# Практика 7. Безопасность контейнеров

Домашнее задание: Необходимо развернуть окружение с веб-сайтом с использованием базы данных, настроить постоянное хранение и создать сетевое окружение. Проверить получившийся контейнер и образ на безопасность. В отчете привести скриншоты и описать последовательность действий. Разобрать вывод сканеров безопасности и предложить меры по их исправлению.

Утилиты для проверки docker-контейнера

1. https://github.com/aquasecurity/trivy
2. https://github.com/docker/docker-bench-security
3. https://docs.docker.com/engine/scan/ (Возможны проблемы, в связи с миграцией на scout)

Share for dojo and Ubuntu Virtual Image - \\10.114.6.6\shared

https://hackmd.io/@0x41/SDL_Docker

Литература:

1. https://docs.docker.com/engine/security/
2. https://katacoda.com/loodse/courses/docker
3. https://disk.yandex.com/d/t0qRW12bDvgubQ
4. https://training.play-with-docker.com/ops-stage2/

## Список вопросов для самопроверки:

1. Что такое виртуализация? Какие типы виртуализации вы знаете?
2. Что такое гипервизор?
3. Расскажите о ключевом различии между виртуализацией и контейнеризацией
4. Опишите составные части архитектуры Docker
5. Дайте определение образу и контейнеру. Чем они отличаются? Какие еще бывают объекты?
6. Назовите наиболее важные команды Docker
7. Что такое пространства имен в Docker?
8. Какие сети доступны по умолчанию в Docker?
9. Если вы остановите контейнер — потеряете данные?
10. Поясните разницу между docker run и docker create

11. Расскажите о CMD и ENTRYPOINT в Dockerfile
12. Опишите все возможные состояния контейнера Docker
13. Какие типы монтирования доступны в Docker?
14. Расскажите о Docker Trusted Registry
15. Какие уязвимости могут быть связаны с использованием Docker?
16. Какие практики обеспечивают безопасность контейнеров Docker?
17. Какие инструменты можно использовать для сканирования Docker-контейнеров на наличие уязвимостей?
18. Какие механизмы в Docker обеспечивают изоляцию контейнеров и защиту от атак сетевого уровня?
19. Какие инструменты можно использовать для мониторинга и аудита безопасности Docker-контейнеров?
20. Какие меры предпринимаются при сборке и поддержки Docker-образов, чтобы снизить риск уязвимостей?
21. Каким образом можно хранить секреты в Docker (https://docs.docker.com/engine/swarm/secrets/)
22. Как запустить Docker в непривилегированном режиме?

# Hello World

## Install docker

```
sudo apt-get remove docker docker-engine docker.io

sudo apt-get update

sudo apt-get install apt-transport-https ca-certificates curl software-properties-comm

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -

sudo apt-key fingerprint 0EBFCD88

sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
xenial \
stable"

sudo apt-get update

sudo apt-get install docker-ce

sudo docker run hello-world

# Add user to group
sudo usermod -aG docker $(whoami)

(c) https://docs.docker.com/engine/install/ubuntu/
```

Для начала найдем интересующий нас образ (первая команда) и установим его:

```
docker search redis
docker run -d redis
```

Для запуска контейнера в фоновом режиме необходимо указать параметр -d. По умолчанию Docker запускает последнюю доступную версию. Если требуется определенная версия, она может быть указана в виде тега, например, версия 3.2 будет

```
docker run -d redis: 3.2.
```

Для просмотра информации о запущенных в фоне контейнерах есть команда:

```
docker ps
```

Так же имеется две полезные команды. Первая выводит дополнительную информацию о контейнере. Вторая выводит его логи.

```
```

docker inspect <friendly-name|container-id>  docker logs <friendly-name|container-id>

Redis установлен, но не доступен вне контейнера. 6379 — используемый порт Redis. Для проброса порта имеется такая команда -p <host-port>:<container-port>. Пример:

```
docker run -d --name redisHostPort -p 6379:6379 redis:latest
```

По умолчанию порт на хосте сопоставлен с 0.0.0.0, что означает доступ со всех IP-адресов. Вы можете указать конкретный IP-адрес при определении сопоставления портов, например, -p 127.0.0.1:6379:6379

Как хранить данные и не потерять их при переустановке контейнера

Если открыть документацию по Redis для Docker , то увидим информацию по хранению данных. Данный образ хранит её в /data .

Любые данные, которые необходимо сохранить на Docker хосте, а не внутри контейнера redis, должны храниться в /opt/docker/data/redis, что задается отдельным параметром: docker run -d --name redisMapped -v

/opt/docker/data/redis:/data redis
Взаимодействие внутри контейнера

Ранее мы использовали -d для выполнения контейнера в отдельном фоновом состоянии.

Без указания этого контейнер будет работать на переднем плане.

Если нужно взаимодействовать с контейнером (например, для доступа к оболочке bash), нужно добавить опции -it.

Примеры:

docker run ubuntu ps запустит контейнер ubuntu и выполнит команду ps. docker run -it ubuntu bash даст доступ к bash shell внутри контейнера.

# HTML страница

Цель урока: создать веб сайт со статичной страницей в контейнере.

## Создадим Dockerfile

Docker images начинаются с базового image(образа), который включает в себя зависимости платформы для приложения. Этот базовый образ определяется как команда в Dockerfile, который является списком инструкций (команд), описывающих как развернуть приложение.

В примерах будет использован как базовый образ NGINX версии Alpine (настроенный веб сервер в дистрибутиве linux alpine).
Так же создайте простой index.html файл в папке, из которой идет работа.
Создадим Dockerfile:

```
FROM nginx:alpine  COPY . /usr/share/nginx/html
```

Первая строчка определяет базовый образ. Вторая строчка копирует контент текущей папки во внутрь контейнера (наш index.html).

## Сборка Docker Image

Для сборки используется команда build. Она принимает несколько параметров. Например, параметр -t позволяет указать имя для и тег для изображения (используется часто как номер версии). Пример:

docker build -t webserver-image:v1 .

Для просмотра списка изображений используйте команду:

docker images

Запускаем контейнер с пробросом 80 порта.

docker run -d -p 80:80 webserver-image:v1

Теперь можно проверить — работает ли наш сайт? Воспользуемся утилитой curl.
curl dockerDocker, лекция 3. Сборка образов

Как собрать образ, основываясь на собственных требованиях.

Для этой лекции контейнер будет запускать статическое HTML-приложение с использованием веб-сервера Nginx.
Имя компьютера, на котором запущен контейнер: Docker. Если вы хотите получить доступ к какой-либо из служб — используйте docker вместо localhost или 0.0.0.0.

## Об образах Docker

```
Образы Docker создаются на основе Dockerfile. Dockerfile определяет все шаги, необходи
Образ позволяет переносить его между различными средами и быть уверенным, что он зара
Dockerfile позволяет пользователям расширять существующие изображения вместо создания
Все образы Docker начинаются с базового образа. Базовый образ – это те же изображения
```

Эти базовые образы используются в качестве основы для запуска вашего приложения. Например, в этом уроке мы требуем, чтобы NGINX был настроен и запущен в системе, прежде чем мы сможем развернуть наши статические HTML-файлы. Поэтому мы хотим использовать NGINX в качестве базового образа.

Dockerfile — это простые текстовые файлы с командой в каждой строке. Чтобы определить базовый образ, нужно использовать инструкцию FROM <image-name>: <tag>.
Добавим образ в Dockerfile.

```
FROM nginx:1.11-alpine
```

Важно: заманчиво использовать тег :lastest, однако есть вероятность построить образ не на той версии, которую бы хотелось. Для исключения ошибок в работе приложения рекомендуется использовать конкретную версию.

# Время команд в Dockerfile

Определив базовый образ, нам нужно запустить различные команды для настройки нашего образа. Есть много команд, которые могут помочь с этим, две главные команды это COPY и RUN.
RUN <команда> позволяет вам выполнить любую команду, как в командной строке, например, установить различные пакеты приложений или выполнить команду сборки. Результаты RUN сохраняются в образе, поэтому важно не оставлять ненужные или временные файлы на диске, так как они будут включены в образ.
COPY <src> <dest> позволяет копировать файлы из каталога, содержащего Dockerfile, в образ контейнера. Это чрезвычайно полезно для исходного кода и

ресурсов, которые вы хотите развернуть внутри своего контейнера.
Для примера создадим новый файл index.html, который добавим в контейнер. На следующей строке после команды FROM созданного ранее dockerfile добавим команду команду COPY, чтобы скопировать index.html в каталог с именем /usr/share/nginx/html

```
COPY index.html /usr/share/nginx/html/index.html
```

```
С настроенным образом Docker и определившись, какие порты будут открыты, нужно указат
С настроенным образом Docker и определением, какие порты мы хотим сделать доступными,
Строка CMD в Dockerfile определяет команду по умолчанию, запускаемую при запуске конт
```

```
["cmd", "-a", "arga value", "-b", "argb-value"]
```

```
Массив будет объединен вместе и итоговая команда ниже будет выполнена:
```

```
cmd -a "arga value" -b argb-value
```

```
Ещё пример — передадим команду nginx. По-умолчанию демон NGINX будет выключен:
```

CMD ["nginx", "-g", "daemon off;"]

## Порты

```
Файлы скопированы в наш образ, теперь нужно определить, на каком порте приложение дол
Используя команду EXPOSE \<port>, нужно сообщить Docker, какие порты должны быть откр
```

```
EXPOSE 80
```

После написания Dockerfile нужно использовать команду docker build, чтобы превратить
Итоговый dockerfile этой лекции приведен ниже. Это nginx версии 1.11, в который скопир

```
FROM nginx:1.11-alpine COPY index.html /usr/share/nginx/html/index.html EXPOSE 80 CMD
```

Итак, выполняем команду docker build для создания образа. Можно дать образу понятное и

```
docker build -t my-nginx-image:latest .
```

Не стоит забывать о команде docker images для просмотра списка образов.
После успешного создания образа можно запустить контейнер так же, как это описано в пр
NGINX предназначен для работы в качестве фоновой службы, поэтому ему нужно включить оп

```
docker run -d -p 80:80 image-id или friendly-tag-name
```

Теперь можно получить доступ к запущенному веб-серверу по имени хоста docker. После за

# Сборка приложения node.js

В этой лекции продолжается изучение того, как создавать и развертывать приложения в вы
Среда настроена с доступом к личному экземпляру Docker, а код для приложения Expressjs
Имя компьютера, на котором запущен Docker, называется Docker. Если нужно получить дост

## Базовый образ

Как описано в предыдущей лекции, все образы начинаются с базового изображения, в идеал
Образ для node 10.0 это node:10-alpine. Это образ на основе alpine linux, который мень

```
Помимо базового образа также необходимо создать базовые каталоги, из которых запускает
Можно определить рабочий каталог, используя WORKDIR <каталог>, чтобы гарантировать, чт
```

В нашем Dockerfile определим выше описанное:

```
FROM node:10-alpine
RUN mkdir -p /src/app
WORKDIR /src/app
```

## Работа с NPM

Базовый образ и местоположение приложения настроены. Следующим этапом является установка зависимостей, необходимых для запуска приложения. Для Node.js это означает запуск установки NPM.

Чтобы свести время сборки к минимуму, Docker кэширует результаты выполнения строки в файле Docker для использования в будущей сборке. Если что-то изменилось, то Docker сделает недействительными текущую и все последующие строки, чтобы убедиться, что все обновлено.

Сделаем так, чтобы npm install запускался только в том случае, если что-то в нашем файле package.json изменилось. Если ничего не изменилось, то продолжим использовать версию кеша для ускорения развертывания. Используя COPY package.json <dest>, можно сделать кеш команды RUN npm install недействительным, если файл package.json изменился. Если файл не изменился, то кэш не будет признан действительным, и будут использованы кэшированные результаты команды установки npm.
Добавим описанные выше действия в dockerfile:

```
COPY package.json /src/app/package.json
RUN npm install
```

Пример package.json:

{ "name": "scrapbook-node-docker-client-as-container", "version": "0.0.0", "private": true, "scripts": { "start": "node ./bin/www" }, "dependencies": { "express": "~4.9.0", "body-parser": "~1.8.1", "cookie-parser": "~1.3.3", "morgan": "~1.3.0", "serve-favicon": "~2.1.3", "debug": "~2.0.0", "jade": "~1.6.0" } }

## Время развертывания

Создадим необходимые шаги в Dockerfile, чтобы завершить развертывание приложения, как это было на прошлой лекции.

Скопируем файлы для развертывания, в Dockerfile следует использовать COPY. <dest dir>.

После того, как исходный код скопирован, порты, к которым требуется доступ к приложению, определяются с помощью EXPOSE <порт>.

Наконец, приложение готово к установке зависимостей. При использовании Node.js следует использовать один хитрый прием: запустить команду npm. Параметр start описан в «scripts» — «start» файла package.json. Ниже приведен пример и краткое описание — почему так.

```
COPY . /src/app
EXPOSE 3000
CMD [ "npm", "start" ]
```

После того, как мы установили наши зависимости с помощью npm install, мы хотим скопировать остальную часть исходного кода нашего приложения. Разделение установки зависимостей и копирование исходного кода позволяет нам использовать кеш при необходимости.

Если мы скопируем наш код перед запуском npm install, он будет запускаться каждый раз, так как наш код изменился бы. Копируя просто package.json, мы можем быть уверены, что кеш становится недействительным только тогда, когда содержимое нашего пакета изменилось. Итоговый dockerfile:

```
FROM node:10-alpine
RUN mkdir -p /src/app
WORKDIR /src/app
COPY package.json /src/app/package.json
RUN npm install
COPY . /src/app
EXPOSE 3000
CMD [ "npm", "start" ]
```

Запускаем

```
docker build -t my-nodejs-app .
docker run -d --name my-running-app -p 3000:3000 my-nodejs-app
```

И проверяем:

```
curl http://docker:3000
```

### Переменные окружения

Образы docker должны быть спроектированы так, чтобы их можно было переносить из одной среды в другую без внесения каких-либо изменений или необходимости восстановления. Следуя этому шаблону, вы можете быть уверены, что если он работает в одной среде, например, в промежуточной, то он будет работать в другой среде, например в рабочей среде.

С помощью Docker переменные среды могут быть определены при запуске контейнера. Например, для приложений Node.js вы должны определить переменную среды для NODE_ENV при запуске.

Используя опцию -e, вы можете установить имя и значение -e NODE_ENV = production

```
docker run -d --name my-production-running-app -e NODE_ENV=production -p 3000:3000 my
```

## Docker OnBuild

В этом сценарии мы рассмотрим, как можно оптимизировать Dockerfile, используя OnBuild инструкции.

Среда лекции настроена с помощью примера приложения Node.js, однако подходы могут быть применены к любому изображению. Имя компьютера, на котором запущен Docker, называется Docker. Если нужно получить доступ к какой-либо из служб, тогда используйте docker вместо localhost или 0.0.0.0.

## Об OnBuild

В то время как Dockerfile выполняется в порядке сверху вниз, с помощью OnBuild вы можете запустить инструкцию, которая будет выполнена позже, когда образ используется в качестве основы для другого образа.
В результате вы можете отложить выполнение, чтобы оно зависело от приложения, которое вы создаете, например, от файла package.json приложения.
Ниже приведен файл Docker Node.js OnBuild. В отличие от сценария предыдущей лекции команды приложения имеют префикс ONBUILD.

```
FROM node:7
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app
ONBUILD COPY package.json /usr/src/app/
ONBUILD RUN npm install
ONBUILD COPY . /usr/src/app
CMD [ "npm", "start" ]
```

В результате образ будет построен, но команды приложения с пометкой ONBUILD не будут выполняться до тех пор, пока построенный образ не будет использован в качестве базового. Затем они будут выполнены как часть сборки базового образа.

## Пример

```
Мы имеем всю логику для копирования кода, установки зависимостей и запуска приложения
Преимущество создания образов OnBuild состоит в том, что наш Dockerfile теперь намного

Пример такого dockerfile:
```

```
FROM node:7-onbuild
EXPOSE 3000
```

### Сборка и запуск контейнера

```
Для начала – сборка первого контейнера из базового docker образа.
```

```
docker build -t my-nodejs-app .
```

```
Теперь выполним команду сборки встроенного образа, который основан на предыдущем:
```

```
docker run -d --name my-running-app -p 3000:3000 my-nodejs-app
```

```
```

```
Сборка первого контейнера заняла около 2 минут при наличии хорошего интернета. Сборка
Как обычно можно проверить доступность сервиса в контейнере командой:
```

```
curl http://docker:3000
```

# Игнорирование файлов при сборке

Как можно игнорировать попадание определенных файлов в образ Docker, что может представлять угрозу безопасности и увеличивать время сборки.

Чтобы не допустить ошибочного включения в изображения конфиденциальных файлов или каталогов, нужно добавить файл с именем .dockerignore.

Пример
Dockerfile копирует рабочий каталог в образ Docker. В результате это может включать потенциально конфиденциальную информацию, такую как файл паролей, которым мы хотели бы управлять за пределами изображения. Добавим файл passwords.txt с нашим условным паролем, а так же напишем Dockerfile:

```
FROM alpine
ADD . /app
COPY cmd.sh /cmd.sh
CMD ["sh", "-c", "/cmd.sh"]
```

Соберем образ и посмотрим файлы в нем (сюрприз — файл passwords.txt будет в контейнере):

```
docker build -t password .
docker run password ls /app
```

**Как игнорировать файл?**

Следующая команда включит passwords.txt в файл .dockerignore и гарантирует, что он случайно не окажется в контейнере.

```
echo passwords.txt >> .dockerignore
```

Файл игнорирования поддерживает каталоги и регулярные выражения для определения ограничений, очень похожих на .gitignore у git.

Этот файл также можно использовать для уменьшения времени сборки, что будет рассмотрено позже. Соберем образ, теперь файла с паролем быть не должно.

```
docker build -t nopassword .
```

Если вам нужно использовать пароли как часть команды RUN, вам нужно скопировать, выполнить и удалить файлы как часть одной команды RUN. Внутри образа сохраняется только конечное состояние контейнера Docker.

### Улучшаем время сборки

```
Файл .dockerignore может гарантировать, что конфиденциальные данные не будут включены
Для примера создадим файл в 100 МБ, например big-temp-file.img и соберем образ. Это у
```

```
docker build -t large-file-context .
```

```
Так же целесообразно игнорировать каталоги .git вместе с зависимостями, которые загру
Таким же образом мы использовали файл .dockerignore для исключения конфиденциальных ф
```

```
echo big-temp-file.img >> .dockerignore
docker build -t no-large-file-context .
```

## Контейнеры данных

```
Контейнеры могут выполнять разные функции. Это может быть контейнер, создающий и хран
Один из подходов, который был ранее, заключается в использовании опции -v <host-dir>:
```

### Создадим контейнер

```
Контейнеры данных — это контейнеры, единственная цель которых заключается в том, чтоб
Как и другие контейнеры, они управляются хост-системой. Однако они не запускаются при
Чтобы создать контейнер данных, мы сначала создаем контейнер с понятным именем для ис
```

При создании контейнера мы также используем опцию -v, чтобы определить, где другие ко

```
docker create -v /config --name dataContainer busybox
```

Теперь, когда контейнер установлен, можно копировать файлы из локального клиентского к
Для копирования файлов в контейнер нужно использовать команду docker cp. Следующая ко

```
docker cp config.conf dataContainer:/config/
```

## Монтирование томов

Теперь контейнер данных имеет конфиг файл и сейчас можно ссылаться на контейнер при за
Используя опцию —volumes-from <container>, можно использовать тома монтирования из дру

```
docker run --volumes-from dataContainer ubuntu ls /config
```

Если каталог /config уже существует, то имя тома будет переопределено. Так же можно пр

## Экспорт\импорт контейнеров

Если мы хотим переместить контейнер данных на другой компьютер, мы можем экспортирова

```
docker export dataContainer > dataContainer.tar
```

Для импорта контейнера данных назад в Docker используйте команду:

```
docker import dataContainer.tar
```

## Создание сетей между контейнерами с использованием Links

```
Как разрешить нескольким контейнерам взаимодействовать друг с другом: как подключить
```

## Запустим reddis

```
Наиболее распространенным сценарием подключения к контейнерам является приложение, по
Настроим:
```

```
docker run -d --name redis-server redis
```

## Создадим Link

```
Для подключения к исходному контейнеру используется команда —link <container-name|id>
Устанавливая alias, мы разделяем, как наше приложение настроено на инфраструктуру. Эт
Рассмотрим на примере. Вызывем контейнер Alpine, который связан с redis-сервером. Опр
Во-первых, Docker установит некоторые переменные окружения на основе ссылки на контей
Можно вывести все переменные окружения, добавив команду env. Например, создадим link
```

```
docker run --link redis-server:redis alpine env
```

```
Во-вторых, Docker обновит файл HOSTS контейнера с записью для исходного контейнера с
```

```
docker run --link redis-server:redis alpine cat /etc/hosts
```

Пример вывода:

```
172.18.0.2 redis a2f3ba2b1d3c redis-server
172.18.0.3 43106ddc9e26
```

После создания link, можно пропинговать исходный контейнер так же, как если бы это бы

```
docker run --link redis-server:redis alpine ping -c 1 redis
```

## Подключение приложения

Создав link, приложения могут подключаться и взаимодействовать с исходным контейнером
Используем простое приложение на node.js, которое будет подключаться к redis:

```
docker run -d -p 3000:3000 --link redis-server:redis katacoda/redis-node-docker-examp
```

В итоге получим работающее приложение, которое можно проверить http запросом:

```
curl docker:3000
```

### Пример с redis cli

Таким же образом можно подключать контейнеры к приложениям и к своим собственным инстр
Эта команда запустит инстанс redis-cli и подключится к серверу по alias:

```
docker run -it --link redis-server:redis redis redis-cli -h redis
```

```
Для проверки работы можно вывести все данные из redis командой:
```

KEYS *

# Создание сетей между контейнерами с использованием NETWORK

```
Как создать сеть docker, которая позволяет контейнерам взаимодействовать по сети. Так
У Docker есть два подхода к работе в сети. Первый — это рассмотренный в прошлой лекци
Альтернативный подход заключается в создании Docker-сети, к которой подключены контей
```

## Создание сети

```
Первым шагом будет создание сети с использованием CLI. Эта сеть позволит нам прикрепит
Создадим сеть с желаемым именем:
```

```
docker network create backend-network
```

```
Теперь при запуске контейнера можно использовать ключ —net, чтобы определить подключе
```

```
docker run -d --name=redis --net=backend-network redis
```

## Сетевое взаимодействие

```
В отличие от использования link, docker-сеть ведет себя как традиционные сети, где уз
Первое отличие, это то, что Docker больше не назначает переменные среды и не обновляет
```

```
docker run --net=backend-network alpine env
docker run --net=backend-network alpine cat /etc/hosts
```

```
Вместо этого контейнеры могут связываться через встроенный DNS-сервер. Этот DNS-серве
```

```
docker run --net=backend-network alpine cat /etc/resolv.conf
```

```
Когда контейнеры пытаются получить доступ к другим контейнерам через имя, такое как R
```

```
docker run --net=backend-network alpine ping -c1 redis
```

## Подключение двух контейнеров

```
Docker поддерживает одновременное подключение сетей и контейнеров к нескольким сетям.
Например, давайте создадим отдельную сеть с приложением Node.js, которое связывается
```

```
docker network create frontend-network
```

```
Используем команду подключения существующего контейнера к сети:
```

```
docker network connect frontend-network redis
```

```
Если запустить тестовое приложение, то оно сможет обмениваться данными с контейнером
```

```
docker run -d -p 3000:3000 --net=frontend-network katacoda/redis-node-docker-example

curl docker:3000
```

### Aliases при использовании сети

```
Link по-прежнему поддерживаются при использовании Docker Network и предоставляют спосо
Другой подход заключается в предоставлении псевдонима при подключении контейнера к се
Следующая команда подключит экземпляр Redis к внешней сети с псевдонимом db:
```

```
docker network create frontend-network2
docker network connect --alias db frontend-network2 redis
```

```
Когда контейнеры пытаются получить доступ к сервису через имя db, ими будет получен I
```

### Отключение от сети

Создав сети, можно использовать CLI для изучения деталей. Выведем список сетей.

```
docker network ls
```

Так же можно посмотреть какие контейнеры подключены и их IP.

```
docker network inspect frontend-network
```

А теперь отключим контейнер redis от сети frontend-network:

```
docker network disconnect frontend-network redis
```

## Сохранение данных с использованием томов (Volumes)

Docker Volumes позволяет обновлять контейнеры, перезагружать машины и обмениваться данными без потери данных. Это важно при обновлении базы данных или версий приложения.

Тома Docker создаются и назначаются при запуске контейнеров. Тома позволяют сопоставить каталог хоста с контейнером для обмена данными.

Тома являются двунаправленными. Это позволяет получать доступ к данным, хранящимся на хосте, из контейнера. Это также означает, что данные, сохраненные процессом внутри контейнера, сохраняются и на хосте.

## Подключение тома

В этом примере Redis будет использоваться для сохранения данных. Запустим контейнер Redis и создадим том данных с помощью параметра -v. Это указывает, что любые данные, сохраненные в контейнере в каталоге /data, должны сохраняться на хосте в каталоге /docker/redis-data.

```
docker run -v /docker/redis-data:/data --name r1 -d redis \redis-server --appendonly
```

Запишем данные в redis командой:

```
cat data | docker exec -i r1 redis-cli --pipe
```

Если команда выполнена успешна, то в папке /docker/redis-data будет создан aof файл.

```
/docker/redis-data
```

Так же путь на хосте можно использовать для других контейнеров. Например, сделаем контейнер с ubuntu, который будет заниматься резервным копированием.

```
docker run -v /docker/redis-data:/backup ubuntu ls /backup
```

### Shared Volumes

Тома, сопоставленные с хостом, отлично подходят для сохранения данных. Однако, чтобы получить доступ к ним из другого контейнера, вам нужно знать точный путь, что может привести к опечаткам и другим ошибкам.

Альтернативный подход заключается в использовании —volumes-from. Параметр отображает тома из исходного контейнера в запускаемый контейнер.

Теперь сопоставляем том нашего контейнера Redis с контейнером Ubuntu. Каталог /data существует только в контейнере Redis, однако с помощью -volumes-from контейнер Ubuntu может получить доступ к данным.

```
docker run --volumes-from r1 -it ubuntu ls /data
```

### Read-only тома

Монтирование томов дает контейнеру полный доступ на чтение и запись к каталогу. Можно указать разрешения только для чтения каталога, добавив ro к монтированию.

ВАЖНО: если контейнер попытается изменить данные в каталоге, произойдет ошибка: Read-only file system.

```
docker run -v /docker/redis-data:/data:ro -it
```

## Лог файлы.

Когда запускается контейнер, Docker будет отслеживать потоки out и error процесса и будет делать их доступными через клиент.

Для вывода логов используется команда

```
docker logs <имя контейнера>
```

### Syslog

По-умолчанию журналы Docker выводятся с использованием json-file logger, т.е. вывод хранится в файле JSON на хосте. Это может привести к переполнению диска файлами логов. Можно изменить драйвер логов для перемещения логов в другое место.

Драйвер журнала syslog запишет все журналы контейнеров в центральный жулнал на хосте docker. Этот лог драйвер хорош, когда syslog собирается и агрегируется внешним ПО.

Команда ниже переведет логи контейнера в syslog.

```
docker run -d --name redis-syslog --log-driver=syslog redis
```

Если попытаться получить доступ к журналам через клиент — будет ошибка FATA[0000] . docker logs поддерживается только драйвером json-file.

### Отключение логов

Для отключения логов достаточно прописать в —log-driver параметр none:

```
docker run -d --name redis-none --log-driver=none redis
```

Как узнать текущий конфиг?

Команда inspect позволяет определить конфигурацию ведения журнала для контейнера, в формате можно указать раздел LogConfig. Пример:

```
docker inspect --format '{{ .HostConfig.LogConfig }}' redis-server
```

## Политики перезапуска

Как и любой процесс, контейнеры могут зависать. В этой лекции узнаем как сохранить контейнеры в рабочем состоянии, и автоматически перезапустить их в случае неожиданного сбоя.

### Stop On Fail

Docker считает любые контейнеры в exit кодом не равным 0 как crashed. По-умолчанию crashed контейнеры остаются остановленными. Для перезапуска используется:

```
docker run -d --name restart-default
```

Не забываем для диагностики смотреть ps и логи:

```
docker ps -a
docker logs restart-default
```

### Restart On Fail

В зависимости от вашего сценария перезапуск сбойного процесса может исправить проблему. Docker может автоматически повторить попытку запуска контейнера определенное количество раз, прежде чем он прекратит попытки.

Опция —restart=on-failure:# позволяет указать, сколько раз Docker должен повторить попытку. В приведенном ниже примере Docker перезапустит

контейнер три раза перед остановкой:

```
docker run -d --name restart-3 --restart=on-failure:3 scrapbook/docker-restart-exampl
```

### Always Restart

Наконец, Docker всегда может перезапустить сбойный контейнер, в этом случае Docker будет продолжать попытки, пока контейнер явно не получит команду остановиться.

```
docker run -d --name restart-always --restart=always scrapbook/docker-restart-example
```

## Метаданные

Когда контейнеры запущены в продакшене важно добавлять дополнительные метаданные, которые помогут в управлении ими. Например: версия запущенного в контейнера кода, список приложений или права пользователей.

Метаданные управляются Docker Labels, которые позволяют установить пользовательские метаданные для контейнера или образа, которые могут быть потом отфильтрованы или просто прочитаны.

### Контейнеры

Label может быть прикреплен к контейнеру когда он запускается командой docker run. Контейнер может иметь несколько labels.
Для добавления label используется команда -l. В примере ниже устанавливается label с именем user с значением.

```
docker run -l user=12345 -d redis
```

Если вы добавляете несколько label, они могут быть получены из внешнего файла. Файл должен иметь label в каждой строке, а затем они будут прикреплены к работающему контейнеру.

Для использования файла используется команда —label-file=<имя_файла>.

### Создадим файл labels с данными:

```
echo 'user=123461' >> labels && echo 'role=cache' >> labels
```

И прикрепим:

```
docker run --label-file=labels -d redis
```

## Images

С образами всё работает похожим способом, только определение label происходит в Dockerfile.
Для добавления label с именем vendor нужно добавить строчку ниже в Dockerfile:

```
LABEL vendor=Katacoda
```

Если необходимо добавить несколько значений, то запись нужно добавлять такого вида:

```
LABEL vendor=Katacoda \ com.katacoda.version=0.0.5 \ com.katacoda.build-date=2016-07-
```

## Просмотр

Для просмотра добавленной информации используется docker inspect. Для примера был создан контейнер rd и образ katakoda-label-example. Чтобы получить label контейнера rd выполним:

```
docker inspect rd
```

Однако это не удобно — выдается вся информация о контейнере, а нам нужна только маленькая часть информации. Для этого есть параметр -f, который позволяет фильтровать JSON ответ. В примере ниже выберем только label.

```
docker inspect -f "{{json .Config.Labels }}" rd
```

С просмотром информации образов отличий почти нет, только вместо .Config.Labels используется .ContainerConfig.Labels:

```
docker inspect -f "{{json .ContainerConfig.Labels }}" katacoda-label-example
```

```
```

```
Если так получилось, что у образа нет имени — можно использовать вместо имени <none>.
```

### Фильтрация

Когда речь идет не о паре контейнеров, а о сотнях, то такой вариант получения информации не очень удобен. Для этого есть фильтрация контейнеров ключом — filter для docker ps.

```
docker ps --filter "label=user=scrapbook"
```

А для образов:

```
docker images --filter "label=vendor=Katacoda"
```

Важно помнить что поиск чувствителен к регистру текста. Поэтому лучше не использовать большие буквы.

## Label для Docker демона\службы

Данная команда может понадобиться для, например, пометки продакшн или тестового Docker.
В дальнейших уроках будет подробнее показаны примеры конфигурации самого Docker, но пока только про label:

```
docker -d \
-H unix:///var/run/docker.sock \
--label com.katacoda.environment="production" \
--label com.katacoda.storage="ssd"
```

## Контейтеры балансировки нагрузки

В этой лекции рассмотрим, как можно использовать веб-сервер NGINX для балансировки запросов между двумя контейнерами, запущенными на хосте.

В Docker есть два основных способа взаимодействия контейнеров друг с другом. Первый — через links, которые конфигурируют контейнер с переменными среды и изменением host файла, позволяя им общаться. Второй — использование шаблонов Service Discovery, в котором используется информация, предоставленная третьими сторонами, в этом случае это будет API докера.

В шаблоне Service Discovery приложение использует стороннюю систему для определения местоположения целевой службы. Например, если наше

приложение хочет общаться с базой данных, оно сначала спросит API, каков IP-адрес базы данных. Этот шаблон позволяет вам быстро перенастроить и масштабировать вашу архитектуру с повышенной отказоустойчивостью, чем фиксированные пути.

Имя компьютера, на котором запущен Docker, называется Docker. Для доступа к службам используется имя docker вместо localhost или 0.0.0.0.

## NGINX Proxy

Мы хотим запустить службу NGINX, которая может динамически обнаруживать и обновлять свою конфигурацию баланса нагрузки при загрузке новых контейнеров. К счастью, это уже существует и называется nginx-proxy.

Nginx-proxy принимает HTTP-запросы и передает их в соответствующий контейнер на основе имени хоста запроса. Это прозрачно для пользователя и происходит без каких-либо дополнительных расходов ресурсов.

При запуске прокси-контейнера необходимо настроить три параметра. Первый — это привязка контейнера к порту 80 на хосте с использованием -p 80:80. Это гарантирует, что все HTTP-запросы обрабатываются прокси.

Второе — смонтировать файл docker.sock. Это соединение с демоном Docker, работающим на хосте, которое позволяет контейнерам получать доступ к его метаданным через API. Nginx-прокси использует это для прослушивания событий, а затем обновляет конфигурацию NGINX на основе IP-адреса контейнера. Монтирование файла работает так же, как и для каталогов, используя -v /var/run/docker.sock:/tmp/docker.sock:ro. Мы указываем: ro, чтобы ограничить доступ только для чтения.

Наконец, мы можем установить необязательный -e DEFAULTHOST = <домен>. Если запрос приходит и не указывает на определенные хосты, то выбирается указанный контейнер, в котором запрос будет обработан. Это позволяет вам запускать несколько веб-сайтов с разными доменами на одном компьютере с помощью отката на известный веб-сайт.

Попробуем на примере:

```
docker run -d -p 80:80 -e DEFAULT_HOST=proxy.example -v /var/run/docker.sock:/tmp/doc
```

Поскольку мы используем DEFAULT_HOST, любые поступающие запросы будут направлены в контейнер, которому назначен HOST proxy.example.

Вы можете сделать запрос к веб-серверу, используя curl http://docker. Поскольку у нас нет контейнеров, он вернет ошибку 503.

## Один хост

Nginx-прокси теперь прослушивает события, которые Docker вызывает при запуске / остановке. Был создан пример веб-сайта katacoda/docker-http-server, который возвращает имя контейнера, на котором он работает. Это позволяет

нам проверить, что наш прокси работает должным образом. Внутренне это приложение PHP и Apache2, прослушивающее порт 80.

Чтобы Nginx-прокси начал отправлять запросы в контейнер, нужно указать переменную среды VIRTUAL_HOST. Эта переменная определяет домен, откуда будут поступать запросы, и должен обрабатываться контейнером.

В этом примере установим наш HOST равным DEFAULT_HOST, чтобы он принимал все запросы.

```
docker run -d -p 80 -e VIRTUAL_HOST=proxy.example katacoda/docker-http-server
```

Если выполнить запрос к нашему прокси с помощью curl http://docker, тогда запрос будет обработан контейнером с сайтом.

## Кластер

Теперь мы успешно создали контейнер для обработки наших HTTP-запросов. Если мы запустим второй контейнер с тем же VIRTUAL_HOST, то nginx-proxy настроит систему в циклическом сценарии с балансировкой нагрузки. Это означает, что первый запрос будет отправлен в один контейнер, второй запрос — во второй контейнер, а затем будет повторяться по кругу. Нет ограничений на количество работающих узлов.

```
docker run -d -p 80 -e VIRTUAL_HOST=proxy.example katacoda/docker-http-server
```

Если мы выполним запрос к нашему прокси с помощью curl http://docker, тогда запрос будет обработан нашим первым контейнером. Второй HTTP-запрос вернет другое имя компьютера, что означает, что он был обработан нашим вторым контейнером.

## Итоговая конфигурация NGINX

Хотя nginx-proxy автоматически создает и настраивает NGINX для нас, если вам интересно, как выглядит окончательная конфигурация, вы можете вывести полный файл конфигурации с помощью docker exec, как показано ниже.

```
docker exec nginx cat /etc/nginx/conf.d/default.conf
```

## Оркестровка с использованием Docker Compose

При работе с несколькими контейнерами может быть сложно управлять запуском, а также настройкой переменных и ссылок. Чтобы решить эту проблему,

в Docker имеется инструмент под названием Docker Compose для управления оркестровкой, запуском контейнеров.

Посетите https://docs.docker.com/compose/install/ для получения инструкций о том, как установить docker compose в вашей локальной среде.

## Определение первого контейнера

Docker Compose основан на файле docker-compose.yml. Этот файл определяет все контейнеры и параметры, необходимые для запуска набора кластеров. Свойства отображаются в зависимости от того, как вы используете команды docker run, теперь хранятся в системе контроля версий и совместно используются с вашим кодом.

Формат файла основан на YAML (Yet Another Markup Language).

```
container_name:
  property: value
    - or options
```

В этом сценарии у нас есть приложение Node.js, которое требует подключения к Redis. Для начала нам нужно определить наш файл docker-compose.yml для запуска приложения Node.js.

Учитывая приведенный выше формат, в файле необходимо присвоить имя контейнера «web» и установить для свойства сборки текущий каталог. Мы рассмотрим другие свойства в следующих шагах. Итого:

```
web:
  build: .
```

## Определение настроек

Docker Compose поддерживает все свойства, которые можно определить с помощью Docker Run.
Чтобы связать два контейнера вместе, укажите свойство links и перечислите необходимые соединения. Например, следующее будет ссылаться на контейнер источника redis, определенный в том же файле, и присвоить псевдониму то же имя. Так же пробросим порты.

```
web:
  build: .

  links:
    - redis

  ports:
```

```
    - "3000"
    - "8000"
```

Дополнительная информация тут — https://docs.docker.com/compose/compose-file/

## Определение второго контейнера

На предыдущем шаге мы использовали Dockerfile в текущем каталоге в качестве базы для нашего контейнера. На этом этапе мы хотим использовать существующий образ из Docker Hub в качестве второго контейнера.

Формат YAML достаточно гибок, чтобы определять несколько контейнеров в одном файле.

```
web:
  build: .

  links:
    - redis

  ports:
    - "3000"
    - "8000"
    - "3001"

redis:
  image: redis:alpine
  volumes:
    - /var/redis/data:/data
```

## Запуск

С созданным файлом docker-compose.yml вы можете запускать все приложения с помощью одной команды up. Если вы хотите вызвать один контейнер, вы можете использовать up <имя>.

Аргумент -d указывает, что контейнеры запускаются в фоновом режиме, аналогично тому, как это используется в Docker run.

```
docker-compose up -d
```

## Управление

Docker Compose не только может управлять начальными контейнерами, но также предоставляет способ управления всеми контейнерами с помощью одной команды.

Например, чтобы увидеть детали запущенных контейнеров, вы можете использовать docker-compose ps.

Для доступа ко всем журналам через один поток вы используете docker-compose logs.
Другие команды следуют той же схеме. Получите их, набрав docker-compose

## Docker Scale

Поскольку Docker Compose понимает, как запускать контейнеры вашего приложения, его также можно использовать для масштабирования количества работающих контейнеров.

Параметр масштаба позволяет указать службу, а затем количество экземпляров, которые вы хотите. Если это число превышает количество уже запущенных экземпляров, он запустит дополнительные контейнеры. Если число меньше, то остановит ненужные контейнеры. Увеличим количество web контейнеров.

```
docker-compose scale web=3
```

Так же ничего не мешает уменьшить их количество:

```
docker-compose scale web=3
```

## Остановка и удаление

Как и при запуске приложения, для остановки набора контейнеров вы можете использовать команду docker-compose stop.

Чтобы удалить все контейнеры, используйте команду docker-compose rm.
Docker, лекция 16. Метрики контейнера с помощью Docker Stats

При запуске контейнеров в продакшн важно следить за показателями выполнения: загрузка ЦП и памяти. Это нужно для того, чтобы убедиться, что они контейнеры ведут себя так, как ожидалось. Эти метрики также могут помочь диагностировать проблемы, если они возникают.

В этом сценарии мы рассмотрим встроенные метрики, предоставляемые Docker, которые могут предоставить дополнительную информацию по работающим контейнерам.

## Один контейнер

В среде этого урока есть контейнер, работающий под именем nginx. Вы можете найти статистику для контейнера с помощью команды:

```
docker stats nginx
```

Команда запускает окно терминала, которое самостоятельно обновляется данными из контейнера.
Для того, чтобы прервать работу команды нажмите CTRL+C.

### Несколько контейнеров

Встроенный Docker позволяет вам предоставлять несколько имен или идентификаторов и отображать их статистику в одном окне.

Сейчас в среде работают три подключенных контейнера. Для просмотра статистики по всем этим контейнерам вы можете использовать pipes и xargs. Pipe передает выходные данные одной команды на вход другой, в то время как xargs позволяет вам предоставлять входные данные в качестве аргументов команды. Пример:

```
docker ps -q | xargs docker stats
```

Данная команда выведет все контейнеры и выведет их статистику.

## Создание оптимизированных Docker Images с помощью Multi-Stage Builds

В этом уроке будет рассказано как использовать функциональность многоэтапной сборки для создания меньших, более оптимизированных образов.

Эта функция идеально подходит для развертывания таких языков, как Golang, в качестве контейнеров. Имея многоэтапные сборки, на первом этапе можно собрать двоичный файл Golang, используя в качестве основы более крупный образ Docker. На втором этапе вновь созданный двоичный файл может быть развернут с использованием намного меньшего базового образа. Конечный результат — оптимизированный образ Docker.

### Начнем с Dockerfile

Функция Multi-Stage позволяет одному Dockerfile содержать несколько этапов, чтобы получить желаемый оптимизированный Docker образ.

Первый этап будет содержать шаги для сборки двоичного файла с использованием контейнера разработки, второй будет оптимизирован и не будет включать инструменты разработки.

Удаляя средства разработки в рабочем образе, вы уменьшаете вероятность атаки и сокращаете время развертывания.

Начнем с развертывания HTTP-сервера Golang. В настоящее время используется двухэтапный подход Docker Build. В этом сценарии будет создан новый файл Docker, который позволяет создавать образ с помощью одной команды.

```
```

```
git clone https://github.com/katacoda/golang-http-server.git
```

И наш Dockerfile.multi:

```
## First Stage
FROM golang:1.6-alpine

RUN mkdir /app
ADD . /app/
WORKDIR /app
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o main .

## Second Stage
FROM alpine
EXPOSE 80
CMD ["/app"]

# Copy from first stage
COPY --from=0 /app/main /app
```

И запустим:

```
docker build -f Dockerfile.multi -t golang-app .
```

Результатом будут два образа. Один без тега, который использовался для первого этапа, а второй, меньший, — наш итоговый образ. Размер первого образа — 293МБ, размер второго — 12МБ.

Если появляется ошибка, COPY —from=0 /build/out /app/ Unknown flag, то это означает, что вы используете старую версию Docker без Multi-Stage поддержки и вам нужно обновится.

Теперь можно запустить приложение и проверить его доступность:

```
docker run -d -p 80:80 golang-app
curl localhost
```

## Форматирование PS вывода.

В этом уроке описано как использовать параметры —format для симпатичной печати вывода из Docker PS и Docker Inspect.

### Имена и образы в виде таблицы

Формат docker ps можно отформатировать так, чтобы отображалась только информация, которая нужна именно сейчас.

Стандартная команда docker ps выводит имя, используемый образ, команду, время работы и информацию о порте.

Чтобы ограничить отображаемые столбцы, используйте параметр — format. Параметр позволяет красиво печатать данные контейнеров с использованием синтаксиса шаблона Go.

Пример ниже выведет: <Имя> контейнер использует образ <имя>.

```
docker ps --format '{{.Names}} container is using {{.Image}} image'
```

Поскольку используются шаблоны Go, то это включает вспомогательные функции, такие как таблица.

```
docker ps --format 'table {{.Names}}\t{{.Image}}'
```

Однако параметр format позволяет поддерживать отображение данных, которые уже доступны с помощью команды docker ps. Если вы хотите включить дополнительную информацию, такую как IP-адрес контейнера, данные должны быть доступны через docker inspect.

К счастью, Docker Inspect также поддерживает красивую печать результатов через шаблон Go. Идентификаторы контейнеров из Docker PS могут быть переданы в Docker Inspect.

Параметр format может получить доступ ко всей информации о контейнере. Ниже приведен пример перечисления всех IP-адресов для работающих контейнеров.

```
docker ps -q | xargs docker inspect --format '{{ .Id }} - {{ .Name }} - {{ .NetworkSe
```

# Методы обеспечения безопасности при работе с Docker

Docker укорачивает циклы разработки программного обеспечения и его развертывания, давая возможность поставлять код быстрее, чем раньше. Однако в комплекте вы также получаете связанные с безопасностью проблемы, о которых следует знать.

Под катом описаны 5 распространенных сценариев, при которых развертывание Docker-образов ведет к появлению вопросов, связанных с безопасностью, а

также представлены замечательные инструменты и приведены рекомендации, которые могут оказаться полезными при их решении.

**1. Подлинность образов**

Начнем с вопроса, заложенного, пожалуй, в саму природу Docker: подлинность образов. Те, кто хотя бы некоторое время пользовался Docker, прекрасно знают, что контейнеры можно создавать на основе практически любых образов, независимо от того, загружены ли они из официального репозитория или получены от третьих лиц.

В результате мы оказываемся перед огромным выбором. Если вам не нравится контейнер, потому что он не удовлетворяет вашим требованиям, его можно заменить на другой. Но безопасно ли это?

Давайте рассмотрим вопрос с точки зрения программиста. При написании приложения можете ли вы доверять любому коду, независимо от того, кем он написан, даже если библиотека предлагается менеджером пакетов вашего языка программирования? Или все же стоит рассматривать не проанализированный вами код с определенной долей недоверия?

Думаю, что если безопасность вам хоть немного важна, перед интеграцией чужого кода в свое приложение нужно должным образом этот код проверить. Или я неправ?

Такая же доля скепсиса должна присутствовать и при работе с Docker-контейнерами. Если разработчик или организация вам незнакомы, можно ли быть уверенным, что выбранный контейнер не содержит скомпрометированных бинарных файлов или другого вредоносного содержимого? Думаю, вряд ли.

Учитывая вышеизложенное, я бы посоветовал сделать три вещи.

Используйте частные (private) и доверенные (trusted) репозитории

В частности, я рекомендую официальные репозитории на Docker Hub.

Там есть базовые образы для:

- операционных систем, таких как Ubuntu;
- языков программирования, таких как PHP и Ruby;
- серверов, таких как MySQL, PostgreSQL и Redis.

Помимо всего прочего, Docker Hub выделяется тем, что образы в нем всегда сканируются с помощью службы проверки безопасности (Security Scanning Service) Docker.

Для тех, кто ранее не слышал об этом сервисе, привожу выдержку из документации:
Docker Cloud и Docker Hub могут сканировать образы в частных репозиториях на предмет наличия уязвимостей, а также формировать отчет о результатах проверки для каждого тега образа.

При использовании официальных репозиториев вы можете быть уверены, что загруженные оттуда контейнеры безопасны и не содержат вредоносный код.

Эта функция доступна как на всех платных, так и на бесплатных тарифных планах, но во втором случае лишь в течение ограниченного периода времени. Если у вас платный тариф, обязательно воспользуйтесь этим сервисом.

Вы также можете создать частный (private) репозиторий для использования внутри организации.

Используйте Docker Content Trust

Еще одним полезным инструментом является Docker Content Trust. Эта функция была представлена сравнительно недавно — в Docker Engine 1.8. Она позволяет издателям заверять созданные ими Docker-образы.

Процитирую статью, посвященную выпуску соответствующего релиза, за авторством Diogo Mónica — ведущего специалиста по безопасности компании Docker:

Docker Engine перед загрузкой в удаленный репозиторий локально подписывает образ закрытым ключом издателя. Когда пользователь загружает этот образ, Docker Engine, используя публичный ключ издателя, удостоверяет, что будет запущен образ, идентичный созданному издателем, и в него не были внесены изменения третьими лицами.

В итоге этот сервис защищает от подделывания образов, атак повторного воспроизведения (replay attacks) и компрометации ключей. Я настоятельно рекомендую прочитать соответствующую статью и официальную документацию.

Docker Bench Security

Не так давно я начал использовать инструмент под названием Docker Bench Security, который:

Проводит проверку на основе десятков зарекомендовавших себя методов, относящихся к развертыванию Docker-контейнеров в production.

Этот инструмент основан на рекомендациях CIS Docker 1.13 Benchmark и проводит проверки в следующих областях:

1. конфигурация хоста;
2. конфигурация демона Docker;
3. конфигурационные файлы демона Docker;
4. образы и сборочные файлы контейнеров;
5. среда исполнения контейнеров;
6. безопасность Docker.

Для установки этого инструмента скопируйте репозиторий:

```
git clone https://github.com/docker/docker-bench-security.git
```

Затем сделайте cd docker-bench-secutity и выполните следующую команду:

```
docker run -it --net host --pid host --cap-add audit_control \
-e DOCKER_CONTENT_TRUST=$DOCKER_CONTENT_TRUST \
```

```
-v /var/lib:/var/lib \
-v /var/run/docker.sock:/var/run/docker.sock \
-v /etc:/etc --label docker_bench_security \
docker/docker-bench-security
```

В итоге будет создан и запущен контейнер, который, в свою очередь, начнет проверять хост и его контейнеры. На рисунке ниже представлен примерный вывод, генерируемый этим инструментом.

Посмотрев на рисунок, легко убедиться, что список проверок и их результаты представлены в удобной для понимания форме. В моем случае, как видите, требуются некоторые улучшения.

Особенно мне приглянулась возможность автоматизации. То есть проверки можно включить в процесс непрерывной интеграции и таким образом постоянно контролировать уровень безопасности контейнеров.

**2. Избыточные привилегии**

Эта проблема появилась вместе с первыми системами контроля доступа, но продолжает быть актуальной и по сей день. При этом неважно, устанавливаете ли вы, например, дистрибутив Linux на голое железо или в качестве гостевой операционной системы в виртуальной машине.

Работа с Docker-контейнерами кардинально не улучшает безопасность. К тому же Docker добавляет новый уровень сложности, и граница между гостевой системой и хостом теперь не так четко различима.

В плане Docker я обращаю особое внимание на 2 вещи:

- контейнеры, работающие в привилегированном режиме;
- избыточные привилегии для контейнеров.

  Начнем с первого пункта. Контейнер в Docker может быть запущен с ключом privileged, который дает ему дополнительные привилегии.

  Процитируем документацию:

> ```
> предоставляет контейнеру все разрешения и снимает все ограничения, наложенные cgro
> ```

```
Другими словами, контейнер может делать практически все то же, что и хост. Этот флаг

Если наличие этой возможности не заставило вас задуматься, я буду крайне удивлен или

Процитирую Armin Braun:
```

> *Если вы используете привилегированные контейнеры, относитесь к ним так же, как и к любым другим процессам, запущенным от имени суперпользователя.*

```
Но даже если вы не используете контейнеры с опцией --privileged, у некоторых из них мо
```

## Отключите ненужные привилегии и разрешения (capabilities)

```
Каков бы ни был ваш хостинг-провайдер, следует прислушаться к руководству по безопасно
```

> *Пользователям рекомендуется отключить все разрешения (capabilities), за исключением тех, которые точно необходимы их процессам.*
> *Ответьте на следующие вопросы:*

- Какое сетевое соединение нужно вашему приложению?
- Нужен ли ему доступ к raw-сокетам?
- Будет ли оно работать по UDP?

Если нет, отключите все эти разрешения. А нужно ли вашему приложению то, что по умолчанию запрещено? Если да, предоставьте необходимое.

Для управления разрешениями контейнеров используются опции –cap-drop и –cap-add.
Предположим, что вашему приложению не нужно изменять разрешения процесса и привязывать привилегированные порты, но необходимо выгружать и загружать модули ядра.

Соответствующие разрешения можно настроить следующим образом:

```
   docker run \
--cap-drop SETPCAP \
--cap-drop NET_BIND_SERVICE \
--cap-add SYS_MODULE \
-ti /bin/sh
```

```
Более подробную информацию об этих опциях вы можете найти в разделе документации «Runt
```

### 3. Безопасность системы

```
Итак, у вас есть удостоверенный издателем образ и уменьшенный набор привилегий контейн

Благодаря пространствам имен (namespaces) и контрольным группам (cgroups) Docker даже

Все они являются зрелыми и хорошо протестированными инструментами, которые в состоянии
```

AppArmor

> *AppArmor — программный инструмент упреждающей защиты, основанный на политиках безопасности (известных также как профили (англ. profiles)), которые определяют, к каким системным ресурсам и с какими привилегиями может получить доступ то или иное приложение. В AppArmor включен набор стандартных профилей, а также инструменты статического анализа и инструменты, основанные на обучении, позволяющие ускорить и упростить построение новых профилей. — Источник: Wikipedia.*

## SELinux

> *SELinux (англ. Security-Enhanced Linux — Linux с улучшенной безопасностью) — реализация системы принудительного контроля доступа, которая может работать параллельно с классической избирательной системой контроля доступа. — Источник: Wikipedia.*

## grsecurity

```
Grsecurity – это проект для Linux, который включает в себя некоторые связанные с безо

Типичной областью применения являются web-серверы и системы, которые принимают удален
```

### seccomp

> *seccomp (сокращение от secure computing mode) — это механизм обеспечения безопасности в ядре Linux. Он был помещен в основную ветку ядра Linux начиная с версии 2.6.12, которая была выпущена 8 марта 2005. Seccomp позволяет процессу перейти в «безопасное» состояние, из которого он не может выйти и в котором способен выполнить лишь ограниченный набор системных вызовов: exit(), sigreturn(), а также read() и write() к уже открытым файловым дескрипторам. При попытке выполнения других системных вызовов процесс будет завершен ядром с помощью SIGKILL. Таким образом, этот механизм не виртуализирует системные ресурсы, а практически полностью изолирует от них запущенный процесс. — Источник: Wikipedia*

```
Рабочие примеры и более глубокое освещение представленных технологий выходит за рамки
```

### 4. Ограничение потребления ресурсов

```
Каковы потребности вашего приложения? Может быть, ему нужно не более 50 МБ памяти? То

Если анализ, профилирование и эталонное тестирование являются частью вашего процесса к
```

```
В результате при развертывании контейнеров необходимо убедиться, что им выданы только
```

- m / — memory: устанавливает ограничения использования памяти;
  — memory-reservation: устанавливает «мягкое» ограничение использования памяти;
  — kernel-memory: ограничение использования памяти ядра;
  — cpus: ограничение использования процессора;
  — device-read-bps: ограничение скорости чтения данных с устройства.

Пример ниже демонстрирует использование некоторых из этих параметров в конфигурационном файле Docker compose, взятом из официальной документации:

```
version: '3'
services:
redis:
image: redis:alpine
deploy:
resources:
limits:
cpus: '0.001'
memory: 50M
reservations:
memory: 20M
```

Более подробную информацию можно получить, воспользовавшись интерактивной справкой Docker или обратившись к разделу документации “Runtime constraints on resources”.

### 5. Поверхность атаки

Последним из рассматриваемых в этой статье аспектов безопасности будет поверхность атаки, увеличение которой является следствием особенностей работы Docker. Это может случиться в любой IT-системе, однако проблема дополнительно усугубляется эфемерностью инфраструктуры, основанной на контейнерах.

Поскольку Docker позволяет создавать, разворачивать и удалять контейнеры очень быстро, бывает сложно уследить, какие точно приложения разворачиваются в вашей системе. Это, в свою очередь, ведет к потенциалу по увеличению поверхности атаки.

Не уверены по поводу статистики развертывания в организации? Задайте себе следующие вопросы:
Какие приложения развернуты в организации в данный момент?

- Кто их развернул?
- Когда они были развернуты?
- Почему они были развернуты?
- Как долго они будут развернуты?
- Кто за них отвечает?

- Когда в последний раз была проведена проверка их безопасности?

Надеюсь, что вы не слишком обеспокоились, обдумывая ответы на эти вопросы. В любом случае давайте рассмотрим действия, которые можно предпринять.

Внедрите журнал аудита и соответствующее логирование

Помимо журналов аудита (audit trail) приложений, учитывающих время создания и активации учетной записи пользователя, последнего обновления пароля и т. д., рассмотрите внедрение журнала аудита контейнеров, создаваемых и разворачиваемых в организации.

Эта система не обязана быть очень сложной, но она должна фиксировать следующие события:

- когда приложение было развернуто;
- кто его развернул;
- почему оно было развернуто;
- каково его назначение;
- когда оно устареет и должно быть выведено из строя.

Большинство инструментов непрерывной разработки должны поддерживать регистрацию этой информации, что может быть реализовано напрямую в инструменте либо с помощью пользовательских скриптов, написанных на языке программирования по вашему выбору.

В дополнение рассмотрите создание оповещений, которые должны отправляться на email или в IRC, Slack, HipChat и т. п. Благодаря этому дополнительному уровню безопасности сотрудники получают информацию о выполненных развертываниях, что делает процесс более прозрачным. Теперь то, что не должно было произойти, уже не сможет пройти незамеченным.

Я не говорю, что вы не должны доверять сотрудникам и коллегам, но все же лучше быть в курсе происходящего.
Прошу понять меня правильно. Здесь важно не переусердствовать и не погрязнуть в создании множества новых процессов контроля. Этим можно добиться лишь потери времени на ненужную работу и утраты всех тех преимуществ, которые дают контейнеры.

При этом по крайней мере обдумывание поставленных вопросов, регулярная переоценка системы на их основе, а также способность принимать решения с их учетом должны помочь в уменьшении вероятности создания неизвестных поверхностей атаки в ваших системах.

# Заключение

Итак, мы рассмотрели 5 вопросов, связанных с безопасностью Docker, а также поговорили о возможных решениях. Надеюсь, что если вы переходите на Docker, раздумываете о таком шаге или уже это сделали, то предложенный материал поможет в усилении безопасности вашей контейнерной инфраструктуры.

Docker — это замечательная технология. Мне бы хотелось, чтобы она появились намного раньше. Теперь Docker с нами, и я надеюсь, что изложенная

информация поможет вам построить работу с этим инструментом таким образом, чтобы он функционировал исключительно в ваших интересах, не подкидывая неприятные сюрпризы в плане безопасности.