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. 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. Развертывание контейнеров с использованием Docker/Podman

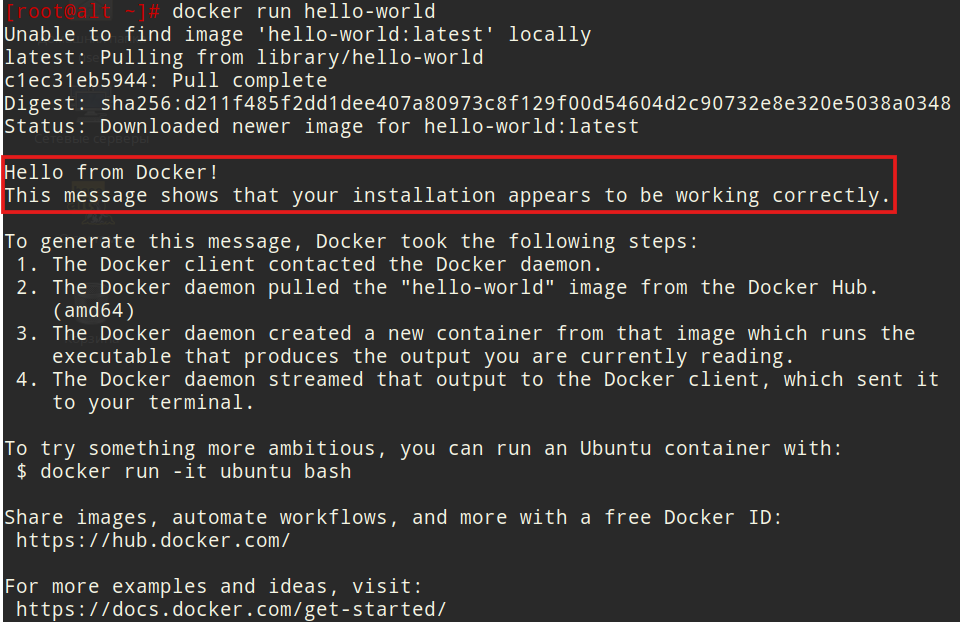
*Docker* — программное обеспечение для автоматизации развёртывания и управления приложениями в среде виртуализации на уровне операционной системы; позволяет «упаковать» приложение со всем его окружением и зависимостями в контейнер, а также предоставляет среду по управлению контейнерами.

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

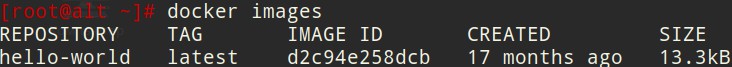
Образ *hello-world* является официальным тестовым образом *Docker*, который предназначен для проверки того, что *Docker* правильно установлен и

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



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”]

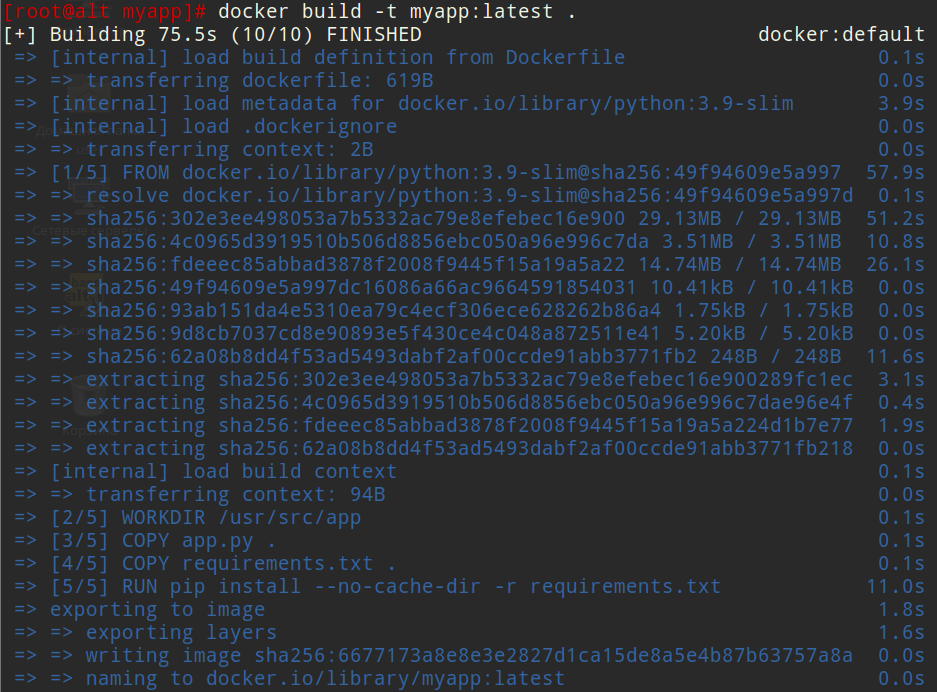
Шаг 3: Сборка образа

Соберем образ на основе *Dockerfile*.

Важно! Убедитесь, что вы находитесь в директории, содержащей

*Dockerfile*.

# docker build -t myapp:latest .

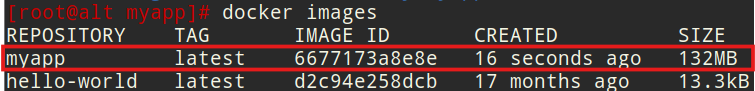


Опция -t указывает на имя образа (в данном случае, *myapp*);

. 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



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:

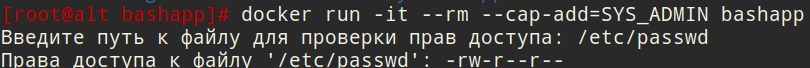


## ЗАДАНИЕ

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:



*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!”



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”]

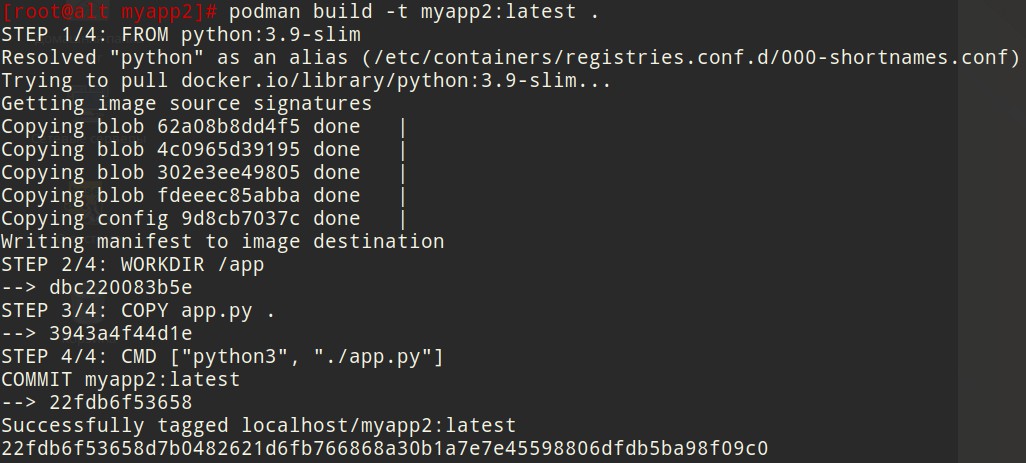
Шаг 3: Сборка образа

Соберем образ на основе *Containerfile*.

Важно! Убедитесь, что вы находитесь в директории, содержащей

*Containerfile*.

# podman build -t myapp2:latest .



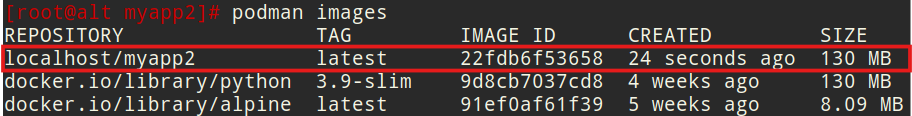
Опция -t указывает на имя образа (в данном случае, *myapp2*);

. 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



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:



## 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.

1. Защита контейнеризации
   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*.

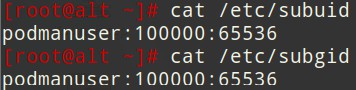
Для применения изменений в *subuid* и *subgid* необходимо выполнить команду:

# podman system migrate

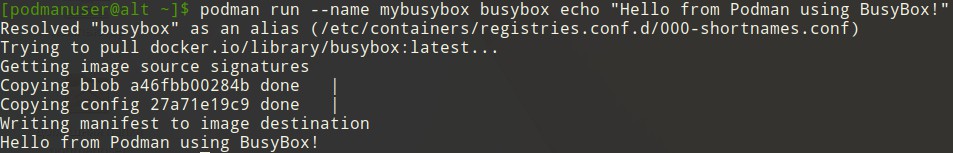
Создадим пользователя *podmanuser* с паролем *P@ssw0rd*. # useradd podmanuser

# passwd podmanuser

Проверим наличие записи пользователя *podmanuser* в файлах

*/etc/subuid* и */etc/subgid*: # cat /etc/subuid # cat /etc/subgid

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!"

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*.

* 1. 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

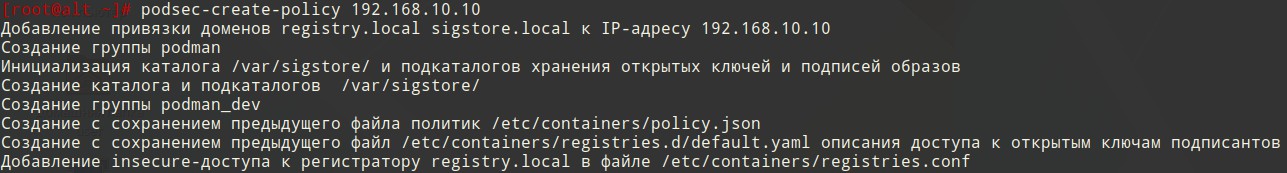
*/etc/containers/policy.json*, */etc/containers/registries.d/default.yaml* максимально защищенную политику доступа к образам — по умолчанию допускается доступ только к подписанным образам локального регистратора *registry.local*. Данная политика распространяется как на пользователей имеющих права суперпользователя, 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

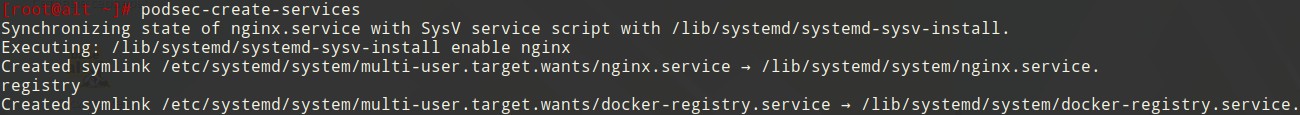


Скрипт *podsec-create-services*

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*](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



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](mailto:%23podsec-create-imagemakeruserpodsecdev@registry.local)

Зайдем в графический сеанс пользователя *podsecdev* и загрузим образ

*base* c 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

