Лабораторная работа №6.
Введение в контейнеризацию и защита виртуализированных сред

Цель работы: Изучение базовых понятий контейнеризации, методов защиты виртуализированных сред и контейнеров.

1. Введение в контейнеризацию

1.1. Основные концепции контейнеризации

Контейнеризация — технология абстракции, которая позволяет упаковывать и исполнять приложения вместе со всеми их зависимостями в изолированных средах, называемых *контейнерами*. Такие среды могут быть запущены практически на любой платформе, главное, чтобы ядро ОС совпадало с тем, что используется в контейнере. Контейнеризацию часто сравнивают с виртуализацией, в процессе которой виртуализируется аппаратное обеспечение и ОС, однако при контейнерном методе абстракции поддаются только уровни приложений выше системного.

Контейнеризация была создана в ответ на сложности, связанные с развертыванием и масштабированием приложений в среде с неоднородными компьютерными системами и различными требованиями к зависимостям. Она позволяет разработчикам упаковывать все необходимые компоненты приложения (app) — код, библиотеки, среду выполнения, настройки — в один контейнер, который затем можно развернуть на любой поддерживаемой платформе без внесения изменений.

Процесс контейнеризации возможен благодаря специальным контейнерным движкам и оркестраторам вроде *Docker*, *Linux Containers* (LXC), *Kubernetes* и другим. С их помощью создается изолированная среда, которая запускает контейнер и обеспечивает ему доступ к ресурсам ОС (обычно это Linux): процессору, оперативной памяти, дисковому пространству. При этом контейнеры работают независимо от операционной системы и программного обеспечения, установленных на конкретной машине. Все зависимости, нужные для правильной работы приложения, уже упакованы в эту среду, поэтому оно функционирует практически в любом окружении. Контейнеризация обеспечивает совместимость приложений даже в условиях, когда у них имеются разные требования к версиям программ и настройкам.

Еще одной задачей контейнеризации является создание удобной рабочей инфраструктуры. Разработчики могут создавать контейнеры с необходимыми настройками, инструментами и зависимостями и легко распространять их между всеми членами команды для более безопасной и комфортной работы. Такой подход обычно используется при организации микросервисной архитектуры. Кроме того, эта технология используется для более эффективного контроля ресурсов и снижения нагрузки на систему.

Суть контейнеризации состоит в том, чтобы отдельные приложения можно было запускать в различном программном окружении без каких-либо конфликтов между контейнерным и установленным на компьютере ПО. При этом важно, чтобы ОС этого компьютера базировалась на том же ядре, на каком функционирует контейнер. Условно говоря, если виртуальную машину с ОС Windows возможно создать на сервере с Linux, то контейнер — нет. В нем можно использовать только приложения, совместимые с семейством системы хоста. Такое ограничение связано с разницей в архитектуре системных ядер.

В остальном содержание контейнера может быть любым. При контейнеризации чаще всего применяются два типа контейнеров: приложений и операционной системы.

Контейнеры приложений (Application Containers)

Самый распространенный вид контейнеров, который в основном используется для организации работы микросервисов с горизонтальной масштабируемостью. В контейнер приложения упаковываются зависимости, необходимые для выполнения одного определенного процесса. Он содержит только те компоненты, благодаря которым работает тот или иной микросервис. То есть контейнер приложения отвечает за запуск отдельного процесса, а тот, в свою очередь, запускает одно приложение. При этом он полностью изолирован как от влияния других контейнеров, так и от влияния хостовой ОС.

Такая изоляция микросервисов позволяет выстраивать сложные многоконтейнерные приложения, чьи процессы выполняются стабильно и безопасно, независимо друг от друга. Контейнеры приложений могут передавать данные, «общаться» между собой, но если в одном из них произойдет сбой, это никак не скажется на работе остальных. Например, вредоносный код не сможет распространиться из одного контейнера в другой. Контейнеризация этого типа предоставляет гарантии воспроизводимости и согласованности работы приложения на разных окружениях. Больше всего подходит для неизменяемых инфраструктур. Наиболее популярным инструментом для создания контейнеров приложений является *Docker*.

Контейнеры ОС (System Containers)

Системные контейнеры по своему функционалу ближе к виртуальным машинам, которые создаются при виртуализации. Они представляют собой не просто среду для выполнения одного приложения, а полноценный экземпляр ОС со своим ядром и всеми необходимыми файлами настроек, только функционирующий внутри хостовой системы. Контейнеры ОС особенно полезны для больших классических приложений, процессы в которых неотделимы и не могут выполняться изолированно. В этом случае контейнеризация позволяет осуществить внутри контейнера запуск дополнительных процессов, помимо основного, и создать независимую настроенную среду для разработки и тестирования.

Создание контейнеров операционной системы обычно осуществляется с использованием шаблонов или образов, которые определяют их структуру и содержимое. С помощью таких образов можно формировать контейнеры с

идентичными средами, применяя одинаковые версии пакетов и конфигурации во всех из них. Таким образом обеспечивается согласованность данных и ПО, а также удобство в управлении контейнерами.

1.2. Развертывание контейнеров с использованием Docker/Podman

Docker — программное обеспечение для автоматизации развёртывания и управления приложениями в среде виртуализации на уровне операционной системы; позволяет «упаковать» приложение со всем его окружением и зависимостями в контейнер, а также предоставляет среду по управлению контейнерами.

Установим *Docker*:

```
# apt-get update && apt-get install docker-engine
```

Запустим и добавим в автозагрузку службу *docker*:

```
# systemctl enable --now docker
```

В зависимости от действий на этапе установки команды можно будет запускать от обычного пользователя или от *root*. Будем выполнять все настройки от суперпользователя.

Важно! В реальной инфраструктуре необходимо работать с Docker от обычного пользователя.

Проверим, что *Docker* был установлен корректно:

```
# docker run hello-world
```

```
[root@alt ~]# docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
c1ec31eb5944: Pull complete
Digest: sha256:d211f485f2dd1dee407a80973c8f129f00d54604d2c90732e8e320e5038a0348
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

Образ *hello-world* является официальным тестовым образом *Docker*, который предназначен для проверки того, что *Docker* правильно установлен и

работает на вашей системе. Сам образ очень легкий и состоит из минимального кода, который просто выводит приветственное сообщение.

Docker сначала проверяет, есть ли образ *hello-world* на вашем локальном компьютере. Если он не найден, *Docker* автоматически загрузит его из *Docker Hub*, который является публичным реестром образов *Docker*.

Выведем список всех загруженных (или созданных) образов в нашей локальной среде *Docker*:

docker images

```
[root@alt ~]# docker images
REPOSITORY    TAG       IMAGE ID       CREATED         SIZE
hello-world   latest    d2c94e258dcb   17 months ago   13.3kB
```

Основная информация:

REPOSITORY — название репозитория образа;

TAG — тег образа, который обычно используется для указания версии;

IMAGE ID — уникальный идентификатор образа, присвоенный *Docker*;

CREATED — дата создания образа;

SIZE — размер образа на диске.

Создадим собственный контейнер *myapp* в *Docker*.

Для этого необходимо выполнить следующие шаги:

Шаг 1: Подготовка приложения

Сначала создадим приложение на *Python*, которое выводит сообщение “Hello, World!”.

Создадим директорию для приложения и перейдем в нее:

mkdir myapp

cd myapp

Создадим файл приложения *app.py*:

vim app.py

Запишем в файл следующий код:

print(“Hello, World!”)

Если ваше приложение требует дополнительных зависимостей, то необходимо создать файл *requirement.txt*:

vim requirement.txt

В данном случае он пустой, но в реальных приложениях вы можете указать библиотеки, которые нужно установить.

Шаг 2: Создание Dockerfile

Создадим файл *Dockerfile* в той же директории с содержимым, описывающим, как нужно собрать образ *myapp*:

vim Dockerfile

Запишем в файл следующий код:

```
# Используем официальный образ Python в качестве базового
FROM python:3.9-slim

# Установим рабочую директорию
WORKDIR /usr/src/app

# Скопируем файлы приложения
COPY app.py .
COPY requirements.txt .

# Установим зависимости (если они есть)
RUN pip install --no-cache-dir -r requirements.txt

# Определим команду, которая будет выполнена при запуске контейнера
CMD [“python3”, “./app.py”]
```

Шаг 3: Сборка образа

Соберем образ на основе *Dockerfile*.

Важно! Убедитесь, что вы находитесь в директории, содержащей *Dockerfile*.

```
# docker build -t myapp:latest .
```

```
[root@alt myapp]# docker build -t myapp:latest .
[+] Building 75.5s (10/10) FINISHED                                docker:default
 => [internal] load build definition from Dockerfile                         0.1s
 => => transferring dockerfile: 619B                                         0.0s
 => [internal] load metadata for docker.io/library/python:3.9-slim           3.9s
 => [internal] load .dockerignore                                            0.0s
 => => transferring context: 2B                                              0.0s
 => [1/5] FROM docker.io/library/python:3.9-slim@sha256:49f94609e5a997      57.9s
 => => resolve docker.io/library/python:3.9-slim@sha256:49f94609e5a997d      0.1s
 => => sha256:302e3ee498053a7b5332ac79e8efebec16e900 29.13MB / 29.13MB      51.2s
 => => sha256:4c0965d3919510b506d8856ebc050a96e996c7da 3.51MB / 3.51MB      10.8s
 => => sha256:fdeeec85abbad3878f2008f9445f15a19a5a22 14.74MB / 14.74MB       26.1s
 => => sha256:49f94609e5a997dc16086a66ac9664591854031 10.41kB / 10.41kB      0.0s
 => => sha256:93ab151da4e5310ea79c4ecf306ece628262b86a4 1.75kB / 1.75kB      0.0s
 => => sha256:9d8cb7037cd8e90893e5f430ce4c048a872511e41 5.20kB / 5.20kB      0.0s
 => => sha256:62a08b8dd4f53ad5493dabf2af00ccde91abb3771fb2 248B / 248B       11.6s
 => => extracting sha256:302e3ee498053a7b5332ac79e8efebec16e900289fc1ec      3.1s
 => => extracting sha256:4c0965d3919510b506d8856ebc050a96e996c7dae96e4f      0.4s
 => => extracting sha256:fdeeec85abbad3878f2008f9445f15a19a5a224d1b7e77      1.9s
 => => extracting sha256:62a08b8dd4f53ad5493dabf2af00ccde91abb3771fb218      0.0s
 => [internal] load build context                                            0.1s
 => => transferring context: 94B                                             0.0s
 => [2/5] WORKDIR /usr/src/app                                               0.1s
 => [3/5] COPY app.py .                                                      0.1s
 => [4/5] COPY requirements.txt .                                            0.1s
 => [5/5] RUN pip install --no-cache-dir -r requirements.txt                11.0s
 => exporting to image                                                       1.8s
 => => exporting layers                                                      1.6s
 => => writing image sha256:6677173a8e8e3e2827d1ca15de8a5e4b87b63757a8a      0.0s
 => => naming to docker.io/library/myapp:latest                              0.0s
```

Опция -t указывает на имя образа (в данном случае, *myapp*);

. указывает на текущую директорию как место, где находится *Dockerfile*.

Проверим, что созданный образ *myapp* есть в списке всех образов в нашей локальной среде *Docker*:

docker images

```
[root@alt myapp]# docker images
REPOSITORY    TAG       IMAGE ID       CREATED          SIZE
myapp         latest    6677173a8e8e   16 seconds ago   132MB
hello-world   latest    d2c94e258dcb   17 months ago    13.3kB
```

Шаг 4: Запуск контейнера

Теперь, когда образ собран, запустим контейнер на его основе:

docker run myapp

После выполнения этой команды вы должны увидеть следующий вывод:

```
[root@alt myapp]# docker run myapp
Hello, World!
```

ЗАДАНИЕ

Создать Docker-контейнер, который запускает простой Bash-скрипт. Скрипт должен проверять установленные права доступа к указанному пользователем файлу или директории.

В отчет вставьте скрины проверки запуска контейнера в интерактивном режиме.

Пример запуска контейнера в интерактивном режиме:

```
[root@alt bashapp]# docker run -it --rm --cap-add=SYS_ADMIN bashapp
Введите путь к файлу для проверки прав доступа: /etc/passwd
Права доступа к файлу '/etc/passwd': -rw-r--r--
```

Podman — инструмент для управления контейнерами, который предоставляет функционал, аналогичный Docker, но без необходимости использования демона. Он позволяет создавать, запускать и управлять контейнерами и образами. Podman поддерживает управление контейнерами как пользователем, так и от имени суперпользователя, что повышает безопасность.

Установим *Podman*:

apt-get update && apt-get install podman

Для проверки запустим контейнер с простым образом *alpine* с командой *echo* для вывода текста:

podman run --name myalpine alpine echo “Hello from Podman!”

```
[root@alt ~]# podman run --name myalpine alpine echo "Hello from Podman!"
Hello from Podman!
```

Образ *alpine* представляет собой легковесный дистрибутив Linux.

Создадим собственный контейнер *myapp2* в *Podman*.

Для этого необходимо выполнить следующие шаги:

Шаг 1: Подготовка приложения

Сначала создадим приложение на *Python*, которое выводит сообщение “Hello, World!”.

Создадим директорию для приложения и перейдем в нее:

```
# mkdir myapp2
# cd myapp2
```

Создадим файл приложения *app.py*:

```
# vim app.py
```

Запишем в файл следующий код:

```
print(“Hello, World!”)
```

Шаг 2: Создание Containerfile

Создадим файл *Containerfile* в той же директории с содержимым, описывающим, как нужно собрать образ *myapp2*:

```
# vim Containerfile
```

Примечание! *Containerfile, Dockerfile* — различные названия для одного и того же файла, где описаны инструкции создания образа. В *Podman* поддерживаются оба формата.

Запишем в файл следующий код:

```
# Используем официальный образ Python в качестве базового
FROM python:3.9-slim

# Установим рабочую директорию
WORKDIR /app

# Скопируем файлы приложения
COPY app.py .

# Определим команду, которая будет выполнена при запуске контейнера
CMD [“python3”, “./app.py”]
```

Шаг 3: Сборка образа

Соберем образ на основе *Containerfile*.

Важно! Убедитесь, что вы находитесь в директории, содержащей *Containerfile*.

podman build -t myapp2:latest .

```
[root@alt myapp2]# podman build -t myapp2:latest .
STEP 1/4: FROM python:3.9-slim
Resolved "python" as an alias (/etc/containers/registries.conf.d/000-shortnames.conf)
Trying to pull docker.io/library/python:3.9-slim...
Getting image source signatures
Copying blob 62a08b8dd4f5 done   |
Copying blob 4c0965d39195 done   |
Copying blob 302e3ee49805 done   |
Copying blob fdeeec85abba done   |
Copying config 9d8cb7037c done   |
Writing manifest to image destination
STEP 2/4: WORKDIR /app
--> dbc220083b5e
STEP 3/4: COPY app.py .
--> 3943a4f44d1e
STEP 4/4: CMD ["python3", "./app.py"]
COMMIT myapp2:latest
--> 22fdb6f53658
Successfully tagged localhost/myapp2:latest
22fdb6f53658d7b0482621d6fb766868a30b1a7e7e45598806dfdb5ba98f09c0
```

Опция -t указывает на имя образа (в данном случае, *myapp2*);

. указывает на текущую директорию как место, где находится *Containerfile*.

Проверим, что созданный образ *myapp2* есть в списке всех образов в нашей локальной среде *Podman*:

podman images

```
[root@alt myapp2]# podman images
REPOSITORY                TAG         IMAGE ID      CREATED         SIZE
localhost/myapp2          latest      22fdb6f53658  24 seconds ago  130 MB
docker.io/library/python  3.9-slim    9d8cb7037cd8  4 weeks ago     130 MB
docker.io/library/alpine  latest      91ef0af61f39  5 weeks ago     8.09 MB
```

Шаг 4: Запуск контейнера

Теперь, когда образ собран, запустим контейнер на его основе:

podman run myapp2

После выполнения этой команды вы должны увидеть следующий вывод:

```
[root@alt myapp2]# podman run myapp2
Hello, World!
```

ЗАДАНИЕ

Создать Podman контейнер, который запускает Bash-скрипт для проверки наличия указанных пользователем открытых портов на машине.

В отчет вставьте скрины проверки запуска контейнера в интерактивном режиме.

2. Защита контейнеризации

2.1. Использование режима rootless в Podman

Для использования *Podman* непривилегированными пользователями, необходимо произвести ряд дополнительных действий по настройке прав.

Разрешим создание пользовательских пространств имен:

```
# sysctl kernel.unprivileged_userns_clone=1
# echo 'kernel.unprivileged_userns_clone=1' >> /etc/sysctl.d/42-podman.conf
```

Предоставим права на запуск исполняемых файлов */usr/bin/newuidmap* и */usr/bin/newgidmap*:

```
# control newgidmap public
# control newuidmap public
```

или

```
# chmod o+x /usr/bin/newuidmap
# chmod o+x /usr/bin/newgidmap
```

Поскольку эти исполняемые файлы обращаются к системным вызовам *setuid()* и *setgid()*, установим файловые разрешения файлов. Такой подход позволяет программам выполнять определенные привилегированные операции без необходимости устанавливать бит *SUID/SGID*.

Установим соответствующие capabilities для файлов */usr/bin/newuidmap* и */usr/bin/newgidmap*:

```
# setcap cap_setuid+ep /usr/bin/newuidmap
# setcap cap_setgid+ep /usr/bin/newgidmap
```

Чтобы непривилегированные пользователя могли запускать *Podman*, для каждого пользователя должна существовать запись конфигурации *subuid* и *subgid*. Новые пользователи, созданные после установки *Podman*, имеют эти записи по умолчанию.

Для пользователей, у которых нет записей в */etc/subuid* и */etc/subgid*, можно создать запись с помощью следующей команды:

```
# usermod --add-subuids 100000-165536 --add-subgids 100000-165536 имя_пользователя
```

Данная команда выделяет заданный диапазон UID и GID пользователю, что позволит пользователю и группе с именем пользователя запускать контейнеры *Podman*.

Указанный выше диапазон *UID* и *GID* уже может быть занят другим пользователем, т.к. это диапазон по умолчанию для первого пользователя. Просмотреть занятые диапазоны можно в файлах */etc/subuid* и */etc/subgid*.

Для применения изменений в *subuid* и *subgid* необходимо выполнить команду:

```
# podman system migrate
```

Создадим пользователя *podmanuser* с паролем *P@ssw0rd*.

```
# useradd podmanuser
# passwd podmanuser
```

Проверим наличие записи пользователя *podmanuser* в файлах */etc/subuid* и */etc/subgid*:

```
# cat /etc/subuid
# cat /etc/subgid
```

```
[root@alt ~]# cat /etc/subuid
podmanuser:100000:65536
[root@alt ~]# cat /etc/subgid
podmanuser:100000:65536
```

Теперь зайдем в графический сеанс пользователя *podmanuser* и запустим контейнер с простым образом *busybox* и командой *echo* для вывода текста.

```
$ podman run --name mybusybox busybox echo "Hello from Podman using BusyBox!"
```

```
[podmanuser@alt ~]$ podman run --name mybusybox busybox echo "Hello from Podman using BusyBox!"
Resolved "busybox" as an alias (/etc/containers/registries.conf.d/000-shortnames.conf)
Trying to pull docker.io/library/busybox:latest...
Getting image source signatures
Copying blob a46fbb00284b done   |
Copying config 27a71e19c9 done   |
Writing manifest to image destination
Hello from Podman using BusyBox!
```

Образ *BusyBox* часто используется для выполнения простых задач в контейнерах.

ЗАДАНИЕ

Запустите любой *Podman* контейнер под учетной записью *user*.

В отчет вставьте скрины запуска контейнера под учетной записью *user*.

2.2. Применение Podsec для защиты контейнеров

Podsec — набор скриптов для развертывания и поддержки безопасных *rootless*-решений для контейнеров.

Пакет *Podsec* включает утилиты, позволяющие:

- настраивать безопасные политики доступа к контейнерным приложениям (каталог */etc/containers/*);
- устанавливать приватный реестр и веб-сервер для управления подписями контейнерных образов;
- создавать пользователей с правами на создание, подпись и размещение *Docker*-образов в реестре;
- создавать пользователей с правами на запуск контейнеров в режиме *rootless*;

- загружать *Docker*-образы из OCI-архива (формат образа, определённый Open Container Initiative), размещать их на локальной системе, подписывать и загружать в реестр;
- развертывать кластер *Kubernetes* в режиме *rootless*.

Установим пакет Podsec:

```
# apt-get update && apt-get install podsec
```

Скрипт *podsec-create-policy*

Настроим политику доступа к образам различным категориям пользователей.

Скрипт *podsec-create-policy* формирует в файлах */etc/containers/policy.json*, */etc/containers/registries.d/default.yaml* максимально защищенную политику доступа к образам — по умолчанию допускается доступ только к подписанным образам локального регистратора *registry.local*. Данная политика распространяется как на пользователей имеющих права суперпользователя, так и на пользователей группы *podsec*, создаваемых *podsec*-скриптом *podsec-create-podmanusers*.

Пользователи группы *podsec-dev*, создаваемые *podsec*-скриптом *podsec-create-imagemakeruser* имеют неограниченные права на доступ, формирование образов, их подпись и помещение на локальный регистратор *registry.local.*

Важно! IP-адрес регистратора и сервера подписей не должен быть локальным 127.0.0.1.

Выполним скрипт для IP-адреса 192.168.10.10:

```
# podsec-create-policy 192.168.10.10
```

```
[root@alt ~]# podsec-create-policy 192.168.10.10
Добавление привязки доменов registry.local sigstore.local к IP-адресу 192.168.10.10
Создание группы podman
Инициализация каталога /var/sigstore/ и подкаталогов хранения открытых ключей и подписей образов
Создание каталога и подкаталогов  /var/sigstore/
Создание группы podman_dev
Создание с сохранением предыдущего файла политик /etc/containers/policy.json
Создание с сохранением предыдущего файл /etc/containers/registries.d/default.yaml описания доступа к открытым ключам подписантов
Добавление insecure-доступа к регистратору registry.local в файле /etc/containers/registries.conf
```

Скрипт *podsec-create-services*

Развернем локальный регистратор образов и сервера подписей образов.

Скрипт запускает и добавляет в автозагрузку сервисы *docker-registry* и *nginx* для поддержки регистратора докер-образов и сервера подписей образов. Регистратор образов принимает запросы по адресу *http://registry.local* и хранит образы в каталоге */var/lib/containers/storage/volumes/registry/_data/*. Сервис подписей образов принимает запросы по адресу *http://sigstore.local:81*, хранит подписи образов в каталоге */var/sigstore/sigstore/*, открытые ключи в каталоге */var/sigstore/keys/*.

```
# podsec-create-services
```

```
[root@alt ~]# podsec-create-services
Synchronizing state of nginx.service with SysV service script with /lib/systemd/systemd-sysv-install.
Executing: /lib/systemd/systemd-sysv-install enable nginx
Created symlink /etc/systemd/system/multi-user.target.wants/nginx.service → /lib/systemd/system/nginx.service.
registry
Created symlink /etc/systemd/system/multi-user.target.wants/docker-registry.service → /lib/systemd/system/docker-registry.service.
```

Скрипт *podsec-create-imagemakeruser*

Создадим пользователя разработчика образов контейнера, который обладает следующими правами:

- менять установленный администратором безопасности средства контейнеризации пароль;
- создавать, модифицировать и удалять образы контейнеров.

При создании каждого пользователя необходимо задать:

- пароль пользователя;
- тип ключа: RSA, DSA и Elgamal, DSA (только для подписи), RSA (только для подписи), имеющийся на карте ключ;
- срок действия ключа;
- полное имя;
- Email (используется в дальнейшем для подписи образов);
- примечание;
- пароль для подписи образов.

Создадим пользователя *podsecdev* со следующими параметрами:
пароль — *P@ssw0rd*;
тип ключа — по умолчанию;
размер ключа — по умолчанию;
срок действия пароля — не ограничен;
полное имя — *Podsec Developer*;
адрес электронной почты — *podsecdev@mail.ru*;
примечаний нет.

```
# podsec-create-imagemakeruser podsecdev@registry.local
```

Зайдем в графический сеанс пользователя *podsecdev* и загрузим образ *base* с тегом *latest* из реестра *registry.altlinux.org*:

```
$ podman pull registry.altlinux.org/alt/base:latest
```

Выведем список всех образов в нашей локальной среде Podman:

```
$ podman images
```

```
[podsecdev@alt ~]$ podman images
REPOSITORY                        TAG         IMAGE ID      CREATED     SIZE
registry.altlinux.org/alt/base    latest      140863f0fff0  7 days ago  387 MB
```