therefore, updating the package will not automatically update the actual boot loader on your system. Use the **grub-install** command manually every time after the package is updated.

For additional information about installing **GRUB**, see the GNU GRUB Manual and the **grub-install(8)** man page. For information about the EFI System Partition, see Section 9.18.1, "Advanced Boot Loader Configuration". For information about the **efibootmgr** tool, see the **efibootmgr(8)** man page.

E.4. TROUBLESHOOTING GRUB

In most cases, **GRUB** will be installed and configured during the initial installation process, unless you used a Kickstart file and specifically disabled this behavior. The installed system should therefore be prepared to boot into your desktop environment or a command line, depending on your package selection. However, in certain cases it is possible that the system's **GRUB** configuration becomes corrupted and the system will no longer be able to boot. This section describes how to fix such problems.

When troubleshooting **GRUB**, keep in mind thatthe grub package does not automatically update the system boot loader when the package is updated using **Yum** or **RPM**. Therefore, updating the package will not automatically update the actual boot loader on your system. To work around this problem, use the **grub-install** command manually every time after the package is updated. See Section E.3, "Installing GRUB" for details about the command.



IMPORTANT

GRUB cannot construct a software RAID. Therefore, the **/boot** directory must reside on a single, specific disk partition. The **/boot** directory cannot be striped across multiple disks, as in a level 0 RAID. To use a level 0 RAID on your system, place **/boot** on a separate partition outside the RAID.

Similarly, because the **/boot** directory must reside on a single, specific disk partition, **GRUB** cannot boot the system if the disk holding that partition fails or is removed from the system. This is true even if the disk is mirrored in a level 1 RAID. The following Red Hat Knowledgebase article describes how to make the system bootable from another disk in the mirrored set: https://access.redhat.com/site/articles/7094

Note that these issues apply only to RAID that is implemented in software, where the individual disks that make up the array are still visible as individual disks on the system. These issues do not apply to hardware RAID where multiple disks are represented as a single device.

The exact steps to fix a broken **GRUB** configuration will vary depending on what kind of problem there is. The GNU GRUB Manual offers a list of all possible error messages displayed by **GRUB** in different stages and their underlying causes. Use the manual for reference.

Once you have determined the cause of the error, you can start fixing it. If you are encountering an error which only appears after you select an entry from the **GRUB** menu, then you can use the menu to fix the error temporarily, boot the system, and then fix the error permanently by running the **grub-install** command to reinstall the boot loader, or by editing the **/boot/grub/grub.conf** or **/boot/efi/EFI/redhat/grub.conf** with a plain text editor. For information about the configuration file structure, see Section E.8, "GRUB Menu Configuration File" .

NOTE

There are two identical files in the **GRUB** configuration directory: **grub.conf** and **menu.lst**. The **grub.conf** configuration file is loaded first; therefore you should make your changes there. The second file, **menu.lst**, will only be loaded if **grub.conf** is not found.

E.5. GRUB TERMINOLOGY

One of the most important things to understand before using GRUB is how the program refers to devices, such as hard drives and partitions. This information is particularly important when configuring GRUB to boot multiple operating systems.

E.5.1. Device Names

When referring to a specific device with GRUB, do so using the following format (note that the parentheses and comma are very important syntactically):

(*<type-of-device><bios-device-number>*,*<partition-number>*)

The *<type-of-device>* specifies the type of device from which GRUB boots. The two most common options are **hd** for a hard disk or **fd** for a 3.5 diskette. A lesser used device type is also available called **nd** for a network disk. Instructions on configuring GRUB to boot over the network are available online at http://www.gnu.org/software/grub/manual/.

The *<bios-device-number>* is the BIOS device number. The primary IDE hard drive is numbered **0** and a secondary IDE hard drive is numbered **1**. This syntax is roughly equivalent to that used for devices by the kernel. For example, the **a** in **hda** for the kernel is analogous to the **0** in **hd0** for GRUB, the **b** in **hdb** is analogous to the **1** in **hd1**, and so on.

The *<partition-number>* specifies the number of a partition on a device. Like the *<bios-device-number>*, most types of partitions are numbered starting at **0**. However, BSD partitions are specified using letters, with **a** corresponding to **0**, **b** corresponding to **1**, and so on.

NOTE

The numbering system for devices under GRUB always begins with **0**, not **1**. Failing to make this distinction is one of the most common mistakes made by new users.

To give an example, if a system has more than one hard drive, GRUB refers to the first hard drive as **(hd0)** and the second as **(hd1)**. Likewise, GRUB refers to the first partition on the first drive as **(hd0,0)** and the third partition on the second hard drive as **(hd1,2)**.

In general the following rules apply when naming devices and partitions under GRUB:

- It does not matter if system hard drives are IDE or SCSI, all hard drives begin with the letters **hd**. The letters **fd** are used to specify 3.5 diskettes.

- To specify an entire device without respect to partitions, leave off the comma and the partition number. This is important when telling GRUB to configure the MBR for a particular disk. For example, **(hd0)** specifies the MBR on the first device and **(hd3)** specifies the MBR on the fourth device.

- If a system has multiple drive devices, it is very important to know how the drive boot order is set in the BIOS. This is a simple task if a system has only IDE or SCSI drives, but if there is a mix of devices, it becomes critical that the type of drive with the boot partition be accessed first.

E.5.2. File Names and Blocklists

When typing commands to GRUB that reference a file, such as a menu list, it is necessary to specify an absolute file path immediately after the device and partition numbers.

The following illustrates the structure of such a command:

(<device-type><device-number>,<partition-number>)</path/to/file>

In this example, replace *<device-type>* with **hd**, **fd**, or **nd**. Replace *<device-number>* with the integer for the device. Replace *</path/to/file>* with an absolute path relative to the top-level of the device.

It is also possible to specify files to GRUB that do not actually appear in the file system, such as a chain loader that appears in the first few blocks of a partition. To load such files, provide a *blocklist* that specifies block by block where the file is located in the partition. Since a file is often comprised of several different sets of blocks, blocklists use a special syntax. Each block containing the file is specified by an offset number of blocks, followed by the number of blocks from that offset point. Block offsets are listed sequentially in a comma-delimited list.

The following is a sample blocklist:

```
0+50,100+25,200+1
```

This sample blocklist specifies a file that starts at the first block on the partition and uses blocks 0 through 49, 100 through 124, and 200.

Knowing how to write blocklists is useful when using GRUB to load operating systems which require chain loading. It is possible to leave off the offset number of blocks if starting at block 0. As an example, the chain loading file in the first partition of the first hard drive would have the following name:

```
(hd0,0)+1
```

The following shows the **chainloader** command with a similar blocklist designation at the GRUB command line after setting the correct device and partition as root:

```
chainloader +1
```

E.5.3. The Root File System and GRUB

The use of the term *root file system* has a different meaning in regard to GRUB. It is important to remember that GRUB's root file system has nothing to do with the Linux root file system.

The GRUB root file system is the top level of the specified device. For example, the image file **(hd0,0)/grub/splash.xpm.gz** is located within the **/grub/** directory at the top-level (or root) of the **(hd0,0)** partition (which is actually the **/boot/** partition for the system).

Next, the **kernel** command is executed with the location of the kernel file as an option. Once the Linux kernel boots, it sets up the root file system that Linux users are familiar with. The original GRUB root file system and its mounts are forgotten; they only existed to boot the kernel file.

Refer to the **root** and **kernel** commands in Section E.7, "GRUB Commands" for more information.

E.6. GRUB INTERFACES

GRUB features three interfaces which provide different levels of functionality. Each of these interfaces allows users to boot the Linux kernel or another operating system.

The interfaces are as follows:

NOTE

The following GRUB interfaces can only be accessed by pressing any key within the three seconds of the GRUB menu bypass screen.

Menu Interface

This is the default interface shown when GRUB is configured by the installation program. A menu of operating systems or preconfigured kernels are displayed as a list, ordered by name. Use the arrow keys to select an operating system or kernel version and press the **Enter** key to boot it. If you do nothing on this screen, then after the time out period expires GRUB will load the default option.

Press the **e** key to enter the entry editor interface or the **c** key to load a command line interface.

Refer to Section E.8, "GRUB Menu Configuration File" for more information on configuring this interface.

Menu Entry Editor Interface

To access the menu entry editor, press the **e** key from the boot loader menu. The GRUB commands for that entry are displayed here, and users may alter these command lines before booting the operating system by adding a command line (**o** inserts a new line after the current line and **O** inserts a new line before it), editing one (**e**), or deleting one (**d**).

After all changes are made, the **b** key executes the commands and boots the operating system. The **Esc** key discards any changes and reloads the standard menu interface. The **c** key loads the command line interface.

NOTE

For information about changing runlevels using the GRUB menu entry editor, refer to Section E.9, "Changing Runlevels at Boot Time" .

Command Line Interface

The command line interface is the most basic GRUB interface, but it is also the one that grants the most control. The command line makes it possible to type any relevant GRUB commands followed by the **Enter** key to execute them. This interface features some advanced shell-like features, including **Tab** key completion based on context, and **Ctrl** key combinations when typing commands, such as **Ctrl**+**a** to move to the beginning of a line and **Ctrl**+**e** to move to the end of a line. In addition, the arrow, **Home**, **End**, and **Delete** keys work as they do in the **bash** shell.

Refer to Section E.7, "GRUB Commands" for a list of common commands.

E.6.1. Interfaces Load Order

When GRUB loads its second stage boot loader, it first searches for its configuration file. Once found, the menu interface bypass screen is displayed. If a key is pressed within three seconds, GRUB builds a menu list and displays the menu interface. If no key is pressed, the default kernel entry in the GRUB menu is used.

If the configuration file cannot be found, or if the configuration file is unreadable, GRUB loads the command line interface, allowing the user to type commands to complete the boot process.

If the configuration file is not valid, GRUB prints out the error and asks for input. This helps the user see precisely where the problem occurred. Pressing any key reloads the menu interface, where it is then possible to edit the menu option and correct the problem based on the error reported by GRUB. If the correction fails, GRUB reports an error and reloads the menu interface.

E.7. GRUB COMMANDS

GRUB allows a number of useful commands in its command line interface. Some of the commands accept options after their name; these options should be separated from the command and other options on that line by space characters.

The following is a list of useful commands:

- **boot** – Boots the operating system or chain loader that was last loaded.
- **chainloader** ***</path/to/file>*** – Loads the specified file as a chain loader. If the file is located on the first sector of the specified partition, use the blocklist notation, **+1**, instead of the file name.

  The following is an example **chainloader** command:

  ```
  chainloader +1
  ```

- **displaymem** – Displays the current use of memory, based on information from the BIOS. This is useful to determine how much RAM a system has prior to booting it.
- **initrd** ***</path/to/initrd>*** – Enables users to specify an initial RAM disk to use when booting. An **initrd** is necessary when the kernel needs certain modules in order to boot properly, such as when the root partition is formatted with the ext3 or ext4 file system.

  The following is an example **initrd** command:

  ```
  initrd /initrd-2.6.8-1.523.img
  ```

- **install** ***<stage-1> <install-disk> <stage-2>*** **p** ***config-file*** – Installs GRUB to the system MBR.
  - ***<stage-1>*** – Signifies a device, partition, and file where the first boot loader image can be found, such as **(hd0,0)/grub/stage1**.
  - ***<install-disk>*** – Specifies the disk where the stage 1 boot loader should be installed, such as **(hd0)**.

- ***<stage-2>*** – Passes the stage 2 boot loader location to the stage 1 boot loader, such as **(hd0,0)/grub/stage2**.
- **p *<config-file>*** – This option tells the **install** command to look for the menu configuration file specified by ***<config-file>***, such as **(hd0,0)/grub/grub.conf**.

WARNING

The **install** command overwrites any information already located on the MBR.

- **kernel *</path/to/kernel> <option-1> <option-N>*** ... – Specifies the kernel file to load when booting the operating system. Replace *</path/to/kernel>* with an absolute path from the partition specified by the root command. Replace *<option-1>* with options for the Linux kernel, such as **root=/dev/VolGroup00/LogVol00** to specify the device on which the root partition for the system is located. Multiple options can be passed to the kernel in a space separated list.

  The following is an example **kernel** command:

  ```
  kernel /vmlinuz-2.6.8-1.523 ro root=/dev/VolGroup00/LogVol00
  ```

  The option in the previous example specifies that the root file system for Linux is located on the **hda5** partition.

- **root (*<device-type><device-number>,<partition>*)** – Configures the root partition for GRUB, such as **(hd0,0)**, and mounts the partition.

  The following is an example **root** command:

  ```
  root (hd0,0)
  ```

- **rootnoverify (*<device-type><device-number>,<partition>*)** – Configures the root partition for GRUB, just like the **root** command, but does not mount the partition.

Other commands are also available; type **help --all** for a full list of commands. For a description of all GRUB commands, refer to the documentation available online at http://www.gnu.org/software/grub/manual/.

E.8. GRUB MENU CONFIGURATION FILE

The configuration file (**/boot/grub/grub.conf** on BIOS systems and **/boot/efi/EFI/redhat/grub.conf** on UEFI systems), which is used to create the list of operating systems to boot in GRUB's menu interface, essentially allows the user to select a pre-set group of commands to execute. The commands given in Section E.7, "GRUB Commands" can be used, as well as some special commands that are only available in the configuration file.

E.8.1. Configuration File Structure