Санкт–Петербургский государственный университет

НИКОЛАЕВА Ирина Николаевна

Выпускная квалификационная работа

Программа для создания минимального размера образов корневых файловых систем для контейнеров приложений

Уровень образования: бакалавриат
Направление 02.03.02 «Фундаментальная информатика и информационные технологии»
Основная образовательная программа СВ.5003.2017 «Программирование и информационные технологии»
Профиль «Автоматизация научных исследований»

Научный руководитель:
доцент, кафедры компьютерного моделировани
и многопроцессорных систем, к.ф. - м.н.
Ганкевич Иван Геннадьевич

Рецензент:
программист, Центр Технологий Распределенных Реестров СПбГУ
Петрунин Вадим Николаевич

Санкт-Петербург
2021 г.

Содержание

Перечень сокращений, условных обозначений, символов, единиц и терминов

Виртуализация — сокрытие настоящей реализации какого-либо процесса или объекта от истинного его представления для того, кто им пользуется.
Docker — технология, реализующая виртуализацию приложений при помощи контейнеров.
Контейнер — подход к виртуализации.
Образ контейнера — описание корневой файловой системы, на основе которой собирается контейнер.
IoT (от англ. Internet of Things) – Интернет Вещей, технологии разработки автоматизированных систем, части которых (встраиваемые системы) взаимодействуют друг с другом через интернет или другие технологии беспроводной связи.
ОС — операционная система.
Хостовая ОС — ОС, установленная на оборудование.
Клиент-серверная архитектура – подход к реализации сложных приложений, когда одна часть, сервер, занимается непосредственными вычислениями и ожидает команд на выполнение от другой части, клиента.
REST — Representational State Transfer; архитектурный стиль взаимодействия компонентов распределенного приложения.
API — Application Programming Interface; описание способов взаимодействия с программным компонентом приложения.
ЯП — язык программирования.
P2P — архитектура компьютерной сети, в которой все участники равноправны.
ПО — программное обеспечение.
Дедубликация — подход, исключающий дубликацию (повторение) данных.
СУБД — система управления базой данных.
Raspberry Pi — микрокомпьютер, используемый в IoT.

Введение

Контейнеры в настоящее время являются одной из наиболее удобных технологий виртуализации и используются как для разработки приложений, так и для их тестирования, так как позволяют изолировать части приложений друг от друга.

Обособленный запуск приложения или его частей достигается созданием для них отдельных окружений. У каждого запущенного в контейнере приложения свои версии корневой файловой системы, списка процессов и сетевых устройств.

Контейнеры имеют преимущества перед другими способами виртуализации, такие как:

- легкость и быстрота развертки,
- работа на основе хостовой ОС,
- легкость в управлении,
- безопасный запуск приложения,
- отсутствие накладных расходов других технологий виртуализации,
- небольшие объемы.

Чаще всего, приложение, построенное с применением технологии контейнеров, работает на основе нескольких контейнеров разных технологий, взаимодействующих друг с другом. Образ одного контейнера может иметь размер больше гигабайта. Тогда весь размер пространства, занимаемого на диске, вместе с выделенной памятью на работающие контейнеры, может достигать нескольких десятков гигабайт, а время на передачу или получение образов - нескольких десятков минут. И если на рабочей машине существует ограничение по свободной памяти или существенна оптимизация нагрузки сети, например, в IoT, появляется необходимость уменьшения размера образов.

Постановка задачи

Образы корневых файловых систем для контейнеров создаются, как правило, менеджерами пакетов таким образом, чтобы человек мог их использовать интерактивно. Из этого следует, что образы контейнеров могут включать в себя программы, которые не запускаются приложением внутри контейнера и файлы, которые никогда не читаются и не пишутся.

Так как внутри контейнера работают заранее известные приложения, количество которых ограниченно, можно обнаружить те файлы и программы, которые необходимы для корректной работы приложения в изолированной среде, и исключить из образа остальные.

Задача заключается в том, чтобы уменьшить размер образов корневых файловых систем, которые используются в контейнерах приложений. Идея данной работы заключается в том чтобы определить с помощью встроенных в Linux средств отладки, какие именно файлы используются конкретными приложениями и удалить все остальные файлы из корневой файловой системы контейнера. Это позволит создать образы, которые, в основном, состоят из

- исполняемых файлов приложений,
- библиотек, которые используются этими исполняемыми файлами,
- набора конфигурационных файлов, используемых этими исполняемыми файлами и библиотеками.

Эти образы будут иметь минимально возможный размер, что позволит эффективнее использовать пропускную способность сети и дисковое пространство, а значит, получить экономическую выгоду.

Для решения данной задачи существует приложение chainsaw. Необходимо провести опыты с образами контейнеров различных приложений, провести оценку функционала и качества работы приложения, а также определить, какие параметры влияют на размер нового образа.

Обзор литературы

Было найдено несколько других приложений, которые рассматривают похожие задачи, но используют совсем другие технологии для их решения.

При разворачивании в распределенной системе крупноразмерного приложения на основе Docker контейнеров и использовании локального хранилища образов, построенного на Docker Registry, было предложено решение, именуемое FID (Faster Image Distribution) [3]. Согласно результатам исследований, при использовании P2P сети можно достичь максимального уменьшения на 91,35% времени доставки образов размером 500 MB на 200 узлах, работающих параллельно. Однако, ввиду поставленных условий, FID не применим локально или в малых системах.

В следующем исследовании проанализировано около 47 TB сжатой информации о всех публичных образах из Docker Hub с точки зрения разных аспектов [4]. Исследователи пришли к выводу, что существует огромный потенциал к дедубликации особо больших локальных хранилищ образов. Это также относится к образам, построенным при помощи менеджеров пакетов и, тем самым, имеющим избыточные данные.

Следующее исследование на практике доказало предыдущее исследование и посвящено уменьшению времени передачи образов внутри небольшой распределенной системы и Docker Registry [5]. Образ контейнера строится на основе уровней, открытых только для чтения. Многие образы содержат внутри себя одни и те же уровни, и их переиспользование на конкретной конфигурации среды, позволило максимально в 7 раз уменьшить время обновления образа. Использовалось решающее дерево для того, чтобы определить, каким методом отправить образ: классическим или при помощи дедубликации. Притом, образы, допускающие дедубликацию, определялись с ошибкой 5,3% — 11,4%.

В Wharf, расширении Docker для распределенных файловых систем, также анализирующем уровни образа, получилось до 12 раз сократить время доставки образов [6]. Более того, данное решение оптимизирует работу Docker на распределенных файловых системах, например, реализует па-

раллельную работу с образами.

Однако, все данные решения разработаны под конкретные системы и не применимы локально, в малых системах или для небольших приложений. Именно поэтому приложение chainsaw, которое рассмотрено в данном исследовании, имеет свою актуальность.

Приложение chainsaw действует изнутри образа и с помощью системного вызова `ptrace` перехватывает системные вызовы, которые работают с файлами, из их аргументов извлекает пути к файлам и сохраняет их.

Подобным образом действует приложение CARE, которое сканирует файловую систему и ее состояние и переменные среды сохраняет в архив, с помощью которого можно добиться полного повторения условий, в которых была запущена программа [7]. В случае chainsaw используемые файлы не сохраняются в архив, а используются для того, чтобы вычислить неиспользуемые файлы и удалить их.

Глава 1. Основные понятия контейнеризации

1.1 Контейнеризация

Контейнер — это стандартизированная единица программного обеспечения, которая содержит в себе приложение и все его зависимости, чтобы запускать приложение быстро и независимо от остальных приложений [1].

Контейнер собирается из образа, который является неизменяемым шаблоном. Образы можно создавать, обновлять или использовать уже готовые. Хранятся образы в реестрах, которыми удобно пользоваться для передачи и получения образов. Большинство новых образов строятся на основе базовых, которые представляют многие популярные технологии.

Образы собираются при помощи специальных файлов, в которых прописываются команды, которые изменяют настройки контейнера, позволяющие приложению внутри него работать изолированно, такие как:

- базовые образы;
- дополнительные файлы;
- переменные среды;
- исполняемые внутри контейнера скрипты;
- настройка окружения внутри контейнера;
- точка запуска приложения.

Таким образом производится описание файловой системы, в которой будет работать выбранное приложение.

1.2 Устройство технологии Docker

Docker — мультиплатформенное решение. Его реализации есть для большинства ОС. Docker-платформа работает на основе хостовой ОС и занимается виртуализацией хостовых ресурсов, которые уже используют контейнеры (Рисунок 1).

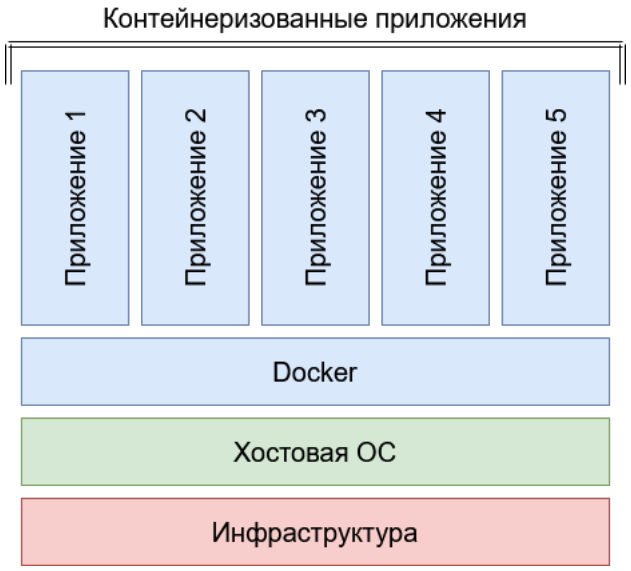

Рис. 1: Виртуализация при помощи контейнеров.

Платформа Docker работает на основе клиент-серверной архитектуры. Ядром платформы, сервером, является docker daemon, который непосредственно занимается разверткой, запуском и управлением контейнерами. Пользователь общается с сервером через клиента, docker Command-line interface (CLI), используя средства командной строки. Клиент и сервер взаимодействуют при помощи REST API. Общая схема платформы представлена на Рисунке 2.

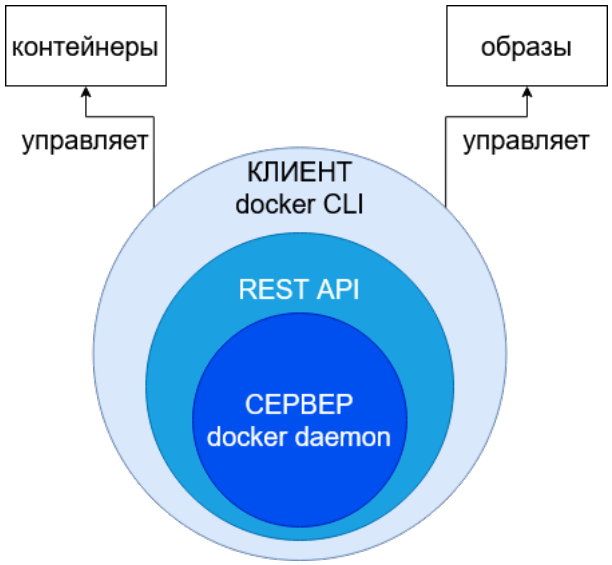

Рис. 2: Компоненты Docker платформы.

Таким образом, схема работы с Docker имеет следующий порядок:

1. Со стороны клиента посылается команда на сервер (`docker run`; `docker pull`; `docker build` ...);
2. Сервер получает эту команду и выполнит действие:
   - запускает контейнер, если локально есть необходимый образ,
   - подтягивает образ из реестра,
   - собирает новый образ,
   - или другую задачу, в зависимости от запроса клиента.

Порядок действий Docker можно рассмотреть на Рисунке 3.

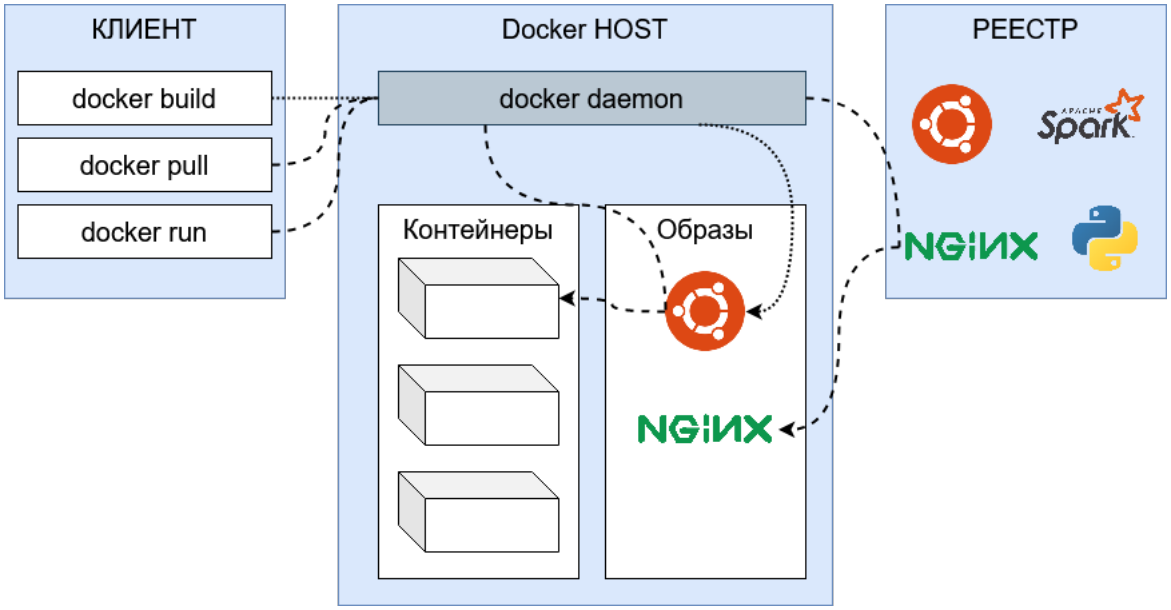

Рис. 3: Архитектура Docker.

Docker использует достигает своей эффективности путем использования некоторых функций ядра ОС Linux [2]. Например, пространство имен (namespaces) для достижения изолирования контейнера. Каждый контейнер запускается в своем собственном пространстве имен, что обеспечивает изоляцию и ограниченный доступ к этому пространству имен.

1.3 Основные понятия Dockerfile

Dockerfile — текстовый документ, который содержит все команды, необходимые для сборки образа в автоматическом режиме.

Docker-контейнеры построены на основе Linux-подобных корневых файловых систем. Поэтому любые команды, которые прописываются в Dockerfile, это консольные команды, допустимые к исполнению в используемом дистрибутиве базового образа. Чаще всего используют в качестве базовых образов два дистрибутива: debian и alpine linux. В зависимости от базового дистрибутива будут различаться команды, исполняемые при сборке образа. Далее будет заметно, как меняется Dockerfile в зависимости от базового дистрибутива рассматриваемого образа.

При сборке образа демон Docker разбирает Dockerfile при помощи специальных директив [8].

- `FROM` — определяет, какой базовый образ будет использован.
- `USER` — дальнейшие команды будут исполняться под указанным пользователем.
- `COPY` — копирует файлы или директории из хостовой ОС или из другого образа.
- `WORDIR` — устанавливает рабочую директорию.
- `ENV` — устанавливает переменные среды.
- `RUN` — запускает консольные инструкции.
- `EXPOSE` — указывает контейнеру, какой порт следует слушать во время работы.
- `VOLUME` — устанавливает путь для директории, которая будет связана с внешним хранилищем, который будет использовать контейнер.
- `STOPSIGNAL` — устанавливается системный сигнал, который будет послан контейнеру для его закрытия.
- `ENTRYPOINT` — устанавливает точку доступа (в виде некоторой инструкции) в контейнер.
- `CMD` — устанавливает запуск контейнера по умолчанию.

Помимо готовых образов в качестве базового можно использовать пустой образ scratch. Весь функционал данного образа будет полностью зависеть

от того, какие файлы будут переданы в него. Этот подход можно использовать, если знать все зависимости, включая системные, которые необходимы для исполнения приложения.

Глава 2. Приложение chainsaw

Для исследовательской работы была выбрана программа chainsaw, написанная на C для оптимизирования размера корневой файловой системы, используемой приложением в контейнере.

2.1 Устройство chainsaw

Программа chainsaw использует встроенные в ОС Linux средства отладки, а именно системный вызов ptrace, позволяющий одному процессу управлять другим.

Программа состоит из четырех частей [9]:

- `chainsaw-blacklist`. Выводит список всех файлов (в формате абсолютного пути) в корне файловой системы в файл blacklist. При этом исключаются домашние директории пользователей и виртуальные файловые системы (`/dev`, `/sys`, `/proc` и другие).
- `chainsaw-whitelist`. Запускает указанную в аргументах программу и с помощью ptrace перехватывает системные вызовы, которые работают с файлами. Из аргументов этих системных вызовов она извлекает пути к файлам и сохраняет их в файл whitelist. Иными словами, в whitelist попадают пути к файлам, которые используются приложением.
- `chainsaw-diff`. Вычисляет разницу между blacklist и whitelist и выводит ее в файл diff. Следовательно, в diff содержатся пути к файлам, которые не используются приложением.
- `chainsaw-cut`. Удаляет все файлы, указанные в diff.

Программа лежит в виде исходного кода в публичном репозитории [9], и для его сборки необходимы несколько зависимостей:

- git для того, чтобы напрямую из репозитория подгрузить исходные файлы,
- python версии 3.5 или выше и pip,
- библиотеки ninja и meson, которые собирают исполняемые файлы.

2.2 Использование chainsaw

Таким образом, использование приложения chainsaw сводится к следующим шагам:

1. Получение исходного образа из хаба или в виде Dockerfile.
2. Написание Dockerfile для нового образа:
   (a) установка зависимостей,
   (b) сборка исполняемых файлов
   (c) запуск компонент приложения chainsaw,
   (d) создание пустого образа scratch,
   (e) копирование корневой файловой системы из модифицированного образа в пустой,
   (f) применение дополнительных настроек образа.
3. Сборка образа.

На псевдокоде Dockerfile будет выглядеть следующим образом:

```
FROM image as build

RUN  install git, python3.5 && \
     pip3 install ninja meson && \
     git clone https://github.com/LazyDareDevil/chainsaw && \
     cd chainsaw && \
     meson build && \
     ninja -C build && \
     ninja -C build install && \
     chainsaw-blacklist / && \
     chainsaw-whitelist some-app && \
     chainsaw-diff && \
     chainsaw-cut && \

FROM scratch
```

```
COPY --from=build / /

ENTRYPOINT ["some-app"]
```

2.3 Постановка задачи в рассмотренных условиях

Таким образом поставленная задача сводится к тому, что необходимо на примере разных образов контейнеров оценить следующие параметры:

- размеры исходного и полученного образов;
- параметры, влияющие на размер полученного образа;
- универсальность относительно разных образов;
- сложность написания Dockerfile.

А также сделать выводы по результатам экспериментов о практической применимости приложения chainsaw.

Глава 3. Проведенные эксперименты

Для исследования работы приложения chainsaw было выбрано пять образов контейнеров, содержащих приложения, написанные на разных ЯП и существенно различающиеся по сложности, размеру, способу запуска внутри контейнера.

Все написанные Dockerfile, а также полученные для каждого образа файлы blacklist, whitelist и diff, лежат в публичном репозитории [9].

3.1 Образ приложения nginx

nginx — это веб-сервер и почтовый прокси-сервер, работающий на Linux-подобных операционных системах, написанный на ЯП C.

Контейнер, собранный из оригинального образа, хостит простой html с небольшой сводкой о nginx.

При первых опытах приложении chainsaw внутри образа было обнаружено, что при работе `chainsaw-cut` появляется ошибка ptrace, обозначающая запрещенность производимых программой операций. Оказалось, что в в коде была небольшая ошибка, исправление которой разрешило данную ситуацию.

Далее было обнаружено, что запущенный на основе нового образа контейнер не запускается. `chainsaw-whitelist` не обнаружила такие системные библиотеки как

- `/lib/x86_64-linux-gnu/ld-2.28.so` ,
- `/lib64/ld-linux-x86-64.so.2` .

Их пришлось добавлять вручную в whitelist. Это происходит из-за того, что эти библиотеки используются ядром системы для динамической загрузки других библиотек, и `chainsaw-whitelist` не может их отловить.

Тоже самое было обнаружено и с самим приложением, находящимся в директории `/usr/sbin/nginx`, исходными файлами в директориях `/usr/share/nginx` и `/etc/nginx`, в также некоторыми файлами в директории `/var/run`. Их также было необходимо добавить в whitelist.

В Приложении А представлен полученный Dockerfile, а размер исходного и полученного образов представлены в Таблице 1.

Образ получился существенно меньше оригинального, и контейнер, собранный на его основе, работает корректно.

3.2 Образ приложения grafana

Для визуализации временных рядов используется веб-сервис grafana, реализованный на TypeScript.

Контейнер, запущенный на основе образа grafana, предоставляет полноценное UI-приложение с возможностью загружать и анализировать файлы, содержащие данные о временных рядах.

Первая отличительная особенность образа grafana — он получен на основе базового образа alpine linux, поэтому некоторые инструкции отличаются от тех, что были запущены в образе nginx.

Опытным путем было обнаружено, что для корректной работы python и библиотек ninja и meson было необходимо загрузить в образ такие библиотеки, как python3-dev, gcc, g++, libc-dev, cmake, make, linux-headers. Это говорит о том, что базовый образ alpine linux уже позволяет создавать достаточно маловесные образы, так как не содержит большого числа редко используемых библиотек.

Далее был изучен `ENTRYPOINT` исходного образа. Им оказался скрипт run.sh, в котором прописывались дополнительные настройки приложения и контейнера, а также сам запуск grafana. Поэтому, для анализа с помощью `chainsaw-whitelist` был выбран именно этот файл.

Далее, вручную в whitelist были добавлены пути к файлам из директорий, указанных в переменных среды, так как в них находятся необходимые для работы приложения настройки и файлы, и, на основе опыта, вынесенного из работы с образом nginx, такие файлы, как:

- `/usr/share/grafana/public` ,
- `/lib/ld-musl-x86_64.so.1` ,
- `/lib64/ld-linux-x86-64.so.2` .

В качестве `ENTRYPOINT` был уже не run.sh, а само приложение, к которому применены необходимые настройки.

В Приложении Б представлен полученный Dockerfile, а размер исходного и полученного образов представлены в Таблице 1.

Размер директории (`$GF_PATHS_HOME` = `/usr/share/grafana`), в которой лежит приложение и необходимые для его работы файлы, занимает в оригинальном контейнере 172,5 MB. Ввиду того, что размер образа в результате стал немногим больше этого значения, можно сделать вывод, что уменьшение размера образа прошло успешно.

Образ получился немногим меньше оригинального, и контейнер, собранный на его основе, работает корректно.

3.3 Образ приложения postgres

postgres — SQL система управления базой данных, написанная на ЯП C++ .

Контейнер, запущенный на основе данного образа, предоставляет полноценную СУБД.

`ENTRYPOINT` контейнера postgres, docker-entrypoint.sh, еще более сложный, чем у grafana, и без его запуска невозможно корректное исполнение СУБД внутри контейнера. Однако, некоторые инструкции внутри него исполняются скрыто (то есть инструкции прописаны неявно, а в виде строки, которая будет исполняться), `chainsaw-whitelist` не отлавливает их, пришлось вручную добавлять в whitelist многие инструкции (id, mkdir, find, chown, gosu, ls, hostname), системные библиотеки:

- `/lib/x86_64-linux-gnu/libreadline.so.7` ,
- `/lib/x86_64-linux-gnu/libreadline.so.7.0` ,

отдельно запускать `chainsaw-whitelist` на приложениях initdb, postgres и инструкции id.

Однако, для корректной работы приложения внутри контейнера нужны были еще дополнительные параметры, которые не переносятся из оригинального. Пришлось вручную перенести `VOLUME`, переменные среды `ENV`, `EXPOSE`, `STOPSIGNAL`, `CMD`, `ENTRYPOINT` в новый образ.

В Приложении В представлен полученный Dockerfile, а размер исходного и полученного образов представлены в Таблице 1.

Образ получился практически в три раза меньше оригинального, и контейнер, собранный на его основе, работает корректно.

3.4 Образ приложения tomcat

tomcat - сервер на Java для хостинга веб-приложений.

Приложению для работы нужны исходные файлы некоторого веб приложения, которые были загружены с официального сайта tomcat.

В целом, схема работы с образом tomcat похожа на то, что производилось с образом nginx. В whitelist попадают пути к файлам, необходимым для исполнения скрипта catalina.sh, а также инструкция bash, которая будет запускать приложение внутри контейнера, файлы в директории `/usr/local/tomcat`, необходимые для работы приложения, файл `/usr/bin/env`.

Однако, для того, чтобы веб приложение корректно запустилось в контейнере из оригинального образа, необходимо после его запуска скопировать файлы из директории `/usr/local/tomcat/webapps.dist` в директорию `/usr/local/tomcat/webapps`, иначе tomcat не видит исходные файлы для запуска веб приложения. Значит, внутри нового образа должна работать инструкция cp. Для этого в файл whitelist добавляются инструкция cp и системные библиотеки, необходимые для ее работы:

- `/lib/x86_64-linux-gnu/libselinux.so.1` ,
- `/usr/lib/x86_64-linux-gnu/libacl.so.1` ,
- `/usr/lib/x86_64-linux-gnu/libacl.so.1.1.2253` ,
- `/usr/lib/x86_64-linux-gnu/libattr.so.1` ,
- `/usr/lib/x86_64-linux-gnu/libattr.so.1.1.2448` ,
- `/lib/x86_64-linux-gnu/libpcre.so.3` ,
- `/lib/x86_64-linux-gnu/libpcre.so.3.13.3` .

Далее работа с новым образом идентична работе с образом postgres: добавление `WORKDIR`, `ENV`, `EXPOSE` и `CMD`.

Так как перед началом работы приложения необходимо скопировать файлы, в команде `CMD` мы указываем исполнение скрипта catalina.sh и инструкции cp.

В Приложении Г представлен полученный Dockerfile, а размер исходного и полученного образов представлены в Таблице 1.

Образ получился практически в три раза меньше оригинального, и контейнер, собранный на его основе, работает корректно.

3.5 Образ приложения peer-calls

peer-calls — приложение для видео конференц-связи, написанное на ЯП Go.

Способом, описанным ранее, с образом peer-calls поработать не удалось, и были открыты его исходные файлы [10]. Как оказалось, он строится на основе scratch.

В качестве эксперимента был собран тестовый образ приложения chainsaw на базе alpine linux [11]. В Приложении Д представлен полученный Dockerfile, а размер исходного и полученного образов представлены в Таблице 1.

Оказалось, что образ получился даже больше, чем исходный. Это связано с тем, чем Go собирает один исполняемый файл, который добавляется в scratch. Это гарантирует, что размер образа будет наименьший.

Размер полученного образа больше исходного, так как приложение в образе могли остаться директории и символьные ссылки, которые chainsaw не трогает.

3.6 Образ приложения Mycodo

Mycodo — приложение для Raspberry Pi, которое позволяет управлять входами и выходами на устройстве, а также отслеживать состояние IoT системы через веб интерфейс. Приложение написано на Python. Представлено в виде исходного кода в публичном репозитории [12].

Данное приложение собирается при помощи файла docker-compose.yml, который является описательный файлом для сборки приложений, состоящим из нескольких контейнеров. Два контейнера из образа, mycodo_flask и mycodo_daemon, запускаются на основе образа app, который и рассматривается в данном эксперименте.

Образ app является основой для двух python приложений, и на начальном этапе запускаются скрипты, которые устанавливают необходимые зависимости и настраивают окружение. Дальше начинает работать chainsaw.

`chainsaw-whitelist` запускается для трех python приложений, которые будут работать в контейнерах: flask, gunicorn и daemon. Также, опираясь на предыдущий опыт, в whitelist были добавлены пути к самим приложениям, файлы из директории, куда скопированы исходные файлы, а также файлы из директорий, которые в docker-compose.yml указаны как `VOLUME`, указана переменная среды, отвечающая за конфигурацию приложения внутри контейнера.

Однако, после запуска системы, контейнеры работали некорректно, возникли ошибки исполнения некоторых библиотек python, а также ошибка выбора пользователя, поэтому в whitelist добавлены файлы из директорий, в которых хранятся библиотеки python и файлы `/etc/passwd` и `/etc/group`.

Далее, опытным путем было обнаружено, что необходимые библиотеки из директории `/lib/x86_64-linux-gnu`, подгружаемые динамически, не были добавлены в whitelist, поэтому были указаны вручную.

В Приложении Е представлен полученный Dockerfile, а размер исходного и полученного образов представлены в Таблице 1.

Образ получился практически в три раза меньше оригинального, и контейнеры, собранные на его основе, работают корректно.

Глава 4. Полученные результаты

Результаты проведенных опытов приведены в Таблице 1.

Образ	**Образ из реестра**	**Полученный образ**
nginx	133,12 MB	8,64 MB
grafana	198,41 MB	181,18 MB
postgres	314,68 MB	104,57 MB
tomcat	667,44 MB	219,11 MB
peer-calls	22,06 MB	22,93 MB
Mycodo app	1,11 GB	366,99 MB

Таблица 1: Сравнение размеров образов

4.1 Анализ экспериментов.

Важным замечанием экспериментов является то, что функционал запущенного на основе нового образа ограничен запущенным им приложением. Как было сказано, большинство функций оболочки, и в некоторых случаях сама оболочка (`/bin/sh`, `/bin/bash`), допускающие интерактивную работу с контейнером, более не доступны.

Другое замечание: результаты минимальных образов для образов nginx, tomcat достигнуты путем использования небольших примеров. При использовании других пользовательских файлов результирующий образ может быть больше.

Образы grafana и peer-calls являются показательными для ситуаций, когда образ уже имеет минимально возможный размер. Образ grafana построен на основе alpine linux и, судя по результатам работы приложения chainsaw, наполнен только необходимыми для работы приложения внутри контейнера. Аналогичная ситуация и у образа peer-calls: внутри образа находится один файл, который является самим приложением. Поэтому этот образ уже имеет минимальный размер. Можно определить связь между этими двумя образами:

- оба используют минимально возможный базовый образ,
- оба написаны на ЯП высокого уровня (TypeScript, Go).

Именно эти признаки будут показателем того, что образ имеет минимальный или почти минимальный размер.

В случае с nginx получилось обратная ситуация. Так как образ построен на основе linux, но приложение и необходимые для его работы файлы занимают небольшой размер памяти, получилось существенно уменьшить размер образа. Например, если взять образ nginx на основе alpine linux, то его размер будет 22,6 MB, следовательно, такого значительного уменьшения образа не получилось бы. Следовательно, это подтверждает предположение о том, что использование более простых базовых образов позволяет добиться размера образа, наиболее приближенного к минимальному.

Однако использование базового образа alpine linux не является достаточным условием, так как с применением chainsaw, образ получился еще меньше, чем образ на alpine. Именно использование приложения, написанного на ЯП низкого уровня, дало возможность уменьшения образа.

Результаты работы с образами postgres и tomcat в целом схожи, как и работа с ними. Так как оба использовали в качестве базового образ linux и приложения, так как написаны на ЯП C++ и Java, представляют собой один исполняемый файл и несколько небольших директорий с пользовательскими файлами и настройками, в результате образ получился в три раза меньше исходного.

Исходный образ Mycodo app строится на основе образа python:3.7-slim-buster, размер которого 113 MB. Так как для работы приложений внутри app устанавливается большое количество зависимостей и приложений для настройки окружения, образ становится больше почти в 10 раз. Однако, многие из установленных приложений далее при работе контейнеров не используются, они удаляются из образа, и новый образ app уменьшается в три раза. Причем, 100 MB из полученных 370 MB — это исходные файлы, которые копируются в образ.

Данный образ является наглядным примером того, как важно следить за памятью внутри образа и удалять неиспользуемые компоненты. В исходном образе app оказалось только около 270 MB полезных файлов, связанных с корректной работой python интерпретатора, что больше, чем

исходный образ python:3.7-slim-buster, за счет установленных библиотек. Так как приложение Mycodo разработано для запуска на Raspberry Pi, у которого память сильно ограничена, использование уменьшенного образа позволит сэкономить 600 MB для хранения статистики IoT системы, которая управляется при помощи Raspberry Pi, что очень важно для систем мониторинга.

Таким образом можно вывести еще одну зависимость между ЯП, на котором написано приложение, и наименьшим размером образа, который можно достичь: чем более высокоуровневый язык используется в приложении внутри контейнера, тем меньшее количество файлов из исходного образа можно удалить, сохранив корректность работы приложения после запуска контейнера.

Выводы

Проведено исследование работы приложения chainsaw на примере пяти образов контейнеров. Были выведены два фактора, влияющих на итоговый размер образа после уменьшения:

- какой базовый образ использовался для сборки рассматриваемого образа,
- какой ЯП используется в работающем внутри контейнера приложении.

При работе с приложением chainsaw пришлось столкнуться с разными проблемами:

- `chainsaw-whitelist` не всегда отлавливает библиотеки, которые подгружаются динамически;
- необходимо вручную указывать пользовательские директории и некоторые директории приложения, которые также не были найдены;
- необходимо вручную переносить переменные среды, связывать директории с внешним хранилищем и указывать точки запуска контейнера;
- сложность отладки проблем, которые возникают при отсутствии в контейнере, собранном на основе уменьшенного образа, необходимых для работы приложения файлов;
- отсутствует универсальность разворачивания приложения chainsaw на разных базовых дистрибутивах.

Тем не менее, приложение chainsaw — полезное решение. На примере приложения Mycodo удалось показать, как можно сэкономить дисковое пространство в системах, имеющих ограниченную доступную память.

Заключение

В процессе данной работы на примере нескольких образов была исследована работа приложения chainsaw. Результатом данной работы стали Dockerfile, в которых наглядно представлены все особенности работы с данным приложением, и анализ факторов, влияющих на размер нового образа.

Приложение chainsaw действительно справляется с поставленной задачей и действительно будет полезно при разработке приложений на основе технологии Docker.

Несмотря на полученные положительные результаты, задача оставляет пространство для дальнейших исследований и экспериментов:

- реализация нескольких образов chainsaw для разных базовых дистрибутивов для повышения удобства использования приложения;
- исследовать обнаруженные в процессе работы проблемы и попытаться разработать пути их решения;
- разработать интерфейс, упрощающий отладку проблем, возникающих при создании нового образа.

Список литературы

[1] Docker Inc. What is a Container? [Электронный ресурс]: URL: `https://www.docker.com/resources/what-container` (дата обращения 23.05.21).

[2] Docker Inc. Docker overview [Электронный ресурс]: URL: `https://docs.docker.com/get-started/overview/` (дата обращения 23.05.21).

[3] Wang Kangjin, Yang Yong, Li Ying et al. FID: a faster image distribution system for Docker platform // IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W). Tucson, AZ. 2017. P. 191–198.

[4] Nannan Zhao, Tarasov V., Hadeel Albahar, Ali Anwar et al. Large-scale analysis of Docker images and performance implications for container storage systems // IEEE Transactions on Parallel and Distributed Systems Submit Manuscript. 2021. Vol. 32. No 4. P. 918–930.

[5] Zhigang Lu, Yuewen Wu, Jiwei Xu, Tao Wang. An acceleration method for Docker image update // IEEE International Conference on Fog Computing (ICFC). Prague, Czech Republic. 2019. P. 15–23.

[6] Chao Zheng, Rupprecht L., Tarasov V. et al. Wharf: Sharing Docker images in a distributed file system // SoCC '18: Proceedings of the ACM Symposium on Cloud Computing. New York, USA: Association for Computing Machinery. 2018. P. 174–185.

[7] CARE [Электронный ресурс]: URL: `https://proot-me.github.io/care/` (дата обращения: 23.05.21).

[8] Docker Inc. Dockerfile reference [Электронный ресурс]: URL: `https://docs.docker.com/engine/reference/builder/` (дата обращения 23.05.21).

[9] GitHub репозиторий приложения chainsaw [Электронный ресурс]: URL: `https://github.com/LazyDareDevil/chainsaw` (дата обращения: 23.05.21).

[10] GitHub репозиторий приложения peer-calls [Электронный ресурс]: URL: `https://github.com/peer-calls/peer-calls` (дата обращения: 23.05.21).

[11] Docker Hub репозиторий образа chainsaw [Электронный ресурс]: URL: `https://hub.docker.com/r/lazydaredevil/chainsaw` (дата обращения: 23.05.21).

[12] GitHub репозиторий приложения Mycodo [Электронный ресурс]: URL: `https://github.com/kizniche/Mycodo` (дата обращения: 23.05.21).

Приложение

Приложение A. Dockerfile для образа nginx

```
FROM nginx:latest as build

RUN apt-get update && \
    apt-get install git python3-pip -y && \
    yes | pip3 install ninja meson && \
    git clone https://github.com/igankevich/chainsaw && \
    cd chainsaw && \
    meson build && \
    ninja -C build && \
    ninja -C build install && \
    cd .. && \
    rm -rf chainsaw

RUN chainsaw-blacklist / && \
    (chainsaw-whitelist timeout 5s
                /usr/sbin/nginx -g 'daemon off;' || true) && \
    echo /usr/sbin/nginx >> whitelist && \
    find /usr/share/nginx -type f >> whitelist && \
    find /etc/nginx -type f >> whitelist && \
    echo /var/run >> whitelist && \
    echo /lib/x86_64-linux-gnu/ld-2.28.so >> whitelist && \
    echo /lib64/ld-linux-x86-64.so.2 >> whitelist && \
    chainsaw-diff && \
    chainsaw-cut --confirm diff

FROM scratch
COPY --from=build / /

ENTRYPOINT ["/usr/sbin/nginx", "-g", "daemon off;"]
```

Приложение Б. Dockerfile для образа grafana

```
FROM grafana/grafana as build

USER root

RUN cd / && \
    apk update && \
    apk add --no-cache git py3-pip && \
    apk add python3-dev gcc g++ libc-dev
                cmake make linux-headers && \
    pip3 install ninja meson && \
    git clone  https://github.com/igankevich/chainsaw && \
    cd chainsaw && \
    meson build && \
    ninja -C build && \
    ninja -C build install && \
    cd / && \
    cp run.sh /usr/share/grafana

RUN cd / && \
    chainsaw-blacklist / && \
    (chainsaw-whitelist timeout 5s /run.sh || true) && \
    find $GF_PATHS_HOME -type f >> whitelist && \
    echo $GF_PATHS_CONFIG >> whitelist && \
    find $GF_PATHS_DATA -type f >> whitelist && \
    find $GF_PATHS_LOGS -type f >> whitelist && \
    find $GF_PATHS_PLUGINS -type f >> whitelist && \
    find $GF_PATHS_PROVISIONING -type f >> whitelist && \
    echo /usr/share/grafana/public >> whitelist && \
    echo /lib/ld-musl-x86_64.so.1 >> whitelist && \
    echo /lib64/ld-linux-x86-64.so.2 >> whitelist && \
    chainsaw-diff && \
```

```
    chainsaw-cut --confirm diff

FROM scratch
COPY --from=build / /

ENTRYPOINT [ "/usr/share/grafana/bin/grafana-server",
          "--homepath=/usr/share/grafana",
          "--config=/etc/grafana/grafana.ini",
          "--packaging=docker", "$@",
          "cfg:default.log.mode=console",
          "cfg:default.paths.data=/var/lib/grafana",
          "cfg:default.paths.logs=/var/log/grafana",
          "cfg:default.paths.plugins=/var/lib/grafana/plugins",
          "cfg:default.paths.provisioning=
                        /etc/grafana/provisioning" ]
```

Приложение B. Dockerfile для образа postgres

```
FROM postgres as build

RUN apt-get update && \
    apt-get install git python3-pip -y && \
    yes | pip3 install ninja meson && \
    git clone https://github.com/igankevich/chainsaw && \
    cd chainsaw && \
    meson build && \
    ninja -C build && \
    ninja -C build install && \
    cd /

RUN chainsaw-blacklist / && \
    (chainsaw-whitelist timeout 10s id || true) && \
    mv whitelist whitelist1 && \
```

```
(chainsaw-whitelist timeout 10s /bin/bash
            docker-entrypoint.sh || true) && \
mv whitelist whitelist2 && \
(chainsaw-whitelist timeout 10s su -c
            initdb postgres || true) && \
mv whitelist whitelist3 && \
(chainsaw-whitelist timeout 10s postgres || true) && \
mv whitelist whitelist4 && \
cat whitelist1 whitelist2 whitelist3
            whitelist4 >> whitelist && \
echo /bin/mkdir >> whitelist && \
echo /bin/chmod >> whitelist && \
echo /usr/bin/find >> whitelist && \
echo /bin/chown >> whitelist && \
echo /usr/sbin/gosu >> whitelist && \
echo /bin/ls >> whitelist && \
echo /docker-entrypoint.sh >> whitelist && \
echo /docker-entrypoint-initdb.d >> whitelist && \
find /usr/lib/postgresql/13 -type f >>
            whitelist && \
find /usr/share/postgresql/13 -type f >>
            whitelist && \
find /usr/bin/env -type f >> whitelist && \
echo /etc/hostname >> whitelist && \
echo /lib/x86_64-linux-gnu/ld-2.28.so >>
            whitelist && \
echo /lib/x86_64-linux-gnu/libreadline.so.7 >>
            whitelist && \
echo /lib/x86_64-linux-gnu/libreadline.so.7.0 >>
            whitelist && \
echo /lib64/ld-linux-x86-64.so.2 >> whitelist && \
chainsaw-diff && \
chainsaw-cut --confirm diff
```

```
FROM scratch
COPY --from=build / /

VOLUME ["/var/lib/postgresql/data"]

ENV PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:
                /sbin:/bin:/usr/lib/postgresql/13/bin
ENV PGDATA=/var/lib/postgresql/data
ENV LD_LIBRARY_PATH=/usr/lib/postgresql/13/lib

EXPOSE 5432/tcp

STOPSIGNAL SIGINT

CMD ["postgres"]
ENTRYPOINT [ "docker-entrypoint.sh" ]
```

Приложение Г. Dockerfile для образа tomcat

```
FROM tomcat:9.0 as build
COPY ./sample/ /usr/local/tomcat/webapps/

RUN cd / && \
    apt-get update && \
    apt-get install git python3-pip -y && \
    yes | pip3 install ninja meson && \
    git clone https://github.com/igankevich/chainsaw && \
    cd chainsaw && \
    meson build && \
    ninja -C build && \
    ninja -C build install && \
    cd /
```

```
RUN chainsaw-blacklist / && \
    (chainsaw-whitelist timeout --signal=KILL 10s
             /usr/local/tomcat/bin/catalina.sh run || true) && \
    find /usr/local/tomcat -type f >> whitelist && \
    echo /bin/cp >> whitelist && \
    echo /bin/bash >> whitelist && \
    echo /usr/bin/env >> whitelist && \
    echo /lib/x86_64-linux-gnu/ld-2.28.so >> whitelist && \
    echo /lib64/ld-linux-x86-64.so.2 >> whitelist && \
    echo /lib/x86_64-linux-gnu/libselinux.so.1 >>
                whitelist && \
    echo /usr/lib/x86_64-linux-gnu/libacl.so.1 >>
                whitelist && \
    echo /usr/lib/x86_64-linux-gnu/libacl.so.1.1.2253 >>
                whitelist && \
    echo /usr/lib/x86_64-linux-gnu/libattr.so.1 >>
                whitelist && \
    echo /usr/lib/x86_64-linux-gnu/libattr.so.1.1.2448 >>
                whitelist && \
    echo /lib/x86_64-linux-gnu/libpcre.so.3 >>
                whitelist && \
    echo /lib/x86_64-linux-gnu/libpcre.so.3.13.3 >>
                whitelist && \
    chainsaw-diff && \
    chainsaw-cut --confirm diff

FROM scratch
COPY --from=build / /

WORKDIR /usr/local/tomcat

ENV PATH=/usr/local/tomcat/bin:/usr/local/openjdk-11/bin:
```

```
          /usr/local/sbin:/usr/local/bin:/usr/sbin:
          /usr/bin:/sbin:/bin
ENV JAVA_HOME=/usr/local/openjdk-11
ENV LANG=C.UTF-8
ENV JAVA_VERSION=11.0.11+9
ENV CATALINA_HOME=/usr/local/tomcat
ENV TOMCAT_NATIVE_LIBDIR=/usr/local/tomcat/native-jni-lib
ENV LD_LIBRARY_PATH=/usr/local/tomcat/native-jni-lib
ENV GPG_KEYS="05AB33110949707C93A279E3D3EFE6B686867BA6
          07E48665A34DCAFAE522E5E6266191C37C037D42
          47309207D818FFD8DCD3F83F1931D684307A10A5
          541FBE7D8F78B25E055DDEE13C370389288584E7
          61B832AC2F1C5A90F0F9B00A1C506407564C17A3
          79F7026C690BAA50B92CD8B66A3AD3F4F22C4FED
          9BA44C2621385CB966EBA586F72C284D731FABEE
          A27677289986DB50844682F8ACB77FC2E86E29AC
          A9C5DF4D22E99998D9875A5110C01C5A2F6059E7
          DCFD35E0BF8CA7344752DE8B6FB21E8933C60243
          F3A04C595DB5B6A5F1ECA43E3B7BBB100D811BBE
          F7DA48BB64BCB84ECBA7EE6935CD23C10D498E23"
ENV TOMCAT_MAJOR=9
ENV TOMCAT_VERSION=9.0.46
ENV TOMCAT_SHA512=4a82ed571d4060ae7cd6730718d7b54a3fa7ea
          af7c8bb0e3e8abbff92d76856db52f3a87c5b5ee4e8452483bb3b
          31a5de55e192a18ea4229305780503ed63951

EXPOSE 8080

CMD [ "bash", "-c", "cp -r webapps.dist/* webapps/ ;
                catalina.sh run" ]
```

Приложение Д. Dockerfile для образа peer-calls

```
FROM node:12-alpine as frontend
COPY package.json package-lock.json /src/

WORKDIR /src

RUN set -ex \
 && apk add --no-cache \
      git \
 && npm ci

COPY ./ /src/

RUN set -ex \
 && npm run build


FROM golang:alpine as server

ENV CGO_ENABLED=0

RUN set -ex \
 && apk add --no-cache \
      git

COPY go.mod go.sum /src/
WORKDIR /src

RUN set -ex \
 && go mod download

COPY                  ./          /src/
```

```
COPY --from=frontend  /src/build/ /src/build/

RUN set -ex \
 && go build \
      -ldflags "-X main.GitDescribe=$(git describe
                          --always --tags --dirty)" \
      -mod=readonly \
      -o peer-calls \
 && ls /bin/


FROM chainsaw:alpine as optimizer
COPY --from=server /src/peer-calls /usr/local/bin/

RUN chainsaw-blacklist / && \
    (chainsaw-whitelist timeout 2s
                   /usr/local/bin/peer-calls || true) && \
    chainsaw-diff && \
    echo whitelist >> diff && \
    echo blacklist >> diff && \
    echo diff >> diff && \
    chainsaw-cut --confirm diff

FROM scratch
COPY --from=optimizer / /

EXPOSE 3000/tcp
STOPSIGNAL SIGINT

ENTRYPOINT ["/usr/local/bin/peer-calls"]
```

Приложение E. Dockerfile для образа Mycodo-app

```
FROM python:3.7-slim-buster as build

ENV DOCKER_CONTAINER TRUE

RUN useradd -ms /bin/bash pi
RUN useradd -ms /bin/bash mycodo

COPY . /home/mycodo

WORKDIR /home/mycodo/mycodo

RUN /home/mycodo/mycodo/scripts/upgrade_commands.sh
          create-files-directories
RUN /home/mycodo/mycodo/scripts/upgrade_commands.sh
          update-apt
RUN /home/mycodo/mycodo/scripts/upgrade_commands.sh
          update-packages
RUN /home/mycodo/mycodo/scripts/upgrade_commands.sh
          install-docker-ce-cli
RUN /home/mycodo/mycodo/scripts/upgrade_commands.sh
          docker-update-pip
RUN /home/mycodo/mycodo/scripts/upgrade_commands.sh
          docker-update-pip-packages
RUN /home/mycodo/mycodo/scripts/upgrade_commands.sh
          ssl-certs-generate
RUN /home/mycodo/mycodo/scripts/upgrade_commands.sh
          docker-compile-translations

RUN yes | pip3 install ninja meson && \
git clone https://github.com/igankevich/chainsaw && \
cd chainsaw && \
```

```
meson build && \
ninja -C build && \
ninja -C build install

RUN cd / && \
chainsaw-blacklist / && \
cd /home/mycodo/mycodo && \
(chainsaw-whitelist timeout --signal=KILL 10s python
                mycodo_daemon.py || true) && \
mv whitelist /whitelist1 && \
export FLASK_APP=app.py && \
(chainsaw-whitelist timeout --signal=KILL 10s flask
                run || true) && \
mv whitelist /whitelist2 && \
(chainsaw-whitelist timeout --signal=KILL 10s gunicorn
                --workers 1
                --bind unix:/home/mycodo/mycodoflask.sock
                start_flask_ui:app || true) && \
mv whitelist /whitelist3 && \
cd / && \
cat whitelist1 whitelist2 whitelist3 >> whitelist && \
echo /usr/bin/gunicorn >> whitelist && \
echo /usr/bin/flask >> whitelist && \
echo /bin/bash >> whitelist && \
echo /bin/sh >> whitelist && \
echo /lib/x86_64-linux-gnu/ld-2.28.so >> whitelist && \
echo /lib64/ld-linux-x86-64.so.2 >> whitelist && \
find /home/mycodo -type f >> whitelist && \
find /var/log -type f >> whitelist && \
find /var/lock -type f >> whitelist && \
find /var/run -type f >> whitelist && \
echo /etc/localtime >> whitelist && \
find /dev -type f >> whitelist && \
```

```
find /usr/local/lib/python3.7/site-packages -type f
                >> whitelist && \
find /usr/lib/python3.7 -type f >> whitelist && \
find /usr/local/lib/python3.7 -type f >> whitelist && \
find /etc/passwd /etc/group -type f >> whitelist && \
find /lib/x86_64-linux-gnu >> whitelist && \
chainsaw-diff && \
chainsaw-cut --confirm diff

FROM scratch

COPY --from=build / /

WORKDIR /home/mycodo/mycodo
ENV DOCKER_CONTAINER=TRUE
```