Red Hat Enterprise Linux 9.4

Using image mode for RHEL to build, deploy, and manage operating systems

Using RHEL bootc images on Red Hat Enterprise Linux 9

Last Updated: 2025-06-10

Red Hat Enterprise Linux 9.4 Using image mode for RHEL to build, deploy, and manage operating systems

Using RHEL bootc images on Red Hat Enterprise Linux 9

Legal Notice

Abstract

RHEL bootc images enable you to build, deploy, and manage the operating system as if it is any other container. You can converge on a single container-native workflow to manage everything from your applications to the underlying OS.

Table of Contents

PROVIDING FEEDBACK ON RED HAT DOCUMENTATION

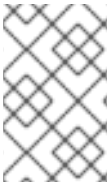
We appreciate your feedback on our documentation. Let us know how we can improve it.

Submitting feedback through Jira (account required)

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

CHAPTER 1. INTRODUCING IMAGE MODE FOR RHEL

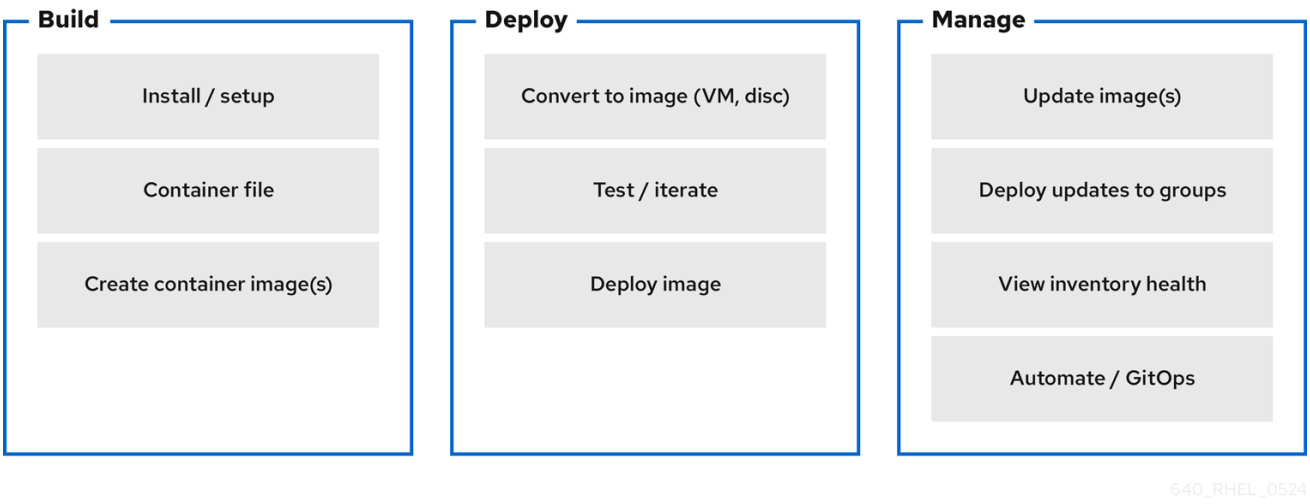
Use image mode for RHEL to build, test, and deploy operating systems by using the same tools and techniques as application containers. Image mode for RHEL is available by using the **registry.redhat.io/rhel9/rhel-bootc** bootc image. The RHEL bootc images differ from the existing application Universal Base Images (UBI) in that they contain additional components necessary to boot that were traditionally excluded, such as, kernel, initrd, boot loader, firmware, among others.



NOTE

The **rhel-bootc** and user-created containers based on **rhel-bootc** container image are subject to the Red Hat Enterprise Linux end user license agreement (EULA) . You are not allowed to publicly redistribute these images.

Figure 1.1. Building, deploying, and managing operating system by using image mode for RHEL



Red Hat provides bootc image for the following computer architectures:

- AMD and Intel 64-bit architectures (x86-64-v2)
- The 64-bit ARM architecture (ARMv8.0-A)
- IBM Power Systems 64-bit Little Endian architecture (ppc64le)
- IBM Z 64-bit architecture (s390x)

The benefits of image mode for RHEL occur across the lifecycle of a system. The following list contains some of the most important advantages:

Container images are easier to understand and use than other image formats and are fast to build

Containerfiles, also known as Dockerfiles, provide a straightforward approach to defining the content and build instructions for an image. Container images are often significantly faster to build and iterate on compared to other image creation tools.

Consolidate process, infrastructure, and release artifacts

As you distribute applications as containers, you can use the same infrastructure and processes to manage the underlying operating system.

Immutable updates

Just as containerized applications are updated in an immutable way, with image mode for RHEL, the operating system is also. You can boot into updates and roll back when needed in the same way that you use **rpm-ostree** systems.

WARNING

The use of **rpm-ostree** to make changes, or install content, is not supported.

Portability across hybrid cloud environments

You can use bootc images across physical, virtualized, cloud, and edge environments.

Although containers provide the foundation to build, transport, and run images, it is important to understand that after you deploy these bootc images, either by using an installation mechanism, or you convert them to a disk image, the system does not run as a container.

- Bootc supports the following container image formats and disk image formats:

Table 1.1. bootc supported image types

Image type	Target environment
OCI container format	Physical, virtualized, cloud, and edge environments.
ami	Amazon Machine Image.
qcow2 (default)	QEMU.
vmdk	VMDK for vSphere.
anaconda-iso	An unattended Anaconda installer that installs to the first disk found.
raw	Unformatted raw disk. Also supported in QEMU and Libvirt
vhd	VHD for Virtual PC, among others.
gce	Google Compute Engine (GCE) environment.

Containers help streamline the lifecycle of a RHEL system by offering the following possibilities:

Building container images

You can configure your operating system at a build time by modifying the Containerfile. Image mode for RHEL is available by using the **registry.redhat.io/rhel9/rhel-bootc** container image. You can use Podman, OpenShift Container Platform, or other standard container build tools to manage your containers and container images. You can automate the build process by using CI/CD pipelines.

Versioning, mirroring, and testing container images

You can version, mirror, introspect, and sign your derived bootc image by using any container tools such as Podman or OpenShift Container Platform.

Deploying container images to the target environment

You have several options on how to deploy your image:

- **Anaconda**: is the installation program used by RHEL. You can deploy all image types to the target environment by using Anaconda and Kickstart to automate the installation process.
- **bootc-image-builder**: is a containerized tool that converts the container image to different types of disk images, and optionally uploads them to an image registry or object storage.
- **bootc**: is a tool responsible for fetching container images from a container registry and installing them to a system, updating the operating system, or switching from an existing ostree-based system. The RHEL bootc image contains the **bootc** utility by default and works with all image types. However, remember that the **rpm-ostree** is not supported and must not be used to make changes.

Updating your operating system

The system supports in-place transactional updates with rollback after deployment. Automatic updates are on by default. A systemd service unit and systemd timer unit files check the container registry for updates and apply them to the system. As the updates are transactional, a reboot is required. For environments that require more sophisticated or scheduled rollouts, disable auto updates and use the **bootc** utility to update your operating system.

RHEL has two deployment modes. Both provide the same stability, reliability, and performance during deployment. See their differences:

1. **Package mode**: You can build package-based images and OSTree images by using RHEL image builder, and you can manage the package mode images by using **composer-cli** or web console. The operating system uses RPM packages and is updated by using the **dnf** package manager. The root filesystem is mutable. However, the operating system cannot be managed as a containerized application. See Composing a customized RHEL system image product documentation.
2. **Image mode**: a container-native approach to build, deploy, and manage RHEL. The same RPM packages are delivered as a base image and updates are deployed as a container image. The root filesystem is immutable by default, except for **/etc** and **/var**, with most content coming from the container image.

You can choose to use either the **Image mode** or the **Package mode** deployment to build, test, and share your operating system. **Image mode** additionally enables you to manage your operating system in the same way as any other containerized application.

1.1. PREREQUISITES

- You have a subscribed RHEL 9 system. For more information, see Getting Started with RHEL System Registration documentation.
- You have a container registry. You can create your registry locally or create a free account on the Quay.io service. To create the Quay.io account, see Red Hat Quay.io page.
- You have a Red Hat account with either production or developer subscriptions. No cost developer subscriptions are available on the Red Hat Enterprise Linux Overview page.

- You have authenticated to registry.redhat.io. For more information, see Red Hat Container Registry Authentication article.

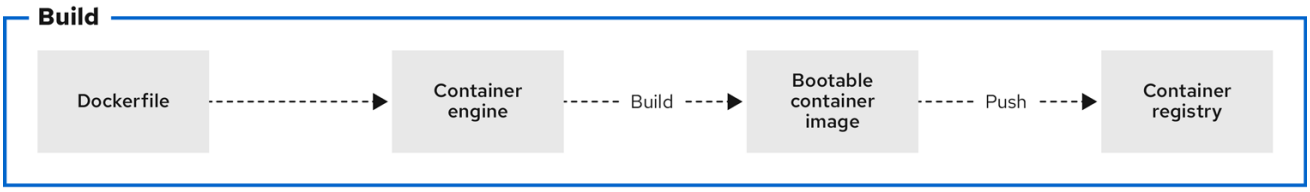
1.2. ADDITIONAL RESOURCES

- Introducing image mode for RHEL and bootc in Podman Desktop quick start guide
- Image mode for Red Hat Enterprise Linux quick start: AI inference quick start guide
- Getting Started with Podman AI Lab blog article
- Customizing Anaconda product documentation
- Automatically installing RHEL product documentation (Kickstart)
- Composing a customized RHEL system image product documentation
- Composing, installing, and managing RHEL for Edge images product documentation

CHAPTER 2. BUILDING AND TESTING RHEL BOOTC IMAGES

The following procedures use Podman to build and test your container image. You can also use other tools, for example, OpenShift Container Platform. For more examples of configuring RHEL systems by using containers, see the rhel-bootc-examples repository.

Figure 2.1. Building an image by using instructions from a Containerfile, testing the container, pushing an image to a registry, and sharing it with others



A general **Containerfile** structure is the following:

```
FROM registry.redhat.io/rhel9/rhel-bootc:latest

RUN dnf -y install [software] [dependencies] && dnf clean all

ADD [application]
ADD [configuration files]

RUN [config scripts]
```

The available commands that are usable inside a **Containerfile** and a **Dockerfile** are equivalent.

However, the following commands in a **Containerfile** are ignored when the **rhel-9-bootc** image is installed to a system:

- **ENTRYPOINT** and **CMD** (OCI: **Entrypoint**/**Cmd**): you can set **CMD** /**sbin**/**init** instead.
- **ENV** (OCI: **Env**): change the **systemd** configuration to configure the global system environment.
- **EXPOSE** (OCI: **exposedPorts**): it is independent of how the system firewall and network function at runtime.
- **USER** (OCI: **User**): configure individual services inside the RHEL bootc to run as unprivileged users instead.

The **rhel-9-bootc** container image reuses the OCI image format.

- The **rhel-9-bootc** container image ignores the container config section (**Config**) when it is installed to a system.
- The **rhel-9-bootc** container image does not ignore the container config section (**Config**) when you run this image by using container runtimes such as **podman** or **docker**.

NOTE

Building custom **rhel-bootc** base images is not supported in this release.

2.1. BUILDING A CONTAINER IMAGE

Use the **podman build** command to build an image using instructions from a **Containerfile**.

Prerequisites

- The **container-tools** meta-package is installed.

Procedure

1. Create a **Containerfile**:

```
$ cat Containerfile
FROM registry.redhat.io/rhel9/rhel-bootc:latest
RUN dnf -y install cloud-init && \
    ln -s ../cloud-init.target /usr/lib/systemd/system/default.target.wants && \
    dnf clean all
```

   This **Containerfile** example adds the **cloud-init** tool, so it automatically fetches SSH keys and can run scripts from the infrastructure and also gather configuration and secrets from the instance metadata. For example, you can use this container image for pre-generated AWS or KVM guest systems.

2. Build the ***<image>*** image by using **Containerfile** in the current directory:

```
$ podman build -t quay.io/<namespace>/<image>:<tag> .
```

Verification

- List all images:

```
$ podman images
REPOSITORY                TAG     IMAGE ID      CREATED             SIZE
localhost/<image>         latest  b28cd00741b3  About a minute ago  2.1 GB
```

Additional resources

- Working with container registries
- Building an image from a Containerfile with Buildah

2.2. BUILDING DERIVED BOOTABLE IMAGES BY USING MULTI-STAGE BUILDS

The deployment image should include only the application and its required runtime, without adding any build tools or unnecessary libraries. To achieve this, use a two-stage **Containerfile**: one stage for building the artifacts and another for hosting the application.

With multi-stage builds, you use multiple **FROM** instructions in your **Containerfile**. Each **FROM** instruction can use a different base, and each of them begins a new stage of the build. You can selectively copy artifacts from one stage to another, and exclude everything you do not need in the final image.

Multi-stage builds offer several advantages:

Smaller image size

By separating the build environment from the runtime environment, only the necessary files and dependencies are included in the final image, significantly reducing its size.

Improved security

Since build tools and unnecessary libraries are excluded from the final image, the attack surface is reduced, leading to a more secure container.

Optimized performance

A smaller image size means faster download, deployment, and startup times, improving the overall efficiency of the containerized application.

Simplified maintenance

With the build and runtime environments separated, the final image is cleaner and easier to maintain, containing only what is needed to run the application.

Cleaner builds

Multi-stage builds help avoid clutter from intermediate files, which could accumulate during the build process, ensuring that only essential artifacts make it into the final image.

Resource efficiency

The ability to build in one stage and discard unnecessary parts minimizes the use of storage and bandwidth during deployment.

Better Layer Caching

With clearly defined stages, Podman can efficiently cache the results of previous stages, by accelerating up future builds.

The following **Containerfile** consists of two stages. The first stage is typically named **builder** and it compiles a golang binary. The second stage copies the binary from the first stage. The default working directory for the go-toolset builder is **opt/ap-root/src**.

```
FROM registry.access.redhat.com/ubi9/go-toolset:latest as builder
RUN echo 'package main; import "fmt"; func main() { fmt.Println("hello world") }' > helloworld.go
RUN go build helloworld.go

FROM registry.redhat.io/rhel9/rhel-bootc:latest
COPY --from=builder /opt/app-root/src/helloworld /
CMD ["/helloworld"]
```

As a result, the final container image includes the **helloworld** binary but no data from the previous stage.

You can also use multi-stage builds to perform the following scenarios:

Stopping at a specific build stage

When you build your image, you can stop at a specified build stage. For example:

```
$ podman build --target build -t hello .
```

For example, you can use this approach to debugging a specific build stage.

Using an external image as a stage

You can use the **COPY --from** instruction to copy from a separate image either using the local image name, a tag available locally or on a container registry, or a tag ID. For example:

```
COPY --from=<image> <source_path> <destination_path>
```

Using a previous stage as a new stage

You can continue where a previous stage ended by using the **FROM** instruction. From example:

```
FROM ubi9 AS stage1
[...]

FROM stage1 AS stage2
[...]

FROM ubi9 AS final-stage
[...]
```

Additional resources

- How to build multi-architecture container images article

2.3. RUNNING A CONTAINER IMAGE

Use the **podman run** command to run and test your container.

Prerequisites

- The **container-tools** meta-package is installed.

Procedure

- Run the container named **mybootc** based on the **quay.io/*<namespace>*/*<image>*:*<tag>*** container image:

  ```
  $ podman run -it --rm --name mybootc quay.io/<namespace>/<image>:<tag> /bin/bash
  ```

  - The **-i** option creates an interactive session. Without the **-t** option, the shell stays open, but you cannot type anything to the shell.
  - The **-t** option opens a terminal session. Without the **-i** option, the shell opens and then exits.
  - The **--rm** option removes the **quay.io/*<namespace>*/*<image>*:*<tag>*** container image after the container exits.

Verification

- List all running containers:

  ```
  $ podman ps
  CONTAINER ID  IMAGE                                COMMAND         CREATED        STATUS
  PORTS   NAMES
  7ccd6001166e  quay.io/<namespace>/<image>:<tag>  /sbin/init  6 seconds ago  Up 5
  seconds ago          mybootc
  ```

Additional resources

- Podman run command

2.4. PUSHING A CONTAINER IMAGE TO THE REGISTRY

Use the **podman push** command to push an image to your own, or a third party, registry and share it with others. The following procedure uses the Red Hat Quay registry.

Prerequisites

- The **container-tools** meta-package is installed.
- An image is built and available on the local system.
- You have created the Red Hat Quay registry. For more information see Proof of Concept - Deploying Red Hat Quay.

Procedure

- Push the **quay.io/*<namespace>*/*<image>*:*<tag>*** container image from your local storage to the registry:

```
$ podman push quay.io/<namespace>/<image>:<tag>
```

Additional resources

- **podman-tag(1)** man page
- **podman-push(1)** man page

CHAPTER 3. BUILDING AND MANAGING LOGICALLY BOUND IMAGES

Logically bound images give you support for container images that are lifecycle bound to the base bootc image. This helps combine different operational processes for applications and operating systems, and the container application images are referenced from the base image as image files or an equivalent. Consequently, you can manage multiple container images for system installations.

You can use containers for lifecycle-bound workloads, such as security agents and monitoring tools. You can also upgrade such workloads by using the **bootc upgrade** command.

3.1. LOGICALLY BOUND IMAGES

Logically bound images enable an association of the container application images to a base bootc system image. The term *logically bound* is used to contrast with physically bound images. The logically bound images offer the following benefits:

- You can update the bootc system without re-downloading the application container images.
- You can update the application container images without modifying the bootc system image, which is especially useful for development work.

The following are examples for lifecycle bound workloads, whose activities are usually not updated outside of the host:

- Logging, for example, journald→remote log forwarder container
- Monitoring, for example, Prometheus node_exporter
- Configuration management agents
- Security agents

Another important property of the logically bound images is that they must be present and available on the host, possibly from a very early stage in the boot process.

Differently from the default usage of tools like Podman or Docker, images might be pulled dynamically after the boot starts, which requires a functioning network. For example, if the remote registry is temporarily unavailable, the host system might run longer without log forwarding or monitoring, which is not desirable. Logically bound images enable you to reference container images similarly to you can with **ExecStart=** in a systemd unit.

When using logically bound images, you must manage multiple container images for the system to install the logically bound images. This is an advantage and also a disadvantage. For example, for a disconnected or offline installation, you must mirror all the containers, not just one. The application images are only referenced from the base image as **.image** files or an equivalent.

Logically bound images installation

When you run **bootc install**, logically bound images must be present in the default **/var/lib/containers** container store. The images will be copied into the target system and present directly at boot, alongside the bootc base image.

Logically bound images lifecycle

Logically bound images are referenced by the bootable container and have guaranteed availability when the bootc based server starts. The image is always upgraded by using **bootc upgrade** and is available as **read-only** to other processes, such as Podman.

Logically bound images management on upgrades, rollbacks, and garbage collection

- During upgrades, the logically bound images are managed exclusively by bootc.
- During rollbacks, the logically bound images corresponding to rollback deployments are retained.
- bootc performs garbage collection of unused bound images.

3.2. USING LOGICALLY BOUND IMAGES

Each logically bound image is defined in a Podman Quadlet **.image** or **.container file**. To use logically bound images, follow the steps:

Prerequisites

- The **container-tools** meta-package is installed.

Procedure

1. Select the image that you want to logically bound.
2. Create a **Containerfile**:

```
$ cat Containerfile
FROM quay.io/<namespace>/<image>:latest
COPY ./<app_1>.image /usr/share/containers/systemd/<app_1>.image
COPY ./<app_2>.container /usr/share/containers/systemd/<app_2>.container

RUN ln -s /usr/share/containers/systemd/<app_1>.image \
 /usr/lib/bootc/bound-images.d/<app_1>.image && \
    ln -s /usr/share/containers/systemd/<app_2>.container \
     /usr/lib/bootc/bound-images.d/<app_2>.container
```

3. In the **.container** definition, use:

```
GlobalArgs=--storage-opt=additionalimagestore=/usr/lib/bootc/storage
```

In the **Containerfile** example, the image is selected to be logically bound by creating a symlink in the **/usr/lib/bootc/bound-images.d** directory pointing to either an **.image** or a **.container** file.

4. Run the **bootc upgrade** command.

```
$ bootc upgrade
```

The bootc upgrade performs the following overall steps:

a. Fetches the new base image from the image repository. See linkhttps://docs.redhat.com/en/documentation/red_hat_enterprise_linux/9/html/using_image users-groups-ssh-key-and-secrets-in-image-mode-for-rhel#configuring-container-pull-

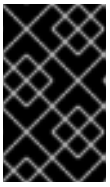
secrets_managing-users-groups-ssh-key-and-secrets-in-image-mode-for-rhel[Configuring container pull secrets].

b. Reads the new base image root file system to discover logically bound images.

c. Automatically pulls any discovered logically bound images defined in the new bootc image into the bootc-owned **/usr/lib/bootc/storage** image storage.

5. Make the bound images become available to container runtimes such as Podman. For that, you must explicitly configure bound images to point to the bootc storage as an "additional image store". For example:

```
podman --storage-opt=additionalimagestore=/usr/lib/bootc/storage run <image>
```



IMPORTANT

Do not attempt to globally enable the **/usr/lib/bootc/storage** image storage in **/etc/containers/storage.conf**. Only use the bootc storage for logically bound images.

The **bootc image store** is owned by **bootc**. The logically bound images will be garbage collected when they are no longer referenced by a file in the **/usr/lib/bootc/bound-images.d** directory.

CHAPTER 4. CREATING BOOTC COMPATIBLE BASE DISK IMAGES WITH BOOTC-IMAGE-BUILDER

The **bootc-image-builder**, available as a Technology Preview, is a containerized tool to create disk images from bootc images. You can use the images that you build to deploy disk images in different environments, such as the edge, server, and clouds.

4.1. INTRODUCING IMAGE MODE FOR RHEL FOR BOOTC-IMAGE-BUILDER

With the **bootc-image-builder** tool, you can convert bootc images into disk images for a variety of different platforms and formats. Converting bootc images into disk images is equivalent to installing a bootc. After you deploy these disk images to the target environment, you can update them directly from the container registry.

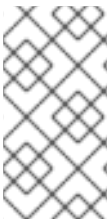
NOTE

The **bootc-image-builder** can only pull and use images from public container repositories. Building base disk images which come from private registries by using **bootc-image-builder** is not supported in this release. If your container image is stored in a private repository, **bootc-image-builder** cannot pull the image because it is not able to authenticate to the registry. If you need to use an image from a private repository, you must authenticate to the registry first and then pull the container image before you use it with **bootc-image-builder**. After pulling the image, you can run the bootc-image-builder command using the **--local** option.

The **bootc-image-builder tool** supports generating the following image types:

- Disk image formats, such as ISO, suitable for disconnected installations.
- Virtual disk images formats, such as:
  - QEMU copy-on-write (QCOW2)
  - Amazon Machine Image (AMI)/ – Raw
  - Virtual Machine Image (VMI)

Deploying from a container image is beneficial when you run VMs or servers because you can achieve the same installation result. That consistency extends across multiple different image types and platforms when you build them from the same container image. Consequently, you can minimize the effort in maintaining operating system images across platforms. You can also update systems that you deploy from these disk images by using the **bootc** tool, instead of re-creating and uploading new disk images with **bootc-image-builder**.



NOTE

Generic base container images do not include any default passwords or SSH keys. Also, the disk images that you create by using the **bootc-image-builder** tool do not contain the tools that are available in common disk images, such as **cloud-init**. These disk images are transformed container images only.

Although you can deploy a **rhel-9-bootc** image directly, you can also create your own customized images that are derived from this bootc image. The **bootc-image-builder** tool takes the **rhel-9-bootc** OCI container image as an input.

Additional resources

- Red Hat products that use cloud-init

4.2. INSTALLING BOOTC-IMAGE-BUILDER

The **bootc-image-builder** is intended to be used as a container and it is not available as an RPM package in RHEL. To access it, follow the procedure.

Prerequisites

- The **container-tools** meta-package is installed. The meta-package contains all container tools, such as Podman, Buildah, and Skopeo.
- You are authenticated to **registry.redhat.io**. For details, see Red Hat Container Registry Authentication.

Procedure

1. Login to authenticate to **registry.redhat.io**:

```
$ sudo podman login registry.redhat.io
```

2. Install the **bootc-image-builder** tool:

```
$ sudo podman pull registry.redhat.io/rhel9/bootc-image-builder
```

Verification

- List all images pulled to your local system:

```
$ sudo podman images
REPOSITORY                                  TAG         IMAGE ID      CREATED       SIZE
registry.redhat.io/rhel9/bootc-image-builder  latest      b361f3e845ea  24 hours ago  676 MB
```

Additional resources

- Red Hat Container Registry Authentication
- Pulling images from registries

4.3. CREATING QCOW2 IMAGES BY USING BOOTC-IMAGE-BUILDER

Build a RHEL bootc image into a QEMU Disk Images (QCOW2) image for the architecture that you are running the commands on.

The RHEL base image does not include a default user. Optionally, you can inject a user configuration with the **--config** option to run the bootc-image-builder container. Alternatively, you can configure the base image with **cloud-init** to inject users and SSH keys on first boot. See Users and groups

configuration - Injecting users and SSH keys by using cloud-init.

Prerequisites

- You have Podman installed on your host machine.
- You have **virt-install** installed on your host machine.
- You have root access to run the **bootc-image-builder** tool, and run the containers in **--privileged** mode, to build the images.

Procedure

1. Optional: Create a **config.toml** to configure user access, for example:

```
[[customizations.user]]
name = "user"
password = "pass"
key = "ssh-rsa AAA ... user@email.com"
groups = ["wheel"]
```

2. Run **bootc-image-builder**. Optionally, if you want to use user access configuration, pass the **config.toml** as an argument.

NOTE

If you do not have the container storage mount and **--local** image options, your image must be public.

a. The following is an example of creating a public QCOW2 image:

```
$ sudo podman run \
    --rm \
    -it \
    --privileged \
    --pull=newer \
    --security-opt label=type:unconfined_t \
    -v ./config.toml:/config.toml \
    -v ./output:/output \
    registry.redhat.io/rhel9/bootc-image-builder:latest \
    --type qcow2 \
    --config /config.toml \
   quay.io/<namespace>/<image>:<tag>
```

b. The following is an example of creating a private QCOW2 image:

```
$ sudo podman run \
    --rm \
    -it \
    --privileged \
    --pull=newer \
    --security-opt label=type:unconfined_t \
    -v $(pwd)/config.toml:/config.toml:ro \
    -v $(pwd)/output:/output \
```

```
-v /var/lib/containers/storage:/var/lib/containers/storage \
registry.redhat.io/rhel9/bootc-image-builder:latest \
--local
--type qcow2 \
quay.io/<namespace>/<image>:<tag>
```

You can find the **.qcow2** image in the output folder.

Next steps

- You can deploy your image. See Deploying a container image using KVM with a QCOW2 disk image.
- You can make updates to the image and push the changes to a registry. See Managing RHEL bootc images.

4.4. CREATING VMDK IMAGES BY USING BOOTC-IMAGE-BUILDER

Create a Virtual Machine Disk (VMDK) from a bootc image and use it within VMware's virtualization platforms, such as vSphere, or use the Virtual Machine Disk (VMDK) in VirtualBox.

Prerequisites

- You have Podman installed on your host machine.
- You have authenticated to the Red Hat Registry by using the **podman login registry.redhat.io**.
- You have pulled the **rhel9/bootc-image-builder** container image.

Procedure

1. Create a **Containerfile** with the following content:

```
FROM registry.redhat.io/rhel9/rhel-bootc:9.4
RUN dnf -y install cloud-init open-vm-tools && \
ln -s ../cloud-init.target /usr/lib/systemd/system/default.target.wants && \
rm -rf /var/{cache,log} /var/lib/{dnf,rhsm} && \
systemctl enable vmtoolsd.service
```

2. Build the bootc image:

```
# podman build . -t localhost/rhel-bootc-vmdk
```

3. Create a VMDK file from the previously created bootc image:

NOTE

If you do not have the container storage mount and --local image options, your image must be public.

a. The following is an example of creating a public VMDK image:

```
# podman run \
```

```
--rm \
-it \
--privileged \
-v /var/lib/containers/storage:/var/lib/containers/storage \
-v ./output:/output \
--security-opt label=type:unconfined_t \
--pull newer \
registry.redhat.io/rhel9/bootc-image-builder:9.4
--local \
--type vmdk \
quay.io/<namespace>/<image>:<tag>
```

b. The following is an example of creating a private VMDK image:

```
# podman run \
  --rm \
  -it \
  --privileged \
  --pull=newer \
  --security-opt label=type:unconfined_t \
  -v $(pwd)/config.toml:/config.toml:ro \
  -v $(pwd)/output:/output \
  -v /var/lib/containers/storage:/var/lib/containers/storage \
  registry.redhat.io/rhel9/bootc-image-builder:latest \
  --local
  --type vmdk \
  quay.io/<namespace>/<image>:<tag>
```

The **--local** option uses the local container storage to source the originating image to produce the VMDK instead of a remote repository.

A VMDK disk file for the bootc image is stored in the **output/vmdk** directory.

Next steps

- You can make updates to the image and push the changes to a registry. See Managing RHEL bootc images.

4.5. CREATING GCE IMAGES BY USING BOOTC-IMAGE-BUILDER

Build a RHEL bootc image into a gce image for the architecture that you are running the commands on. The RHEL base image does not include a default user. Optionally, you can inject a user configuration with the **--config** option to run the bootc-image-builder container. Alternatively, you can configure the base image with **cloud-init** to inject users and SSH keys on first boot. See Users and groups configuration - Injecting users and SSH keys by using cloud-init.

Prerequisites

- You have Podman installed on your host machine.
- You have root access to run the **bootc-image-builder** tool, and run the containers in **--privileged** mode, to build the images.

Procedure

1. Optional: Create a **config.toml** to configure user access, for example:

```
[[customizations.user]]
name = "user"
password = "pass"
key = "ssh-rsa AAA ... user@email.com"
groups = ["wheel"]
```

2. Run **bootc-image-builder**. Optionally, if you want to use user access configuration, pass the **config.toml** as an argument.

NOTE

If you do not have the container storage mount and **--local** image options, your image must be public.

a. The following is an example of creating a **gce** image:

```
$ sudo podman run \
    --rm \
    -it \
    --privileged \
    --pull=newer \
    --security-opt label=type:unconfined_t \
    -v ./config.toml:/config.toml \
    -v ./output:/output \
    registry.redhat.io/rhel9/bootc-image-builder:latest \
    --type gce \
    --config /config.toml \
   quay.io/<namespace>/<image>:<tag>
```

b. The following is an example of creating a private **gce** image:

```
$ sudo podman run \
    --rm \
    -it \
    --privileged \
    --pull=newer \
    --security-opt label=type:unconfined_t \
    -v $(pwd)/config.toml:/config.toml:ro \
    -v $(pwd)/output:/output \
    -v /var/lib/containers/storage:/var/lib/containers/storage \
    registry.redhat.io/rhel9/bootc-image-builder:latest \
    --local
    --type gce \
    quay.io/<namespace>/<image>:<tag>
```

You can find the **gce** image in the output folder.

Next steps

- You can deploy your image. See Deploying a container image using KVM with a QCOW2 disk image.

- You can make updates to the image and push the changes to a registry. See Managing RHEL bootc images.

4.6. CREATING AMI IMAGES BY USING BOOTC-IMAGE-BUILDER AND UPLOADING IT TO AWS

Create an Amazon Machine Image (AMI) from a bootc image and use it to launch an Amazon Web Service EC2 (Amazon Elastic Compute Cloud) instance.

Prerequisites

- You have Podman installed on your host machine.
- You have an existing **AWS S3** bucket within your AWS account.
- You have root access to run the **bootc-image-builder** tool, and run the containers in **--privileged** mode, to build the images.
- You have the **vmimport** service role configured on your account to import an AMI into your AWS account.

Procedure

1. Create a disk image from the bootc image.
   - Configure the user details in the Containerfile. Make sure that you assign it with sudo access.
   - Build a customized operating system image with the configured user from the Containerfile. It creates a default user with passwordless sudo access.
2. Optional: Configure the machine image with **cloud-init**. See Users and groups configuration - Injecting users and SSH keys by using cloud-init. The following is an example:

```
FROM registry.redhat.io/rhel9/rhel-bootc:9.4

RUN dnf -y install cloud-init && \
    ln -s ../cloud-init.target /usr/lib/systemd/system/default.target.wants && \
    rm -rf /var/{cache,log} /var/lib/{dnf,rhsm}
```

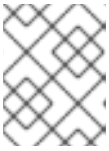
NOTE

You can also use **cloud-init** to add users and additional configuration by using instance metadata.

3. Build the bootc image. For example, to deploy the image to an **x86_64** AWS machine, use the following commands:

```
$ podman build -t quay.io/<namespace>/<image>:<tag> .
$ podman push quay.io/<namespace>/<image>:<tag> .
```

4. Use the **bootc-image-builder** tool to create an AMI from the bootc container image.



NOTE

If you do not have the container storage mount and **--local** image options, your image must be public.

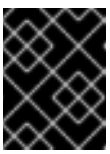
a. The following is an example of creating a public AMI image:

```
$ sudo podman run \
  --rm \
  -it \
  --privileged \
  --pull=newer \
  -v $HOME/.aws:/root/.aws:ro \
  --env AWS_PROFILE=default \
  registry.redhat.io/rhel9/bootc-image-builder:latest \
  --type ami \
  --aws-ami-name rhel-bootc-x86 \
  --aws-bucket rhel-bootc-bucket \
  --aws-region us-east-1 \
quay.io/<namespace>/<image>:<tag>
```

b. The following is an example of creating a private AMI image:

```
$ sudo podman run \
    --rm \
    -it \
    --privileged \
    --pull=newer \
    -v $HOME/.aws:/root/.aws:ro \
    --env AWS_PROFILE=default \
    registry.redhat.io/rhel9/bootc-image-builder:latest \
    --type ami \
    --aws-ami-name rhel-bootc-x86 \
    --aws-bucket rhel-bootc-bucket \
    --local
  quay.io/<namespace>/<image>:<tag>
```

You can also inject all your AWS configuration parameters by using **--env AWS_***.



IMPORTANT

The following flags must be specified all together. If you do not specify any flag, the AMI is exported to your output directory.

- **--aws-ami-name** - The name of the AMI image in AWS
- **--aws-bucket** - The target S3 bucket name for intermediate storage when you are creating the AMI
- **--aws-region** - The target region for AWS uploads
  The **bootc-image-builder** tool builds an AMI image and uploads it to your AWS s3 bucket by using your AWS credentials to push and register an AMI image after building it.

Next steps

- You can deploy your image. See Deploying a container image to AWS with an AMI disk image .
- You can make updates to the image and push the changes to a registry. See Managing RHEL bootc images.

Additional resources

- AWS CLI documentation

4.7. CREATING RAW DISK IMAGES BY USING BOOTC-IMAGE-BUILDER

You can convert a bootc image to a Raw image with an MBR or GPT partition table by using **bootc-image-builder**. The RHEL base image does not include a default user, so optionally, you can inject a user configuration with the **--config** option to run the **bootc-image-builder** container. Alternatively, you can configure the base image with **cloud-init** to inject users and SSH keys on first boot. See Users and groups configuration - Injecting users and SSH keys by using cloud-init.

Prerequisites

- You have Podman installed on your host machine.
- You have root access to run the **bootc-image-builder** tool, and run the containers in **--privileged** mode, to build the images.
- You have pulled your target container image in the container storage.

Procedure

1. Optional: Create a **config.toml** to configure user access, for example:

```
[[customizations.user]]
name = "user"
password = "pass"
key = "ssh-rsa AAA ... user@email.com"
groups = ["wheel"]
```

2. Run **bootc-image-builder**. If you want to use user access configuration, pass the **config.toml** as an argument:

NOTE

If you do not have the container storage mount and **--local** image options, your image must be public.

a. The following is an example of creating a public RAW image:

```
$ sudo podman run \
    --rm \
    -it \
    --privileged \
    --pull=newer \
    --security-opt label=type:unconfined_t \
```

```
  -v /var/lib/containers/storage:/var/lib/containers/storage \
  -v ./config.toml:/config.toml \
  -v ./output:/output \
  registry.redhat.io/rhel9/bootc-image-builder:latest \
  --local \
  --type raw \
  --config /config.toml \
quay.io/<namespace>/<image>:<tag>
```

b. The following is an example of creating a private RAW image:

```
$ sudo podman run \
  --rm \
  -it \
  --privileged \
  --pull=newer \
  --security-opt label=type:unconfined_t \
  -v $(pwd)/config.toml:/config.toml:ro \
  -v $(pwd)/output:/output \
  -v /var/lib/containers/storage:/var/lib/containers/storage \
  registry.redhat.io/rhel9/bootc-image-builder:latest \
  --local
  --type raw \
quay.io/<namespace>/<image>:<tag>
```

You can find the **.raw** image in the output folder.

Next steps

- You can deploy your image. See Deploying a container image by using KVM with a QCOW2 disk image.
- You can make updates to the image and push the changes to a registry. See Managing RHEL bootc images.

4.8. CREATING ISO IMAGES BY USING BOOTC-IMAGE-BUILDER

You can use **bootc-image-builder** to create an ISO from which you can perform an offline deployment of a bootable container.

Prerequisites

- You have Podman installed on your host machine.
- You have root access to run the **bootc-image-builder** tool, and run the containers in **--privileged** mode, to build the images.

Procedure

1. Optional: Create a **config.toml** to configure user access, for example:

```
[[customizations.user]]
name = "user"
password = "pass"
```

```
key = "ssh-rsa AAA ... user@email.com"
groups = ["wheel"]
```

2. Run **bootc-image-builder**. If you do not want to add any configuration, omit the **-v $(pwd)/config.toml:/config.toml** argument.

NOTE

If you do not have the container storage mount and --local image options, your image must be public.

a. The following is an example of creating a public ISO image:

```
$ sudo podman run \
  --rm \
  -it \
  --privileged \
  --pull=newer \
  --security-opt label=type:unconfined_t \
  -v /var/lib/containers/storage:/var/lib/containers/storage \
  -v $(pwd)/config.toml:/config.toml \
  -v $(pwd)/output:/output \
  registry.redhat.io/rhel9/bootc-image-builder:latest \
  --type iso \
  --config /config.toml \
 quay.io/<namespace>/<image>:<tag>
```

b. The following is an example of creating a private ISO image:

```
$ sudo podman run \
  --rm \
  -it \
  --privileged \
  --pull=newer \
  --security-opt label=type:unconfined_t \
  -v $(pwd)/config.toml:/config.toml:ro \
  -v $(pwd)/output:/output \
  -v /var/lib/containers/storage:/var/lib/containers/storage \
  registry.redhat.io/rhel9/bootc-image-builder:latest \
  --local
  --type iso \
  quay.io/<namespace>/<image>:<tag>
```

You can find the **.iso** image in the output folder.

Next steps

- You can use the ISO image on unattended installation methods, such as USB sticks or Install-on-boot. The installable boot ISO contains a configured Kickstart file. See Deploying a container image by using Anaconda and Kickstart.

WARNING

Booting the ISO on a machine with an existing operating system or data can be destructive, because the Kickstart is configured to automatically reformat the first disk on the system.

- You can make updates to the image and push the changes to a registry. See Managing RHEL bootable images.

4.9. USING BOOTC-IMAGE-BUILDER TO BUILD ISO IMAGES WITH A KICKSTART FILE

You can use a Kickstart file to configure various parts of the installation process, such as setting up users, customizing partitioning, and adding an SSH key. You can include the Kickstart file in an ISO build to configure any part of the installation process, except the deployment of the base image. For ISOs with bootc container base images, you can use a Kickstart file to configure anything except the **ostreecontainer** command.

For example, you can use a Kickstart to perform either a partial installation, a full installation, or even omit the user creation. Use **bootc-image-builder** to build an ISO image that contains the custom Kickstart to configure your installation process.

Prerequisites

- You have Podman installed on your host machine.
- You have root access to run the **bootc-image-builder** tool, and run the containers in **--privileged** mode, to build the images.

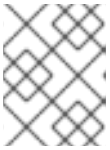
Procedure

1. Create your Kickstart file. The following Kickstart file is an example of a fully unattended Kickstart file configuration that contains user creation, and partition instructions.

```
[customizations.installer.kickstart]
contents = """
lang en_GB.UTF-8
keyboard uk
timezone CET

user --name <user> --password <password> --plaintext --groups <groups>
sshkey --username <user> ssh-<type> <public key>
rootpw --lock

zerombr
clearpart --all --initlabel
autopart --type=plain
reboot --eject
"""
```

2. Save the Kickstart configuration in the **toml** format to inject the Kickstart content. For example, **config.toml**.

3. Run **bootc-image-builder**, and include the Kickstart file configuration that you want to add to the ISO build. The **bootc-image-builder** automatically adds the **ostreecontainer** command that installs the container image.



NOTE

If you do not have the container storage mount and **--local** image options, your image must be public.

a. The following is an example of creating a public ISO image:

```
$ sudo podman run \
  --rm \
  -it \
  --privileged \
  --pull=newer \
  --security-opt label=type:unconfined_t \
  -v /var/lib/containers/storage:/var/lib/containers/storage \
  -v $(pwd)/config.toml:/config.toml \
  -v $(pwd)/output:/output \
  registry.redhat.io/rhel9/bootc-image-builder:latest \
  --type iso \
  --config /config.toml \
 quay.io/<namespace>/<image>:<tag>
```

b. The following is an example of creating a private ISO image:

```
$ sudo podman run \
  --rm \
  -it \
  --privileged \
  --pull=newer \
  --security-opt label=type:unconfined_t \
  -v $(pwd)/config.toml:/config.toml:ro \
  -v $(pwd)/output:/output \
  -v /var/lib/containers/storage:/var/lib/containers/storage \
  registry.redhat.io/rhel9/bootc-image-builder:latest \
  --local
  --type iso \
 quay.io/<namespace>/<image>:<tag>
```

You can find the **.iso** image in the output folder.

Next steps

- You can use the ISO image on unattended installation methods, such as USB sticks or Install-on-boot. The installable boot ISO contains a configured Kickstart file. See Deploying a container image by using Anaconda and Kickstart.

WARNING

Booting the ISO on a machine with an existing operating system or data can be destructive, because the Kickstart is configured to automatically reformat the first disk on the system.

- You can make updates to the image and push the changes to a registry. See Managing RHEL bootable images.

4.10. VERIFICATION AND TROUBLESHOOTING

If you have any issues configuring the requirements for your AWS image, see the following documentation

- AWS IAM account manager
- Using high-level (s3) commands with the AWS CLI .
- S3 buckets.
- Regions and Zones.
- Launching a customized RHEL image on AWS .

For more details on users, groups, SSH keys, and secrets, see

- Managing users, groups, SSH keys, and secrets in image mode for RHEL

CHAPTER 5. CUSTOMIZING DISK IMAGES OF RHEL IMAGE MODE WITH ADVANCED PARTITIONING

There are 2 options to customize advanced partitions:

- Disk customizations
- Filesystem customizations

However, the two customizations are incompatible with each other. You cannot use both customizations in the same blueprint.

5.1. UNDERSTANDING PARTITIONS

The following are the general principles about partitions:

- The full disk image size is always larger than the size of the sum of the partitions, due to requirements for headers and metadata. Consequently, all sizes are treated as minimum requirements, whether for specific filesystems, partitions, logical volumes, or the image itself.
- When the partition is automatically added, the partition that contains the root filesystem is always the last in the partition table layout. This is valid for a plain formatted partition, an LVM Volume Group, or a Btrfs partition. For disk customizations, the order that you defined is respected.
- For the raw partitioning, that is, with no LVM, the partition containing the root filesystem is grown to fill any leftover space on the partition table. Logical Volumes are not grown to fill the space in the Volume Group because they are simple to grow on a live system. Some directories have hard-coded minimum sizes which cannot be overridden. These are 1 GiB for / and 2 GiB for /**usr**. As a result, if /**usr** is not on a separate partition, the root filesystem size is at least 3 GiB.

5.2. THE DISK CUSTOMIZATIONS OPTION

The disk customizations option provides a more powerful interface to control the whole partitioning layout of the image.

Allowed mountpoints

When using **bootc-image-builder**, only the following directories allow customization:

- The / (root) directory.
- Custom directories under /**var**, but not /**var** itself.

Not allowed mountpoints

Under /**var**, the following mount points do not allow customization:

- /**var**/**home**
- /**var**/**lock**
- /**var**/**mail**
- /**var**/**mnt**

- **/var/roothome**
- **/var/run**
- **/var/srv**
- **/var/usrlocal**

5.3. DESCRIBING DISK CUSTOMIZATIONS IN A BLUEPRINT

When using the Disk customizations, you can describe the partition table almost entirely by using a blueprint. The customizations have the following structure:

- **Partitions**: The top level is a list of partitions.
  - **type**: Each partition has a type, which can be either **plain** or **lvm**. If the type is not set, it defaults to **plain**. The remaining required and optional properties of the partition depend on the type.
    - **plain**: A plain partition is a partition with a filesystem. It supports the following properties:
      - **fs_type**: The filesystem type, which should be one of **xfs**, **ext4**, **vfat**, or **swap**. Setting it to **swap** will create a swap partition. The mountpoint for a swap partition must be empty.
      - **minsize**: The minimum size of the partition, as an integer (in bytes) or a string with a data unit (for example 3 GiB). The final size of the partition in the image might be larger for specific mountpoints. See Understanding partitions section.
      - **mountpoint** The mountpoint for the filesystem. For swap partitions, this must be empty.
      - **label**: The label for the filesystem (optional).
    - **lvm**: An lvm partition is a partition with an LVM volume group. Only single Persistent Volumes volume groups are supported. It supports the following properties:
      - **name**: The name of the volume group (optional; if unset, a name will be generated automatically).
      - **minsize**: The minimum size of the volume group, as an integer (in bytes) or a string with a data unit (for example 3 GiB). The final size of the partition and volume group in the image might be larger if the value is smaller than the sum of logical volumes it contains.
      - **logical_volumes**: One or more logical volumes for the volume group. Each volume group supports the following properties:
        - **name**: The name of the logical volume (optional; if unset, a name will be generated automatically based on the mountpoint).
        - **minsize**: The minimum size of the logical volume, as an integer (in bytes) or a string with a data unit (for example 3 GiB). The final size of the logical volume in the image might be larger for specific mountpoints. See the General principles chapter for more details).

- **label**: The label for the filesystem (optional).
- **fs_type**: The filesystem type, which should be one of **xfs**, **ext4**, **vfat**, or **swap**. Setting it to swap will create a swap logical volume. The mountpoint for a swap logical volume must be empty.
- **mountpoint**: The mountpoint for the logical volume's filesystem. For swap logical volumes, this must be empty.

- **Order**:

The order of each element in a list is respected when creating the partition table. The partitions are created in the order they are defined, regardless of their type.

- **Incomplete partition tables:**

Incomplete partitioning descriptions are valid. Partitions, LVM logical volumes, are added automatically to create a valid partition table. The following rules are applied:

- A root filesystem is added if one is not defined. This is identified by the mountpoint /. If an LVM volume group is defined, the root filesystem is created as a logical volume, otherwise it will be created as a plain partition with a filesystem. The type of the filesystem, for plain and LVM, depends on the distribution (xfs for RHEL and CentOS, ext4 for Fedora). See Understanding partitions section for information about the sizing and order of the root partition.
- A boot partition is created if needed and if one is not defined. This is identified by the mountpoint /**boot**. A boot partition is needed when the root partition (mountpoint /) is on an LVM logical volume. It is created as the first partition after the ESP (see next item).
- An EFI system partition (ESP) is created if needed. This is identified by the mountpoint /**boot/efi**. An ESP is needed when the image is configured to boot with UEFI. This is defined by the image definition and depends on the image type, the distribution, and the architecture. The type of the filesystem is always **vfat**. By default, the ESP is 200 MiB and is the first partition after the BIOS boot (see next item).
- A 1 MiB unformatted BIOS boot partition is created at the start of the partition table if the image is configured to boot with BIOS. This is defined by the image definition and depends on the image type, the distribution, and the architecture. Both a BIOS boot partition and an ESP are created for images that are configured for hybrid boot.

- **Combining partition types:**

You can define multiple partitions. The following combination of partition types are valid:

- **plain** and **lvm**: Plain partitions can be created alongside an LVM volume group. However, only one LVM volume group can be defined.
- **Examples:** Blueprint to define two partitions

The following blueprint defines two partitions. The first is a 50 GiB partition with an **ext4** filesystem that will be mounted at /**data**. The second is an LVM volume group with three logical volumes, one for root /, one for home directories /**home**, and a **swap** space in that order. The LVM volume group will have 15 GiB of non-allocated space.

```
[[customizations.disk.partitions]]
type = "plain"
label = "data"
```

```
mountpoint = "/data"
fs_type = "ext4"
minsize = "50 GiB"

[[customizations.disk.partitions]]
type = "lvm"
name = "mainvg"
minsize = "20 GiB"

[[customizations.disk.partitions.logical_volumes]]
name = "rootlv"
mountpoint = "/"
label = "root"
fs_type = "ext4"
minsize = "2 GiB"

[[customizations.disk.partitions.logical_volumes]]
name = "homelv"
mountpoint = "/home"
label = "home"
fs_type = "ext4"
minsize = "2 GiB"

[[customizations.disk.partitions.logical_volumes]]
name = "swaplv"
fs_type = "swap"
minsize = "1 GiB"
```

5.4. THE FILESYSTEM CUSTOMIZATION OPTION

The filesystem customization option provides the final partition table of an image that you built with image builder, and it is determined by a combination of the following factors:

- The base partition table for a given image type.
- The relevant blueprint customizations:
  - Partitioning mode.
  - Filesystem customizations.
- The image size parameter of the build request:
  - On the command line, this is the **--size** option of the **composer-cli compose start** command.

The following describes how these factors affect the final layout of the partition table.

- **Modifying partition tables**

You can modify the partition table by taking the following aspects in consideration:

- **Partitioning modes**

The partitioning mode controls how the partition table is modified from the image type's default layout.

- The **raw** partition type does not convert any partition to LVM.
- The **lvm** partition type always converts the partition that contains the / root mountpoint to an LVM Volume Group and creates a root Logical Volume. Except from /**boot**, any extra mountpoint is added to the Volume Group as new Logical Volumes.
- The **auto-lvm** mode is the default mode and converts a raw partition table to an LVM-based one if and only if new mountpoints are defined in the filesystems customization. See the **Mountpoints** entry for more details.
    - **Mountpoints**

You can define new filesystems and minimum partition sizes by using the filesystems customization in the blueprint. By default, if new mountpoints are created, a partition table is automatically converted to LVM. See the **Partitioning modes** entry for more details.

- **Image size** The minimum size of the partition table is the size of the disk image. The final size of the image will either be the value of the size parameter or the sum of all partitions and their associated metadata, depending on which one is the larger.

5.5. CREATING IMAGES WITH SPECIFIC SIZES

To create a disk image of a very specific size, you must ensure that the following requirements are met:

- You must specify the exact **[Image size]** in the build request.
- The mountpoints that you define as customizations must specify sizes that are smaller than the total size in sum. This is required, because the partition table, partitions, and other entities often require extra space for metadata and headers, so the space required to fit all the mountpoints is always larger than the sum of the size of the partitions. However, the exact size of the extra space that is required varies based on many factors, such as image type, for example.

The following are overall steps to create a disk image of a very specific size in the TOML file:

1. Set the **[Image size]** parameter to the size that you want.
2. Add any extra mountpoints with their required minimum sizes. Ensure that the sum of the sizes is smaller than the image size by at least 3.01 GiB if there is no /**usr** mountpoint, or at least 1.01 GiB if there is. The extra 0.01 MiB is more than enough for the headers and metadata for which extra space might be reserved.
3. Do not specify a size for the / mountpoint.

With this, you create a disk with a partition table of the desired size with each partition sized to fit the desired mountpoints. The root partition, root LVM Logical Volume, will be at least 3 GiB, or 1 GiB if /**usr** is specified. See Understanding partitions for more details.

- If the partition table does not have any LVM Volume Groups (VG), the root partition will be grown to fill the remaining space.
- If the partition table contains an LVM Volume Group (VG), the VG will have unallocated extents that can be used to grow any of the Logical Volumes.

5.6. USING BOOTC-IMAGE-BUILDER TO ADD WITH ADVANCED PARTITIONING TO DISK IMAGES OF IMAGE MODE

You can customize your bootc-image-builder blueprint to implement advanced partitioning for **osbuild-composer**. The following are possible custom mountpoints:

- You can create LVM-based images under all partitions on LVs except, /**boot** and /**boot/efi**.
- You can create an LV-based swap.
- You can give VGs and LVs custom names.

Include partitioning configurations in the base image that **bootc-image-builder** will read to create the partitioning layout, making the container itself the source of truth for the partition table. Mountpoints for partitions and logical volumes should be created in the base container image used to build the disk. This is particularly important for top-level mountpoints, such as the /**app** mountpoint. The **bootc-image-builder** will validate the configuration against the bootc container before building, in order to avoid creating unbootable images.

Prerequisites

- You have Podman installed on your host machine.
- You have root access to run the **bootc-image-builder** tool, and run the containers in **--privileged** mode, to build the images.
- QEMU is installed.

Procedure

1. Create a **config.tom** file with the following content:
   a. Add a user to the image.

```
[[customizations.user]]
name = "user1"
password = "user2"
key = "ssh-rsa AAA ... user@email.com"
groups = ["wheel"]

# Set a size for the root partition:

[[customizations.partitioning.plain.filesystems]]
mountpoint = "/"
type = "ext4"
minsize = "3 GiB"

# Add an extra data partition
[[customizations.partitioning.plain.filesystems]]
mountpoint = "/var/data"
type = "xfs"
minsize = "2 GiB"
label = "data"


# Add an app partition with a top-level mountpoint.
# Requires derived container.
[[customizations.partitioning.plain.filesystems]]
mountpoint = "/app"
```

```
type = "xfs"
minsize = "1 GiB"
label = "app"

# Add the LVM configuration:
# Define the LVM Volume Group name and size
[[customizations.partitioning.lvm.volume_groups]]
name = "centosvg"
minsize = 10737418240  # 10 GiB


# Add a data Logical Volume to the group
[[customizations.partitioning.lvm.volume_groups.logical_volumes]]
name = "datalv"
mountpoint = "/var/data"
label = "data"
minsize = "1 GiB"
type = "xfs"


# The root Logical Volume is created automatically if not defined, but setting
# it lets us set the name, label, and size explicitly
[[customizations.partitioning.lvm.volume_groups.logical_volumes]]
name = "rootlv"
mountpoint = "/"
label = "system"
minsize = "2 GiB"
type = "ext4"


# Add an app Logical Volume with a top-level mountpoint.
# Requires derived container.
[[customizations.partitioning.lvm.volume_groups.logical_volumes]]
mountpoint = "/app"
type = "xfs"
minsize = "1 GiB"
label = "app"
name = "applv"
```

2. Run the **bootc-image-builder**. Optionally, if you want to use user access configuration, pass the **config.toml** as an argument. The following is an example of creating a public QCOW2 image:

```
sudo podman run \
    --rm \
    -it \
    --privileged \
    --pull=newer \
    --security-opt label=type:unconfined_t \
    -v ./config.toml:/config.toml:ro \
    -v ./output:/output \
     registry.redhat.io/rhel9/bootc-image-builder:latest \
    --type qcow2 \
    --config /config.toml \
  quay.io/<namespace>/<image>:<tag>
```

You can find the **.qcow2** image in the output folder.

Verification

1. Run the resulting QCOW2 file on a virtual machine.

```
qemu-system-x86_64 \
    -enable-kvm \
    -cpu host \
    -m 8G \
    -bios /usr/share/edk2/ovmf/OVMF_CODE.fd \
    -snapshot \
    -drive file="${path}/output/qcow2/disk.qcow2"
```

2. Access the system in the virtual machine launched with SSH.

```
# ssh -i /<path_to_private_ssh-key> <user1>@<ip-address>
```

Next steps

- You can deploy your image. See Deploying a container image using KVM with a QCOW2 disk image.
- You can make updates to the image and push the changes to a registry. See Managing RHEL bootc images.

5.7. BUILDING DISK IMAGES OF IMAGE MODE RHEL WITH ADVANCED PARTITIONING

Create image mode disk images with advanced partitioning by using **bootc-image-builder**. The image mode disk images that you create of image mode RHEL with custom mount points, include custom mount options, LVM-based partitions and LVM-based SWAP. With that you can, for example, change the size of the / and the /**boot** directories by using a **config.toml** file. When installing the RHEL image mode on bare-metal machines, you can benefit from all partitioning features available on Anaconda.

Prerequisites

- You have Podman installed on your host machine.
- You have **virt-install** installed on your host machine.
- You have root access to run the **bootc-image-builder** tool, and run the containers in **--privileged** mode, to build the images.

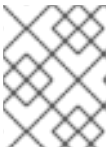
Procedure

1. Create a **config.toml** to configure custom mount options, for example:

```
name = "lvm"
description = "A base system with custom LVM partitioning"

[customizations.disk]
```

```
[[customizations.disk.partitions]]
mountpoint = "/var/data"
minsize = "1 GiB"
label = "data"
fs_type = "ext4"

[[customizations.disk.partitions]]
mountpoint = "/"
minsize = "2 GiB"
label = "root"
fs_type = "ext4"
```

2. Run **bootc-image-builder**, passing the **config.toml** as an argument.



NOTE

If you do not have the container storage mount and --local image options, your image must be public.

```
$ sudo podman run \
  --rm \
  -it \
  --privileged \
  --pull=newer \
  --security-opt label=type:unconfined_t \
  -v ./config.toml:/config.toml \
  -v ./output:/output \
  registry.redhat.io/rhel9/bootc-image-builder:latest \
  --type <image_type>\
  --config config.toml \
quay.io/<namespace>/<image>:<tag>
```

You can find your image with the customized advanced partitioning in the output folder.

Next steps

- Deploy the disk image with advanced partitioning layout. See Deploying your customized images.

Additional resources

- link: Creating an LVM2 logical volume for swap

CHAPTER 6. BEST PRACTICES FOR RUNNING CONTAINERS USING LOCAL SOURCES

You can access content hosted in an internal registry that requires a custom Transport Layer Security (TLS) root certificate, when running RHEL bootc images.

There are two options available to install content to a container by using only local resources:

- Bind mounts: Use for example **-v /etc/pki:/etc/pki** to override the container's store with the host's.
- Derived image: Create a new container image with your custom certificates by building it using a **Containerfile**.

You can use the same techniques to run a **bootc-image-builder** container or a **bootc** container when appropriate.

6.1. IMPORTING CUSTOM CERTIFICATE TO A CONTAINER BY USING BIND MOUNTS

Use bound mounts to override the container's store with the host's.

Procedure

- Run RHEL bootc image and use bind mount, for example **-v /etc/pki:/etc/pki**, to override the container's store with the host's:

```
# podman run \
  --rm \
  -it \
  --privileged \
  --pull=newer \
  --security-opt label=type:unconfined_t \
  -v $(pwd)/output:/output \
  -v /etc/pki:/etc/pki \
  localhost/<image> \
  --type iso \
  --config /config.toml \
  quay.io/<namespace>/<image>:<tag>
```

Verification

- List certificates inside the container:

```
# ls -l /etc/pki
```

6.2. IMPORTING CUSTOM CERTIFICATES TO A CONTAINER BY USING CONTAINERFILE

Create a new container image with your custom certificates by building it using a **Containerfile**.

Procedure

1. Create a **Containerfile**:

```
FROM <internal_repository>/<image>
RUN  mkdir -p /etc/pki/ca-trust/extracted/pem/
COPY tls-ca-bundle.pem /etc/pki/ca-trust/extracted/pem/
RUN  rm -rf /etc/yum.repos.d/*
COPY echo-rhel9_4.repo /etc/yum.repos.d/
```

2. Build the custom image:

```
# podman build -t <your_image> .
```

3. Run the **<your_image>**:

```
# podman run -it --rm <your_image>
```

Verification

- List the certificates inside the container:

```
# ls -l /etc/pki/ca-trust/extracted/pem/
tls-ca-bundle.pem
```

CHAPTER 7. DEPLOYING THE RHEL BOOTC IMAGES

You can deploy the **rhel-bootc** container image by using the following different mechanisms.

- Anaconda
- **bootc-image-builder**
- **bootc install**

The following bootc image types are available:

- Disk images that you generated by using the **bootc image-builder** such as:
  - QCOW2 (QEMU copy-on-write, virtual disk)
  - Raw (Mac Format)
  - AMI (Amazon Cloud)
  - ISO: Unattended installation method, by using an USB Sticks or Install-on-boot.

After you have created a layered image that you can deploy, there are several ways that the image can be installed to a host:

- You can use RHEL installer and Kickstart to install the layered image to a bare metal system, by using the following mechanisms:
  - Deploy by using USB
  - PXE
- You can also use **bootc-image-builder** to convert the container image to a bootc image and deploy it to a bare metal or to a cloud environment.

The installation method happens only one time. After you deploy your image, any future updates will apply directly from the container registry as the updates are published.

Figure 7.1. Deploying a bootc image by using a basic build installer, bootc install, or deploying a container image by using Anaconda and Kickstart

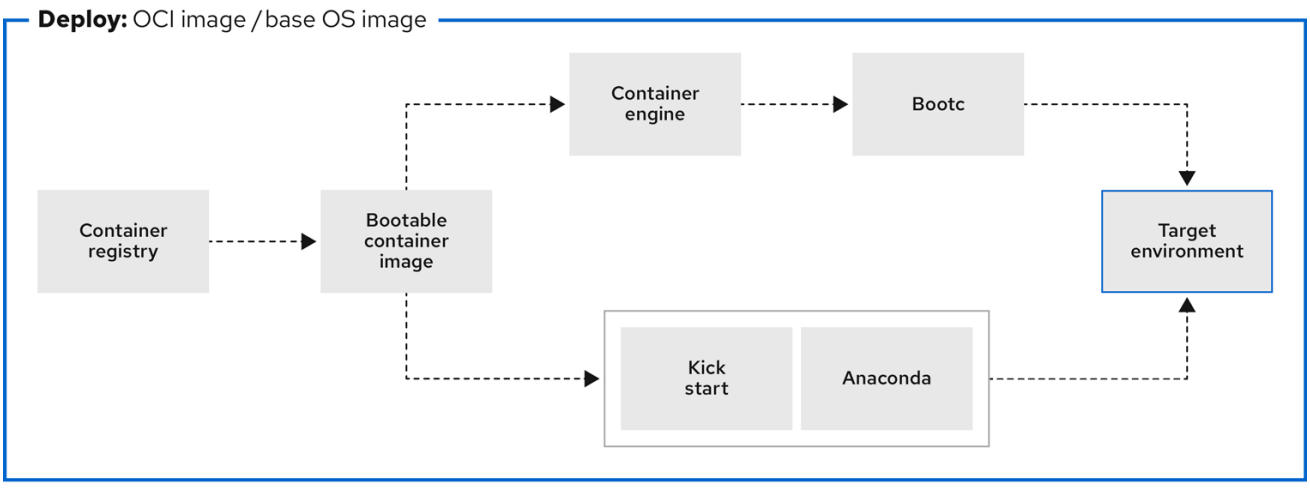
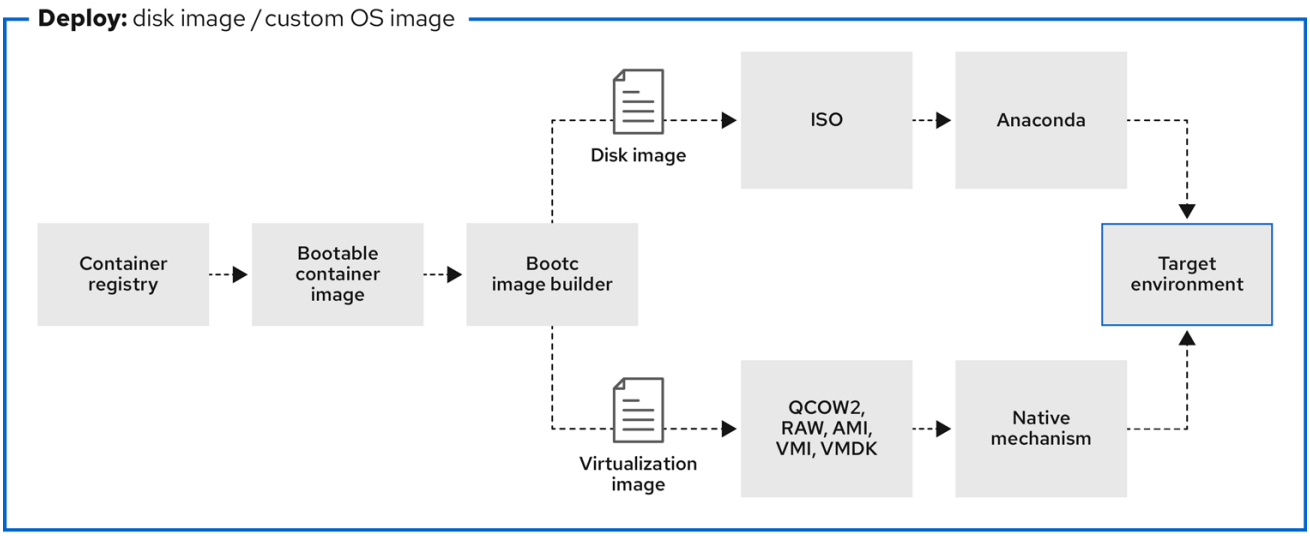


Figure 7.2. Using bootc-image-builder to create disk images from bootc images and deploying disk images in different environments, such as the edge, servers, and clouds by using Anaconda, bootc-image-builder or bootc install



7.1. DEPLOYING A CONTAINER IMAGE BY USING KVM WITH A QCOW2 DISK IMAGE

After creating a QEMU disk image from a RHEL bootc image by using the **bootc-image-builder** tool, you can use a virtualization software to boot it.

Prerequisites

- You created a container image. See Creating QCOW2 images by using bootc-image-builder .
- You pushed the container image to an accessible repository.

Procedure

- Run the container image that you create by using either **libvirt**. See Creating virtual machines by using the command line for more details.
  - The following example uses **libvirt**:

```
$ sudo virt-install \
  --name bootc \
  --memory 4096 \
  --vcpus 2 \
  --disk qcow2/disk.qcow2 \
  --import \
  --os-variant rhel9-unknown
```

Verification

- Connect to the VM in which you are running the container image. See Connecting to virtual machines for more details.

Next steps

- You can make updates to the image and push the changes to a registry. See Managing RHEL bootc images.

Additional resources

- Configuring and managing virtualization

7.2. DEPLOYING A CONTAINER IMAGE TO AWS WITH AN AMI DISK IMAGE

After using the **bootc-image-builder** tool to create an AMI from a bootc image, and uploading it to a AWS s3 bucket, you can deploy a container image to AWS with the AMI disk image.

Prerequisites

- You created an Amazon Machine Image (AMI) from a bootc image. See Creating AMI images by using bootc-image-builder and uploading it to AWS.
- **cloud-init** is available in the Containerfile that you previously created so that you can create a layered image for your use case.

Procedure

1. In a browser, access Service→EC2 and log in.
2. On the AWS console dashboard menu, choose the correct region. The image must have the **Available** status, to indicate that it was correctly uploaded.
3. On the AWS dashboard, select your image and click **Launch**.
4. In the new window that opens, choose an instance type according to the resources you need to start your image. Click **Review and Launch**.
5. Review your instance details. You can edit each section if you need to make any changes. Click **Launch**.
6. Before you start the instance, select a public key to access it. You can either use the key pair you already have or you can create a new key pair.
7. Click **Launch Instance** to start your instance. You can check the status of the instance, which displays as **Initializing**.
   After the instance status is **Running**, the **Connect** button becomes available.
8. Click **Connect**. A window appears with instructions on how to connect by using SSH.
9. Run the following command to set the permissions of your private key file so that only you can read it. See Connect to your Linux instance .

   ```
   $ chmod 400 <your-instance-name.pem>
   ```

10. Connect to your instance by using its Public DNS:

    ```
    $ ssh -i <your-instance-name.pem>ec2-user@<your-instance-IP-address>
    ```

NOTE

Your instance continues to run unless you stop it.

Verification

After launching your image, you can:

- Try to connect to http://*<your_instance_ip_address>* in a browser.
- Check if you are able to perform any action while connected to your instance by using SSH.

Next steps

- After you deploy your image, you can make updates to the image and push the changes to a registry. See Managing RHEL bootc images.

Additional resources

- Pushing images to AWS Cloud AMI
- Amazon Machine Images (AMI)

7.3. DEPLOYING A CONTAINER IMAGE FROM THE NETWORK BY USING ANACONDA AND KICKSTART

You can deploy an ISO image by using Anaconda and Kickstart to install your container image. The installable boot ISO already contains the **ostreecontainer** Kickstart file configured that you can use to provision your custom container image.

Prerequisites

- You have downloaded the 9.4 Boot ISO for your architecture from Red Hat. See Downloading RH boot images.

Procedure

1. Create an **ostreecontainer** Kickstart file to fetch the image from the network. For example:

```
# Basic setup
text
network --bootproto=dhcp --device=link --activate
# Basic partitioning
clearpart --all --initlabel --disklabel=gpt
reqpart --add-boot
part / --grow --fstype xfs

# Reference the container image to install - The kickstart
# has no %packages section. A container image is being installed.
ostreecontainer --url quay.io/<namespace>/<image>:<tag> . bootc-image-builder:latest

firewall --disabled
services --enabled=sshd
```

```
# Only inject a SSH key for root
rootpw --iscrypted locked
sshkey --username root "<your-key>"
reboot
```

2. Boot a system by using the 9.4 Boot ISO installation media.
   a. Append the Kickstart file with the following to the kernel argument:

   ```
   inst.ks=http://<path_to_your_kickstart>
   ```

3. Press **CTRL+X** to boot the system.

Next steps

- After you deploy your container image, you can make updates to the image and push the changes to a registry. See Managing RHEL bootc images.

Additional resources

- ostreecontainer documentation
- bootc upgrade fails when using local rpm-ostree modifications (Red Hat Knowledgebase)

7.4. DEPLOYING A CUSTOM ISO CONTAINER IMAGE IN DISCONNECTED ENVIRONMENTS

By using **using bootc-image-builder** to convert a bootc image to an ISO image, you create a system similar to the RHEL ISOs available for download, except that your container image content is embedded in the ISO disk image. You do not need to have access to the network during installation. You can install the ISO disk image that you created from **bootc-image-builder** to a bare metal system.

Prerequisites

- You have created an ISO image with your bootc image embedded.

Procedure

1. Copy your ISO disk image to a USB flash drive.
2. Perform a bare-metal installation by using the content in the USB stick into a disconnected environment.

Next steps

- After you deploy your container image, you can make updates to the image and push the changes to a registry. See Managing RHEL bootc images.

7.5. DEPLOYING AN ISO BOOTC IMAGE OVER PXE BOOT

You can use a network installation to deploy the RHEL ISO image over PXE boot to run your ISO bootc image.

Prerequisites

Prerequisites

- You have downloaded the 9.4 Boot ISO for your architecture from Red Hat. See Downloading RH boot images.
- You have configured the server for the PXE boot. Choose one of the following options:
  - For HTTP clients, see Configuring the DHCPv4 server for HTTP and PXE boot .
  - For UEFI-based clients, see Configuring a TFTP server for UEFI-based clients .
  - For BIOS-based clients, see Configuring a TFTP server for BIOS-based clients .
- You have a client, also known as the system to which you are installing your ISO image.

Procedure

1. Export the RHEL installation ISO image to the HTTP server. The PXE boot server is now ready to serve PXE clients.
2. Boot the client and start the installation.
3. Select PXE Boot when prompted to specify a boot source. If the boot options are not displayed, press the Enter key on your keyboard or wait until the boot window opens.
4. From the Red Hat Enterprise Linux boot window, select the boot option that you want, and press Enter.
5. Start the network installation.

Next steps

- You can make updates to the image and push the changes to a registry. See Managing RHEL bootc images.

Additional resources

- Preparing to install from the network using PXE
- Booting the installation from a network by using PXE

7.6. INJECTING CONFIGURATION IN THE RESULTING DISK IMAGES WITH BOOTC-IMAGE-BUILDER

You can inject configuration into a custom image by using a **build config**, that is, a **.toml** or a **.json file with customizations for the resulting image. The `build config** file is mapped into the container directory to **/config.toml**. The following example shows how to add a user to the resulting disk image:

Procedure

1. Create a **./config.toml**. The following example shows how to add a user to the disk image.

```
[[customizations.user]]
name = "user"
password = "pass"
```

```
key = "ssh-rsa AAA ... user@email.com"
groups = ["wheel"]
```

- **name** - Mandatory. Name of the user.
- **password** - Not mandatory. Nonencrypted password.
- **key** - Not mandatory. Public SSH key contents.
- **groups** - Not mandatory. An array of groups to add the user into.

2. Run **bootc-image-builder** and pass the following arguments, including the **config.toml**:

```
$ sudo podman run \
    --rm \
    -it \
    --privileged \
    --pull=newer \
    --security-opt label=type:unconfined_t \
    -v $(pwd)/config.toml:/config.toml \
    -v $(pwd)/output:/output \
    registry.redhat.io/rhel9/bootc-image-builder:latest \
    --type qcow2 \
    --config config.toml \
    quay.io/<namespace>/<image>:<tag>
```

3. Launch a VM, for example, by using **virt-install**:

```
$ sudo virt-install \
  --name bootc \
  --memory 4096 \
  --vcpus 2 \
  --disk qcow2/disk.qcow2 \
  --import \
  --os-variant rhel9
```

Verification

- Access the system with SSH:

```
# ssh -i /<path_to_private_ssh-key> <user1>_@_<ip-address>
```

Next steps

- After you deploy your container image, you can make updates to the image and push the changes to a registry. See Managing RHEL bootable images.

7.7. DEPLOYING A CONTAINER IMAGE TO BARE METAL BY USING BOOTC INSTALL

You can perform a bare-metal installation to a device by using a RHEL ISO image. Bootc contains a basic build installer and it is available as the following methods: **bootc install to-disk** or **bootc install to-filesystem**.

- **bootc install to-disk**: By using this method, you do not need to perform any additional steps to deploy the container image, because the container images include a basic installer.
- **bootc install to-filesystem**: By using this method, you can configure a target device and root filesystem by using a tool of your choice, for example, LVM.

Prerequisites

- You have downloaded a RHEL 10 Boot ISO from Red Hat for your architecture. See Downloading RHEL boot images.
- You have created a configuration file.

Procedure

- Inject a configuration into the running ISO image.
  - By using **bootc install to-disk**:

```
$ podman run \
--rm --privileged \
--pid=host
-v /dev:/dev \
-v /var/lib/containers:/var/lib/containers \
--security-opt label=type:unconfined_t
<image>
bootc install to-disk <path-to-disk>
```

  - By using **bootc install to-filesystem**:

```
$ podman run \
--rm --privileged \
--pid=host
-v /:/target \
-v /dev:/dev \
-v /var/lib/containers:/var/lib/containers \
--security-opt label=type:unconfined_t
<image>
bootc install to-filesystem <path-to-disk>
```

Next steps

- After you deploy your container image to a bare-metal environment, you can make updates to the image and push the changes to a registry. See Managing RHEL bootable images.

7.8. DEPLOYING A CONTAINER IMAGE BY USING A SINGLE COMMAND

The **system-reinstall-bootc** command provides an interactive CLI that wraps the **bootc install to-existing root** command. You can deploy a container image into a RHEL cloud instance by using a signal command. The **system-reinstall-bootc** command performs the following actions:

- Pull the supplied image to set up SSH keys or access the system.

- Run the **bootc install to-existing-root** command with all the bind mounts and SSH keys configured.

The following procedure deploys a bootc image to a new RHEL 10 instance on AWS. When launching the instance, make sure to select your SSH key, or create a new one. Otherwise, the default instance configuration settings can be used.

Prerequisites

- Red Hat Account or Access to Red Hat RPMS
- A package-based RHEL (9.6 / 10.0 or greater) virtual system running in an AWS environment.
- Ability and permissions to SSH into the package system and make "destructive changes."

Procedure

1. After the instance starts, connect to it by using SSH using the key you selected when creating the instance:

```
$ ssh -i <ssh-key-file> <cloud-user@ip>
```

2. Make sure that the **system-reinstall-bootc** subpackage is installed:

```
# rpm -q system-reinstall-bootc
```

   If not, install the **system-reinstall-bootc** subpackage:

```
# dnf -y install system-reinstall-bootc
```

3. Convert the system to use a bootc image:

```
# system-reinstall-bootc <image>
```

   - You can use the container image from the Red Hat Ecosystem Catalog or the customized bootc image built from a Containerfile.

4. Select users to import to the bootc image by pressing the "a" key.
5. Confirm your selection twice and wait until the image is downloaded.
6. Reboot the system:

```
# reboot
```

7. Remove the stored SSH host key for the given **<ip>** from your **/.ssh/known_hosts** file:

```
# ssh-keygen -R <ip>
```

   The bootc system is now using a new public SSH host key. When attempting to connect to the same IP address with a different key than what is stored locally, SSH will raise a warning or refuse the connection due to a host key mismatch. Since this change is expected, the existing host key entry can be safely removed from the **~/.ssh/known_hosts** file using the following command.

8. Connect to the bootc system:

```
# ssh -i <ssh-key-file> root@<ip>
```

Verification

- Confirm that the system OS has changed:

```
# bootc status
```

7.9. ADVANCED INSTALLATION WITH TO-FILESYSTEM

The **bootc install** contains two subcommands: **bootc install to-disk** and **bootc install to-filesystem**.

- The **bootc-install-to-filesystem** performs installation to the target filesystem.
- The **bootc install to-disk** subcommand consists of a set of opinionated lower level tools that you can also call independently. The command consist of the following tools:
  - **mkfs.$fs /dev/disk**
  - **mount /dev/disk /mnt**
  - **bootc install to-filesystem --karg=root=UUID=*<uuid of /mnt>* --imgref $self /mnt**

7.9.1. Using bootc install to-existing-root

The **bootc install to-existing-root** is a variant of **install to-filesystem**. You can use it to convert an existing system into the target container image.

WARNING

This conversion eliminates the **/boot** and **/boot/efi** partitions and can delete the existing Linux installation. The conversion process reuses the filesystem, and even though the user data is preserved, the system no longer boots in package mode.

Prerequisites

- You must have root permissions to complete the procedure.
- You must match the host environment and the target container version, for example, if your host is a RHEL 9 host, then you must have a RHEL 9 container. Installing a RHEL container on a Fedora host by using **btrfs** as the RHEL kernel will not support that filesystem.

Procedure

- Run the following command to convert an existing system into the target container image. Pass the target **rootfs** by using the **-v /:/target** option.

```
# podman run --rm --privileged -v /dev:/dev -v /var/lib/containers:/var/lib/containers -v
/:/target \
        --pid=host --security-opt label=type:unconfined_t \
        <image> \
        bootc install to-existing-root
```

This command deletes the data in **/boot**, but everything else in the existing operating system is not automatically deleted. This can be useful because the new image can automatically import data from the previous host system. Consequently, container images, database, the user home directory data, configuration files in **/etc** are all available after the subsequent reboot in **/sysroot**.

You can also use the **--root-ssh-authorized-keys** flag to inherit the root user SSH keys, by adding **--root-ssh-authorized-keys /target/root/.ssh/authorized_keys**. For example:

```
# podman run --rm --privileged -v /dev:/dev -v /var/lib/containers:/var/lib/containers -v
/:/target \
        --pid=host --security-opt label=type:unconfined_t \
        <image> \
        bootc install to-existing-root --root-ssh-authorized-keys
/target/root/.ssh/authorized_keys
```

CHAPTER 8. CREATING BOOTC IMAGES FROM SCRATCH

With bootc images from scratch, you can have control over the underlying image content, and tailor your system environment to your requirements.

You can use the **bootc-base-imagectl** command to create a bootc image from scratch by using an existing bootc base image as a build environment, providing greater control over the content included in the build process. This process takes the user RPMs as input, so you need to rebuild the image if the RPMs change.

The custom base derives from the base container, and does not automatically consume changes to the default base image unless you make them part of a container pipeline.

You can use the **bootc-base-imagectl rechunk** subcommand on any bootc container image.

If you want to perform kernel management, you do not need to create a bootc image from scratch. See Managing kernel arguments in bootc systems .

8.1. USING PINNED CONTENT TO BUILD IMAGES

To ensure the base image version contains a set of packages at exactly specific versions, for example, defined by a lockfile, or an **rpm-md** or **yum repository**, you can use several tools to manage snapshots of **rpm-md** or **yum repository** repositories.

With the **bootc image from scratch** feature, you can configure and override package information in source RPM repositories, while referencing mirrored, pinned, or snapshotted repository content. Consequently, you gain control over package versions and their dependencies.

For example, you might want to gain control over package versions and their dependencies in the following situations:

- You need to use a specific package version because of strict certification and compliance requirements.
- You need to use specific software versions to support critical dependencies.

Prerequisites

- A standard bootc base image.

Procedure

- The following example creates a bootc image from scratch with pinned content:

```
# Begin with a standard bootc base image that serves as a "builder" for our custom image.
FROM registry.redhat.io/rhel10/rhel-bootc:latest
# Configure and override source RPM repositories, if necessary. The following step is required when referencing specific content views or target mirrored/snapshotted/pinned versions of content.
RUN rm -vf /etc/yum.repos.d
COPY mypinnedcontent.repo /etc/yum.repos
# Add additional repositories to apply customizations to the image. However, referencing a custom manifest in this step is not currently supported without forking the code.
# Build the root file system by using the specified repositories and non-RPM content from the "builder" base image.
```

```
# If no repositories are defined, the default build will be used. You can modify the scope of packages in the base image by changing the manifest between the "standard" and "minimal" sets.
RUN /usr/libexec/bootc-base-imagectl build-rootfs --manifest=standard /target-rootfs
# Create a new, empty image from scratch.
FROM scratch
# Copy the root file system built in the previous step into this image.
COPY --from=builder /target-rootfs/ /
# Apply customizations to the image. This syntax uses "heredocs" https://www.docker.com/blog/introduction-to-heredocs-in-dockerfiles/ to pass multi-line arguments in a more readable format.
RUN <<EORUN
# Set pipefail to display failures within the heredoc and avoid false-positive successful builds.
set -xeuo pipefail
# Install necessary packages, run scripts, etc.
dnf -y install NetworkManager emacs
# Remove leftover build artifacts from installing packages in the final built image.
dnf clean all
rm /var/{log,cache,lib}/* -rf
EORUN
# Define required labels for this bootc image to be recognized as such.
LABEL containers.bootc 1
LABEL ostree.bootable 1
# Optional labels that only apply when running this image as a container. These keep the default entry point running under systemd.
STOPSIGNAL SIGRTMIN+3
CMD ["/sbin/init"]
# Run the bootc linter to avoid encountering certain bugs and maintain content quality. Place this command last in your Containerfile.
RUN bootc container lint
```

Verification steps

1. Save and build your image.

```
$ podman build -t quay.io/<namespace>/<image>:<tag> . --cap-add=all --security-opt=label=type:container_runtime_t --device /dev/fuse
```

2. Build <_image_> image by using the **Containerfile** in the current directory:

```
$ podman build -t quay.io/<namespace>/<image>:<tag> .
```

8.2. BUILDING A BASE IMAGE UP FROM MINIMAL

Previously, you could build just a standard image by using image mode for RHEL. The standard image is roughly a headless server-oriented installation, although you can use it for desktops as well, and includes many opinionated packages for networking, CLI tool, among others.

You now have the option to generate from the standard image a new minimal image which only starts from bootc, kernel, and dnf. This image can then be extended further in a multi-stage build. At the current time the minimal image is not shipped pre-built in the registry.

The base images include the **/usr/libexec/bootc-base-imagectl** tool that enables you to generate a custom base image. By using the tool, you can build a root file system that is based on the RPM packages that you selected in the base image.

Prerequisites

- A standard bootc base image.

Procedure

- The following example creates a custom minimal base image:

```
# Begin with a standard bootc base image that is reused as a "builder" for the custom image.
FROM registry.redhat.io/rhel10/rhel-bootc:latest
# Configure and override source RPM repositories, if necessary. This step is not required when building up from minimal unless referencing specific content views or target mirrored/snapshotted/pinned versions of content.
# Add additional repositories to apply customizations to the image. However, referencing a custom manifest in this step is not currently supported without forking the code.
# Build the root file system by using the specified repositories and non-RPM content from the "builder" base image.
# If no repositories are defined, the default build will be used. You can modify the scope of packages in the base image by changing the manifest between the "standard" and "minimal" sets.
RUN /usr/libexec/bootc-base-imagectl build-rootfs --manifest=minimal /target-rootfs
# Create a new, empty image from scratch.
FROM scratch
# Copy the root file system built in the previous step into this image.
COPY --from=builder /target-rootfs/ /
# Apply customizations to the image. This syntax uses "heredocs" https://www.docker.com/blog/introduction-to-heredocs-in-dockerfiles/ to pass multi-line arguments in a more readable format.
RUN <<EORUN
# Set pipefail to display failures within the heredoc and avoid false-positive successful builds.
set -xeuo pipefail
# Install required packages for our custom bootc image. Note that using a minimal manifest means we need to add critical components specific to our use case and environment.
dnf -y install NetworkManager cowsay
# Remove leftover build artifacts from installing packages from the final built image.
dnf clean all
rm /var/{log,cache,lib}/* -rf
# Close the shell command.
EORUN
# Define required labels for this bootc image to be recognized as such.
LABEL containers.bootc 1
LABEL ostree.bootable 1
# Optional labels that only apply when running this image as a container. These keep the default entry point running under systemd.
STOPSIGNAL SIGRTMIN+3
CMD ["/sbin/init"]
# Run the bootc linter to avoid encountering certain bugs and maintain content quality. Place this command last in your Containerfile.
RUN bootc container lint
```

8.3. BUILDING REQUIRED PRIVILEGES

If you want to generate a new root file system, and do not modify the existing container, you can use container features, such as the mount namespacing, that are not enabled by default in many container build environments, to be able to gain building required privileges.

Prerequisites

- The **container-tools** meta-package is installed.

Procedure

- Generate a new root file system, providing these arguments at a minimum to **podman build**:

```
--cap-add=all --security-opt=label=type:container_runtime_t --device /dev/fuse
```

8.4. GENERATING YOUR BOOTC IMAGES FROM SCRATCH

Create bootc images from scratch from a custom RHEL bootc default base container image to get a small root content set.

Prerequisites

- The **container-tools** metapackage is installed.

Procedure

- Create a **Containerfile**. The following is an example:

```
# The following example reuses the default base image as a "builder" image. Optionally,  you can use the commented instructions to configure or override the RPM repositories in /etc/yum.repos.d to, for example, refer to pinned versions
FROM registry.redhat.io/rhel10/rhel-bootc:latest
# RUN rm -rf /etc/yum.repos.d/*
# COPY mycustom.repo /etc/yum.repos.d
RUN /usr/libexec/bootc-base-imagectl build-rootfs --manifest=minimal /target-rootfs
# Create a new, empty image from scratch.
FROM scratch
# Copy the root file system built in the previous step into this image.
COPY --from=builder /target-rootfs/ /
# You can make arbitrary changes such as copying the systemd units and other tweaks from the baseconfig container image. This example uses the heredocs syntax, to improve and make it easy to add complex instructions, and install critical components
RUN <<EORUN
set -xeuo pipefail
# Install networking support and SSH which are not in minimal
dnf -y install NetworkManager openssh-server
dnf clean all
rm /var/{log,cache,lib}/* -rf
bootc container lint
EORUN
# This label is required
LABEL containers.bootc 1
LABEL ostree.bootable 1
# These labels are optional but useful if you want to keep the default of running under
```

```
systemd when run as a container image.
STOPSIGNAL SIGRTMIN+3
CMD ["/sbin/init"]
```

Next steps

- After creating your **Containerfile**, you get an image with a single tar file large layer. Every change, such as pushing to the registry, pulling for clients, results in copying the single large tar file, and increases the container image size. You can optimize the container image that you created for a smaller version.

Optionally, in RHEL 9.6 systems, you can use **rpm-ostree** to take a large image and resize it to a smaller version.

8.5. OPTIMIZING CONTAINER IMAGES TO A SMALLER VERSION

You can use the **bootc-base-imagectl** rechunk subcommand to interact with **rpm-ostree** and perform the image rechunking.

rpm-ostree can take a large image and "rechunk" it. This process optimizes the ordering and grouping of installed packages into many layers in the final image, which provides better network efficiency since several layers can be reused without causing a transfer.

The output of a scratch build is an image with a single large layer (**tar**). Every change that you do to the input, for example, a kernel update, results in a new layer including the entire contents of the bootc image. This new layer must then be pushed, stored by registries, and pulled by clients.

The **bootc-base-imagectl** is shipped as part of the bootc images. You can use pre-existing images to rechunk custom base images by mapping the host containers-storage into the container, and running **bootc-base-imagectl** within to do the rechunking operation.

Prerequisites

- You have a previously-built base image.

Procedure

- Run the following command to rechunk your base image.

```
$ sudo podman run --rm --privileged -v /var/lib/containers:/var/lib/containers \
    registry.redhat.io/rhel10/rhel-bootc:latest \
    /usr/libexec/bootc-base-imagectl rechunk \
      quay.io/exampleos/rhel-bootc:single \
      quay.io/exampleos/rhel-bootc:chunked
```

CHAPTER 9. ENABLING THE FIPS MODE WHILE BUILDING A BOOTC IMAGE

FIPs include standards for cryptographic operations. You can enable the FIPS mode during the bootc image build time, when building a bootc image, to configure the system to use only FIPS approved modules. There are 2 options to enable FIPS mode:

- By using the **bootc-image-builder** tool: you must enable the FIPS crypto policy into the Containerfile.
- When performing an Anaconda installation: apart from enabling the FIPS crypto policy into the Containerfile, you must add the **fips=1** kernel argument during the boot time.

FIPS dracut module is built-in to the base image. It defaults to a **boot=UUID= karg** in **bootc install-to-filesystem**.

9.1. ENABLING THE FIPS MODE BY USING BOOTC-IMAGE-BUILDER

Create a disk image by using **bootc-image-builder** or **bootc install to-disk**, and enable the FIPS mode by passing the custom Containerfile as an argument when building the image.

Prerequisites

- You have Podman installed on your host machine.
- You have **virt-install** installed on your host machine.
- You have root access to run the **bootc-image-builder** tool, and run the containers in **--privileged** mode, to build the images.

Procedure

1. Create a **01-fips.toml** to configure FIPS enablement, for example:

```
# Enable FIPS
kargs = ["fips=1"]
```

2. Create a Containerfile with the following instructions to enable the **fips=1** kernel argument:

```
FROM registry.redhat.io/rhel9/rhel-bootc:latest
# Enable fips=1 kernel argument: https://bootc-dev.github.io/bootc/building/kernel-arguments.html
COPY 01-fips.toml /usr/lib/bootc/kargs.d/
# Enable the FIPS crypto policy
# crypto-policies-scripts is not installed by default in RHEL-10
RUN dnf install -y crypto-policies-scripts && update-crypto-policies --no-reload --set FIPS
```

3. Create your bootc ***<image>*** compatible base disk image by using **Containerfile** in the current directory:

```
$ podman build -t quay.io/<namespace>/<image>:<tag> .
```

Verification

- After login in to the system, check that FIPS mode is enabled:

```
$ cat /proc/sys/crypto/fips_enabled
1
$ update-crypto-policies --show
FIPS
```

Additional resources

- Installing the system with FIPS mode enabled

9.2. ENABLING THE FIPS MODE TO PERFORM AN ANACONDA INSTALLATION

To create a disk image and enable the FIPS mode when performing an Anaconda installation, follow the steps:

Prerequisites

- You have Podman installed on your host machine.
- You have **virt-install** installed on your host machine.
- You have root access to run the **bootc-image-builder** tool, and run the containers in **--privileged** mode, to build the images.

Procedure

1. Create a **01-fips.toml** to configure FIPS enablement, for example:

```
# Enable FIPS
kargs = ["fips=1"]
```

2. Create a Containerfile with the following instructions to enable the **fips=1** kernel argument:

```
FROM registry.redhat.io/rhel9/rhel-bootc:latest
# Enable fips=1 kernel argument: https://bootc-dev.github.io/bootc/building/kernel-arguments.html
COPY 01-fips.toml /usr/lib/bootc/kargs.d/
# Install and enable the FIPS crypto policy
RUN dnf install -y crypto-policies-scripts && update-crypto-policies --no-reload --set FIPS
```

3. Create your bootc ***<image>*** compatible base disk image by using **Containerfile** in the current directory:

```
$ sudo podman run \
  --rm \
  -it \
  --privileged \
  --pull=newer \
  --security-opt label=type:unconfined_t \
  -v $(pwd)/config.toml:/config.toml:ro \
  -v $(pwd)/output:/output \
```

```
-v /var/lib/containers/storage:/var/lib/containers/storage \
registry.redhat.io/rhel9/bootc-image-builder:latest \
--local
--type iso \
quay.io/<namespace>/<image>:<tag>
```

4. Enable FIPS mode during the system installation:

   a. When booting the RHEL Anaconda installer, on the installation screen, press the TAB key and add the **fips=1** kernel argument.
   After the installation, the system starts in FIPS mode automatically.

Verification

- After login in to the system, check that FIPS mode is enabled:

```
$ cat /proc/sys/crypto/fips_enabled
1
$ update-crypto-policies --show
FIPS
```

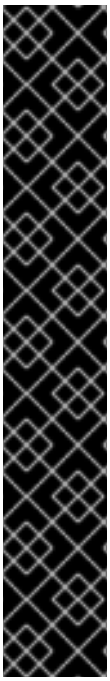
Additional resources

- Installing the system with FIPS mode enabled

CHAPTER 10. SECURITY HARDENING AND COMPLIANCE OF BOOTABLE IMAGES

Image mode for RHEL provides security compliance features and supports workloads that require compliant configuration. However, the process of hardening systems and verifying compliance status is different than in package mode.

The key part of using Image mode for RHEL is creating a bootable container image. The deployed system mirrors the image. Therefore, the built image must contain all packages and configuration settings that are required by the security policy.



IMPORTANT

When a bootable image is run as a container, some of the hardening configuration is not in effect. To get a system that is fully configured in accordance with the security profile, you must boot the image in a bare metal or virtual machine instead of running as a container. Main differences of a container deployment include the following:

- Systemd services that are required by security profiles do not run on containers because systemd is not running in the container. Therefore, the container cannot comply with the related policy requirements.
- Other services cannot run in containers, although they are configured correctly. This means that **oscap** reports them as correctly configured, even if they are not running.
- Configurations defined by the compliance profile are not enforcing. Requests from other packages or installation prescripts can change the compliance state. Always check the compliance of the installed product and alter your Containerfile to fit your requirements.

10.1. BUILDING HARDENED BOOTABLE IMAGES

You can build hardened bootable images more easily by including the **oscap-im** tool in the **Containerfile** that you use to build your bootable container image.

Although **oscap-im** can consume any SCAP content, the SCAP source data streams shipped in **scap-security-guide** are specifically adjusted and tested to be compatible with bootable containers.

Prerequisites

- The **container-tools** meta-package is installed.
- You know the ID of the profile within the baseline with which the system should comply. To find the ID, see the Viewing profiles for configuration compliance section.

Procedure

1. Create a **Containerfile**:

```
FROM registry.redhat.io/rhel9/rhel-bootc:latest

# Install OpenSCAP scanner and security content to the image
RUN dnf install -y openscap-utils scap-security-guide && dnf clean all
```

```
# Add sudo user 'admin' with password 'admin123'.
# The user can be used with profiles that prevent
# ssh root logins.
RUN useradd -G wheel -p "\$6\$Ga6Zn
IlytrWpuCzO\$q0LqT1USHpahzUafQM9jyHCY9BiE5/ahXLNWUMiVQnFGblu0WWGZ1e6icTa
CGO4GNgZNtspp1Let/qpM7FMVB0" admin

# Run scan and hardening
RUN oscap-im --profile <profileID> /usr/share/xml/scap/ssg/content/ssg-rhel9-ds.xml
```

This **Containerfile** performs the following tasks:

- Installs the **openscap-utils** package that provides the **oscap-im** tool and the **scap-security-guide** package that provides the data streams with the Security Content Automation Protocol (SCAP) content.
- Adds a user with **sudoer** privileges for profiles that prevent SSH root logins.
- Scans and remediates the image for compliance with the selected profile.

2. Build the image by using the **Containerfile** in the current directory:

```
$ podman build -t quay.io/<namespace>/<image>:<tag> .
```

Verification

- List all images:

```
$ podman images
REPOSITORY                      TAG     IMAGE ID      CREATED             SIZE
quay.io/<namespace>/<image>             <tag>  b28cd00741b3  About a minute ago  2.1
GB
```

Next steps

- You can deploy hardened bootable images by using any of the normal bootable image deployment methods. For more information, see Deploying the RHEL bootc images . The deployment method, however, can affect the compliance state of the target system.
- You can verify the compliance of a running system in Image Mode RHEL by using the **oscap** tool with the same syntax and usage as in package mode RHEL. For more information, see Configuration compliance scanning.

10.2. CUSTOMIZING HARDENED BOOTABLE IMAGES

You can apply a customized profile to a bootable image by using the **oscap-im** tool. You can customize a security profile by changing parameters in certain rules, for example, minimum password length, removing rules that you cover in a different way, and selecting additional rules, to implement internal policies. You cannot define new rules by customizing a profile.

Prerequisites

- The **container-tools** meta-package is installed.

- You have a customization file for your profile. For more information, see Customizing a security profile with SCAP Workbench.

Procedure

1. Create a **Containerfile**:

```
FROM registry.redhat.io/rhel9/rhel-bootc:latest

# Copy a tailoring file into the Containerfile
COPY tailoring.xml /usr/share/

# Install OpenSCAP scanner and security content to the image
RUN dnf install -y openscap-utils scap-security-guide && dnf clean all


# Add sudo user 'admin' with password 'admin123'.
# The user can be used with profiles that prevent
# ssh root logins.
RUN useradd -G wheel -p "\$6\$Ga6Zn
IlytrWpuCzO\$q0LqT1USHpahzUafQM9jyHCY9BiE5/ahXLNWUMiVQnFGblu0WWGZ1e6icTa
CGO4GNgZNtspp1Let/qpM7FMVB0" admin

# Run scan and hardening including the tailoring file
RUN oscap-im --tailoring-file /usr/share/tailoring.xml --profile stig_customized
/usr/share/xml/scap/ssg/content/ssg-rhel9-ds.xml
```

This **Containerfile** performs the following tasks:

- Injects the tailoring file to your image.
- Installs the **openscap-utils** package that provides the **oscap-im** tool and the **scap-security-guide** package that provides the data streams with the Security Content Automation Protocol (SCAP) content.
- Adds a user with **sudoer** privileges for profiles that prevent SSH root logins.
- Scans and remediates the image for compliance with the selected profile.

1. Build the image by using the **Containerfile** in the current directory:

```
$ podman build -t quay.io/<namespace>/<image>:<tag> .
```

Verification

- List all images:

```
$ podman images
REPOSITORY                    TAG      IMAGE ID      CREATED             SIZE
quay.io/<namespace>/<image>   <tag>    b28cd00741b3  About a minute ago  2.1
GB
```

Next steps

- You can deploy hardened bootable images by using any of the normal bootable image deployment methods. For more information, see Deploying the RHEL bootc images . The deployment method, however, can affect the compliance state of the target system.

NOTE

Some customizations performed during the deployment, in blueprint for **bootc-image-builder** or in Kickstart for Anaconda, can interfere with the configuration present in the container image. Do not use customizations that conflict with the security policy requirements.

- You can verify the compliance of a running system in Image Mode RHEL by using the **oscap** tool with the same syntax and usage as in package mode RHEL. For more information, see Configuration compliance scanning.

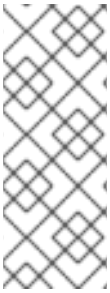
CHAPTER 11. MANAGING RHEL BOOTC IMAGES

After installing and deploying RHEL bootc images, you can perform management operations on your container images, such as changing or updating the systems. The system supports in-place transactional updates with rollback after deployment.

This kind of management, also known as Day 2 management baseline, consists of transactionally fetching new operating system updates from a container registry and booting the system into them, while supporting manual, or automated rollbacks in case of failures.

You can also rely on automatic updates, that are turned on by default. The **systemd service unit** and the **systemd timer unit** files check the container registry for updates and apply them to the system. You can trigger an update process with different events, such as updating an application. There are automation tools watching these updates and then triggering the CI/CD pipelines. A reboot is required, because the updates are transactional. For environments that require more sophisticated or scheduled rollouts, you must disable auto updates and use the **bootc** utility to update your operating system.

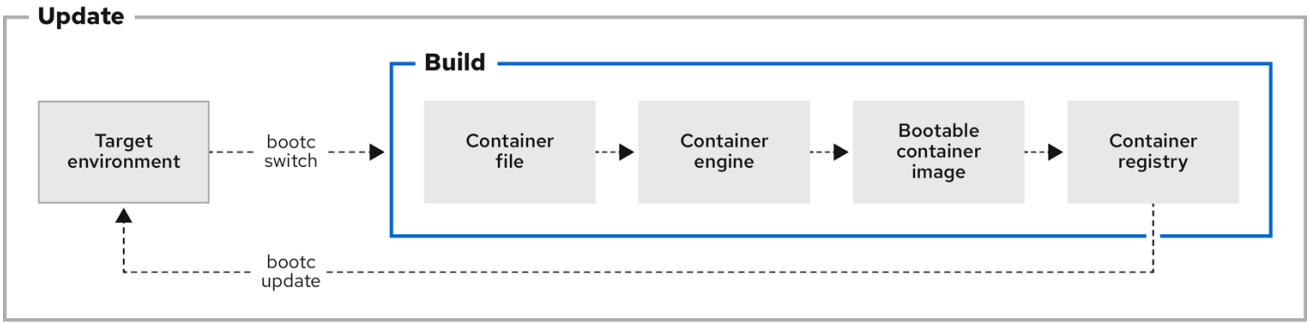
See Day 2 operations support for more details.



NOTE

The **rhel-bootc** images are rebuilt whenever their underlying inputs, such as RPM packages, are updated. These rebuilds occur at least monthly, or more frequently if critical updates are released. As a user, you maintain full control over when to push the update images. A newly published base image does not trigger automatic rebuilds or redeployments of your custom images. You configure the update cadence and only push changes as required.

Figure 11.1. Manually updating an installed operating system, changing the container image reference or rolling back changes if needed



640_RHEL_0524

11.1. SWITCHING THE CONTAINER IMAGE REFERENCE

You can change the container image reference used for upgrades by using the **bootc switch** command. For example, you can switch from the stage to the production tag. The **bootc switch** command performs the same operations as the **bootc upgrade** command and additionally changes the container image reference.

To manually switch an existing **ostree-based** container image reference, use the **bootc switch** command.

WARNING

The use of **rpm-ostree** to make changes, or install content, is not supported.

Prerequisites

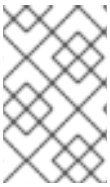
- A booted system using **bootc**.

Procedure

- Run the following command:

```
$ sudo bootc switch [--apply] quay.io/<namespace>/<image>:<tag>
```

Optionally, you can use the **--apply** option when you want to automatically take actions, such as rebooting if the system has changed.



NOTE

The **bootc switch** command has the same effect as **bootc upgrade**. The only difference is the container image reference is changed. This allows preserving the existing states in **/etc** and **/var**, for example, host SSH keys and home directories.

Additional resources

- The **bootc-switch** man page

11.2. ADDING MODULES TO THE BOOTC IMAGE INITRAMFS

The **rhel9/rhel-bootc** image uses the **dracut** infrastructure to build an initial RAM disk, the **initrd** during the image build time. The **initrd** is built and included in the **/usr/lib/modules/$kver/initramfs.img** location inside the container.

You can use a drop-in configuration file to override the **dracut** configuration, and place it in **/usr/lib/dracut/dracut.conf.d/*<50-custom-added-modules.conf>*** And thus re-create **initrd** with the modules you want to add.

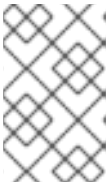
Prerequisites

- A booted system using bootc.

Procedure

- Re-create the **initrd** as part of a container build:

```
FROM <baseimage>
COPY <50-custom-added-modules>.conf /usr/lib/dracut/dracut.conf.d
RUN set -x; kver=$(cd /usr/lib/modules && echo *); dracut -vf
/usr/lib/modules/$kver/initramfs.img $kver
```



NOTE

By default the command attempts to pull the running kernel version, which causes an error. Explicitly pass to **dracut** the kernel version of the target to avoid errors.

11.3. MODIFYING AND REGENERATING INITRD

The default container image includes a pre-generated initial RAM disk (initrd) in **/usr/lib/modules/$kver/initramfs.img**. To regenerate the **initrd**, for example, to add a dracut module, follow the steps:

Procedure

1. Write your drop-in configuration file. For example:

```
dracutmodules = "module"
```

2. Place your drop-in configuration file in the location that **dracut** normally uses: **/usr**. For example:

```
/usr/lib/dracut/dracut.conf.d/50-custom-added-modules.conf
```

3. Regenerate the **initrd** as part of the container build. You must explicitly pass the kernel version to target to **dracut**, because it tries to pull the running kernel version, which can cause an error. The following is an example:

```
FROM <baseimage>
COPY 50-custom-added-modules.conf /usr/lib/dracut/dracut.conf.d
RUN set -x; kver=$(cd /usr/lib/modules && echo *); dracut -vf
/usr/lib/modules/$kver/initramfs.img $kver
```

11.4. PERFORMING MANUAL UPDATES FROM AN INSTALLED OPERATING SYSTEM

Installing image mode for RHEL is a one time task. You can perform any other management task, such as changing or updating the system, by pushing the changes to the container registry.

When using image mode for RHEL, you can choose to perform manual updates for your systems. Manual updates are also useful if you have an automated way to perform updates, for example, by using Ansible. Because the automatic updates are enabled by default, to perform manual updates you must turn the automatic updates off. You can do this by choosing one of the following options:

- Running the **bootc upgrade** command
- Modifying the **systemd** timer file

11.5. TURNING OFF AUTOMATIC UPDATES

To perform manual updates you must turn off automatic updates. You can do this by choosing one of the following options in the procedure below.

Procedure

- Disable the timer of a container build.
  - By running the **systemctl mask** command:

```
$ systemctl mask bootc-fetch-apply-updates.timer
```

  - By modifying the **systemd** timer file. Use **systemd** "drop-ins" to override the timer. In the following example, updates are scheduled for once a week.
    1. Create an **updates.conf** file with the following content:

```
[Timer]
# Clear previous timers
OnBootSec= OnBootSec=1w OnUnitInactiveSec=1w
```

    2. Add you file to the directory you created:

```
$ mkdir -p /usr/lib/systemd/system/bootc-fetch-apply-updates.timer.d
$ cp updates.conf /usr/lib/systemd/system/bootc-fetch-apply-updates.timer.d
```

11.6. MANUALLY UPDATING AN INSTALLED OPERATING SYSTEM

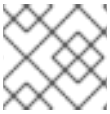
To manually fetch updates from a registry and boot the system into the new updates, use **bootc upgrade**. This command fetches the transactional in-place updates from the installed operating system to the container image registry. The command queries the registry and queues an updated container image for the next boot. It stages the changes to the base image, while not changing the running system by default.

Procedure

- Run the following command:

```
$ bootc upgrade [--apply]
```

  The **apply** argument is optional and you can use it when you want to automatically take actions, such as rebooting if the system has changed.



NOTE

The **bootc upgrade** and **bootc update** commands are aliases.

Additional resources

- The **bootc-upgrade** man page

11.7. PERFORMING ROLLBACKS FROM A UPDATED OPERATING SYSTEM

You can roll back to a previous boot entry to revert changes by using the **bootc rollback** command. This command changes the boot loader entry ordering by making the deployment under **rollback** queued for the next boot. The current deployment then becomes the rollback. Any staged changes, such as a

queued upgrade that was not applied, are discarded.

After a rollback completes, the system reboots and the update timer runs within 1 to 3 hours which automatically updates and reboots your system to the image you just rolled back from.

WARNING

If you perform a rollback, the system will automatically update again unless you turn off auto-updates. See Turning off automatic updates .

NOTE

When performing a rollback, for example, by using the **bootc rollback** command, changes made to files in the **/etc** directory do not carry over to the rolled-back deployment. Instead, the files in the **/etc** directory revert to the state they were in during the previous deployment.

The **bootc rollback** command reorders existing deployments but does not create new ones. The **/etc** directory is merged when new deployments are created.

To preserve a modified **/etc** file for use after a rollback, copy it to a directory under **/var**, such as **/var/home/*<user>***, for a specific ***<user>***, or under **/var/root/**, for the root user. These directories are unaffected by rollbacks, as they store user content.

When returning to the original state, either through a temporary rollback or another bootc rollback, the **/etc** directory reverts to its state from the original deployment.

Alternatively, if the issue you are rolling back does not involve configuration files in the **/etc** directory and you want to revert to an older deployment, use the **bootc switch** command. This command performs the necessary **/etc** merge and deploy the previous version of the software.

Prerequisites

- You performed an update to the system.

Procedure

- Run the following command:

```
$ bootc rollback [-h|--help] [-V|--version]
```

Verification

- Use **systemd journal** to check the logged message for the detected rollback invocation.

```
$ journalctl -b
```

You can see a log similar to:

```
MESSAGE_ID=26f3b1eb24464d12aa5e7b544a6b5468
```

Additional resources

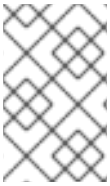
- The **bootc-rollback** man page

11.8. DEPLOYING UPDATES TO SYSTEM GROUPS

You can change the configuration of your operating system by modifying the Containerfile. Then you can build and push your container image to the registry. When you next boot your operating system, an update will be applied.

You can also change the container image source by using the **bootc switch** command. The container registry is the source of truth. See Switching the container image reference.

Usually, when deploying updates to system groups, you can use a central management service to provide a client to be installed on each system which connects to the central service. Often, the management service requires the client to perform a one time registration. The following is an example on how to deploy updates to system groups. You can modify it to create a persistent **systemd** service, if required.



NOTE

For clarity reasons, the Containerfile in the example is not optimized. For example, a better optimization to avoid creating multiple layers in the image is by invoking RUN a single time.

You can install a client into an image mode for RHEL image and run it at startup to register the system.

Prerequisites

- The management-client handles future connections to the server, by using a **cron** job or a separate **systemd** service.

Procedure

- Create a management service with the following characteristics. It determines when to upgrade the system.
  1. Disable **bootc-fetch-apply-updates.timer** if it is included in the base image.
  2. Install the client by using **dnf**, or some other method that applies for your client.
  3. Inject the credentials for the management service into the image.

11.9. CHECKING INVENTORY HEALTH

Health checks are one of the Day 2 Operations. You can manually check the system health of the container images and events that are running inside the container.

You can set health checks by creating the container on the command line. You can display the health check status of a container by using the **podman inspect** or **podman ps** commands.

You can monitor and print events that occur in Podman by using the **podman events** command. Each event includes a timestamp, a type, a status, a name, if applicable, and an image, if applicable.

For more information about health checks and events, see chapter Monitoring containers.

11.10. AUTOMATION AND GITOPS

You can automate the building process by using CI/CD (Continuous Integration and Continuous Delivery) pipelines so that an update process can be triggered by events, such as updating an application. You can use automation tools that track these updates and trigger the CI/CD pipelines. The pipeline keeps the systems up to date by using the transactional background operating system updates.

For more details on resources to create image mode for RHEL instances, check the specific implementations available to create image mode for RHEL instances:RHEL Image Mode CI/CD.

11.11. USING TOOLBX TO INSPECT BOOTC CONTAINERS

Installing software on a system presents certain risks: it can change a system's behavior, and can leave unwanted files and directories behind after they are no longer needed. You can prevent these risks by installing your favorite development and debugging tools, editors, and software development kits (SDKs) into the Toolbx utility included in the RHEL bootc, an image fully mutable container without affecting the base operating system. You can perform changes on the host system with commands such as **less**, **lsof**, **rsync**, **ssh**, **sudo**, and **unzip**.

The Toolbx utility performs the following actions:

1. Pulling the **registry.access.redhat.com/ubi9/toolbox:latest** image to your local system
2. Starting up a container from the image
3. Running a shell inside the container from which you can access the host system

NOTE

Toolbx can run a root container or a rootless container, depending on the rights of the user who creates the Toolbx container. Utilities that would require root rights on the host system also should be run in root containers.

The default container name is **rhel-toolbox**. To inspect bootc containers, follow the steps:

Procedure

1. Start a Toolbx container by using the **toolbox create** command and enter the container with the **toolbox enter** command.
   - As a rootless user:

     ```
     $ toolbox create <mytoolbox>
     ```

   - As a root user:

     ```
     $ sudo toolbox create <mytoolbox>
     Created container: <mytoolbox>
     Enter with: toolbox enter
     ```

- Verify that you pulled the correct image:

```
[user@toolbox ~]$ toolbox list
IMAGE ID      IMAGE NAME    CREATED
fe0ae375f149   registry.access.redhat.com/ubi{ProductVersion}/toolbox 5 weeks ago

CONTAINER ID  CONTAINER NAME  CREATED        STATUS   IMAGE NAME
5245b924c2cb  <mytoolbox>       7 minutes ago   created
registry.access.redhat.com/ubi{ProductVersion}/toolbox:8.9-6
```

  a. Enter the Toolbx container:

```
[user@toolbox ~]$ toolbox enter <mytoolbox>
```

  b. Optional: Check if you pulled the correct image

- Enter a command inside the ***<mytoolbox>*** container and display the name of the container and the image:

```
⬢ [user@toolbox ~]$ cat /run/.containerenv
engine="podman-4.8.2"
name="<mytoolbox>"
id="5245b924c2cb..."
image="registry.access.redhat.com/ubi{ProductVersion}/toolbox"
imageid="fe0ae375f14919cbc0596142e3aff22a70973a36e5a165c75a86ea7ec5d8d65c"
```

2. Use the Toolbx to install the development tools:

  a. Install the tools of your choice, for example, the Emacs text editor, GCC compiler and GNU Debugger (GDB):

```
⬢[user@toolbox ~]$ sudo dnf install emacs gcc gdb
```

  b. Optional: Verify that the tools are installed:

```
⬢[user@toolbox ~]$  dnf repoquery --info --installed <package_name>
```

  After installation, you can continue using those tools as a rootless user.

3. Use Toolbx to troubleshoot the host system without installing them on the host system.

  a. Install the **systemd** suite to be able to run the **journalctl** command:

```
⬢[root@toolbox ~]# dnf install systemd
```

  b. Display log messages for all processes running on the host:

```
⬢[root@toolbox ~]# j journalctl --boot -0
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: microcode: updated ear>
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: Linux version 6.6.8-10>
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: Command line: BOOT_IMA>
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: x86/split lock detecti>
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: BIOS-provided physical>
```

c. Display log messages for the kernel:

```
⬢[root@toolbox ~]# journalctl --boot -0 --dmesg
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: microcode: updated ear>
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: Linux version 6.6.8-10>
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: Command line: BOOT_IMA>
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: x86/split lock detecti>
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: BIOS-provided physical>
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: BIOS-e820: [mem 0x0000>
```

d. Install the **nmap** network scanning tool:

```
⬢[root@toolbox ~]# dnf install nmap
```

e. Scan IP addresses and ports in a network:

```
⬢[root@toolbox ~]# nmap -sS scanme.nmap.org
Starting Nmap 7.93 ( https://nmap.org ) at 2024-01-02 10:39 CET
Stats: 0:01:01 elapsed; 0 hosts completed (0 up), 256 undergoing Ping Scan
Ping Scan Timing: About 29.79% done; ETC: 10:43 (0:02:24 remaining)
Nmap done: 256 IP addresses (0 hosts up) scanned in 206.45 seconds
```

- The **-sS** option performs a TCP SYN scan. Most of Nmap's scan types are only available to privileged users, because they send and receive raw packets, which requires root access on UNIX systems.

4. Stop the Toolbx bootc container.

a. Leave the container and return to the host:

```
⬢ [user@toolbox ~]$ exit
```

b. Stop the toolbox container:

```
⬢ [user@toolbox ~]$ podman stop <mytoolbox>
```

c. Optional: Remove the toolbox container:

```
⬢ [user@toolbox ~]$ toolbox rm <mytoolbox>
```

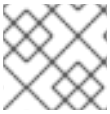
Alternatively, you can also use the **podman rm** command to remove the bootc container.

Additional resources

- Debugging image mode hosts article

CHAPTER 12. MANAGING KERNEL ARGUMENTS IN BOOTC SYSTEMS

You can use **bootc** to configure kernel arguments. By default, **bootc** uses the boot loader configuration files that are stored in **/boot/loader/entries**. This directory defines arguments provided to the Linux kernel. The set of kernel arguments is machine-specific state, but you can also manage the kernel arguments by using container updates. The boot loader menu entries are shared between multiple operating systems and boot loaders are installed on one device.



NOTE

Currently, the boot loader entries are written by an OSTree backend.

12.1. HOW TO ADD SUPPORT TO INJECT KERNEL ARGUMENTS WITH BOOTC

The **bootc** tool uses generic operating system kernels. You can add support to inject kernel arguments by adding a custom configuration, in the TOML format, in **/usr/lib/bootc/kargs.d**. For example:

```
# /usr/lib/bootc/kargs.d/10-example.toml
kargs = ["mitigations=auto,nosmt"]
```

You can also make these kernel arguments architecture-specific by using the **match-architectures** key. For example:

```
# /usr/lib/bootc/kargs.d/00-console.toml
kargs = ["console=ttyS0,114800n8"]
match-architectures = ["x86_64"]
```

12.2. HOW TO MODIFY KERNEL ARGUMENTS BY USING BOOTC INSTALL CONFIGS

You can use **bootc install** to add kernel arguments during the install time in the following ways:

- Adding kernel arguments into the container image.
- Adding kernel arguments by using the **bootc install --karg** command.

You can use the kernel arguments on Day 2 operations, by adding the arguments and applying them on a switch, upgrade, or edit. Adding kernel arguments and using it for Day 2 operations involves the following high-level steps:

1. Create files within **/usr/lib/bootc/kargs.d** with kernel arguments.
2. Fetch the container image to get the OSTree commit.
3. Use the OSTree commit to return the file tree.
4. Navigate to **/usr/lib/bootc/kargs.d**.
5. Read each file within the directory.
6. Push the contents of each **kargs** file into a file containing all the needed **kargs**.

7. Pass the **kargs** to the ***stage()*** function.
8. Apply these arguments to switch, upgrade, or edit.

12.3. HOW TO INJECT KERNEL ARGUMENTS IN THE CONTAINERFILE

To add kernel arguments into a container image, use a Containerfile. The following is an example:

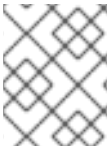
```
FROM registry.redhat.io/rhel9/rhel-bootc:latest

RUN mkdir -p /usr/lib/bootc/kargs.d
RUN cat <<EOF >> /usr/lib/bootc/kargs.d/console.toml
kargs = ["console=ttyS0,114800n8"]
match-architectures = ["x86_64"]
EOF

RUN cat <<EOF >> /usr/lib/bootc/kargs.d/01-mitigations.toml
kargs = ["mitigations=on", "systemd.unified_cgroup_hierarchy=0"]
match-architectures = ["x86_64", "aarch64"]
EOF
```

12.4. HOW TO INJECT KERNEL ARGUMENTS AT INSTALLATION TIME

You can use **boot install** with the **--karg** to inject kernel arguments during installation time. As a result, the kernel arguments become machine-local state.

For example, to inject kernel arguments, use the following command:

```
# bootc install to-filesystem --karg
```



NOTE

Currently, bootc does not have an API to manipulate kernel arguments. This is only supported by **rpm-ostree** , by using the **rpm-ostree kargs** command.

12.5. HOW TO ADD INSTALL-TIME KERNEL ARGUMENTS WITH BOOTC-IMAGE-BUILDER

The **bootc-image-builder** tool supports the **customizations.kernel.append** during install-time.

To add the kernel arguments with **bootc-image-builder**, use the following customization:

```
{
  "customizations": {
    "kernel": {
      "append": "mitigations=auto,nosmt"
    }
  }
}
```

12.6. ABOUT CHANGING KERNEL ARGUMENTS POST-INSTALL WITH KARGS.D

The changes that you make to **kargs.d** files and include in a container build are applied after the installation, and the difference between the set of kernel arguments is applied to the current boot loader configuration. This preserves any machine-local kernel arguments. You can use any tool to edit the **/boot/loader/entries** files, which are in a standardized format. The **/boot** file has read-only access to limit the set of tools that can write to this filesystem.

12.7. HOW TO EDIT KERNEL ARGUMENTS IN BOOTC SYSTEMS

To perform machine local changes, you also can edit kernel arguments on a bootc system or an **rpm-ostree** system, by using the **rpm-ostree kargs** command. The changes are made through the **user/lib/bootc/kargs.d** path, which also handles "Day 2" changes, besides the first boot changes.

The following are the options that you can use to add, modify or remove kernel arguments.

rpm-ostree kargs [option]

--append=KEY=VALUE

Appends a kernel argument. It is useful with, for example, **console=** that can be used multiple times. You can use an empty value for an argument.

--replace=KEY=VALUE=NEWVALUE

Replaces an existing kernel argument. You can replace an argument with **KEY=VALUE** only if one value already exists for that argument.

--delete=KEY=VALUE

Deletes a specific kernel key-value pair argument or an entire argument with a single key-value pair.

--append-if-missing=KEY=VALUE

Appends a kernel argument. Does nothing if the key is already present.

--delete-if-present=KEY=VALUE

Deletes a specific kernel key-value pair argument. Does nothing if the key is missing.

--editor

Uses an editor to modify the kernel arguments.

For more information, check the help:

```
# rpm-ostree kargs --help
```

The following is an example:

```
# rpm-ostree kargs --append debug
Staging deployment... done
Freed: 40.1 MB (pkgcache branches: 0)
Changes queued for next boot. Run "systemctl reboot" to start a reboot
```

CHAPTER 13. MANAGING FILE SYSTEMS IN IMAGE MODE FOR RHEL

Currently, image mode for RHEL uses OSTree as a backend, and enables **composefs** for storage by default. The /**opt** and /**usr**/**local** paths are plain directories, and not symbolic links into /**var**. This enables you to easily install third-party content in derived container images that write into /**opt** for example.

13.1. PHYSICAL AND LOGICAL ROOT WITH /SYSROOT

When a system is fully booted, it is similar to **chroot**, that is, the operating system changes the apparent root directory for the current running process and its children. The physical host root filesystem is mounted at /**sysroot**. The **chroot** filesystem is called a deployment root.

The remaining filesystem paths are part of a deployment root which is used as a final target for the system boot. The system uses the **ostree=kernel** argument to find the deployment root.

/**usr**

This filesystem keeps all operating system content in /**usr**, with directories such as /**bin** working as symbolic links to /**usr**/**bin**.

NOTE

composefs enabled /**usr** is not different from /. Both directories are part of the same immutable image, so you do not need to perform a full **UsrMove** with a bootc system.

/**usr**/**local**

The base image is configured with /**usr**/**local** as the default directory.

/**etc**

The /**etc** directory contains mutable persistent state by default, but it supports enabling the **etc.transient config** option. When the directory is in mutable persistent state, it performs a 3-way merge across upgrades:

- Uses the new default /**etc** as a base
- Applies the diff between current and previous /**etc** to the new /**etc** directory
- Retains locally modified files that are different from the default /**usr**/**etc** of the same deployment in /**etc**.

The **ostree-finalize-staged.service** executes these tasks during shutdown time, before creating the new boot loader entry.

This happens because many components of a Linux system ship default configuration files in the /**etc** directory. Even if the default package does not ship it, by default the software only checks for config files in /**etc**. Non bootc image based update systems with no distinct versions of /**etc** are populated only during the installation time, and will not be changed at any point after installation. This causes the /**etc** system state to be influenced by the initial image version and can lead to problems to apply a change, for example, to /**etc**/**sudoers.conf**, and requires external intervention. For more details about file configuration, see Building and testing RHEL bootc images .

/**var**

The content in the **/var** directory is persistent by default. You can also make **/var** or subdirectories mount points be persistent, whether network or **tmpfs**.

There is just one **/var** directory. If it is not a distinct partition, then physically the **/var** directory is a bind mount into **/ostree/deploy/$stateroot/var** and is shared across the available boot loader entries deployments.

By default, the content in **/var** acts as a volume, that is, the content from the container image is copied during the initial installation time, and is not updated thereafter.

The **/var** and the **/etc** directories are different. You can use **/etc** for relatively small configuration files, and the expected configuration files are often bound to the operating system binaries in **/usr**. The **/var** directory has arbitrarily large data, for example, system logs, databases, and by default, will not be rolled back if the operating system state is rolled back.

For example, making an update such as **dnf downgrade postgresql** should not affect the physical database in **/var/lib/postgres**. Similarly, making a **bootc update** or **bootc rollback** do not affect this application data.

Having **/var** separate also makes it work cleanly to stage new operating system updates before applying them, that is, updates are downloaded and ready, but only take effect on reboot. The same applies for Docker volume, as it decouples the application code from its data.

You can use this case if you want applications to have a pre-created directory structure, for example, **/var/lib/postgresql**. Use **systemd tmpfiles.d** for this. You can also use **StateDirectory=*<directory>*** in units.

Other directories

There is no support to ship content in **/run**, **/proc** or other API Filesystems in container images. Apart from that, other top level directories such as **/usr**, and **/opt**, are lifecycled with the container image.

/opt

With **bootc** using **composefs**, the **/opt** directory is read-only, alongside other top level directories such as **/usr**.

When a software needs to write to its own directory in **/opt/exampleapp**, a common pattern is to use a symbolic link to redirect to, for example, **/var** for operations such as log files:

```
RUN rmdir /opt/exampleapp/logs && ln -sr /var/log/exampleapp /opt/exampleapp/logs
```

Optionally, you can configure the systemd unit to launch the service to do these mounts dynamically. For example:

```
BindPaths=/var/log/exampleapp:/opt/exampleapp/logs
```

Enabling transient root

To enable a software to transiently (until the next reboot) write to all top-level directories, including **/usr** and **/opt**, with symlinks to **/var** for content that should persist, you can enable transient root. To enable a fully transient writable **rootfs** by default, set the following option in **/usr/lib/ostree/prepare-root.conf**.

```
[root]
transient = true
```

This enables a software to transiently write to **/opt**, with symlinks to **/var** for content that must persist.

Additional resources

- Enabling transient root documentation

13.2. VERSION SELECTION AND BOOTUP

Image mode for RHEL uses GRUB by default, with exception to **s390x** architectures. Each version of image mode for RHEL currently available on a system has a menu entry.

The menu entry references an OSTree deployment which consists of a Linux kernel, an **initramfs** and a hash linking to an OSTree commit, that you can pass by using the **ostree=kernel** argument.

During bootup, OSTree reads the kernel argument to determine which deployment to use as the root filesystem. Each update or change to the system, such as package installation, addition of kernel arguments, creates a new deployment.

This enables rolling back to a previous deployment if the update causes problems.

CHAPTER 14. APPENDIX: MANAGING USERS, GROUPS, SSH KEYS, AND SECRETS IN IMAGE MODE FOR RHEL

Learn more about users, groups, SSH keys, and secrets management in image mode for RHEL.

14.1. USERS AND GROUPS CONFIGURATION

Image mode for RHEL is a generic operating system update and configuration mechanism. You cannot use it to configure users or groups. The only exception is the **bootc install** command that has the **--root-ssh-authorized-keys** option.

Users and groups configuration for generic base images

Usually, the distribution base images do not have any configuration. Do not encrypt passwords and SSH keys with publicly-available private keys in generic images because of security risks.

Injecting SSH keys through systemd credentials

You can use **systemd** to inject a root password or SSH **authorized_keys** file in some environments. For example, use System Management BIOS (SMBIOS) to inject SSH keys system firmware. You can configure this in local virtualization environments, such as **qemu**.

Injecting users and SSH keys by using cloud-init

Many Infrastructure as a service (IaaS) and virtualization systems use metadata servers that are commonly processed by software such as **cloud-init** or **ignition**. See AWS instance metadata. The base image you are using might include **cloud-init** or Ignition, or you can install it in your own derived images. In this model, the SSH configuration is managed outside of the bootc image.

Adding users and credentials by using container or unit custom logic

Systems such as **cloud-init** are not privileged. You can inject any logic you want to manage credentials in the way you want to launch a container image, for example, by using a **systemd** unit. To manage the credentials, you can use a custom network-hosted source, for example, FreeIPA.

Adding users and credentials statically in the container build

In package-oriented systems, you can use the derived build to inject users and credentials by using the following command:

```
RUN useradd someuser
```

You can find issues in the default **shadow-utils** implementation of **useradd**: Users and groups IDs are allocated dynamically, and this can cause drift.

User and group home directories and /var directory

For systems configured with persistent **/home → /var/home**, any changes to **/var** made in the container image after initial installation will not be applied on subsequent updates.
For example, if you inject **/var/home/someuser/.ssh/authorized_keys** into a container build, existing systems do not get the updated **authorized_keys** file.

Using DynamicUser=yes for systemd units

Use the **systemd DynamicUser=yes** option where possible for system users.
This is significantly better than the pattern of allocating users or groups at package install time, because it avoids potential UID or GID drift.

Using systemd-sysusers

Use **systemd**-sysusers, for example, in your derived build. For more information, see the **systemd**-sysusers documentation.

```
COPY mycustom-user.conf /usr/lib/sysusers.d
```

The **sysusers** tool makes changes to the traditional **/etc/passwd** file as necessary during boot time. If **/etc** is persistent, this can avoid **UID** or **GID** drift. It means that the **UID** or **GID** allocation depends on how a specific machine was upgraded over time.

Using systemd JSON user records

See JSON user records **systemd** documentation. Unlike **sysusers**, the canonical state for these users lives in **/usr**. If a subsequent image drops a user record, then it also vanishes from the system.

Using nss-altfiles

With **nss-altfiles**, you can remove the **systemd** JSON user records. It splits system users into **/usr/lib/passwd** and **/usr/lib/group**, aligning with the way the OSTree project handles the 3 way merge for **/etc** as it relates to **/etc/passwd**. Currently, if the **/etc/passwd** file is modified in any way on the local system, then subsequent changes to **/etc/passwd** in the container image are not applied. Base images built by **rpm-ostree** have **nns-altfiles** enabled by default.

Also, base images have a system users pre-allocated and managed by the NSS file to avoid UID or GID drift.

In a derived container build, you can also append users to **/usr/lib/passwd**, for example. Use **sysusers.d** or **DynamicUser=yes**.

Machine-local state for users

The filesystem layout depends on the base image.
By default, the user data is stored in both **/etc**, **/etc/passwd**, **/etc/shadow** and **groups**, and **/home**, depending on the base image. However, the generic base images have to both be machine-local persistent state. In this model **/home** is a symlink to **/var/home/*user***.

Injecting users and SSH keys at system provisioning time

For base images where **/etc** and **/var** are configured to persist by default, you can inject users by using installers such as Anaconda or Kickstart.
Typically, generic installers are designed for one time bootstrap. Then, the configuration becomes a mutable machine-local state that you can change in Day 2 operations, by using some other mechanism.

You can use the Anaconda installer to set the initial password. However, changing this initial password requires a different in-system tool, such as **passwd**.

These flows work equivalently in a **bootc-compatible** system, to support users directly installing generic base images, without requiring changes to the different in-system tool.

Transient home directories

Many operating system deployments minimize persistent, mutable, and executable state. This can damage user home directories.
The **/home** directory can be set as **tmpfs**, to ensure that user data is cleared across reboots. This approach works especially well when combined with a transient **/etc** directory.

To set up the user's home directory to, for example, inject SSH **authorized_keys** or other files, use the **systemd tmpfiles.d** snippets:

-

```
f~ /home/user/.ssh/authorized_keys 600 user user - <base64 encoded data>
```

SSH is embedded in the image as: **/usr/lib/tmpfiles.d/<username-keys.conf**. Another example is a service embedded in the image that can fetch keys from the network and write them. This is the pattern used by **cloud-init**.

UID and GID drift

The **/etc/passwd** and similar files are a mapping between names and numeric identifiers. When the mapping is dynamic and mixed with "stateless" container image builds, it can cause issues. Each container image build might result in the UID changing due to RPM installation ordering or other reasons. This can be a problem if that user maintains a persistent state. To handle such cases, convert it to use **sysusers.d** or use **DynamicUser=yes**.

14.2. INJECTING SECRETS IN IMAGE MODE FOR RHEL

Image mode for RHEL does not have an opinionated mechanism for secrets. You can inject container pull secrets in your system for some cases, for example:

- For **bootc** to fetch updates from a registry that requires authentication, you must include a pull secret in a file. In the following example, the **creds** secret contains the registry pull secret.

```
FROM registry.redhat.io/rhel9/bootc-image-builder:latest
COPY containers-auth.conf /usr/lib/tmpfiles.d/link-podman-credentials.conf
RUN --mount=type=secret,id=creds,required=true cp /run/secrets/creds /usr/lib/container-auth.json && \
    chmod 0600 /usr/lib/container-auth.json && \
    ln -sr /usr/lib/container-auth.json /etc/ostree/auth.json
```

  To build it, run **podman build --secret id=creds,src=$HOME/.docker/config.json**. Use a single pull secret for **bootc** and Podman by using a symlink to both locations to a common persistent file embedded in the container image, for example **/usr/lib/container-auth.json**.

- For Podman to fetch container images, include a pull secret to **/etc/containers/auth.json**. With this configuration, the two stacks share the **/usr/lib/container-auth.json** file.

  #### Injecting secrets by embedding them in a container build

  You can include secrets in the container image if the registry server is suitably protected. In some cases, embedding only bootstrap secrets into the container image is a viable pattern, especially alongside a mechanism for having a machine authenticate to a cluster. In this pattern, a provisioning tool, whether run as part of the host system or a container image, uses the bootstrap secret to inject or update other secrets, such as SSH keys, certificates, among others.

  #### Injecting secrets by using cloud metadata

  Most production Infrastructure as a Service (IaaS) systems support a metadata server or equivalent which can securely host secrets, particularly bootstrap secrets. Your container image can include tools such as **cloud-init** or **ignition** to fetch these secrets.

  #### Injecting secrets by embedding them in disk images

  You can embed **bootstrap secrets** only in disk images. For example, when you generate a cloud disk image from an input container image, such as AMI or OpenStack, the disk image can contain secrets that are effectively machine-local state. Rotating them requires an additional management tool or refreshing the disk images.

  #### Injecting secrets by using bare metal installers

Installer tools usually support injecting configuration through secrets.

Injecting secrets through systemd credentials

The **systemd** project has a credential concept for securely acquiring and passing credential data to systems and services, which applies in some deployment methodologies. See the systemd credentials documentation for more details.

Additional resources

- Example bootc images

14.3. CONFIGURING CONTAINER PULL SECRETS

To be able to fetch container images, you must configure a host system with a "pull secret", which includes the host updates itself. See the appendix about Injecting secrets in image mode for RHEL documentation for more details.

You can configure the container pull secrets to an image already built. If you use an external installer such as Anaconda for bare metal, or **bootc-image-builder**, you must configure the systems with any applicable pull secrets.

The host bootc updates write the configuration to the **/etc/ostree/auth.json** file, which is shared with **rpm-ostree**.

Podman does not have system wide credentials. Podman accepts the **containers-auth** locations that are underneath the following directories:

- **/run**: The content of this directory vanishes on reboot, which is not desired.
- **/root**: Part of root home directory, which is local mutable state by default.

To unify **bootc** and Podman credentials, use a single default global pull secret for both **bootc** and Podman. The following container build is an example to unify the **bootc** and the Podman credentials. The example expects a secret named **creds** to contain the registry pull secret to build.

Procedure

1. Create a symbolic link between **bootc** and Podman to use a single pull secret. By creating the symbolic link, you ensure that both locations are present to a common persistent file embedded in the container image.
2. Create the **/usr/lib/container-auth.json** file.

```
FROM quay.io/<namespace>/<image>:<tag>
COPY containers-auth.conf /usr/lib/tmpfiles.d/link-podman-credentials.conf
RUN --mount=type=secret,id=creds,required=true cp /run/secrets/creds /usr/lib/container-auth.json && \
    chmod 0600 /usr/lib/container-auth.json && \
    ln -sr /usr/lib/container-auth.json /etc/ostree/auth.json
```

When you run the Containerfile, the following actions happen:

- The Containerfile makes **/run/containers/0/auth.json** a transient runtime file.
- It creates a symbolic link to the **/usr/lib/container-auth.json**.

- It also creates a persistent file, which is also symbolic linked from **/etc/ostree/auth.json**.

14.4. INJECTING PULL SECRETS FOR REGISTRIES AND DISABLING TLS

You can configure container images, pull secrets, and disable TLS for a registry within a system. These actions enable containerized environments to pull images from private or insecure registries.

You can include container pull secrets and other configuration to access a registry inside the base image. However, when installing by using Anaconda, the installation environment might need a duplicate copy of "bootstrap" configuration to access the targeted registry when fetching over the network.

To perform arbitrary changes to the installation environment before the target bootc container image is fetched, you can use the Anaconda **%pre** command.

See the **containers-auth.json(5)** for more detailed information about format and configurations of the **auth.json** file.

Procedure

1. Configure a pull secret:

```
%pre
mkdir -p /etc/ostree
cat > /etc/ostree/auth.json << 'EOF'
{
        "auths": {
                "quay.io": {
                        "auth": "<your secret here>"
                }
        }
}
EOF
%end
```

   With this configuration, the system pulls images from **quay.io** using the provided authentication credentials, which are stored in **/etc/ostree/auth.json**.

2. Disable TLS for an insecure registry:

```
%pre
mkdir -p /etc/containers/registries.conf.d/
cat > /etc/containers/registries.conf.d/local-registry.conf << 'EOF'

[[registry]]
location="[IP_Address]:5000"
insecure=true
EOF
%end
```

   With this configuration, the system pulls container images from a registry that is not secured with TLS. You can use it in development or internal networks.

You can also use **%pre** to:

- Fetch data from the network by using binaries included in the installation environment, such as **curl**.
- Inject trusted certificate authorities into the installation environment **/etc/pki/ca-trust/source/anchors** by using the **update-ca-trust** command.

You can configure insecure registries similarly by modifying the **/etc/containers** directory.

Additional resources

- Working with container registries

CHAPTER 15. APPENDIX: SYSTEM CONFIGURATION

15.1. TRANSIENT RUNTIME RECONFIGURATION

You can perform a dynamic reconfiguration in the base image configuration. For example, you can run the **firewall-cmd --permanent** command to achieve persistent changes across a reboot.

WARNING

The **/etc** directory is persistent by default. If you perform changes made by using tools, for example **firewall-cmd --permanent**, the contents of the **/etc** on the system can differ from the one described in the container image.

In the default configuration, first make the changes in the base image, then queue the changes without restarting running systems, and then simultaneously write to apply the changes to existing systems only in memory.

You can configure the **/etc** directory to be transient by using bind mounts. In this case, the **etc** directory is a part of the machine's local root filesystem. For example, if you inject static IP addresses by using Anaconda kickstarts, they persist across upgrades.

A 3-way merge is applied across upgrades and each "deployment" has its own copy of **/etc**.

The /run directory

The **/run** directory is an API filesystem that is defined to be deleted when the system is restarted. Use the **/run** directory for transient files.

Dynamic reconfiguration models

In the Pull model, you can include code directly embedded in your base image or a privileged container that contacts the remote network server for configuration, and subsequently launch additional container images, by using the Podman API.

In the Push model, some workloads are implemented by tooling such as Ansible.

systemd

You can use systemd units for dynamic transient reconfiguration by writing to **/run/systemd** directory. For example, the **systemctl edit --runtime myservice.service** dynamically changes the configuration of the **myservice.service** unit, without persisting the changes.

NetworkManager

Use a **/run/NetworkManager/conf.d** directory for applying temporary network configuration. Use the **nmcli connection modify --temporary** command to write changes only in memory. Without the **--temporary** option, the command writes persistent changes.

podman

Use the **podman run --rm** command to automatically remove the container when it exits. Without the **--rm** option, the **podman run** command creates a container that persists across system reboots.

15.2. USING DNF

The **rhel9/rhel-bootc** container image includes **dnf**. There are several use cases:

Using dnf as a part of a container build

You can use the **RUN dnf install** directive in the Containerfile.

Using dnf at runtime

WARNING

The functionality depends on the **dnf** version. You might get an error: **error: can't create transaction lock on /usr/share/rpm/.rpm.lock (Read-only file system)**.

You can use the **bootc-usr-overlay** command to create a writable overlay filesystem for **/usr** directory. The **dnf install** writes to this overlay. You can use this feature for installing debugging tools. Note that changes will be lost on reboot.

Configuring storage

The supported storage technologies are the following:

- **xfs/ext4**
- Logical volume management (LVM)
- Linux Unified Key Setup (LUKS)

You can add other storage packages to the host system.

- **Storage with bootc-image-builder** You can use the bootc-image-builder tool to create a disk image. The available configuration for partitioning and layout is relatively fixed. The default filesystem type is derived from the container image's bootc install configuration.

Storage with bootc install You can use the **bootc install to-disk** command for flat storage configurations and **bootc install to-filesytem** command for more advanced installations. For more information see Advanced installation with **to-filesystem**.

15.3. NETWORK CONFIGURATION

The default images include the **NetworkManager** dynamic network control and configuration system, and bootc attempts to connect by using DHCP on every interface with a cable plugged in. You can apply a temporary network configuration, by setting up the **/run/NetworkManager/conf.d** directory.

However, if you need to use static addressing or more complex networking such as VLANs, bonds, bridges, teams, among others, you can use different ways. Regardless of the way you choose to configure networking, it results as a configuration for NetworkManager, which takes the form of **NetworkManager** keyfiles.

Host Network Configuration options

Complex networking configuration often also requires per-machine state. You can generate machine-specific container images that have, for example, static IP addressing included. You can

also include code to generate network configuration from inside the image by inspecting the MAC address of the host.

Network configuration options available

The following are the available options for configuring static IP, and how the configuration should be done:

- By using a Containerfile: Create a container image with static IP or include code to generate network configuration from inside the image based on MAC address.
- By using Anaconda: You can use an Anaconda Kickstart to configure networking, including Wi-Fi, for bare-metal installations. The configuration is stored by default in **/etc/NetworkManager/system-connections/**, and is inherently per-machine state.
- By using kernel arguments: Add kernel parameters on first boot to define networking configuration. On the first boot of a machine, enter kernel arguments that define networking configuration. The kernel arguments are mostly defined in the **dracut.cmdline** man page. You can apply these kernel arguments on first boot by using different methods. When using **bootc install**, you can also set per-machine kernel arguments by using **--karg**.
- By using NetworkManager key files: **nmcli** or **nm-initrd-generator**

Generating a NetworkManager keyfiles by using nmcli

The **nmcli** NetworkManager command line tool provides an offline mode that does not communicate with the NetworkManager daemon and just writes the keyfile content to standard output.

- Run the **nmcli** tool for each connection profile you want to create:

```
$ nmcli --offline connection add \
      type ethernet ifname enp1s0 \
      ipv4.method manual ipv4.addresses 192.0.0.1/24 \
      ipv6.method disabled

[connection]
id=ethernet-enp1s0
uuid=ff242096-f803-425f-9a77-4c3ec92686bd
type=ethernet
interface-name=enp1s0

[ethernet]

[ipv4]
address1=192.0.0.1/24
method=manual

[ipv6]
addr-gen-mode=default
method=disabled
[proxy]
```

See the settings man page for a list of the properties that can be specified by using **nmcli**. Bash autocompletion is available.

Generating NetworkManager Keyfiles by using nm-initrd-generator

NetworkManager contains the **nm-initrd-generator** tool, that can generate keyfiles from **dracut** kernel argument syntax. You can use the tool to either convert from kernel arguments to keyfiles or to just quickly generate some keyfiles giving a small amount of input and then modify some more detailed settings.

- Generate keyfiles for a bond by using **nm-initrd-generator**:

```
$ podman run --rm -ti quay.io/<namespace>/<image>:<tag> /usr/libexec/nm-initrd-generator
-s -- "ip=bond0:dhcp" "bond=bond0:ens2,ens3:mode=active-backup,miimon=100"
"nameserver=8.8.8.8"

* Connection 'bond0' *

[connection]
id=bond0
uuid=643c17b5-b364-4137-b273-33f450a45476
type=bond
interface-name=bond0
multi-connect=1
permissions=

[ethernet]
mac-address-blacklist=

[bond]
miimon=100
mode=active-backup

[ipv4]
dns=8.8.8.8;
dns-search=
may-fail=false
method=auto

[ipv6]
addr-gen-mode=eui64
dns-search=
method=auto

[proxy]

* Connection 'ens3' *

[connection]
id=ens3
uuid=b42cc917-fd87-47df-9ac2-34622ecddd8c
type=ethernet
interface-name=ens3
master=643c17b5-b364-4137-b273-33f450a45476
multi-connect=1
permissions=
slave-type=bond

[ethernet]
mac-address-blacklist=
```

```
* Connection 'ens2' *

[connection]
id=ens2
uuid=e111bb4e-3ee3-4612-afc2-1d2dfff97671
type=ethernet
interface-name=ens2
master=643c17b5-b364-4137-b273-33f450a45476
multi-connect=1
permissions=
slave-type=bond

[ethernet]
mac-address-blacklist=
```

The command generates three keyfiles for each interface: **bond0**, **ens3**, and **ens2**. You can use the generated output, add more settings or modify existing settings, and then commit the files into a container image.

Configuring a Static IP

- You can use the following **dracut** kernel arguments:
  Template:

```
ip=${ip}::${gateway}:${netmask}:${hostname}:${interface}:none:${nameserver}
```

Example:

```
ip=10.10.10.10::10.10.10.1:255.255.255.0:myhostname:ens2:none:8.8.8.8
```

Writing configuration embedded in container images

Store the NetworkManager configuration embedded in container images in **/usr/lib/NetworkManager/system-connections/** because this form is part of the immutable image state. You can also write configuration to **/etc/NetworkManager/system-connections/** as part of the container image. The default OSTree 3-way merge, that is, using the old default configuration, the active **/etc** system, and the new default configuration, applies with any machine-specific configuration.

The keyfiles must have the **600** root-only access permissions, otherwise **NetworkManager** ignores them.

Disabling automatic configuration of Ethernet devices

By default, **NetworkManager** attempts to autoconfigure by using the DHCP or SLAAC addresses on every interface with a cable plugged in. In some network environments this might not be desirable. For that, it is possible to change the NetworkManager behavior by adding a configuration file, such as **/usr/lib/NetworkManager/conf.d/noauto.conf**.

- Disable the **NetworkManager** autoconfiguration of Ethernet devices

```
[main]
# Do not do automatic (DHCP or SLAAC) configuration on ethernet devices
# with no other matching connections.
no-auto-default=*
```

15.4. SETTING A HOSTNAME

To set a custom hostname for your system, modify the **/etc/hostname** file. You can set the hostname by using Anaconda, or with a privileged container.

Once you boot a system, you can verify the hostname by using the **hostnamectl** command.

15.5. PROXIED INTERNET ACCESS

If you are deploying to an environment requiring internet access by using a proxy, you need to configure services so that they can access resources as intended.

This is done by defining a single file with required environment variables in your configuration, and to reference this by using **systemd** drop-in unit files for all such services.

Defining common proxy environment variables

This common file has to be subsequently referenced explicitly by each service that requires internet access.

```
# /etc/example-proxy.env
https_proxy="http://example.com:8080"
all_proxy="http://example.com:8080"
http_proxy="http://example.com:8080"
HTTP_PROXY="http://example.com:8080"
HTTPS_PROXY="http://example.com:8080"
no_proxy="*.example.com,127.0.0.1,0.0.0.0,localhost"
```

Defining drop-in units for core services

The **bootc** and **podman** tools commonly need proxy configuration. At the current time, **bootc** does not always run as a **systemd** unit.

```
# /usr/lib/systemd/system/bootc-fetch-apply-updates.service.d/99-proxy.conf
[Service]
EnvironmentFile=/etc/example-proxy.env
```

Defining proxy use for podman systemd units

Using the Podman **systemd** configuration, similarly add **EnvironmentFile=/etc/example-proxy.env**. You can set the configuration for proxy and environment settings of **podman** and containers in the **/etc/containers/containers.conf** configuration file as a root user or in the **$HOME/.config/containers/containers.conf** configuration file as a non-root user.

CHAPTER 16. APPENDIX: GETTING THE SOURCE CODE OF CONTAINER IMAGES

You can find the source code for bootc image in the Red Hat Ecosystem Catalog .

Procedure

1. Access the Red Hat Ecosystem Catalog and search for **rhel-bootc**.
2. In the **Get this image** tab, click **Get the source** and follow the instructions.
3. After you extract the content, the input RPM package list and other content resources are available in the **extra_src_dir** directory.
   The .tar files are snapshots of the input git repository, and contain YAML files with the package lists.

CHAPTER 17. APPENDIX: CONTRIBUTING TO THE UPSTREAM PROJECTS

You can contribute to the following upstream bootc projects:

- The upstream git repository is in CentOS Stream.
- The CentOS Stream sources primarily track the Fedora upstream project.