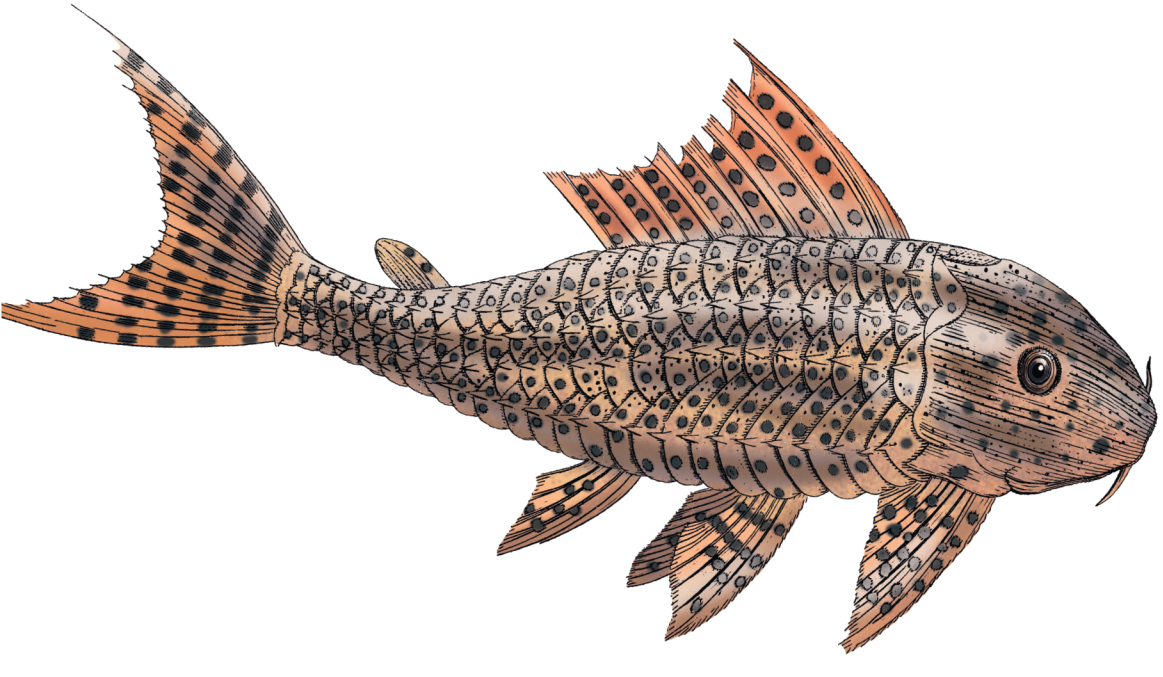
O'REILLY®

Безопасность контейнеров

Фундаментальный подход к защите контейнеризированных приложений



Лиз Райс

Container Security

Fundamental Technology Concepts that Protect Containerized Applications

Liz Rice

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Безопасность контейнеров

Фундаментальный подход к защите контейнеризированных приложений

Лиз Райс

ПИТЕР®

Санкт-Петербург • Москва • Минск

2021

ББК 32.988.02-018-07
УДК 004.056.53
Р18

Райс Лиз

Р18 Безопасность контейнеров. Фундаментальный подход к защите контейнеризированных приложений. — СПб.: Питер, 2021. — 224 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-1850-2

Во многих организациях приложения работают в облачных средах, обеспечивая масштабируемость и отказоустойчивость с помощью контейнеров и средств координации. Но достаточно ли защищена развернутая система? В этой книге, предназначенной для специалистов-практиков, изучаются ключевые технологии, с помощью которых разработчики и специалисты по защите данных могут оценить риски для безопасности и выбрать подходящие решения.

Лиз Райс исследует вопросы построения контейнерных систем в Linux. Узнайте, что происходит при развертывании контейнеров, и научитесь оценивать возможные риски для безопасности развертываемой системы. Приступайте, если используете Kubernetes или Docker и знаете базовые команды Linux.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.988.02-018-07
УДК 004.056.53

ISBN 978-1492056706 англ.

Authorized Russian translation of the English edition of Container Security ISBN 9781492056706 © 2020 Vertical Shift Ltd.
This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

ISBN 978-5-4461-1850-2

Краткое содержание

Оглавление

Предисловие

Во многих организациях приложения работают в нативных облачных средах, обеспечивая масштабируемость и отказоустойчивость с помощью контейнеров и средств координации. Как участнику команды Ops, DevOps или даже DevSecOps, отвечающему за настройку подобной среды для своей компании, гарантировать безопасность развертываемых приложений? Как специалисту по безопасности, имеющему опыт использования традиционных систем на основе серверов или виртуальных машин, адаптировать свои знания к контейнерному развертыванию? И что разработчику нативных облачных приложений стоит учесть, чтобы повысить безопасность своих контейнеризованных приложений? В данной книге описаны ключевые технологии, лежащие в основе контейнеров и нативного облачного программирования. Поэтому после ее прочтения вы сможете лучше оценить риски для безопасности и решения, подходящие для конкретной среды, а также избежать нерекомендуемых приемов, которые подвергают опасности технологии, развернутые вами.

С помощью этой книги вы изучите многие базовые технологии и механизмы, часто применяемые в контейнерных системах, а также способы их построения в операционной системе Linux. Вместе мы углубимся в основы функционирования контейнеров и их взаимодействия и ответим не только на вопрос *«что»* относительно безопасности контейнеров, но и, главное, *«почему»*. При написании данной книги я ставила перед собой цель помочь читателю лучше разобраться в происходящем при развертывании контейнеров. Мне хотелось бы вдохновить вас на создание ментальных моделей, позволяющих вам самостоятельно оценить потенциальные риски для безопасности при развертывании.

В основном в этой книге обсуждаются контейнеры приложений, которые в настоящее время используют многие организации, чтобы запускать свои приложения в таких системах, как Kubernetes и Docker, а не контейнеры

для систем наподобие LXC и LXD из проекта Linux Containers Project (https://linuxcontainers.org/). В контейнере приложений можно запускать неизменяемые контейнеры с помощью кода объемом не больше, чем требуется для запуска приложения. Наряду с этим в среде системного контейнера выполняется полный дистрибутив Linux, и работают с ним скорее как с виртуальной машиной. Вполне допустимо подключаться к системному контейнеру по SSH. Если же вы захотите подключиться по SSH к контейнеру приложений, то специалисты по их безопасности посмотрят на вас косо (по причинам, изложенным далее в этой книге). Впрочем, основные механизмы создания контейнеров для систем и приложений совпадают: контрольные группы, пространства имен и изменение корневого каталога. Так что фундамент, заложенный в данной книге, позволит вам и далее изучать разницу в подходах, которые используются в различных проектах контейнеров.

Для кого эта книга

Неважно, кем вы себя считаете: разработчиком, специалистом в области безопасности, оператором или менеджером, — эта книга подойдет, если вам интересно дойти до самой сути того, как функционирует что-либо, и нравится проводить время за терминалом Linux.

Если же вы ищете пошаговое руководство по безопасности контейнеров, то, возможно, эта книга не для вас. Я не верю в существование универсального рецепта, подходящего для всех приложений во всех средах и для всех организаций. Напротив, я хочу помочь вам разобраться, что происходит при запуске приложений в контейнерах и как работают различные механизмы безопасности, чтобы вы могли сами оценить риски.

Как вы узнаете далее, контейнеры основаны на сочетании некоторых функциональных возможностей ядра Linux. Механизмы обеспечения безопасности контейнеров во многом схожи с механизмами безопасности для хоста Linux (под словом «хост» я понимаю как виртуальные машины, так и физические серверы). Я разложу по полочкам все нюансы функционирования этих механизмов, а затем продемонстрирую их применение в контейнерах. Опытные системные администраторы могут спокойно пропустить часть разделов и перейти сразу к информации, относящейся к контейнерам.

Я предполагаю, что вы хотя бы поверхностно знакомы с контейнерами и, возможно, хотя бы немного «играли» с Docker и Kubernetes. А также понимаете, как минимум в общих чертах, выражения типа «извлечь образ контейнера из реестра» или «запустить контейнер», даже если не знаете в точности, что происходит «под капотом» при подобных действиях. Я не жду от вас знаний нюансов работы конвейеров, по крайней мере до того, как вы прочитаете данную книгу.

Структура издания

Мы начнем в главе 1 с моделей угроз и векторов атак, встречающихся при контейнерном развертывании, а также нюансов безопасности контейнеров по сравнению с безопасностью при обычном развертывании. Цель оставшейся части книги — помочь вам разобраться в контейнерах и связанных с ними угрозах безопасности, равно как и в защите от них.

Прежде чем заняться собственно безопасностью контейнеров, необходимо разобраться, как они работают. Глава 2 описывает основные механизмы Linux, в частности системные вызовы и привилегии, используемые для контейнеров. Далее, в главах 3 и 4, мы углубимся в обсуждение компонентов Linux, из которых состоят контейнеры. Благодаря этому вы поймете, чем в действительности являются контейнеры и насколько они изолированы друг от друга. А в главе 5 мы сравним их степень изоляции с изоляцией виртуальных машин.

В главе 6 вы узнаете о содержимом образов контейнеров и некоторых практических рекомендациях по обеспечению их безопасности. А в главе 7 научитесь определять известные программные уязвимости контейнеров.

В главе 8 мы рассмотрим некоторые необязательные меры безопасности Linux, выходящие за рамки базовой реализации, представленной в главе 4, позволяющие усилить безопасность контейнеров. Вдобавок в главе 9 обсудим способы нарушения изоляции контейнеров в результате опасных, хотя и очень распространенных ошибок конфигурации.

Далее мы обратимся к вопросу взаимодействия контейнеров. Глава 10 рассказывает о способах обмена информацией между контейнерами и улучшения безопасности с помощью соединений между ними. Глава 11 посвящена

основам ключей и сертификатов, благодаря которым контейнеризованные компоненты могут идентифицировать друг друга и создавать между собой защищенное соединение. Хотя их использование в случае контейнеров не отличается от любых других компонентов, мы включили этот вопрос в нашу книгу, поскольку в распределенных системах ключи и сертификаты часто вызывают затруднения. В главе 12 мы покажем, как безопасно (и как не столь безопасно) передавать сертификаты и прочие учетные данные в контейнеры во время выполнения.

В главе 13 мы поговорим о способах, позволяющих предотвращать атаки во время выполнения благодаря возможностям контейнеров.

Наконец, в главе 14 мы рассмотрим список из десяти основных рисков для безопасности, опубликованный открытым проектом обеспечения безопасности веб-приложений (Open Web Application Security Project, OWASP) и рассмотрим подходы к их устранению на основе контейнеров. Спойлер: некоторые риски для безопасности устраняются одинаково для контейнеризованных и неконтейнеризованных приложений.

Примечание относительно Kubernetes

Используемые в настоящее время контейнеры работают в основном под управлением механизма координации Kubernetes (https://kubernetes.io/). Он автоматизирует процесс выполнения различной рабочей нагрузки на кластере машин, и в некоторых местах книги я предполагаю, что в общих чертах эта идея вам понятна. В целом я старалась сосредоточиться на понятиях уровня контейнеров — «плоскости данных» при развертывании с помощью Kubernetes.

Поскольку рабочие задания Kubernetes выполняются в контейнерах, эта книга в определенной степени относится и к безопасности Kubernetes, однако ни в коем случае не является всесторонним обсуждением вопросов безопасности Kubernetes или нативных облачных развертываний. Многие прочие вопросы настройки и использования компонентов плоскости управления также выходят за рамки данной книги. Если вы хотите получить больше информации по этим вопросам, то рекомендую заглянуть в книгу Kubernetes Security издательства O'Reilly (https://oreil.ly/Of6yK) (которую я написала совместно с Майклом Хаузенбласом (Michael Hausenblas)).

Примеры

В данной книге вы найдете множество примеров, и я рекомендую вам поэкспериментировать с ними.

В этих примерах я предполагаю, что вы хорошо знакомы с основными утилитами командной строки Linux, наподобие `ps` и `grep`, а также с основными операциями, необходимыми для запуска контейнерных приложений с помощью таких утилит, как `kubectl` или `docker`. Первый из этих наборов утилит поможет нам пояснить очень многое из того, что происходит при использовании второго!

Чтобы следить за ходом примеров, вам понадобится доступ к машине с Linux или соответствующей виртуальной машине. Я создавала примеры с помощью виртуальной машины Ubuntu 19.04, работавшей в VirtualBox (https://www.virtualbox.org/) на моем Mac. Кроме того, для создания, запуска и останова моих виртуальных машин я применяла Vagrant (https://www.vagrantup.com/). Аналогичные результаты можно получить на самых различных дистрибутивах Linux, а также виртуальных машин от вашего излюбленного поставщика облачных сервисов.

Запуск контейнеров

Многие люди непосредственно запускали контейнеры в основном (а иногда и только) с помощью Docker. Он сильно упростил использование контейнеров за счет набора удобных утилит для разработчиков. Управлять контейнерами и их образами из командной строки позволяет команда `docker`.

Утилита `docker` фактически представляет собой тонкую прослойку, которая вызывает API основного компонента Docker, производящего всю основную работу демона. При каждом запуске контейнера вызывается компонент демона под названием `containerd`. Данный компонент проверяет наличие требуемого образа контейнера, после чего вызывает компонент `runc`, фактически создающий объект контейнера.

При желании можно запустить контейнер самостоятельно, вызвав `containerd` или даже непосредственно `runc`. Компания Docker в 2017 году безвозмездно передала проект `containerd` фонду Cloud Native Computing Foundation (CNCF) (https://cncf.io/).

В Kubernetes используется Container Runtime Interface (CRI), который пользователи могут при желании выбрать в качестве среды выполнения контейнеров. На сегодняшний день чаще всего применяются упомянутый выше `containerd` (https://containerd.io/) и `CRI-O` (https://cri-o.io/) (который, прежде чем его безвозмездно передали CNCF, был частью Red Hat).

Интерфейс командной строки Docker — лишь один из вариантов управления контейнерами и образами. Существуют и другие возможности запуска контейнеров приложений, наподобие тех, что обсуждаются в этой книге. Один из подобных вариантов — утилита `podman` из дистрибутива Red Hat, изначально предназначенная для устранения зависимости от компонента-демона.

В примерах, представленных в этой книге, используется множество разнообразных контейнерных утилит, чтобы показать большое количество реализаций контейнеров, значительная часть функциональности которых совпадает.

Обратная связь

В дополнение к этой книге существует сайт containersecurity.tech. Не стесняйтесь сообщать туда о найденных проблемах и поправках, которые вы хотели бы увидеть в следующих изданиях.

Условные обозначения

В данной книге используются следующие условные обозначения.

Курсив

Курсивом выделены новые термины и важные слова.

`Моноширинный шрифт`

Используется для листингов программ, а также внутри абзацев для обозначения таких элементов, как переменные и функции, базы данных, типы данных, переменные среды, операторы и ключевые слова, имена файлов и их расширений.

`Моноширинный жирный шрифт`

Показывает команды или другой текст, который пользователь должен ввести самостоятельно.

Моноширинный курсив

Показывает текст, который должен быть заменен значениями, введенными пользователем, или значениями, определяемыми контекстом.

Шрифт без засечек

Используется для обозначения URL, адресов электронной почты, названий кнопок, каталогов.

Этот рисунок указывает на общее примечание.

Использование примеров кода

Дополнительные материалы (примеры кода, упражнения и т. д.) доступны для скачивания по адресу https://containersecurity.tech/.

Если при использовании примеров кода у вас возникнут технические вопросы или проблемы, то, пожалуйста, напишите нам на электронную почту по адресу bookquestions@oreilly.com.

Помочь вам делать вашу работу — вот цель создания этой книги. Если к ней прилагается какой-либо пример кода, то вы можете задействовать его в ваших программах и документации. Обращаться к нам за разрешением нет необходимости, разве что вы копируете значительную часть кода. Так, написание программы, в которой используется несколько фрагментов кода из этой книги, не требует отдельного разрешения, в отличие от продажи или распространения компакт-дисков с примерами из книг O'Reilly. Вы можете свободно цитировать эту книгу с примерами кода, отвечая на некий вопрос. А вот если хотите включить существенную часть приведенного здесь кода в документацию своего программного продукта, то вам следует связаться с нами.

Мы приветствуем, хотя и не требуем, ссылки на первоисточник. Она включает название, автора, издательство и ISBN. Например: «Лиз Райс. Безопасность контейнеров. СПб.: Питер, 2021. 978-5-4461-1850-2».

Если вам кажется, что ваше обращение с примерами кода выходит за рамки правомерного использования или условий, перечисленных выше, то можете обратиться к нам по адресу permissions@oreilly.com.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

Благодарности

Я благодарна множеству людей, которые помогали мне и поддерживали, пока я писала эту книгу:

- моему редактору из издательства O'Reilly Вирджинии Вилсон (Virginia Wilson), державшей весь процесс под контролем и следившей, чтобы эта книга оказалась на должной высоте;
- научным редакторам за их вдумчивые замечания и конструктивную обратную связь: Ахилю Белю (Akhil Behl), Эндрю Мартину (Andrew Martin), Эрику Ст. Мартину (Erik St. Martin), Филу Эстесу (Phil Estes), Рани Оснату (Rani Osnat) и Роберту П. Дж. Дею (Robert P. J. Day);
- моим коллегам из компании Aqua Security, за долгие годы научившим меня столь многому о безопасности контейнеров;
- Филу Перлу (Phil Pearl) — моему мужу, лучшему критику и учителю и в то же время моему лучшему другу.

ГЛАВА 1

Угрозы безопасности контейнеров

За последние несколько лет масштабы применения контейнеров резко возросли. Соответствующие концепции контейнеров существовали еще за несколько лет до Docker. Но большинство наблюдателей сходятся во мнении, что именно появление в 2013 году Docker с его удобными в использовании утилитами командной строки дало толчок популярности контейнеров среди сообщества разработчиков.

У контейнеров есть множество достоинств. Как гласит рекламный слоган Docker, с их помощью можно «создать один раз, выполнять где угодно» — благодаря объединению в пакет приложения и всех его зависимостей и изоляции приложения от остальной части машины, на которой оно работает. У контейнеризованного приложения есть все необходимое, его легко можно упаковать в образ контейнера, который будет работать одинаково на моем/вашем ноутбуке и на сервере в центре обработки данных (ЦОД).

Следствие этой изоляции — возможность параллельного выполнения нескольких различных контейнеров, которые не будут мешать друг другу. До появления контейнеров мешанина зависимостей могла легко превратиться в настоящий кошмар для разработчиков, в котором двум приложениям требовались различные версии одних и тех же пакетов. Проще всего решить данную проблему, выполняя приложения на отдельных машинах. Контейнеры изолируют зависимости друг от друга, и потому выполнение нескольких приложений на одном сервере не доставляет никаких проблем. Все быстро поняли, что благодаря контейнеризации можно запускать несколько приложений на одном хосте (неважно, виртуальной машине или реальном сервере), не беспокоясь о зависимостях.

Следующий логичный шаг — распределение контейнеризованных приложений по кластеру серверов. Благодаря средствам координации наподобие Kubernetes этот процесс автоматизируется до такой степени, что больше не нужно вручную устанавливать приложения на конкретных машинах,

достаточно сообщить средству координации, какие контейнеры необходимо запустить, и оно само найдет подходящую машину для каждого из них.

С точки зрения безопасности контейнеризованная среда во многом схожа с обычным развертыванием. Нарушители пытаются похитить данные, или изменить поведение системы, или, скажем, использовать вычислительные ресурсы других людей для майнинга криптовалюты. При переходе к контейнерам ничего из этого не меняется. Однако контейнеры существенно меняют способ работы приложений, что приводит к иному набору рисков для безопасности.

Риски, угрозы и уменьшение их последствий

Риск (risk) — это потенциальная проблема, а также ее возможные последствия.

Угроза (threat) — путь реализации этого риска.

Уменьшение их последствий (mitigation) — контрмеры, с помощью которых можно предотвратить угрозу или по крайней мере уменьшить вероятность ее успешной реализации.

Скажем, существует риск, что какой-нибудь злоумышленник украдет из вашего дома ключи от вашей же машины и уедет на ней. Угрозы в данном случае — это различные способы кражи ключей: разбить окно, запустить руку и схватить ключи; просунуть удочку через щель для почты; постучать в дверь и отвлечь вас, пока сообщник быстро проскользнет внутрь и схватит ключи. Чтобы уменьшить последствия всех этих угроз, можно, например, убрать ключи от машины с видного места.

Риски очень различаются в разных организациях. Основной риск для банка, хранящего клиентские деньги, — их кража. Для интернет-магазина основная головная боль — мошеннические транзакции. Ведущий личный блог пользователь может бояться, например, что кто-то взломает его учетную запись, выдаст себя за него и начнет публиковать непристойные комментарии. В разных странах законодательство о защите персональной информации имеет свои особенности, поэтому различается и риск утечки личных данных пользователей — во многих странах риски «лишь» репутационные, в то время как в Европе Общий регламент защиты персональных данных (General Data Protection Regulation, GDPR) допускает штрафы до 4 % общего дохода компании (https://oreil.ly/guQg3).

А поскольку риски весьма разнятся, то сильно различается и относительная значимость потенциальных угроз, равно как и соответствующий набор средств уменьшения последствий. В основе управления рисками лежит процесс их систематизации, перечисления возможных угроз, расстановки их по приоритетам и выбора подхода к уменьшению их последствий.

Моделирование угроз (threat modeling) — процесс распознавания и перечисления возможных угроз системе. За счет планомерного анализа ее компонентов и вероятных векторов атаки модель угроз помогает определить места системы, наиболее уязвимые для атак.

Единой всеобъемлющей модели угроз не существует, все зависит от рисков конкретной среды, организации и запускаемых приложений. Но можно перечислить некоторые потенциальные угрозы, общие для многих, если не всех, контейнерных развертываний.

Модель угроз для контейнеров

В частности, модель угроз можно рассматривать с точки зрения ее участников, в число которых могут входить:

- *внешние нарушители* (external attackers), пытающиеся извне получить доступ к развернутой системе;
- *внутренние нарушители* (internal attackers), сумевшие получить доступ к некой части развернутой системы;
- *внутренние действующие лица-злоумышленники* (malicious internal actors), например, разработчики и администраторы с определенным уровнем полномочий доступа к развернутой системе;
- *небрежные внутренние действующие лица* (inadvertent internal actors), которые могут неумышленно вызывать проблемы;
- *процессы приложений* (application processes) — не люди-злоумышленники, тем не менее имеющие определенный программный доступ к системе.

Необходимо учитывать набор прав доступа каждого из действующих лиц.

- Какой доступ к системе есть у этого лица в соответствии с его учетными данными? Например, есть ли у него доступ к пользовательским учетным записям на машинах хостов, где работает развернутая система?

- Какие права доступа оно имеет в системе? В Kubernetes этот пункт относится к настройкам управления доступом для всех пользователей, в том числе анонимных, на основе ролей.
- Какие права доступа к сети есть у этого лица? Например, какие части системы включены в виртуальное частное облако (virtual private cloud, VPC)?

Существует несколько возможных путей атаки на развернутую контейнеризованную систему, и чтобы их систематизировать, можно, например, проанализировать потенциальные векторы атак на каждом из этапов жизненного цикла контейнера (рис. 1.1).

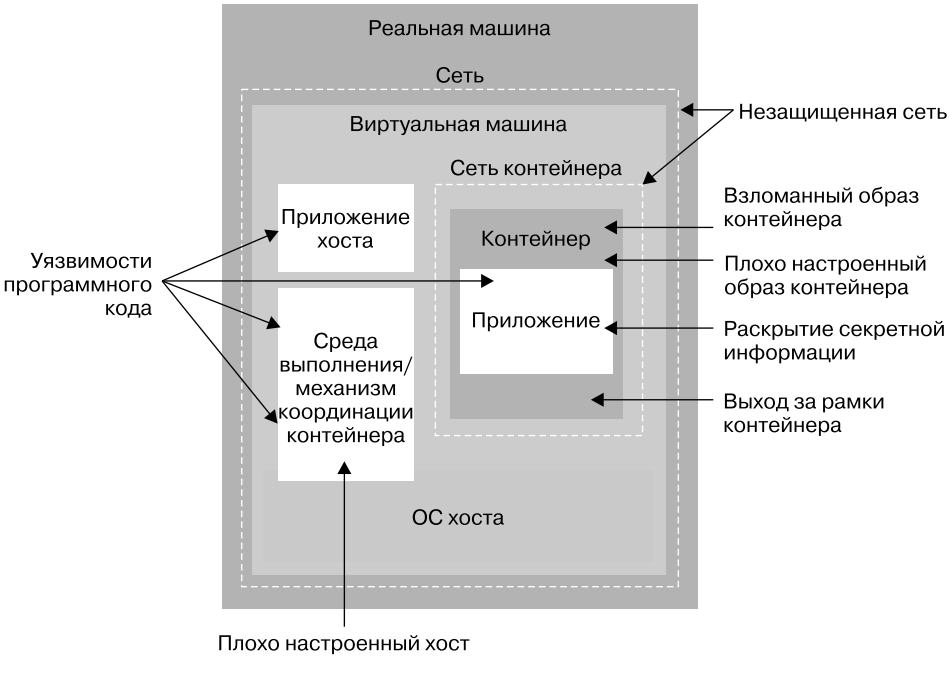


Рис. 1.1. Векторы атак на контейнеры

Рассмотрим эти векторы более подробно.

- *Уязвимый код*. Жизненный цикл приложения начинается с написания разработчиком его кода. Он, равно как и его зависимости, может содержать изъяны (уязвимости). Существуют списки из тысяч известных уязвимостей, которыми (если они есть в приложении) могут воспользоваться злоумышленники. Образы необходимо анализировать, как вы увидите

в главе 7, чтобы не применять контейнеры с известными уязвимостями. Причем делать это нужно регулярно, поскольку уязвимости обнаруживаются в уже существующем коде постоянно. В процессе анализа должны также выявляться контейнеры с устаревшим ПО, которое необходимо обновить, установив исправления безопасности. Кроме того, есть анализаторы, способные выявлять встроенное в образы вредоносное программное обеспечение.

- *Плохо настроенные образы контейнеров.* Написанный код встраивается в образ контейнера. В ходе конфигурации сборки образа контейнера возникает множество возможностей создать уязвимости, которые открывают дорогу для дальнейших атак на работающий контейнер. В их число входит выполнение контейнера от имени суперпользователя, в результате чего у него оказывается больше полномочий, чем нужно. Больше информации об этом — в главе 6.

- *Атаки на систему сборки.* Если злоумышленник может изменить сборку образа контейнера или как-то повлиять на нее, то сможет вставить вредоносный код, который потом будет запущен в среде промышленной эксплуатации. Кроме того, возможность закрепиться внутри среды сборки — плацдарм для злоумышленника, позволяющий в дальнейшем проникать в среду промышленной эксплуатации. Этот вопрос также обсуждается в главе 6.

- *Атаки на цепь поставок.* Собранный образ контейнера сохраняется в реестре, откуда извлекается перед запуском. Как гарантировать соответствие извлекаемого образа тому, который был ранее помещен в реестр? Не могли ли злоумышленники внести в него изменения? Любой, кто может заменить образ или модифицировать его в промежутке между сборкой и развертыванием, сможет выполнить любой код в развернутой системе.

- *Плохо настроенные контейнеры.* Как мы обсудим в главе 9, контейнер можно запустить с настройками, в результате которых у них появляются ненужные, а порой и незапланированные полномочия. Скачивая файлы конфигурации YAML из Интернета, пожалуйста, не запускайте их, не убедившись в отсутствии в них небезопасных настроек!

- *Уязвимые хосты.* Контейнеры выполняются на хост-компьютерах, поэтому нужно проверять работающий на них код на наличие уязвимостей (например, отслеживать старые версии компонентов механизма

координации, с известными уязвимостями). Имеет смысл уменьшить до минимума объем запущенного на каждом хосте программного обеспечения, чтобы сократить поверхность атаки. Кроме того, необходимо задать правильную конфигурацию хостов в соответствии с практическими рекомендациями по обеспечению безопасности. Все это обсуждается в главе 4.

- *Общедоступные секретные данные*. Чтобы взаимодействовать с другими компонентами системы, код приложения часто требует учетные данные, токены или пароли. При развертывании в контейнере эти секретные значения необходимо передавать в контейнеризованный код. Как вы увидите в главе 12, существует несколько различных вариантов решения этой задачи, имеющих разную степень безопасности.

- *Незащищенная сеть*. Контейнерам обычно требуется взаимодействовать друг с другом или с окружающим миром. В главе 10 обсуждается передача данных по сети в контейнерах, а в главе 11 — установление защищенных соединений между компонентами.

- *Уязвимости выхода за рамки контейнера*. Широко используемые среды выполнения контейнеров, включая `containerd` и `CRI-O`, уже хорошо проверены в деле, однако не исключено, что в них все же остаются программные ошибки, вследствие которых вредоносный код, работающий внутри контейнера, может просочиться за пределы контейнера, в хост. Одна из таких проблем — Runcescape (https://oreil.ly/cFSaJ) — обнаружилась в 2019 году. В главе 4 можно прочитать об изоляции, предназначенной для ограничения кода приложения рамками контейнера. Ущерб от выхода за рамки контейнера для ряда приложений может быть столь велик, что имеет смысл задуматься о применении более эффективных механизмов изоляции, таких как те, которые обсуждаются в главе 8.

Некоторые векторы атак выходят за рамки данной книги.

- Исходный код обычно хранится в репозиториях, потенциально доступных для атак, цель которых — взлом приложения. Необходимо обеспечить должный контроль доступа пользователя к репозиторию.

- Хост-компьютеры связываются между собой сетью, обычно подключенной к Интернету, причем в целях безопасности при этом часто применяется VPC. Как и при обычном развертывании, необходимо защитить хост-компьютеры (или виртуальные машины) от доступа злоумышленников.

Безопасные настройки сети, использование брандмауэра, а также управление идентификацией и доступом для нативного облачного развертывания ничуть не менее релевантны, чем для обычного.

- Контейнеры обычно работают под управлением механизма координации — в современных развертываниях его роль обычно играет Kubernetes, хотя есть и другие варианты, например Docker Swarm и Hashicorp Nomad. Недостаточная безопасность настроек средства координации или отсутствие должного контроля над доступом с правами администратора открывают злоумышленникам дополнительные векторы атак.

Больше информации о моделях угроз при развертываниях на основе Kubernetes можно найти в отчете «Модель угроз Kubernetes» (Kubernetes Threat Model) (https://bit.ly/3sg7aBn), заказанном CNCF.

Кроме того, команда Financial User Group проекта CNCF опубликовала дерево атак Kubernetes (Kubernetes Attack Tree) (https://bit.ly/3mPGZR2), созданное на основе методологии STRIDE (https://oreil.ly/rNmPN).

Границы зон безопасности

Границы зон безопасности (иногда называемые границами доверия) между частями системы означают, что наборы прав доступа в этих частях отличаются. Иногда границы задаются в ходе администрирования — например, в системах под управлением Linux системный администратор может модифицировать границы зон безопасности, указывая, какие группы файлов доступны пользователю. Администратор делает это, изменяя группы, в которых состоит данный пользователь. Если вы подзабыли систему прав доступа к файлам Linux, то мы напомним ее в главе 2.

Контейнер представляет собой зону безопасности. Код приложения должен работать внутри него и не иметь доступа к коду или данным вне контейнера, за исключением случаев, когда явным образом получает подобное разрешение (например, путем подключения к нему внешнего тома).

Чем строже граница зон безопасности между злоумышленником и его целью (например, данными пользователей), тем сложнее ему достичь этой цели.

Векторы атак, описанные в разделе «Модель угроз для контейнеров» на с. 23, можно связывать цепочкой — чтобы проникнуть через несколько границ зон безопасности. Приведу следующие примеры.

- Злоумышленник может обнаружить, что уязвимость в одной из зависимостей приложения позволяет ему выполнять код удаленно, внутри контейнера.
- Допустим, у взломанного компьютера нет прямого доступа к каким-либо ценным данным. Злоумышленнику необходим способ выйти за рамки контейнера, в другой контейнер или на хост-компьютер. Один из путей наружу — уязвимость выхода за рамки контейнера; другой — небезопасная конфигурация контейнера. Если один из этих путей открыт для злоумышленника, то он может получить доступ к хосту.
- Следующий шаг — поиск способов получить на хосте полномочия суперпользователя, что может оказаться тривиальной задачей, если код приложения выполняется от имени суперпользователя внутри контейнера, как вы увидите в главе 4.
- Имея полномочия суперпользователя на хост-компьютере, злоумышленник может получить доступ ко всему, что доступно с хоста или из любого контейнера, запущенного на этом хосте.

Добавление и укрепление границ безопасности при развертывании значительно усложняет жизнь взломщиков.

Важный аспект модели угроз — учет возможности атак изнутри среды, в которой работает приложение. При развертывании в облаке отдельные ресурсы порой задействуются совместно с другими пользователями и их приложениями. Совместное использование ресурсов компьютера называется *мультиарендностью* (multitenancy) и серьезно влияет на модель угроз.

Мультиарендность

В мультиарендной среде различные пользователи — *арендаторы* (tenants) — выполняют задания на общем оборудовании. (Термин «мультиарендность» можно также встретить в контексте программных приложений, где означает нескольких пользователей, задействующих один и тот же экземпляр программы, но в нашем случае совместно применяется лишь аппаратное обе-

спечение.) В зависимости от владельцев этих различных рабочих заданий, а также степени взаимного доверия арендаторов, могут понадобиться более жесткие границы между ними, чтобы предотвратить возможное негативное влияние их друг на друга.

Мультиарендность — идея, появившаяся еще во времена мейнфреймов в 1960-х, когда пользователи арендовали время CPU, память и место на диске на совместно используемой ими машине. Эта концепция не так уж сильно отличается от современных общедоступных облачных сервисов, например Amazon AWS, Microsoft Azure и Google Cloud Platform, в которых пользователи арендуют время CPU, оперативную память, место для хранения наряду с прочими возможностями и управляемыми сервисами. С тех пор как в 2006 году Amazon AWS предоставил EC2, можно арендовать экземпляры виртуальных машин, запущенные на стойках серверов в центрах обработки данных, разбросанных по всему миру. На одной реальной машине может работать несколько виртуальных (VM), и работающий на группе виртуальных машин пользователь может не знать, кто работает на соседней VM.

Совместно используемые машины

В некоторых случаях одну (возможно, виртуальную) машину Linux задействуют несколько пользователей. Так, очень часто подобный пример истинной мультиарендности встречается в университетах, где пользователи не доверяют друг другу и, говоря откровенно, администраторы не доверяют пользователям. В такой среде доступ пользователей ограничивается благодаря средствам управления доступом Linux. У каждого пользователя есть свой идентификатор входа, и доступ ограничивается с помощью средств управления доступом Linux, чтобы, например, пользователь мог изменять только файлы в своих каталогах. Можете представить, какой бы возник беспорядок, если бы студенты могли читать или — хуже того — редактировать файлы своих однокурсников?

Как вы увидите в главе 4, ядро у всех контейнеров одного хоста — общее. Если на машине применяется демон Docker, то у всех пользователей, имеющих права на выполнение команд `docker`, по сути, есть доступ с полномочиями суперпользователя, и вряд ли администратору следует предоставлять не доверенным пользователям эту возможность.

В корпоративной и особенно в нативной облачной среде подобные совместно используемые машины встречаются реже. Вместо этого пользователи (или команды, доверяющие друг другу) обычно задействуют собственные ресурсы, выделенные им в виде виртуальных машин.

Виртуализация

Вообще говоря, считается, что виртуальные машины достаточно надежно изолированы друг от друга, то есть маловероятно, что ваши соседи смогут увидеть или вмешаться в происходящее в ваших виртуальных машинах. Как достигается подобная изоляция, можно узнать в главе 5. На самом же деле в соответствии с общепринятым определением (https://oreil.ly/yfkQI) виртуализация вообще не является мультиарендностью. Мультиарендность — это когда различные группы пользователей совместно работают с одним экземпляром программного обеспечения, а при виртуализации у пользователей нет доступа к гипервизору, управляющему их виртуальными машинами, поэтому они не используют совместно никаких программ.

Впрочем, изоляция виртуальных машин отнюдь не идеальна, и пользователи ранее жаловались на проблемы с «шумными соседями», когда совместное использование реальной машины может привести к неожиданным колебаниям производительности. Компания Netflix одной из первых начала применять общедоступные облачные сервисы. В 2010 году они опубликовали в своем блоге сообщение, в разделе «Совместная аренда — вещь непростая» (Cotenancy is hard) (https://oreil.ly/CGlZ0) которого признаются, что создаваемые ими системы могли осознанно прекратить выполнение подзадачи, если она выполнялась слишком медленно. В последнее время можно услышать утверждения, что проблема «шумных соседей» больше не актуальна (https://oreil.ly/iE4qE).

Известны случаи, когда уязвимости в программном обеспечении приводили к нарушению границ между виртуальными машинами.

Возможные последствия несанкционированного доступа для некоторых приложений и организаций (особенно правительственных, финансовых и медицинских) настолько серьезны, что оправданно полное физическое разделение. Чтобы гарантировать полную изоляцию рабочих заданий, можно использовать частное облако, работающее в вашем собственном центре обработки данных либо под управлением выбранного вами поставщика сервисов. В частных облаках иногда встречаются дополнительные меры

защиты, например дополнительные проверки анкетных данных персонала, имеющего доступ к ЦОД.

Многие поставщики облачных сервисов предоставляют вариант VM, при котором гарантируется, что одну реальную машину использует только один клиент. Можно также арендовать выделенные физические серверы, обслуживаемые поставщиками облачных сервисов. В обоих этих сценариях полностью устраняется проблема «шумных соседей», плюс вы получаете преимущества, вызванные более надежной изоляцией между реальными машинами.

Неважно, арендуете ли вы реальные или виртуальные машины в облаке или используете собственные серверы, но при работе с контейнерами может понадобиться учесть границы зон безопасности между различными группами пользователей.

Мультиарендность контейнеров

В главе 4 вы увидите, что изоляция контейнеров не так строга, как изоляция виртуальных машин. Хотя это зависит от профиля рисков, вряд ли имеет смысл использовать контейнеры на одной машине в качестве не доверенных сторон.

Даже если вы или люди, которым вы полностью доверяете, управляете всеми запущенными на ваших машинах контейнерами, все равно следует сделать скидку на склонность людей к ошибкам и сделать так, чтобы контейнеры не могли влиять друг на друга.

В Kubernetes с помощью *пространств имен* (namespaces) кластер компьютеров можно разбить на части, используемые различными людьми, командами или приложениями.

«Пространство имен» — термин с несколькими значениями. В Kubernetes пространство имен представляет собой высокоуровневую абстракцию, которая предназначена для разбиения ресурсов кластера с возможным применением к ним различных режимов управления доступом. В Linux пространство имен — низкоуровневый механизм изоляции ресурсов машины, доступных процессу. Более подробно эти пространства имен будут описаны в главе 4.

Используйте механизмы управления доступом на основе ролей (role-based access control, RBAC), чтобы указать, какие пользователи и компоненты могут обращаться к тем или иным пространствам имен Kubernetes. Подробная инструкция по выполнению этого выходит за рамки данной книги. Хотелось бы лишь упомянуть, что RBAC Kubernetes позволяет контролировать только действия, производимые через API Kubernetes. Контейнеры приложений в модулях Kubernetes, работающих на одном хосте, защищены друг от друга лишь с помощью изоляции контейнеров, как описано в данной книге, даже если находятся в разных пространствах имен. Если злоумышленнику удастся выйти за рамки контейнера на хост, то границы пространств имен Kubernetes ни на йоту не изменят его возможностей влиять на другие контейнеры.

Экземпляры контейнеров

Такие облачные сервисы, как Amazon AWS, Microsoft Azure и Google Cloud Platform, предоставляют множество *управляемых сервисов* (managed services), с помощью которых пользователи могут арендовать программное обеспечение, хранилище и другие компоненты, не прибегая к их установке или управлению ими. Классический пример — сервис реляционных баз данных (Relational Database Service, RDS) Amazon; он позволяет легко подготовить к работе базы данных, использующие такое популярное ПО, как PostgreSQL, а для резервного копирования данных достаточно поставить нужную галочку (и оплатить счет, конечно).

Управляемые сервисы распространились и на мир контейнеров. С помощью сервисов Azure Container Instances и AWS Fargate можно запускать контейнеры, не задумываясь о том, на каком компьютере (или виртуальной машине) они будут работать.

Это снимает с наших плеч немалый груз администрирования и позволяет легко масштабировать развертываемую систему. Однако (по крайней мере теоретически) экземпляры контейнеров разных пользователей можно располагать на одной виртуальной машине. Уточните у вашего поставщика облачных сервисов на всякий случай.

Теперь вам известно немало угроз, потенциально опасных для развертываемых вами систем. Прежде чем перейти к оставшейся части книги, я бы хотела познакомить вас с основными принципами безопасности, которые пригодятся вам при выборе утилит и процессов безопасности, требуемых при развертывании.

Принципы безопасности

Обычно считают, что эти общие принципы применимы независимо от того, о безопасности чего идет речь.

Минимум полномочий

Принцип минимума полномочий предполагает ограничение прав доступа пользователя или компонента до мизера, необходимого для выполнения их задачи. Например, для микросервиса, выполняющего поиск товара в приложении электронной коммерции, принцип минимума полномочий означает, что учетная запись этого микросервиса должна предоставлять ему доступ к базе данных только для чтения. Ему не требуется, скажем, информация о пользователях или платежах, равно как нет нужды записывать в базу данных информацию о товарах.

Многослойная защита

Как вы увидите позже, есть множество способов повысить безопасность развертываемой системы и работающих в ней приложений. Принцип многослойной защиты гласит, что защита должна состоять из нескольких отдельных слоев. Если злоумышленник сумеет преодолеть один слой, то следующие все равно не позволят ему нанести вред развернутой системе или похитить данные.

Минимальная поверхность атаки

Как правило, чем больше система, тем выше вероятность существования в ней уязвимостей. Затруднить атаки на нее можно путем упрощения системы, в том числе:

- сокращения количества точек доступа за счет максимального уменьшения и упрощения интерфейсов;
- ограничения количества пользователей и компонентов, имеющих доступ к сервису;
- минимизации объема кода.

Ограничение радиуса поражения

Разбиение средств управления безопасностью на меньшие подкомпоненты гарантирует, что даже при худшем варианте развития событий негативные последствия будут ограничены. Контейнеры прекрасно подходят для реализации этого принципа, ведь контейнер сам может ограничивать зону безопасности за счет разбиения архитектуры на несколько экземпляров микросервиса.

Разграничение обязанностей

Принципу минимума полномочий и ограничения радиуса поражения родственна идея разграничения обязанностей с тем, чтобы компоненты/люди отвечали за как можно меньшую часть всей системы. Этот подход ограничивает ущерб, который может нанести отдельный привилегированный пользователь, поскольку определенные операции требуют полномочий более чем одного пользователя.

Реализация принципов безопасности с помощью контейнеров

Как вы увидите далее, благодаря контейнерам можно применить все эти принципы безопасности.

- *Минимум полномочий*. Различным контейнерам можно назначать разные наборы полномочий, каждый из которых будет иметь лишь те права, которые необходимы для выполнения поставленной задачи.
- *Многослойная защита*. Контейнеры — это еще один рубеж для обеспечения безопасности.
- *Минимальная поверхность атаки*. Разбиение монолитного приложения на простые микросервисы создает между ними «чистые» интерфейсы, позволяющие при тщательном проектировании снизить сложность, а значит, и ограничить поверхность атаки. С другой стороны, в систему добавляется нетривиальный слой координации контейнеров, в результате чего возникает дополнительная поверхность атаки.
- *Ограничение радиуса поражения*. В случае взлома контейнеризованного приложения средства безопасности помогают ограничить атаку рамками контейнера, не позволяя ей затронуть остальную систему.

- *Разграничение обязанностей*. Если передавать права и учетные данные только в те контейнеры, которым они нужны, то утечка одного набора секретных данных не обязательно будет означать потерю их всех.

Все описанные преимущества — хороши, но только с теоретической точки зрения. На практике их легко могут перевесить ненадежная конфигурация системы, плохая организация образов контейнера или небезопасные приемы. К концу этой книги вы будете знать все, что нужно, чтобы избежать типовых ошибок в сфере безопасности, которые могут встретиться при развертывании контейнеризованной системы, и сможете воспользоваться всеми ее преимуществами.

Резюме

Мы рассмотрели в общих чертах виды атак, способных повлиять на контейнерное развертывание, а также принципы безопасности, применяемые для защиты от этих атак. В оставшейся части книги вы погрузитесь в механизмы, лежащие в основе контейнеров, чтобы разобраться в том, как можно реализовать эти принципы путем сочетания утилит безопасности и рекомендуемых практик.

ГЛАВА 2

Системные вызовы Linux, права доступа и привилегии

В большинстве случаев контейнеры запускаются на компьютере, работающем под управлением операционной системы Linux. Поэтому желательно разобраться в ее базовых возможностях, чтобы понимать, как они влияют на безопасность, в частности, применительно к контейнерам. Мы рассмотрим системные вызовы, файловые права доступа, привилегии, а в заключение — повышение полномочий. Если вы хорошо знакомы с этими понятиями, то можете пропустить данную главу и перейти сразу к следующей.

Эти понятия жизненно важны, поскольку *контейнеры запускают процессы Linux, видимые с хоста*. Контейнеризованный процесс использует системные вызовы и требует таких же прав доступа и полномочий, что и обычный. Но контейнеры позволяют иначе управлять назначением этих прав доступа во время выполнения или процесса сборки образа контейнера, что существенно влияет на безопасность.

Системные вызовы

Приложения выполняются в так называемом *пользовательском пространстве* (user space), уровень полномочий которого ниже, чем у ядра операционной системы. Чтобы обратиться к файлу, передать данные по сети или даже узнать время суток, приложению приходится просить ядро выполнить соответствующее действие. Программный интерфейс, с помощью которого код из пользовательского пространства выполняет подобные запросы к ядру, называется интерфейсом *системных вызовов* (system call).

Существует более 300 различных системных вызовов, конкретное количество зависит от версии ядра Linux. Ниже приведены несколько примеров:

- `read` — чтение данных из файла;
- `write` — запись данных в файл;
- `open` — открытие файла для последующего чтения или записи;
- `execve` — запуск исполняемой программы;
- `chown` — изменение владельца файла;
- `clone` — создание нового процесса.

Разработчикам приложений практически (а то и вовсе) никогда не приходится непосредственно иметь дело с системными вызовами, обычно те обернуты в высокоуровневые программные абстракции. Вероятно, самая низкоуровневая абстракция из тех, с которыми вы можете столкнуться как разработчик приложений — библиотека `glibc` или пакет `syscall` языка Golang. Однако на практике они обычно также обернуты в более высокоуровневые слои абстракций.

Узнать больше о системных вызовах можно из моего доклада «Руководство по системным вызовам для начинающих» (A Beginner's Guide to Syscalls) (https://oreil.ly/HrZzJ), доступного на образовательной платформе издательства O'Reilly.

В коде приложения системные вызовы применяются совершенно одинаково, вне зависимости от того, запущено оно в контейнере или нет. Но, как вы увидите далее в этой книге, совместное использование всеми контейнерами на хосте одного ядра (выполнения к нему системных вызовов) вызывает далеко идущие последствия для безопасности.

Не всем приложениям необходимы системные вызовы, поэтому — следуя принципу минимума полномочий — существуют средства защиты Linux, с помощью которых пользователи могут ограничивать набор системных вызовов, доступных различным программам. В главе 8 вы увидите, как это применяется в контексте контейнеров.

Я вернусь к вопросу пользовательского пространства и полномочий уровня ядра в главе 5. А пока обсудим, как в Linux происходит управление правами доступа к файлам.

Права доступа к файлам

В любой системе на основе Linux, неважно, идет ли речь о контейнере, права доступа к файлам — краеугольный камень безопасности. Говорят, что все в Linux является файлами (https://oreil.ly/QTxzb). Код приложений, данные, конфигурация, журналы и т. д. — все хранится в файлах. Даже физические устройства, например мониторы и принтеры, представлены в виде файлов. Права доступа к файлам определяют, какие пользователи могут обращаться к ним и какие действия выполнять над ними. Иногда эти права доступа к файлам называют *избирательным управлением доступом* (discretionary access control, DAC)[1].

Изучим это понятие более подробно.

Если вам приходилось немало времени проводить в терминале Linux, то наверняка вы выполняли команду `ls -l`, чтобы получить информацию о файлах и их атрибутах.

В примере на рис. 2.1 приведена информация об относящемся к группе staff файле `myapp`, владельцем которого является пользователь liz. Атрибуты прав доступа указывают, какие действия пользователи могут выполнять с этим файлом, в зависимости от того, кем являются. В данном примере можно видеть девять символов, соответствующих атрибутам прав доступа, которые следует рассматривать в группах по три символа:

- первая группа из трех символов описывает права доступа для владельца данного файла (в этом примере — пользователя liz);
- вторая группа описывает права доступа для членов группы, с которой связан данный файл (в данном случае staff);
- последние три символа отражают, что могут делать с этим файлом прочие пользователи (не liz и не члены группы staff).

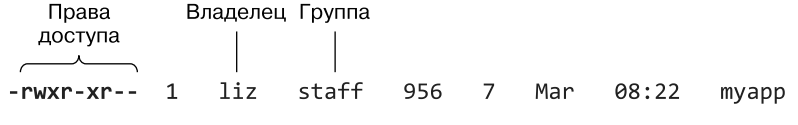


Рис. 2.1. Пример прав доступа к файлу в Linux

[1] В русскоязычной литературе встречаются также варианты «разграничительное управление доступом» и «дискреционное управление доступом». — *Здесь и далее примеч. пер.*

В зависимости от того, установлены ли биты `r`, `w` и `x`, пользователи могут производить над этим файлом три действия: чтение, запись и выполнение. Три символа в каждой группе соответствуют установленным или неустановленным битам, демонстрируя таким образом, какие из трех указанных действий разрешены, — тире означает, что бит не установлен.

В данном примере производить запись в файл может только его владелец, поскольку бит `w` установлен лишь в первой группе символов, соответствующей правам доступа владельца. Выполнять файл может как владелец, так и любой из членов группы `staff`. Читать файл могут все пользователи, поскольку бит `r` установлен во всех трех группах символов.

Больше подробностей о правах доступа Linux можно найти в статье https://oreil.ly/7DKZw.

Очень может быть, что вы уже знаете о битах `r`, `w` и `x`, но это еще не все. На права доступа влияют также биты `setuid`, `setgid` и `sticky`. С точки зрения безопасности важны первые два из них, поскольку с их помощью процессы могут получать дополнительные права, которыми затем могут воспользоваться злоумышленники в своих целях.

Биты setuid и setgid

При обычном выполнении файла запускаемый процесс наследует ваш идентификатор пользователя. Если же у файла установлен бит `setuid`, то процесс получит идентификатор пользователя владельца файла. В следующем примере задействована копия исполняемого файла `sleep`, владельцем которого является несуперпользователь:

```
vagrant@vagrant:~$ ls -l `which sleep`
-rwxr-xr-x 1 root root 35000 Jan 18  2018 /bin/sleep
vagrant@vagrant:~$ cp /bin/sleep ./mysleep
vagrant@vagrant:~$ ls -l mysleep
-rwxr-xr-x 1 vagrant vagrant 35000 Oct 17 08:49 mysleep
```

Информация, которую выводит `ls`, показывает, что владельцем этой копии является пользователь `vagrant`. Запустите данную команду от имени суперпользователя с помощью `sudo sleep 100`, и во втором терминале сможете посмотреть на работающий процесс — `100` означает, что у вас есть на это

100 секунд, прежде чем процесс завершится (чтобы сделать вывод более понятным, я убрала из него несколько строк):

```
vagrant@vagrant:~$ ps ajf
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
 1315  1316  1316  1316 pts/0     1502 Ss    1000   0:00 -bash
 1316  1502  1502  1316 pts/0     1502 S+       0   0:00  \_ sudo ./mysleep 100
 1502  1503  1502  1316 pts/0     1502 S+       0   0:00      \_ ./mysleep 100
```

UID, равный 0, означает, что как процесс `sudo`, так и процесс `mysleep` выполняются под UID суперпользователя. Попробуем теперь установить бит `setuid`:

```
vagrant@vagrant:~$ chmod +s mysleep
vagrant@vagrant:~$ ls -l mysleep
-rwsr-sr-x 1 vagrant vagrant 35000 Oct 17 08:49 mysleep
```

Снова выполните `sudo ./mysleep 100` и опять посмотрите в другом терминале на запущенные процессы:

```
vagrant@vagrant:~$ ps ajf
PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
1315  1316  1316  1316 pts/0     1507 Ss    1000   0:00 -bash
1316  1507  1507  1316 pts/0     1507 S+       0   0:00  \_ sudo ./mysleep 100
1507  1508  1507  1316 pts/0     1507 S+    1000   0:00      \_ ./mysleep 100
```

Процесс `sudo` снова выполняется от имени суперпользователя, а процесс `mysleep` унаследовал UID от владельца файла.

Обычно с помощью этого бита программе предоставляются полномочия, необходимые ей, но недоступные обычным пользователям. Классический пример — команда `ping`, которой требуется право на открытие сетевых сокетов прямого доступа для отправки сообщений ping. (Для предоставления этого права доступа используется механизм `capability`, который мы рассмотрим в разделе «Привилегии Linux» на с. 44.) Администратор может не возражать против того, чтобы его пользователи применяли утилиту `ping`, но отнюдь не разрешать им открывать сокеты прямого доступа для других целей. Поэтому для исполняемого файла `ping` обычно сразу устанавливается бит `setuid` и владельцем его является суперпользователь, так что команда `ping` может использовать полномочия, обычно доступные лишь суперпользователю.

Я очень осторожно выбирала слова для предыдущего предложения. Как вы увидите далее в этом разделе, в действительности команда `ping` прилагает немало усилий, лишь бы не выполняться от имени суперпользователя. Но прежде, чем обсудить это, посмотрим на бит `setuid` в действии.

Можно поэкспериментировать с правами, необходимыми для эффективной работы `ping`, создав собственную копию от имени несуперпользователя.

На самом деле можно опрашивать с помощью `ping` и несуществующий адрес; нас интересует только то, достаточно ли у `ping` прав для открытия сетевого сокета прямого доступа. Убедитесь, что можете выполнить команду `ping`, вот так:

```
vagrant@vagrant:~$ ping 10.0.0.1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
^C
--- 10.0.0.1 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1017ms
```

Проверив возможность выполнения исполняемого файла `ping` от имени непривилегированного пользователя, создайте его копию и проверьте, можете ли выполнить ее:

```
vagrant@vagrant:~$ ls -l `which ping`
-rwsr-xr-x 1 root root 64424 Jun 28 11:05 /bin/ping
vagrant@vagrant:~$ cp /bin/ping ./myping
vagrant@vagrant:~$ ls -l ./myping
-rwxr-xr-x 1 vagrant vagrant 64424 Nov 24 18:51 ./myping
vagrant@vagrant:~$ ./myping 10.0.0.1
ping: socket: Operation not permitted
```

При копировании исполняемого файла атрибуты принадлежности файла устанавливаются в соответствии с вашим ID пользователя и бит `setuid` не переносится. У `myping`, запущенного от имени обычного пользователя, недостаточно полномочий для открытия сокета прямого доступа. Внимательно изучив биты полномочий, вы увидите бит `s` (`setuid`) вместо обычного `x` у исходного исполняемого файла `ping`.

Можете попробовать поменять владельца данного файла на `root` (для этого вам понадобится команда `sudo`), но если его не запустить от имени `root`, то полномочий все равно будет недостаточно:

```
vagrant@vagrant:~$ sudo chown root ./myping
vagrant@vagrant:~$ ls -l ./myping
-rwxr-xr-x 1 root vagrant 64424 Nov 24 18:55 ./myping
vagrant@vagrant:~$ ./myping 10.0.0.1
ping: socket: Operation not permitted
vagrant@vagrant:~$ sudo ./myping 10.0.0.1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
^C
--- 10.0.0.1 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1012ms
```

Теперь установите бит `setuid` для исполняемого файла и попробуйте снова:

```
vagrant@vagrant:~$ sudo chmod +s ./myping
vagrant@vagrant:~$ ls -l ./myping
-rwsr-sr-x 1 root vagrant 64424 Nov 24 18:55 ./myping
```

```
vagrant@vagrant:~$ ./myping 10.0.0.1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
^C
--- 10.0.0.1 ping statistics ---
3 packets transmitted, 0 received, 100% packet loss, time 2052ms
```

Как вы скоро увидите в разделе «Привилегии Linux» на с. 44, существует еще один способ предоставить `myping` полномочия, достаточные для открытия сокета, не предоставляя ему всех полномочий суперпользователя.

Теперь наша копия `ping` работает благодаря биту `setuid`, который позволяет ей работать от имени суперпользователя. Но если вы откроете второй терминал и посмотрите на соответствующий процесс с помощью `ps`, то будете весьма удивлены:

```
vagrant@vagrant:~$ ps uf -C myping
USER       PID %CPU %MEM  VSZ   RSS  TTY   STAT START  TIME COMMAND
vagrant  5154  0.0  0.0  18512 2484 pts/1 S+   00:33  0:00 ./myping localhost
```

Как видите, процесс *не* выполняется от имени `root`, несмотря на установленный бит `setuid` и то, что владельцем файла является `root`. Что произошло? Дело вот в чем: в современных версиях утилиты `ping` исполняемый файл запускается от имени `root`, после чего явным образом задает только нужные ему привилегии и сбрасывает свой UID в UID исходного пользователя. Именно это я имела в виду, когда говорила о прилагаемых `ping` усилиях.

Если вы хотите изучить сами все это более подробно, то можете воспользоваться strace для просмотра системных вызовов исполняемого файла ping (или myping). Найдите идентификатор процесса вашей командной оболочки, после чего в другом терминале, запущенном от имени суперпользователя, выполните strace -f -p <ID процесса командной оболочки> для трассировки всех системных вызовов, выполненных из этой командной оболочки, включая все запущенные в ней исполняемые файлы. Найдите системный вызов setuid(), сбрасывающий идентификатор пользователя. Вы увидите, что это происходит вскоре после нескольких системных вызовов setcap(), задающих привилегии, необходимые данному потоку выполнения.

Не все исполняемые файлы сбрасывают идентификатор пользователя подобным образом. Можете задействовать копию исполняемого файла команды `sleep`, упомянутой ранее в этой главе, чтобы посмотреть на более традицион-

ное поведение `setuid`. Поменяйте владельца файла на `root`, установите бит `setuid` (он сбрасывается при смене владельца файла), после чего запустите от имени несуперпользователя:

```
vagrant@vagrant:~$ sudo chown root mysleep
vagrant@vagrant:~$ sudo chmod +s mysleep
vagrant@vagrant:~$ ls -l ./mysleep
-rwsr-sr-x 1 root vagrant 35000 Dec  2 00:36 ./mysleep
vagrant@vagrant:~$ ./mysleep 100
```

Можете применить команду `ps` в другом терминале, чтобы убедиться в выполнении этого процесса с UID суперпользователя:

```
vagrant@vagrant:~$ ps uf -C mysleep
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root      6646  0.0  0.0   7468   764 pts/2    S+   00:38   0:00 ./mysleep 100
```

Проведя эти эксперименты с битом `setuid`, мы можем приступить к оценке его потенциального влияния на безопасность.

Влияние бита setuid на безопасность. Представьте, что получится, если установить бит `setuid`, скажем, для исполняемого файла `bash`. Все выполняющие его пользователи окажутся в командной оболочке, работающей от имени суперпользователя. На практике не все так просто, конечно, поскольку большинство командных оболочек ведут себя подобно `ping` и сбрасывают идентификатор пользователя, чтобы его нельзя было применить для такого очевидного повышения полномочий. Но можно очень легко написать собственную программу, которая бы устанавливала себе бит `setuid`, после чего, перейдя в статус суперпользователя, запускала командную оболочку (https://oreil.ly/viKwm).

А поскольку бит `setuid` — опасный путь к повышению полномочий, некоторые анализаторы образов контейнеров (обсуждаемые в главе 7) сообщают о наличии файлов с установленным битом `setuid`. Можно также запретить его использование, прибегнув к флагу `--no-new-privileges` команды `docker run`.

Бит `setuid` ведет свою историю еще с тех времен, когда механизм полномочий был гораздо более простым: у процесса либо были полномочия суперпользователя, либо нет. Бит служил механизмом предоставления дополнительных полномочий несуперпользователям. В версии 2.2 ядра Linux появилась возможность более детально контролировать эти дополнительные полномочия с помощью *привилегий* (capabilities).

Привилегии Linux

В нынешнем ядре Linux существует более 30 привилегий. Они назначаются потокам выполнения и определяют, может ли данный поток производить некие действия. Например, чтобы выполнять привязку к какому-либо порту с номером меньше 1024, потоку требуется привилегия `CAP_NET_BIND_SERVICE`. Привилегия `CAP_SYS_BOOT` предназначена для того, чтобы указать, какие исполняемые файлы должны иметь право на перезагрузку системы. Для загрузки и выгрузки модулей ядра служит привилегия `CAP_SYS_MODULE`.

Я уже упоминала ранее, что утилита `ping` выполняется от имени суперпользователя ровно столько времени, сколько ей нужно для того, чтобы предоставить себе привилегию на открытие потоком выполнения сетевого сокета прямого доступа, а именно привилегии `CAP_NET_RAW`.

Подробную информацию о привилегиях можно получить на машине под управлением Linux с помощью команды man capabilities.

Посмотреть список привилегий процесса позволяет команда `getpcaps`. Например, у процессов, запущенных несуперпользователями, обычно привилегий нет:

```
vagrant@vagrant:~$ ps
  PID TTY          TIME CMD
22355 pts/0    00:00:00 bash
25058 pts/0    00:00:00 ps
vagrant@vagrant:~$ getpcaps 22355
Capabilities for '22355': =
```

Если же запустить процесс от имени `root`, то все сразу меняется:

```
vagrant@vagrant:~$ sudo bash
root@vagrant:~# ps
  PID TTY          TIME CMD
25061 pts/0    00:00:00 sudo
25062 pts/0    00:00:00 bash
25070 pts/0    00:00:00 ps
root@vagrant:~# getpcaps 25062
Capabilities for '25062': = cap_chown,cap_dac_override,cap_dac_read_search,
cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_setpcap
cap_linux_immutable,cap_net_bind_service,cap_net_broadcast,cap_net_admin,
cap_net_raw,cap_ipc_lock,cap_ipc_owner,cap_sys_module,cap_sys_rawio,
```

```
cap_sys_chroot,cap_sys_ptrace,cap_sys_pacct,cap_sys_admin,cap_sys_boot,
cap_sys_nice,cap_sys_resource,cap_sys_time,cap_sys_tty_config,cap_mknod,
cap_lease,cap_audit_write,cap_audit_control,cap_setfcap,cap_mac_override
cap_mac_admin,cap_syslog,cap_wake_alarm,cap_block_suspend,cap_audit_read+ep
```

Можно назначать привилегии непосредственно файлам. Ранее мы видели, что копию `ping` нельзя было запустить из-под несуперпользователя без установленного бита `setuid`. Есть и другой подход: назначать необходимые привилегии непосредственно исполняемому файлу. Возьмите копию исполняемого файла `ping` и убедитесь, что она имеет обычные права (бит `setuid` не установлен). Открывать сокеты она не может:

```
vagrant@vagrant:~$ cp /bin/ping ./myping
vagrant@vagrant:~$ ls -l myping
-rwxr-xr-x 1 vagrant vagrant 64424 Feb 12 18:18 myping
vagrant@vagrant:~$ ./myping 10.0.0.1
ping: socket: Operation not permitted
```

С помощью команды `setcap` добавьте для этого файла привилегию `CAP_NET_RAW`, которая позволит ему открывать сетевые сокеты прямого доступа. Смена привилегий требует полномочий суперпользователя. Точнее, необходима только привилегия `CAP_SETFCAP`, автоматически предоставляемая пользователю `root`:

```
vagrant@vagrant:~$ setcap 'cap_net_raw+p' ./myping
unable to set CAP_SETFCAP effective capability: Operation not permitted
vagrant@vagrant:~$ sudo setcap 'cap_net_raw+p' ./myping
```

На выводимых `ls` полномочиях это никак не отразится, но можно просмотреть привилегии с помощью `getcap`:

```
vagrant@vagrant:~$ ls -l myping
-rwxr-xr-x 1 vagrant vagrant 64424 Feb 12 18:18 myping
vagrant@vagrant:~$ getcap ./myping
./myping = cap_net_raw+p
```

Благодаря этой привилегии наша копия `ping` может делать все, что нужно:

```
vagrant@vagrant:~$ ./myping 10.0.0.1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
^C
```

Более детальное обсуждение влияния друг на друга полномочий файлов и процессов можно найти в сообщении Эдриена Моата (Adrian Mouat) «Привилегии Linux на практике» (Linux capabilities in practice) (https://oreil.ly/DE8e-).

Исходя из принципа минимума полномочий, имеет смысл предоставлять процессам только необходимые для их работы привилегии. При использовании контейнера можно управлять разрешаемыми привилегиями, как вы увидите в главе 8.

Теперь, познакомившись с основными идеями прав доступа и полномочий в Linux, мы можем обсудить понятие повышения полномочий.

Повышение полномочий

Термин «повышение полномочий» означает выход за пределы имеющихся полномочий для выполнения действий, которые не были разрешены изначально. Для повышения полномочий злоумышленник может воспользоваться, например, системной уязвимостью или неудачными настройками и предоставить себе дополнительные права.

Зачастую злоумышленники начинают работу в качестве непривилегированных пользователей и пытаются получить на машине полномочия суперпользователя. Распространенный метод повышения полномочий — найти запущенную от имени `root` программу и задействовать ее известные уязвимости. Так, программное обеспечение веб-сервера может включать уязвимость, позволяющую злоумышленникам удаленно выполнять код, например уязвимость Struts (https://oreil.ly/ydu-a). Если веб-сервер запущен от имени суперпользователя, то весь код, удаленно выполняемый злоумышленником, будет запущен с полномочиями суперпользователя. Поэтому программное обеспечение желательно запускать от имени непривилегированных пользователей, насколько это возможно.

Как вы узнаете далее, по умолчанию *контейнеры запускаются от имени суперпользователя*. Это значит, запущенные в контейнерах приложения с намного большей вероятностью, чем в обычной машине под управлением Linux, будут выполняться от имени суперпользователя. Злоумышленнику, захватившему контроль над процессом внутри контейнера, конечно, все равно придется найти способ выйти за рамки контейнера, но затем он сразу получит права суперпользователя на хосте, не прибегая к какому-либо дальнейшему повышению полномочий. Все это более подробно обсуждается в главе 9.

Но даже если контейнер запущен не от имени `root`, все равно остается возможность повысить полномочия на основе механизмов прав доступа Linux, которые обсуждались выше в этой главе:

- образов контейнеров, содержащих исполняемые файлы с установленным битом `setuid`;
- предоставления дополнительных привилегий контейнеру, запущенному от имени несуперпользователя.

Далее в книге вы узнаете, с помощью каких подходов можно уменьшить последствия этих проблем.

Резюме

В текущей главе вы узнали (или вспомнили) об основных механизмах Linux, необходимых для понимания дальнейших глав книги. Описанные механизмы играют множество важных ролей в безопасности; все средства безопасности контейнеров, представленные далее, основаны на этих базовых элементах.

Разобравшись с основными средствами безопасности Linux, мы можем обсудить механизмы, лежащие в основе контейнеров, чтобы вам стало понятно, почему `root` на хосте и в контейнере, по существу, является одним и тем же.

ГЛАВА 3

Контрольные группы

В этой главе вам предстоит познакомиться с одним из главных «кирпичиков» создания контейнеров: *контрольными группами* (control groups, cgroups).

Контрольные группы служат для ограничения доступных группе процессов ресурсов, например оперативной памяти, времени CPU и ресурсов сетевого ввода/вывода. С точки зрения безопасности хорошо настроенная контрольная группа позволяет гарантировать, что процесс не сможет влиять на поведение других процессов, захватывая все ресурсы — например, все ресурсы CPU или память, — не оставляя ничего остальным приложениям. Существует также контрольная группа pid, предназначенная для ограничения общего количества процессов в контрольной группе, тем самым сводя на нет результативность так называемой fork-бомбы.

Fork-бомба — это программа, создающая процессы, которые, в свою очередь, тоже создают процессы, что приводит к экспоненциальному росту объема используемых ресурсов и в итоге выводит машину из строя. В видео доклада, сделанного мной несколько лет назад (https://oreil.ly/Us75y), демонстрируется, как с помощью контрольной группы pid можно ограничить эффект fork-бомбы.

В главе 4 во всех подробностях будет показано, что контейнеры выполняются как обычные процессы Linux, вследствие чего можно использовать контрольные группы для ограничения ресурсов, доступных каждому из контейнеров. Посмотрим на то, как устроены контрольные группы.

Иерархии контрольных групп

Для каждого из типов ресурсов есть своя иерархия контрольных групп, управляемая контроллером контрольных групп. Всякий процесс Linux является членом одной контрольной группы для каждого типа ресурсов.

Любой создаваемый процесс наследует контрольные группы своего родительского процесса.

Ядро Linux обменивается информацией о контрольных группах через набор псевдофайловых систем, располагающихся обычно по пути `/sys/fs/cgroup`. Чтобы просмотреть все типы контрольных групп в системе, можно вывести содержимое этого каталога:

```
root@vagrant:/sys/fs/cgroup$ ls
blkio    cpu,cpuacct  freezer  net_cls           perf_event  systemd
cpu      cpuset       hugetlb  net_cls,net_prio  pids        unified
cpuacct  devices      memory   net_prio          rdma
```

Работа с контрольными группами требует чтения и записи в файлы и каталоги в этой иерархии. Посмотрим на контрольную группу memory в качестве примера:

```
root@vagrant:/sys/fs/cgroup$ ls memory/
cgroup.clone_children               memory.limit_in_bytes
cgroup.event_control                memory.max_usage_in_bytes
cgroup.procs                        memory.move_charge_at_immigrate
cgroup.sane_behavior                memory.numa_stat
init.scope                          memory.oom_control
memory.failcnt                      memory.pressure_level
memory.force_empty                  memory.soft_limit_in_bytes
memory.kmem.failcnt                 memory.stat
memory.kmem.limit_in_bytes          memory.swappiness
memory.kmem.max_usage_in_bytes      memory.usage_in_bytes
memory.kmem.slabinfo                memory.use_hierarchy
memory.kmem.tcp.failcnt             notify_on_release
memory.kmem.tcp.limit_in_bytes      release_agent
memory.kmem.tcp.max_usage_in_bytes  system.slice
memory.kmem.tcp.usage_in_bytes      tasks
memory.kmem.usage_in_bytes          user.slice
```

Управлять контрольной группой можно с помощью записи в одни из этих файлов, в то время как другие содержат информацию о состоянии группы, записанную ядром Linux. Не изучив документацию (https://oreil.ly/LQxKB), легко различить параметры и информационные файлы не получится, но можно, наверное, угадать, для чего предназначены некоторые из них, просто по названиям. Например, файл `memory.limit_in_bytes` содержит перезаписываемое значение, задающее объем памяти, доступной процессам группы; `memory.max_usage_in_bytes` указывает максимальное количество использовавшейся в группе памяти.

Этот каталог группы memory находится на верхнем уровне иерархии и в отсутствие других контрольных групп содержит информацию о памяти для

всех запущенных процессов. Чтобы ограничить объем памяти, доступной процессу, необходимо создать новую контрольную группу и прикрепить к ней данный процесс.

Создание контрольных групп

Создание подкаталога внутри каталога группы memory приводит к созданию контрольной группы, а ядро Linux автоматически заполняет его различными файлами для параметров и статистики этой группы.

```
root@vagrant:/sys/fs/cgroup$ mkdir memory/liz
root@vagrant:/sys/fs/cgroup$ ls memory/liz/
cgroup.clone_children               memory.limit_in_bytes
cgroup.event_control                memory.max_usage_in_bytes
cgroup.procs                        memory.move_charge_at_immigrate
memory.failcnt                      memory.numa_stat
memory.force_empty                  memory.oom_control
memory.kmem.failcnt                 memory.pressure_level
memory.kmem.limit_in_bytes          memory.soft_limit_in_bytes
memory.kmem.max_usage_in_bytes      memory.stat
memory.kmem.slabinfo                memory.swappiness
memory.kmem.tcp.failcnt             memory.usage_in_bytes
memory.kmem.tcp.limit_in_bytes      memory.use_hierarchy
memory.kmem.tcp.max_usage_in_bytes  notify_on_release
memory.kmem.tcp.usage_in_bytes      tasks
memory.kmem.usage_in_bytes
```

Подробное описание смысла всех этих файлов выходит за рамки данной книги, но в некоторых из них содержатся параметры для управления ограничениями ресурсов контрольной группы, а в других — показатели текущего использования ресурсов в данной контрольной группе. Можно предположить, например, что файл `memory.usage_in_bytes` указывает, сколько памяти задействует группа в настоящий момент. А максимально доступный группе объем памяти задается файлом `memory.limit_in_bytes`.

При запуске контейнера среда выполнения создает для него новые контрольные группы. Утилита `lscgroup` (которую на Ubuntu можно установить с пакетом `cgroup-tools`) позволяет просмотреть эти контрольные группы на хост-компьютере. Поскольку их довольно много, посмотрим только на различия в контрольной группе memory до и после запуска нового контейнера с помощью команды `runc`. Сделайте снимок файловой системы контрольной группы memory:

```
root@vagrant:~$ lscgroup memory:/ > before.memory
```

Запустите в другом терминале контейнер:

```
vagrant@vagrant:alpine-bundle$ sudo runc run sh
/ $
```

Сделайте еще один снимок файловой системы и сравните их:

```
root@vagrant:~$ lscgroup memory:/ > after.memory
root@vagrant:~$ diff before.memory after.memory
4a5
> memory:/user.slice/user-1000.slice/session-43.scope/sh
```

Иерархия отсчитывается от корня контрольной группы memory, расположенного обычно по пути `/sys/fs/cgroup/memory`. Во время работы контейнера можно просмотреть эту контрольную группу с хоста:

```
root@vagrant:/sys/fs/cgroup/memory$ ls user.slice/user-1000.slice/session-43.
scope/sh/
cgroup.clone_children               memory.limit_in_bytes
cgroup.event_control                memory.max_usage_in_bytes
cgroup.procs                        memory.move_charge_at_immigrate
memory.failcnt                      memory.numa_stat
memory.force_empty                  memory.oom_control
memory.kmem.failcnt                 memory.pressure_level
memory.kmem.limit_in_bytes          memory.soft_limit_in_bytes
memory.kmem.max_usage_in_bytes      memory.stat
memory.kmem.slabinfo                memory.swappiness
memory.kmem.tcp.failcnt             memory.usage_in_bytes
memory.kmem.tcp.limit_in_bytes      memory.use_hierarchy
memory.kmem.tcp.max_usage_in_bytes  notify_on_release
memory.kmem.tcp.usage_in_bytes      tasks
memory.kmem.usage_in_bytes
```

Изнутри контейнера можно увидеть список его собственных контрольных групп в каталоге /proc:

```
/ $ cat /proc/$$/cgroup
12:cpu,cpuacct:/sh
11:cpuset:/sh
10:hugetlb:/sh
9:blkio:/sh
8:memory:/user.slice/user-1000.slice/session-43.scope/sh
7:pids:/user.slice/user-1000.slice/session-43.scope/sh
6:freezer:/sh
5:devices:/user.slice/sh
4:net_cls,net_prio:/sh
3:rdma:/
2:perf_event:/sh
1:name=systemd:/user.slice/user-1000.slice/session-43.scope/sh
0::/user.slice/user-1000.slice/session-43.scope
```

Обратите внимание: контрольная группа memory — точно такая же, как если смотреть с хоста. Менять параметры в имеющихся контрольных группах можно путем записи в соответствующие файлы.

Вероятно, вас заинтриговали части `user.slice/user-1000` в предшествующем списке контрольных групп. Они связаны с подсистемой `systemd`, автоматически создающей некоторые иерархии контрольных групп для управления ресурсами. Если вам интересно, то в документации Red Hat можно найти подробное описание (https://oreil.ly/i4OWd).

Установка ограничений на ресурсы

Объем оперативной памяти, доступный контрольной группе для использования, можно узнать, заглянув в ее файл `memory.limit_in_bytes`:

```
root@vagrant:/sys/fs/cgroup/memory$ cat user.slice/user-1000.slice/session-43.sco
pe/sh/memory.limit_in_bytes
9223372036854771712
```

По умолчанию объем памяти не ограничен, так что это гигантское число соответствует всей памяти, доступной виртуальной машине, на которой я генерировала данный пример.

Если не ограничить объем памяти, доступный процессу, то он может оставить без памяти другие процессы на том же хост-компьютере. Это может происходить либо непреднамеренно, в результате утечки памяти в приложении, либо в результате атаки на истощение ресурсов, при которой злоумышленник специально применяет утечку памяти, чтобы использовать как можно больше памяти. Установка ограничений памяти и прочих ресурсов, доступных процессу, позволяет уменьшить последствия подобных атак и гарантировать, что прочие процессы смогут продолжать нормально работать.

Чтобы ограничить память, выделяемую контрольной группе при создании контейнера, можно изменить файл `config.json` в комплекте `runc`. Ограничения контрольной группы задаются в разделе `linux:resources` этого файла:

```
  "linux": {
    "resources": {
      "memory": {
        "limit": 1000000
      },
      ...
    }
  }
```

Чтобы учесть эти новые настройки, необходимо остановить работу контейнера и перезапустить команду `runc`. Если использовать то же самое название контейнера, то название контрольной группы тоже останется прежним (можете проверить это, выполнив `cat /proc/$$/cgroup` внутри контейнера). После этого значение параметра `memory.limit_in_bytes` будет соответствовать заданному ограничению, вероятно округленному до килобайтов:

```
root@vagrant:/sys/fs/cgroup/memory$ cat user.slice/user-1000.slice/session-43.sco
pe/sh/memory.limit_in_bytes
999424
```

Именно `runc` изменила это значение. Чтобы установить ограничение для контрольной группы, достаточно записать значение в файл, соответствующий ограничиваемому параметру.

Теперь вы знаете, как задавать ограничения. Осталась последняя часть пазла под названием «контрольные группы» — распределение процессов по контрольным группам.

Приписываем процесс к контрольной группе

Как и при установке ограничений на ресурсы, чтобы приписать процесс к определенной контрольной группе, необходимо просто записать его идентификатор в файл `cgroup.procs` этой группы. В следующем примере 29903 — идентификатор процесса командной оболочки:

```
root@vagrant:/sys/fs/cgroup/memory/liz$ echo 100000 > memory.limit_in_bytes
root@vagrant:/sys/fs/cgroup/memory/liz$ cat memory.limit_in_bytes
98304
root@vagrant:/sys/fs/cgroup/memory/liz$ echo 29903 > cgroup.procs
root@vagrant:/sys/fs/cgroup/memory/liz$ cat cgroup.procs
29903
root@vagrant:/sys/fs/cgroup/memory/liz$ cat /proc/29903/cgroup | grep memory
8:memory:/liz
```

Командная оболочка теперь относится к этой контрольной группе, а объем доступной ей памяти ограничен значением чуть меньше 100 Кбайт. Не так уж много, поэтому даже попытка выполнить `ls` в данной командной оболочке приведет к превышению этого ограничения:

```
$ ls
Killed
```

При попытке превысить ограничение на память выполнение процесса прекращается.

Контрольные группы в Docker

Выше было показано управление контрольными группами с помощью изменения файлов для конкретных типов ресурсов в файловой системе cgroup. Это несложно увидеть в действии в Docker.

Чтобы следить за ходом этих примеров, вам понадобится Docker, работающий непосредственно на (виртуальной) машине Linux. Docker для Mac/Windows работает внутри виртуальной машины. Это значит (как вы увидите в главе 5), что приведенные примеры в нем работать не будут, поскольку демон Docker и контейнеры выполняются на отдельном ядре в этой виртуальной машине.

Docker автоматически создает собственные контрольные группы всех типов. Просмотреть их список можно, выбрав каталоги docker в иерархии контрольных групп:

```
root@vagrant:/sys/fs/cgroup$ ls */docker | grep docker
blkio/docker:
cpuacct/docker:
cpu,cpuacct/docker:
cpu/docker:
cpuset/docker:
devices/docker:
freezer/docker:
hugetlb/docker:
memory/docker:
net_cls/docker:
net_cls,net_prio/docker:
net_prio/docker:
perf_event/docker:
pids/docker:
systemd/docker:
```

При запуске контейнера автоматически создается еще один набор контрольных групп внутри контрольных групп `docker`. Создайте контейнер и задайте для него ограничения памяти, который мы увидим позже внутри контрольной группы memory. Этот пример запускает контейнер в фоновом режиме, который бездействует в течение времени, достаточного для просмотра его контрольных групп:

```
root@vagrant:~$ docker run --rm --memory 100M -d alpine sleep 10000
68fb008c5fd3f9067e1aa245b4522a9f3675720d8953371ecfcf2e9faf91b8a0
```

Если посмотреть иерархию контрольных групп, то вы увидите созданные для этого контейнера новые контрольные группы с его идентификатором в качестве названия контрольной группы:

```
root@vagrant:/sys/fs/cgroup$ ls memory/docker/
68fb008c5fd3f9067e1aa245b4522a9f3675720d8953371ecfcf2e9faf91b8a0
cgroup.clone_children
cgroup.event_control
cgroup.procs
memory.failcnt
memory.force_empty
memory.kmem.failcnt
memory.kmem.limit_in_bytes
memory.kmem.max_usage_in_bytes
...
```

Проверяем ограничение на память в байтах внутри контрольной группы memory:

```
root@vagrant:/sys/fs/cgroup$ cat memory/docker/68fb008c5fd3f9067e1aa245b4522a9f36
75720d8953371ecfcf2e9faf91b8a0/memory.limit_in_bytes
104857600
```

Можно также убедиться, что наш процесс бездействия является членом этой контрольной группы:

```
root@vagrant:/sys/fs/cgroup$ cat memory/docker/68fb008c5fd3f9067e1aa245b4522a9f36
75720d8953371ecfcf2e9faf91b8a0/cgroup.procs
19824
root@vagrant:/sys/fs/cgroup$ ps -eaf | grep sleep
root     19824 19789  0 18:22 ?        00:00:00 sleep 10000
root     20486 18862  0 18:28 pts/1    00:00:00 grep --color=auto sleep
```

Контрольные группы версии 2

В ядре Linux в 2016 году[1] появилась версия 2 контрольных групп, и первым дистрибутивом Linux, где она используется по умолчанию, стала Fedora в середине 2019-го. Впрочем, на момент написания данной книги большинство популярных реализаций сред выполнения контейнеров использует версию 1 контрольных групп и не поддерживает версию 2 (хотя в этом направлении и ведутся определенные работы, кратко описанные Акихиро Суда (Akihiro Suda) в публикации в блоге) (https://oreil.ly/pDTZ6).

[1] В упомянутом ниже сообщении Акихиро Суда говорится, что в 2014-м.

Основное различие между версиями состоит в том, что в версии 2 процесс не может сочетать различные группы для различных контроллеров. В версии 1 процесс может входить в `/sys/fs/cgroup/memory/mygroup` и `/sys/fs/cgroup/pids/yourgroup`. В версии 2 все проще: процесс входит в `/sys/fs/cgroup/ourgroup` и управляется всеми контроллерами `ourgroup`.

Версия 2 контрольных групп также лучше поддерживает контейнеры, не требующие полномочий суперпользователя, что позволяет ограничивать ресурсы и для них. Мы поговорим об этом в подразделе «Контейнеры, не требующие полномочий суперпользователя» на с. 152.

Резюме

Контрольные группы ограничивают ресурсы, доступные различным Linux-процессам. Чтобы воспользоваться возможностями контрольных групп, не обязательно применять контейнеры, но Docker и прочие среды выполнения контейнеров предоставляют удобный интерфейс для них: при запуске контейнера очень удобно задавать ограничения ресурсов, а контрольные группы обеспечивают проведение этих ограничений в жизнь.

Ограничение ресурсов обеспечивает защиту от класса атак, нацеленных на нарушение работы развернутой системы путем потребления чрезмерного количества ресурсов и лишения других приложений, таким образом, законного доступа к этим ресурсам. Рекомендую задавать ограничения на память и ресурсы CPU при запуске контейнеризованных приложений.

Теперь, разобравшись с ограничением ресурсов в контейнерах, вы должны быть готовы узнать об остальных составных частях пазла контейнеров: о пространствах имен и изменении корневого каталога. О них я и расскажу в главе 4.

ГЛАВА 4

Изоляция контейнеров

Именно в этой главе вы узнаете, как на самом деле работают контейнеры! Жизненно важно понимать, насколько они изолированы друг от друга и от хост-компьютера. Вы сами сможете оценить прочность границ безопасности, окружающих контейнеры.

Если вы когда-нибудь выполняли команду `docker exec <образ> bash`, то наверняка знаете, что контейнер изнутри очень похож на виртуальную машину. Если при наличии доступа к контейнеру через командную оболочку выполнить команду `ps`, то будут видны только работающие внутри него процессы. У контейнера свой сетевой стек и, на первый взгляд, собственная файловая система с корневым каталогом, никак не связанным с корневым каталогом хост-компьютера. Можно выполнять контейнеры с ограничением ресурсов, например объема оперативной памяти или доступных ресурсов CPU. Сделать все это позволяют возможности Linux, которые мы более подробно обсудим в данной главе.

Однако, как бы ни были похожи на первый взгляд эти два типа изоляции, важно отдавать себе отчет, что контейнеры — это *не* виртуальные машины, и в главе 5 мы обсудим их различия. По моему опыту, без полноценного осознания их различий совершенно невозможно оценить степень эффективности обычных мер безопасности контейнеров и понять, где необходимы средства, предназначенные специально для них.

Вы увидите, что для контейнеров используются такие конструктивные элементы Linux, как пространства имен и `chroot`, а также контрольные группы, которые мы обсуждали в главе 3. Имея понимание соответствующих концепций, вы узнаете, насколько хорошо защищены ваши приложения, запущенные внутри контейнеров.

И хотя общие идеи этих элементов достаточно просты, они могут взаимодействовать с прочей функциональностью ядра Linux довольно запутанным образом. Именно тонкие нюансы взаимодействия пространств имен, привилегий и файловых систем играют главную роль в уязвимостях выхода за рамки контейнера (например, CVE-2019-5736 (https://oreil.ly/NtcRv) — серьезной уязвимости, обнаруженной как в `runc`, так и в `LXC`).

Пространства имен Linux

Контрольные группы позволяют контролировать ресурсы, доступные процессу, а *пространства имен* (namespaces) — ресурсы, которые он видит. Помещение процесса в пространство имен позволяет ограничить видимые ему ресурсы.

Пространства имен ведут свою историю от операционной системы Plan 9 (https://oreil.ly/BCi9W). В те времена у большинства операционных систем было единое пространство имен файлов. Операционные системы Unix позволили монтировать файловые системы, но лишь в единое для всей системы пространство имен файлов. В Plan 9 все процессы входили в одну из групп процессов со своим пространством имен — иерархией файлов (и файлоподобных объектов), видимых данной группе процессов. Каждый процесс может монтировать собственный набор файловых систем, невидимых друг другу.

Первое пространство имен появилось в версии 2.4.19 ядра Linux еще в 2002-м, а именно пространство имен монтирования с функциональностью, аналогичной операционной системе Plan 9. На сегодняшний день Linux поддерживает несколько видов пространств имен:

- систему разделения времени Unix (Unix Timesharing System, UTS) — звучит довольно сложно, но фактически это пространство имен служит для изоляции хост-имени и доменного имени системы с точки зрения процесса;
- идентификаторы процессов (Process IDs, PID);
- точки монтирования;
- сеть;
- идентификаторы пользователей и групп;
- обмен информацией между процессами (inter-process communication, IPC)[1];
- контрольные группы.

Вероятно, в будущих версиях ядра Linux появятся пространства имен и для других ресурсов. Например, ходят дискуссии (https://oreil.ly/NZqb-) о пространстве имен для времени.

Процесс всегда относится ровно к одному пространству имен каждого типа. При запуске системы Linux в ней есть одно пространство имен каждого типа,

[1] В русскоязычной литературе нередко можно также встретить термин «межпроцессное взаимодействие».

но, как вы увидите, можно создавать пространства имен и распределять по ним процессы. Просмотреть пространства имен на машине можно с помощью команды `lsns`:

```
vagrant@myhost:~$ lsns
        NS TYPE   NPROCS   PID USER    COMMAND
4026531835 cgroup      3 28459 vagrant /lib/systemd/systemd --user
4026531836 pid         3 28459 vagrant /lib/systemd/systemd --user
4026531837 user        3 28459 vagrant /lib/systemd/systemd --user
4026531838 uts         3 28459 vagrant /lib/systemd/systemd --user
4026531839 ipc         3 28459 vagrant /lib/systemd/systemd --user
4026531840 mnt         3 28459 vagrant /lib/systemd/systemd --user
4026531992 net         3 28459 vagrant /lib/systemd/systemd --user
```

Все выглядит идеально, по одному пространству имен для каждого из вышеупомянутых типов. К сожалению, эта картина неполна! Как сообщает справка (man) (https://oreil.ly/nd0Eh) по команде `lsns`, «она читает информацию непосредственно из файловой системы `/proc` и для несуперпользователей может возвращать неполную информацию». Посмотрим, что она выведет при запуске от имени `root`:

```
vagrant@myhost:~$ sudo lsns
        NS TYPE   NPROCS   PID USER            COMMAND
4026531835 cgroup     93     1 root            /sbin/init
4026531836 pid        93     1 root            /sbin/init
4026531837 user       93     1 root            /sbin/init
4026531838 uts        93     1 root            /sbin/init
4026531839 ipc        93     1 root            /sbin/init
4026531840 mnt        89     1 root            /sbin/init
4026531860 mnt         1    15 root            kdevtmpfs
4026531992 net        93     1 root            /sbin/init
4026532170 mnt         1 14040 root            /lib/systemd/systemd-udevd
4026532171 mnt         1   451 systemd-network /lib/systemd/systemd-networkd
4026532190 mnt         1   617 systemd-resolve /lib/systemd/systemd-resolved
```

Пользователю `root` видны дополнительные пространства имен монтирования, кроме того, ему видно намного больше процессов, чем несуперпользователям. Это говорит о том, что запускать `lsns` необходимо от имени `root` (или задействовать `sudo`), чтобы видеть полную картину.

Теперь посмотрим, как с помощью пространств имен создать нечто, по поведению напоминающее контейнер.

В примерах из этой главы для создания контейнера используются инструкции командной оболочки Linux. Инструкции по созданию контейнера с помощью языка программирования Go можно найти по адресу https://github.com/lizrice/containers-from-scratch.

Изоляция хост-имени

Начнем с пространства имен для системы разделения времени Unix (UTS). Как уже упоминалось, она охватывает хост-имя и доменное имя. Помещение процесса в отдельное пространство имен UTS позволяет менять хост-имя для этого процесса независимо от хост-имени машины (или виртуальной машины), на которой он запущен.

Просмотреть хост-имя в терминале Linux можно следующим образом:

```
vagrant@myhost:~$ hostname
myhost
```

Большинство (возможно, все?) систем контейнеров присваивают всем контейнерам случайные ID, используемые по умолчанию в качестве хост-имен. Убедиться в этом можно, запустив контейнер и подключившись к нему через командную оболочку. Например, в Docker можно сделать следующее:

```
vagrant@myhost:~$ docker run --rm -it --name hello ubuntu bash
root@cdf75e7a6c50:/$ hostname
cdf75e7a6c50
```

Кстати, из этого примера видно, что, даже если присвоить в Docker контейнеру название (в данном случае я задала `--name hello`), оно не используется в качестве хост-имени контейнера.

Хост-имя контейнера может отличаться, поскольку Docker создает его с собственным пространством имен UTS. Тот же результат можно получить, воспользовавшись командой `unshare` для создания процесса с отдельным пространством имен UTS.

Как описано на соответствующей странице справки (которую можно просмотреть с помощью команды `man unshare`), `unshare` позволяет «запускать программы с пространствами имен, отдельными от родительского процесса». Углубимся в это описание. При «запуске программы» ядро создает новый процесс и выполняет в нем программу. Это происходит в контексте текущего — *родительского* — процесса, а новый процесс при этом называется *дочерним*. Слово unshare («не разделять») означает, что дочерний процесс получает собственное пространство имен, а не делит его с родительским.

Поэкспериментируем с этим. Вам понадобятся полномочия суперпользователя, поэтому в начале строки можно видеть `sudo`:

```
vagrant@myhost:~$ sudo unshare --uts sh
$ hostname
```

```
myhost
$ hostname experiment
$ hostname
experiment
$ exit
vagrant@myhost:~$ hostname
myhost
```

В этом примере запускается в новом процессе командная оболочка `sh` с новым пространством имен UTS. Все запускаемые в ней программы наследуют ее пространства имен. Команда `hostname` выполняется в новом пространстве имен UTS, изолированном от пространства имен UTS хост-компьютера.

Если открыть новое окно терминала на том же хосте перед командой `exit`, то можно убедиться, что хост-имя не поменялось для (виртуальной) машины в целом. Изменение хост-имени на хосте не влияет на хост-имя, используемое заключенным в новое пространство имен процессом, и наоборот.

Эта возможность — ключевой элемент работы контейнеров. Благодаря пространствам имен они получают набор ресурсов (в данном случае хост-имя), независимых от хост-компьютера и прочих контейнеров. Но процесс все равно выполняется тем же самым ядром Linux. Это имеет последствия для безопасности, которые я обсужу далее в данной главе. А пока рассмотрим еще один пример пространства имен, которое позволяет контейнерам видеть запущенные процессы со своей точки зрения.

Изоляция идентификаторов процессов

Если выполнить команду `ps` внутри контейнера Docker, то вы увидите только работающие внутри него процессы и не увидите никаких процессов, запущенных на хосте:

```
vagrant@myhost:~$ docker run --rm -it --name hello ubuntu bash
root@cdf75e7a6c50:/$ ps -eaf
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 18:41 pts/0    00:00:00 bash
root        10     1  0 18:42 pts/0    00:00:00 ps -eaf
root@cdf75e7a6c50:/$ exit
vagrant@myhost:~$
```

Это происходит благодаря пространству имен идентификаторов процессов, ограничивающему множество видимых идентификаторов процессов.

Попробуйте снова выполнить команду `unshare`, однако на сей раз указав, что вам нужно новое пространство имен PID, с помощью флага `--pid`:

```
vagrant@myhost:~$ sudo unshare --pid sh
$ whoami
root
$ whoami
sh: 2: Cannot fork
$ whoami
sh: 3: Cannot fork
$ ls
sh: 4: Cannot fork
$ exit
vagrant@myhost:~$
```

Не слишком успешно: после первой команды `whoami` не получается выполнить ни одной команды! Но в выведенных результатах есть несколько интересных артефактов.

Первый процесс под `sh` сработал нормально, но все команды после него завершились неудачей, поскольку невозможно создать ветку. Ошибка выводится в виде `<`*`команда`*`>: <`*`ID процесса`*`>: <`*`сообщение`*`>`, и, как видите, идентификатор процесса всякий раз увеличивается на единицу. Исходя из данной последовательности, можно предположить, что первая команда `whoami` выполняется в виде процесса с ID 1, поэтому пространство имен PID как-то работает, по крайней мере в том, что нумерация идентификаторов процессов была начата заново. Но пользы от него немного, раз нельзя запустить больше одного процесса!

Ключ к причине проблемы можно найти в описании флага `--fork` на странице справки для команды `unshare`: «Лучше порождать ветку заданной программы в виде дочернего процесса с помощью `unshare`, а не выполнять непосредственно. Это удобно при создании нового пространства имен PID».

Можете поэкспериментировать с этим, выполнив команду `ps`, чтобы просмотреть иерархию из другого окна терминала:

```
vagrant@myhost:~$ ps fa
  PID TTY      STAT   TIME COMMAND
...
30345 pts/0    Ss     0:00 -bash
30475 pts/0    S      0:00  \_ sudo unshare --pid sh
30476 pts/0    S      0:00      \_ sh
```

Процесс `sh` является дочерним не для `unshare`, а для процесса `sudo`.

Теперь попробуйте то же самое с параметром `--fork`:

```
vagrant@myhost:~$ sudo unshare --pid --fork sh
$ whoami
root
$ whoami
Root
```

Явный прогресс, мы теперь можем выполнить более одной команды без возникновения ошибки Cannot fork. Если посмотреть на иерархию во втором окне терминала, то станет заметным существенное отличие:

```
vagrant@myhost:~$ ps fa
  PID TTY      STAT   TIME COMMAND
...
30345 pts/0    Ss     0:00 -bash
30470 pts/0    S      0:00  \_ sudo unshare --pid --fork sh
30471 pts/0    S      0:00      \_ unshare --pid --fork sh
30472 pts/0    S      0:00          \_ sh
...
```

При использовании параметра `--fork` командная оболочка `sh` выполняется в виде процесса, дочернего для `unshare`, поэтому можно выполнять столько дочерних команд в этой командной оболочке, сколько нужно.

Результаты выполнения команды `ps` могут вас удивить, учитывая тот факт, что эта командная оболочка находится в отдельном пространстве идентификаторов процессов:

```
vagrant@myhost:~$ sudo unshare --pid --fork sh
$ ps
  PID TTY          TIME CMD
14511 pts/0    00:00:00 sudo
14512 pts/0    00:00:00 unshare
14513 pts/0    00:00:00 sh
14515 pts/0    00:00:00 ps
$ ps -eaf
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 Mar27 ?        00:00:02 /sbin/init
root         2     0  0 Mar27 ?        00:00:00 [kthreadd]
root         3     2  0 Mar27 ?        00:00:00 [ksoftirqd/0]
root         5     2  0 Mar27 ?        00:00:00 [kworker/0:0H]
...еще много строк для разных процессов...
$ exit
vagrant@myhost:~$
```

Как видите, команда `ps` по-прежнему отображает все процессы на всем хосте, хотя и работает внутри нового пространства имен идентификаторов процессов. Чтобы `ps` вела себя так, как в контейнере Docker, нового пространства

имен идентификаторов процессов недостаточно. Причина этого изложена на странице справки для команды `ps`: «Работа команды `ps` основана на чтении виртуальных файлов в `/proc`».

Посмотрим на каталог /proc и упомянутые в нем виртуальные файлы. В вашей системе список будет похожим, но не идентичным, поскольку отличается набор работающих в ней процессов:

```
vagrant@myhost:~$ ls /proc
1      14553  292    467        cmdline      modules
10     14585  3      5          consoles     mounts
1009   14586  30087  53         cpuinfo      mpt
1010   14664  30108  538        crypto       mtrr
1015   14725  30120  54         devices      net
1016   14749  30221  55         diskstats    pagetypeinfo
1017   15     30224  56         dma          partitions
1030   156    30256  57         driver       sched_debug
1034   157    30257  58         execdomains  schedstat
1037   158    30283  59         fb           scsi
1044   159    313    60         filesystems  self
1053   16     314    61         fs           slabinfo
1063   160    315    62         interrupts   softirqs
1076   161    34     63         iomem        stat
1082   17     35     64         ioports      swaps
11     18     3509   65         irq          sys
1104   19     3512   66         kallsyms     sysrq-trigger
1111   2      36     7          kcore        sysvipc
1175   20     37     72         keys         thread-self
1194   21     378    8          key-users    timer_list
12     22     385    85         kmsg         timer_stats
1207   23     392    86         kpagecgroup  tty
1211   24     399    894        kpagecount   uptime
1215   25     401    9          kpageflags   version
12426  26     403    966        loadavg      version_signature
125    263    407    acpi       locks        vmallocinfo
13     27     409    buddyinfo  mdstat       vmstat
14046  28     412    bus        meminfo      zoneinfo
14087  29     427    cgroups    misc
```

Все числовые каталоги в `/proc` соответствуют идентификаторам процессов, и внутри каждого из каталогов можно найти много интересной информации о соответствующем процессе. Например, `/proc/<pid>/exe` — символическая ссылка на исполняемый файл, запущенный в рамках конкретного процесса, как видно из данного примера:

```
vagrant@myhost:~$ ps
  PID TTY          TIME CMD
28441 pts/1    00:00:00 bash
28558 pts/1    00:00:00 ps
```

```
vagrant@myhost:~$ ls /proc/28441
attr              fdinfo      numa_maps      smaps
autogroup         gid_map     oom_adj        smaps_rollup
auxv              io          oom_score      stack
cgroup            limits      oom_score_adj  stat
clear_refs        loginuid    pagemap        statm
cmdline           map_files   patch_state    status
comm              maps        personality    syscall
coredump_filter   mem         projid_map     task
cpuset            mountinfo   root           timers
cwd               mounts      sched          timerslack_ns
environ           mountstats  schedstat      uid_map
exe               net         sessionid      wchan
fd                ns          setgroups
vagrant@myhost:~$ ls -l /proc/28441/exe
lrwxrwxrwx 1 vagrant vagrant 0 Oct 10 13:32 /proc/28441/exe -> /bin/bash
```

Вне зависимости от того, в каком пространстве имен ID процессов она выполняется, команда `ps` ищет информацию о запущенных процессах в каталоге /proc. Чтобы команда `ps` возвращала информацию только о процессах, запущенных внутри нового пространства имен, нужна отдельная копия каталога /proc, в которую ядро могло бы записывать информацию о процессах, заключенных в отдельное пространство имен. А поскольку каталог /proc располагается в корне дерева каталогов, то это означает необходимость изменения корневого каталога.

Изменение корневого каталога

Изнутри контейнера не видна вся файловая система хоста, а только ее подмножество, поскольку при создании контейнера корневой каталог изменяется.

Изменить корневой каталог в Linux можно с помощью команды `chroot`. Фактически она делает так, что корневой каталог текущего процесса начинает указывать на другую точку файловой системы. После того как команда `chroot` будет выполнена, теряется доступ ко всему, что располагается в иерархии файлов выше нового корневого каталога, поскольку подняться выше корня файловой системы невозможно, как показано на рис. 4.1.

Описание на странице справки для команды `chroot` гласит: «Выполняет *`команду`* с корневым каталогом, указывающим на *`новый_корневой_каталог`*. [...] Если команда не задана, то выполняет *`${командная_оболочка}`* `-i` (по умолчанию: */bin/sh* `-i`)».

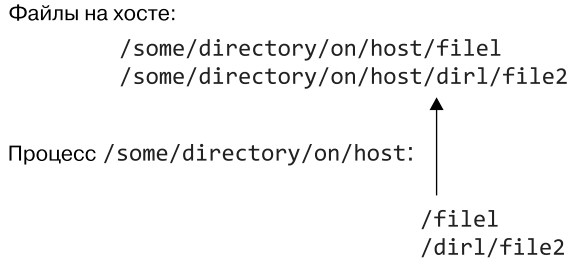


Рис. 4.1. Изменение корневого каталога, после чего процессу становится видна лишь часть файловой системы

Из этого видно, что `chroot` не просто меняет каталог, но и выполняет указанную команду, а если другая команда не задана, то возвращается к выполнению командной оболочки.

Создайте новый каталог и попробуйте заменить корневой на него с помощью команды `chroot`:

```
vagrant@myhost:~$ mkdir new_root
vagrant@myhost:~$ sudo chroot new_root
chroot: failed to run command '/bin/bash': No such file or directory
vagrant@myhost:~$ sudo chroot new_root ls
chroot: failed to run command 'ls': No such file or directory
```

Не работает! Дело в том, что внутри нового корневого каталога нет каталога bin, из-за чего невозможно запустить командную оболочку `/bin/bash`. Аналогично не удастся найти и команду `ls`. Для запуска любых команд необходимо, чтобы соответствующие файлы были доступны внутри нового корневого каталога. Именно это и происходит в «настоящем» контейнере: из образа контейнера создается экземпляр контейнера, который включает видимую контейнером файловую систему. Если внутри нее нет нужного исполняемого файла, то контейнер не сможет его найти и выполнить.

Попробуем запустить Alpine Linux в контейнере. Alpine — минималистичный дистрибутив Linux, предназначенный как раз для контейнеров. Для начала необходимо скачать файловую систему:

```
vagrant@myhost:~$ mkdir alpine
vagrant@myhost:~$ cd alpine
vagrant@myhost:~/alpine$ curl -o alpine.tar.gz http://dl-cdn.alpinelinux.org/
alpine/v3.10/releases/x86_64/alpine-minirootfs-3.10.0-x86_64.tar.gz
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 2647k  100 2647k    0     0  16.6M      0 --:--:-- --:--:-- --:--:-- 16.6M
vagrant@myhost:~/alpine$ tar xvf alpine.tar.gz
```

Получили копию файловой системы Alpine внутри созданного нами каталога alpine. Удалите архив и перейдите обратно в родительский каталог:

```
vagrant@myhost:~/alpine$ rm alpine.tar.gz
vagrant@myhost:~/alpine$ cd ..
```

Можете посмотреть на содержимое файловой системы с помощью команды `ls alpine`. Она выглядит точно так же, как корневой каталог файловой системы Linux, с каталогами bin, lib, var, tmp и т. д.

После того как дистрибутив Alpine будет распакован, можно переместиться в каталог alpine с помощью команды `chroot`, если, конечно, указать команду, существующую внутри иерархии этого каталога.

В действительности есть некоторые нюансы, поскольку исполняемый файл должен располагаться по системному пути нового процесса. Данный процесс наследует среду родительского процесса, включая переменную среды PATH. Подкаталог bin в каталоге alpine становится каталогом /bin нового процесса. Поэтому если ваш обычный системный путь включает каталог /bin, то можно использовать исполняемый файл `ls` из этого каталога, не задавая путь к нему явным образом:

```
vagrant@myhost:~$ sudo chroot alpine ls
bin    etc    lib    mnt    proc   run    srv    tmp    var
dev    home   media  opt    root   sbin   sys    usr
vagrant@myhost:~$
```

Обратите внимание: этот новый корневой каталог предназначен только для дочернего процесса (в данном случае процесса, выполняющего команду `ls`). По его завершении контроль возвращается к родительскому процессу. Если запустить командную оболочку как дочерний процесс, то ее выполнение не закончится сразу же, что позволит лучше рассмотреть эффект от смены корневого каталога:

```
vagrant@myhost:~$ sudo chroot alpine sh
/ $ ls
bin    etc    lib    mnt    proc   run    srv    tmp    var
dev    home   media  opt    root   sbin   sys    usr
/ $ whoami
root
/ $ exit
vagrant@myhost:~$
```

Если же попытаться запустить командную оболочку `bash`, то ничего не получится, поскольку дистрибутив Alpine ее не включает, вследствие чего внутри нового корневого каталога ее нет. В файловой системе же дистрибутивов наподобие Ubuntu, включающих `bash`, все будет работать нормально.

Подытожим: команда `chroot` буквально меняет корневой каталог процесса. После чего процесс (и его дочерние процессы) сможет обращаться только к файлам и каталогам, расположенным ниже в иерархии, чем этот новый корневой каталог.

Помимо chroot, существует и системный вызов pivot_root. В данной главе не играет роли, какой из них использовать; главное, что у контейнера должен быть свой корневой каталог. В этих примерах я использовала команду chroot, поскольку она проще и привычнее для многих.

С точки зрения безопасности у pivot_root есть несколько преимуществ по сравнению с chroot, поэтому на практике в реализации среды выполнения контейнеров обычно можно встретить первую из них. Основное различие: pivot_root использует пространство имен монтирования, старый корневой каталог размонтируется и более не доступен внутри пространства имен монтирования. Системный же вызов chroot данный подход не использует, поэтому старый корневой каталог остается доступным через точки монтирования.

Вот каким образом можно создать для контейнера собственную корневую файловую систему. Мы обсудим это подробнее в главе 6, а пока посмотрим, как благодаря подобной отдельной корневой файловой системе ядро Linux показывает контейнеру лишь ограниченное множество ресурсов, заключенных в отдельное пространство имен.

Сочетание возможностей пространств имен и изменения корневого каталога

До сих пор мы рассматривали пространства имен и изменение корневого каталога как нечто отдельное, но их можно использовать совместно, выполняя команду `chroot` в новом пространстве имен:

```
me@myhost:~$ sudo unshare --pid --fork chroot alpine sh
/ $ ls
bin    etc    lib    mnt    proc   run    srv    tmp    var
dev    home   media  opt    root   sbin   sys    usr
```

Как вы помните из раздела «Изоляция идентификаторов процессов» на с. 61, благодаря собственному корневому каталогу контейнера можно

создать для него каталог /proc, независимый от каталога /proc хоста. Чтобы заполнить его информацией о процессах, необходимо смонтировать его в виде псевдофайловой системы типа proc. Сочетание пространства имен идентификаторов процессов и независимого каталога /proc позволяет команде ps теперь выводить только процессы внутри пространства имен идентификаторов процессов:

```
/ $ mount -t proc proc proc
/ $ ps
PID   USER     TIME  COMMAND
    1 root      0:00 sh
    6 root      0:00 ps
/ $ exit
vagrant@myhost:~$
```

Получилось! Не так просто, как изолировать хост-имя контейнера, но, комбинируя создание пространства имен идентификаторов процессов, изменение корневого каталога и монтирование псевдофайловой системы для информации процессов, мы смогли ограничить «поле зрения» контейнера, поэтому ему теперь видны лишь его собственные процессы.

Это далеко не все пространства имен. Посмотрим на пространство имен монтирования.

Пространство имен монтирования

Обычно нежелательно, чтобы точки монтирования файловых систем контейнера совпадали с точками монтирования файловых систем его хоста. Разделить их можно, если создать для контейнера его собственное пространство имен монтирования.

Вот пример создания простой связанной точки монтирования для процесса с отдельным пространством имен монтирования:

```
vagrant@myhost:~$ sudo unshare --mount sh
$ mkdir source
$ touch source/HELLO
$ ls source
HELLO
$ mkdir target
$ ls target
$ mount --bind source target
$ ls target
HELLO
```

После того как связанная точка монтирования будет создана, содержимое каталога `source` становится доступно в `target`. Всего точек монтирования (при взгляде изнутри этого процесса), наверное, будет немало, но если вы выполнили предыдущий пример, то сможете найти созданную вами `target` с помощью команды, показанной ниже:

```
$ findmnt target
TARGET    SOURCE                FSTYPE OPTIONS
/home/vagrant/target
          /dev/mapper/vagrant--vg-root[/home/vagrant/source]
                                ext4   rw,relatime,errors=remount-
ro,data=ordered
```

С точки зрения хоста она не видна, что можно проверить, выполнив ту же самую команду из другого окна терминала и убедившись, что она ничего не возвращает.

Попробуйте снова выполнить `findmnt` внутри пространства имен монтирования, но на этот раз без каких-либо параметров — и полу́чите длинный список точек монтирования. Возможность для контейнера видеть все точки монтирования на хосте может показаться нелогичной. Очень похоже на ситуацию с пространством имен идентификаторов процессов: ядро использует каталог /proc/<PID>/mounts для обмена информацией относительно точек монтирования процессов. Если создать процесс, имеющий отдельное пространство имен монтирования, однако использующий каталог /proc хоста, то окажется, что его файл `/proc/<PID>/mounts` содержит все уже существующие точки монтирования хоста. (Для получения списка точек монтирования можно воспользоваться командой `cat`.)

Чтобы множество точек монтирования для контейнеризованного процесса было полностью изолированным, необходимо сочетать создание нового пространства имен монтирования с новой корневой файловой системой и новой точкой монтирования `proc` вот так:

```
vagrant@myhost:~$ sudo unshare --mount chroot alpine sh
/ $ mount -t proc proc proc
/ $ mount
proc on /proc type proc (rw,relatime)
/ $ mkdir source
/ $ touch source/HELLO
/ $ mkdir target
/ $ mount --bind source target
/ $ mount
proc on /proc type proc (rw,relatime)
/dev/sda1 on /target type ext4 (rw,relatime,data=ordered)
```

Linux Alpine не включает команды `findmnt`, поэтому в данном примере список точек монтирования генерируется с помощью команды `mount` без параметров. (Если вы скептически относитесь к подобной замене, то попробуйте выполнить предыдущий пример с командой `mount` вместо `findmnt`, и проверьте, что результаты не отличаются.)

Наверное, вам знакома идея монтирования каталогов хоста к контейнеру с помощью команды `docker run -v <каталог_хоста>:<каталог_контейнера> ...`. Для этого, после того как будет создана корневая файловая система для контейнера, создается каталог target контейнера, после чего производится связанное монтирование каталога source хоста к этому каталогу target. А поскольку у каждого контейнера — свое пространство имен монтирования, то каталоги хоста, смонтированные подобным образом, не видны из других контейнеров.

Видимая хосту точка монтирования не очищается автоматически при завершении процесса контейнера. Ее необходимо удалять явным образом с помощью команды umount. То же самое относится и к псевдофайловым системам /proc. Никакого вреда от них нет, но ради аккуратности можете удалить их, задействовав umount proc. Размонтировать последнюю точку монтирования /proc, используемую хостом, система вам не позволит.

Пространство имен сети

Благодаря пространству имен сети контейнер получает независимое представление сетевых интерфейсов и таблицы маршрутизации. Посмотреть это представление для процесса, созданного с отдельным пространством имен сети, можно с помощью команды `lsns`:

```
vagrant@myhost:~$ sudo lsns -t net
        NS TYPE NPROCS PID USER    NETNSID NSFS COMMAND
4026531992 net      93   1 root unassigned      /sbin/init
vagrant@myhost:~$ sudo unshare --net bash
root@myhost:~$ lsns -t net
        NS TYPE NPROCS   PID USER    NETNSID NSFS COMMAND
4026531992 net      92     1 root unassigned      /sbin/init
4026532192 net       2 28586 root unassigned      bash
```

Вам может встретиться команда ip netns, но здесь она не пригодится. Команда unshare --net создает анонимное пространство имен сети, а такие пространства ip netns list не выводит.

Если поместить процесс в отдельное пространство имен сети, то он запускается с одним лишь интерфейсом loopback:

```
vagrant@myhost:~$ sudo unshare --net bash
root@myhost:~$ ip a
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

А с одним только интерфейсом loopback контейнер не сможет взаимодействовать с окружающим миром. Для этого необходимо создать для него виртуальный интерфейс Ethernet — или, точнее говоря, пару виртуальных интерфейсов Ethernet, играющих роль двух концов воображаемого кабеля, который соединяет пространство имен сети контейнера с пространством имен сети по умолчанию.

Можете создать пару виртуальных интерфейсов Ethernet во втором окне терминала от имени суперпользователя, указав два анонимных пространства имен с соответствующими идентификаторами процессов, вот так:

```
root@myhost:~$ ip link add ve1 netns 28586 type veth peer name ve2 netns 1
```

- Команда `ip link add` сообщает, что мы хотим добавить сетевое подключение.
- `ve1` — название одного из «концов» виртуального Ethernet-«кабеля».
- `netns 28586` указывает, что этот конец «подключается» в пространство имен сети, соответствующее процессу с ID 28586 (показанному в выводе команды `lsns -t net` в примере в начале данного раздела).
- `type veth` указывает, что речь идет о виртуальной паре Ethernet.
- `peer name ve2` задает название второго «конца кабеля».
- `netns 1` указывает, что этот конец «подключается» в пространство имен сети, соответствующее процессу с ID 1.

Виртуальный интерфейс Ethernet `ve1` теперь виден внутри процесса «контейнера»:

```
root@myhost:~$ ip a
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: ve1@if3: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN group ...
    link/ether 7a:8a:3f:ba:61:2c brd ff:ff:ff:ff:ff:ff link-netnsid 0
```

Подключение находится в состоянии DOWN, поэтому его необходимо включить, прежде чем использовать. Причем сделать это необходимо с обоими концами соединения.

Сначала включаем ve2 на хосте:

```
root@myhost:~$ ip link set ve2 up
```

После того как конец ve1 в контейнере будет включен, соединение должно перейти в состояние UP:

```
root@myhost:~$ ip link set ve1 up
root@myhost:~$ ip a
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: ve1@if3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
...
    link/ether 7a:8a:3f:ba:61:2c brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet6 fe80::788a:3fff:feba:612c/64 scope link
       valid_lft forever preferred_lft forever
```

Для передачи IP-трафика необходимо добавить IP-адрес, связанный с этим интерфейсом. В контейнере:

```
root@myhost:~$ ip addr add 192.168.1.100/24 dev ve1
```

И на хосте:

```
root@myhost:~$ ip addr add 192.168.1.200/24 dev ve2
```

Помимо прочего, при этом в таблицу маршрутизации в контейнере будет добавлен IP-маршрут:

```
root@myhost:~$ ip route
192.168.1.0/24 dev ve1 proto kernel scope link src 192.168.1.100
```

В начале данного раздела уже упоминалось, что пространство имен сети изолирует как интерфейсы, так и таблицу маршрутизации, поэтому данная информация о маршрутизации не зависит от таблицы маршрутизации IP на хосте. Пока контейнер может отправлять трафик только по адресам подсети 192.168.1.0/24. Протестировать это можно, выполнив ping к удаленному концу подключения изнутри контейнера:

```
root@myhost:~$ ping 192.168.1.100
PING 192.168.1.100 (192.168.1.100) 56(84) bytes of data.
64 bytes from 192.168.1.100: icmp_seq=1 ttl=64 time=0.355 ms
64 bytes from 192.168.1.100: icmp_seq=2 ttl=64 time=0.035 ms
^C
```

В главе 10 мы более подробно изучим вопросы передачи данных по сети и сетевую безопасность контейнеров.

Пространство имен пользователей

Благодаря пространству имен пользователей процессы получают свое представление идентификаторов пользователей и групп. Подобно идентификаторам процессов, пользователи и группы остаются на хосте, но могут получать другие идентификаторы. Основным достоинством этого подхода является возможность связать идентификатор суперпользователя 0 в контейнере с какой-либо непривилегированной учетной записью на хосте. С точки зрения безопасности преимущества огромны, ведь программное обеспечение может запускаться от имени суперпользователя внутри контейнера, но выбравшийся за пределы контейнера на хост злоумышленник получит доступ лишь к некой отличной от `root`, непривилегированной учетной записи. Как вы увидите в главе 9, неправильные настройки контейнера, упрощающие злоумышленнику выход за его пределы на хост, не такая уж редкость. Благодаря пространствам имен пользователей от захвата хоста злоумышленником вас отделяет уже не одно неверное движение, как раньше.

На момент написания данной книги пространства имен пользователей не слишком сильно распространены. Эта возможность не включена по умолчанию в Docker (см. пункт «Ограничения пространств имен пользователей в Docker» на с. 77) и вовсе не поддерживается Kubernetes, хотя последнее сейчас активно обсуждается (https://oreil.ly/YBN-i).

Вообще говоря, для создания новых пространств имен необходимы полномочия суперпользователя, поэтому демон Docker выполняется от имени суперпользователя, но пространство имен пользователей — исключение:

```
vagrant@myhost:~$ unshare --user bash
nobody@myhost:~$ id
uid=65534(nobody) gid=65534(nogroup) groups=65534(nogroup)
nobody@myhost:~$ echo $$
31196
```

Идентификатор пользователя внутри нового пространства имен пользователей — `nobody`. Необходимо задать соответствие идентификаторов пользователей внутри и вне пространства имен, как показано на рис. 4.2.

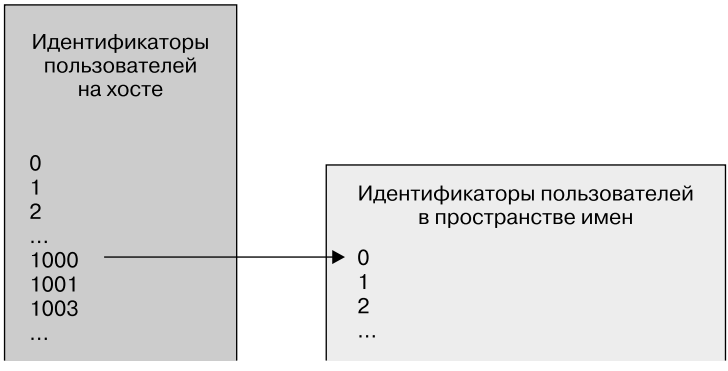


Рис. 4.2. Привязываем несуперпользователя на хосте к суперпользователю в контейнере

Упомянутое соответствие описано в файле `/proc/<pid>/uid_map`, который можно редактировать с полномочиями пользователя `root` (на хосте). В этом файле содержится три поля:

- наименьший идентификатор для отображения с точки зрения дочернего процесса;
- наименьший идентификатор, которому он должен соответствовать на хосте;
- количество отображаемых идентификаторов.

Приведу такой пример: на моей машине идентификатор пользователя `vagrant` — 1000. Чтобы `vagrant` получил идентификатор суперпользователя внутри дочернего процесса, в первых двух полях должны содержаться значения 0 и 1000. Последнее поле может содержать 1, если нужно привязать только один идентификатор (например, если внутри контейнера вам нужен лишь один пользователь). Команда, с помощью которой я задавала это соответствие, выглядит так:

```
vagrant@myhost:~$ sudo echo '0 1000 1' > /proc/31196/uid_map
```

Сразу же после того, как она была выполнена, внутри пространства имен пользователей соответствующий процесс стал суперпользователем. Не удивляйтесь, что в командной строке bash по-прежнему отображается `nobody`; здесь значение обновится только после повторного выполнения сценариев, выполняемых при запуске новой командной оболочки (например, `~/.bash_profile`):

```
nobody@myhost:~$ id
uid=0(root) gid=65534(nogroup) groups=65534(nogroup)
```

Аналогичным образом задается соответствие групп, используемых внутри дочернего процесса.

Теперь у нашего процесса есть множество привилегий:

```
nobody@myhost:~$ capsh --print | grep Current
Current: = cap_chown,cap_dac_override,cap_dac_read_search,cap_fowner,cap_fsetid,
cap_kill,cap_setgid,cap_setuid,cap_setpcap,cap_linux_immutable,
cap_net_bind_service,cap_net_broadcast,cap_net_admin,cap_net_raw,cap_ipc_lock,
cap_ipc_owner,cap_sys_module,cap_sys_rawio,cap_sys_chroot,cap_sys_ptrace,
cap_sys_pacct,cap_sys_admin,cap_sys_boot,cap_sys_nice,cap_sys_resource,
cap_sys_time,cap_sys_tty_config,cap_mknod,cap_lease,cap_audit_write,
cap_audit_control,cap_setfcap,cap_mac_override,cap_mac_admin,cap_syslog,
cap_wake_alarm,cap_block_suspend,cap_audit_read+ep
```

Как вы видели в главе 2, привилегии дают процессам различные полномочия. При создании нового пространства имен пользователей ядро выделяет процессу все эти привилегии, чтобы псевдосуперпользователь внутри пространства имен мог создавать другие пространства имен, настраивать сеть и т. д., а также осуществлять все прочее, без чего контейнер не будет настоящим.

Фактически же если создать процесс одновременно с несколькими пространствами имен, то первым будет создано пространство имен пользователей, чтобы обеспечить полный набор привилегий для создания других пространств имен:

```
vagrant@myhost:~$ unshare --uts bash
unshare: unshare failed: Operation not permitted
vagrant@myhost:~$ unshare --uts --user bash
nobody@myhost:~$
```

Пространства имен пользователей позволяют непривилегированному пользователю фактически играть роль суперпользователя в контейнеризованном процессе. Благодаря этому обычные пользователи могут запускать контейнеры с помощью *контейнеров, не требующих полномочий суперпользователя*, которые мы обсудим в главе 9.

По общему мнению, пространства имен пользователей полезны для безопасности, поскольку позволяют запускать меньшее количество контейнеров от имени «настоящего» суперпользователя (то есть суперпользователя с точки зрения хоста). Однако есть ряд уязвимостей (например, CVE-2018-18955) (https://oreil.ly/764Ia), непосредственно связанных с некорректным преобразованием полномочий при переходе в пространство имен пользователей или выходе из него. Ядро Linux — очень сложная программная система, и ничего удивительного, что время от времени в нем находят ошибки.

Ограничения пространств имен пользователей в Docker. Пространства имен пользователей в Docker можно включить, хотя по умолчанию они отключены, поскольку несовместимы с возможностями, которые нужны пользователям Docker.

Перечисленное ниже может коснуться вас и в случае применения пространств имен пользователей с другими средами выполнения контейнеров.

- Пространства имен пользователей несовместимы с совместным применением идентификатора процесса или пространства имен сети с хостом.
- Даже если процесс выполняется от имени суперпользователя внутри контейнера, на самом деле у него нет всех полномочий суперпользователя. Например, у него нет привилегии `CAP_NET_BIND_SERVICE`, поэтому он не может привязываться к портам с номерами меньше 1024 (больше информации о привилегиях Linux можно найти в главе 2).
- Для работы с файлами контейнеризованному процессу необходимы соответствующие полномочия (например, доступ для записи, чтобы изменить файл). При монтировании файла с хоста важен фактический идентификатор пользователя на хосте.

  Это удобно для защиты файлов хоста от несанкционированного доступа из контейнера, но может сбивать с толку, если, допустим, пользователю, играющему роль `root` внутри контейнера, не разрешается изменять файл.

Пространство имен обмена информацией между процессами

В Linux различные процессы могут обмениваться информацией с помощью доступа к разделяемой области памяти либо очереди сообщений. Для доступа к общему множеству идентификаторов для этих механизмов два процесса должны входить в одно пространство имен обмена информацией между процессами (inter-process communications, IPC).

Вообще говоря, *нежелательно*, чтобы контейнеры могли обращаться к разделяемой памяти друг друга, поэтому им назначаются отдельные пространства имен IPC.

Чтобы посмотреть на это в действии, создайте разделяемый блок памяти и просмотрите текущее состояние IPC с помощью команды `ipcs`:

```
$ ipcmk -M 1000
Shared memory id: 98307
$ ipcs

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 0          root       644        80         2
0x00000000 32769      root       644        16384      2
0x00000000 65538      root       644        280        2
0xad291bee 98307      ubuntu     644        1000       0

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0x000000a7 0          root       600        1
```

В этом примере созданный новый блок разделяемой памяти (идентификатор которого указан в столбце `shmid`) отображается последним в блоке «Разделяемые сегменты памяти» (Shared Memory Segments). Вдобавок там отображается несколько уже существующих объектов IPC, созданных ранее `root`.

Процесс с отдельным пространством имен IPC не видит ни одного из этих объектов IPC:

```
$ sudo unshare --ipc sh
$ ipcs

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
```

Пространство имен контрольных групп

Последнее из пространств имен (по крайней мере на момент написания данной книги) — пространство имен контрольных групп. Оно чем-то напоминает `chroot` для файловой системы контрольных групп — не дает про-

цессу видеть конфигурацию контрольной группы, которая по иерархии каталогов контрольных групп расположена выше, чем его собственная контрольная группа.

Большинство пространств имен были добавлены в версии 3.8 ядра Linux, но пространство имен контрольных групп было добавлено позднее, в версии 4.6. В относительно старых дистрибутивах Linux (например, Ubuntu 16.04) эта возможность не поддерживается. Проверить версию ядра Linux можно с помощью команды uname -r.

Посмотреть на пространство имен контрольных групп в действии можно, сравнив содержимое каталога /proc/self/cgroup вне и внутри пространства имен контрольных групп:

```
vagrant@myhost:~$ cat /proc/self/cgroup
12:cpu,cpuacct:/
11:cpuset:/
10:hugetlb:/
9:blkio:/
8:memory:/user.slice/user-1000.slice/session-51.scope
7:pids:/user.slice/user-1000.slice/session-51.scope
6:freezer:/
5:devices:/user.slice
4:net_cls,net_prio:/
3:rdma:/
2:perf_event:/
1:name=systemd:/user.slice/user-1000.slice/session-51.scope
0::/user.slice/user-1000.slice/session-51.scope
vagrant@myhost:~$
vagrant@myhost:~$ sudo unshare --cgroup bash
root@myhost:~# cat /proc/self/cgroup
12:cpu,cpuacct:/
11:cpuset:/
10:hugetlb:/
9:blkio:/
8:memory:/
7:pids:/
6:freezer:/
5:devices:/
4:net_cls,net_prio:/
3:rdma:/
2:perf_event:/
1:name=systemd:/
0::/
```

Мы обсудили все типы пространств имен и их использование совместно с `chroot` для изоляции представления процессом своего окружения. В сочетании с информацией о контрольных группах, приведенной в предыдущей главе, этого должно быть достаточно для понимания всего, что требуется для создания контейнера.

Прежде чем перейти к следующей главе, имеет смысл взглянуть на контейнер с точки зрения хоста, на котором он работает.

Процессы контейнера с точки зрения хоста

Хотя обычно используется название «контейнеры», правильнее было бы называть их «контейнеризованные процессы». Контейнер остается процессом Linux, работающим на хост-компьютере, просто видит лишь ограниченную часть этого хост-компьютера и имеет доступ только к части файловой системы, а возможно, и к ограниченному контрольными группами набору ресурсов. И поскольку он является всего лишь процессом, то существует в контексте операционной системы хоста и использует одно и то же ядро с хостом, как показано на рис. 4.3.

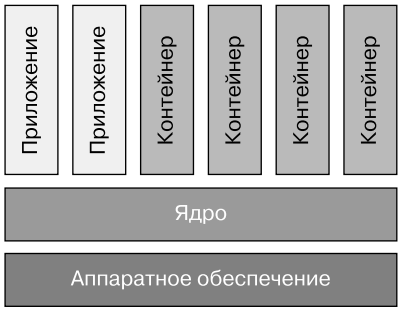


Рис. 4.3. Контейнеры используют одно ядро с хостом

В следующей главе мы сравним их с виртуальными машинами, но прежде изучим более подробно степень изоляции контейнеризованного процесса от хоста, а также прочих контейнеризованных процессов на этом хосте. Для этого проведем несколько экспериментов над контейнером Docker. Запустите процесс контейнера на основе Ubuntu (или другого дистрибутива

Linux, который вам по душе), запустите в нем командную оболочку, а в ней выполните команду длительного бездействия, вот так:

```
$ docker run --rm -it ubuntu bash
root@1551d24a $ sleep 1000
```

В этом примере выполняется команда `sleep` длительностью 1000 секунд, но учтите, что данная команда выполняется как процесс внутри контейнера. При нажатии клавиши Enter после ввода команды `sleep` Linux клонирует новый процесс с новым идентификатором процесса и запускает исполняемый файл `sleep` в этом процессе.

Можно перевести этот процесс `sleep` на выполнение в фоновом режиме (`Ctrl-Z` для приостановки процесса, а затем `bg %1` для перевода его в фоновый режим). Теперь выполните `ps` внутри контейнера, чтобы посмотреть на тот же процесс с точки зрения контейнера:

```
me@myhost:~$ docker run --rm -it ubuntu bash
root@ab6ea36fce8e:/$ sleep 1000
^Z
[1]+  Stopped                 sleep 1000
root@ab6ea36fce8e:/$ bg %1
[1]+ sleep 1000 &
root@ab6ea36fce8e:/$ ps
  PID TTY          TIME CMD
    1 pts/0    00:00:00 bash
   10 pts/0    00:00:00 sleep
   11 pts/0    00:00:00 ps
root@ab6ea36fce8e:/$
```

А пока команда `sleep` еще выполняется, откройте второй терминал на том же хосте и посмотрите на тот же процесс бездействия с точки зрения хоста:

```
me@myhost:~$ ps -C sleep
  PID TTY          TIME CMD
30591 pts/0    00:00:00 sleep
```

Параметр `-C sleep` указывает, что нас интересуют только процессы, в которых запущен исполняемый файл `sleep`.

У контейнера есть собственное пространство имен идентификаторов процессов, поэтому логично, что идентификаторы его процессов меньше в числовом выражении, и именно это мы и наблюдаем при выполнении `ps` в контейнере. Однако с точки зрения хоста у процесса бездействия другой, больший идентификатор. В предыдущем примере отображается только один процесс, с идентификатором 30591 на хосте и десять в контейнере. (Его идентификатор может отличаться, в зависимости от того, что еще выполняется сейчас или запускалось ранее на этой машине, но, вероятно, число будет намного больше.)

Чтобы лучше понимать контейнеры и обеспечиваемый ими уровень изоляции, важно понимать следующее: хотя идентификаторы процессов и отличаются, они оба указывают на *один и тот же процесс*. Просто с точки зрения хоста числовой идентификатор процесса намного больше.

То, что процессы контейнера видны с хоста, — одно из фундаментальных различий контейнеров и виртуальных машин. Злоумышленник, получивший доступ к хост-компьютеру, может видеть и влиять на *все работающие на этом хосте процессы*, особенно имея полномочия суперпользователя. Как вы увидите в главе 9, существует немало на удивление простых способов непреднамеренно открыть злоумышленнику возможность выйти за пределы взломанного контейнера на хост.

Хост-компьютеры контейнеров

Как вы видели, контейнеры и их хост-компьютеры используют одно ядро, что определенным образом влияет на практические рекомендации по хост-компьютерам для контейнеров. В случае взлома хоста все расположенные на нем контейнеры — потенциальные жертвы, особенно если злоумышленник получает полномочия суперпользователя или какой-либо другой повышенный уровень полномочий (например, членство в группе `docker`, имеющее права администрировать контейнеры, в которых в качестве среды выполнения используется Docker).

Настоятельно рекомендуется запускать контейнерные приложения на выделенных хост-компьютерах (неважно, виртуальных машинах или реальных), в основном из соображений безопасности.

- Применение механизма координации для работы контейнеров означает, что людям (почти) не нужен доступ к хостам. Если не используются никакие другие приложения, то достаточно очень маленького набора учетных записей пользователей на хост-компьютерах, что упрощает администрирование и делает более заметными попытки неавторизованного входа.
- В качестве операционной системы хоста для запуска контейнеров Linux можно использовать любой дистрибутив Linux, но существует несколько «тонких» дистрибутивов операционных систем, специально предназначенных для запуска контейнеров. Они включают только необходимые для работы контейнеров компоненты и за счет этого сокращают поверхность атаки на хост. В их числе: RancherOS, Fedora CoreOS от Red Hat

и Photon OS от VMware. Чем меньше компонентов установлено на хост-компьютере, тем ниже вероятность существования в них уязвимостей (см. главу 7).

- Все хост-компьютеры в кластере могут использовать одну конфигурацию, без каких-либо требований, относящихся к конкретным приложениям. Это упрощает автоматизацию подготовки хост-компьютеров и означает, что их можно считать, по сути, неизменяемыми. Если хост-компьютеру требуется обновление, то нужно не устанавливать на нем патчи, а просто исключить его из кластера и заменить новой машиной. Неизменяемость машин упрощает выявление вторжений.

Мы вернемся к преимуществам неизменяемости в главе 6.

Использование «тонких» операционных систем сокращает набор возможных настроек, но не исключает их полностью. Например, на каждом хост-компьютере будет работать среда выполнения контейнеров (возможно, Docker) плюс код средства координации (возможно, kubelet Kubernetes). Эти компоненты имеют множество настроек, часть которых связана с безопасностью. Центр по интернет-безопасности (Center for Internet Security, CIS) (https://cisecurity.org/) публикует оценки эффективности практических рекомендаций по настройке и выполнению различных программных компонентов, включая Docker, Kubernetes и Linux.

В корпоративной среде следует искать решение по обеспечению безопасности контейнеров, которое бы также защищало хосты, сообщая об уязвимостях и потенциально небезопасных значениях настроек. Кроме того, не помешает журналирование и оповещение о входах и попытках входа в систему на уровне хоста.

Резюме

Поздравляю! Раз вы дошли до конца этой главы, то должны уже хорошо представлять себе, чем в действительности является контейнер. Я показала вам три основных механизма ядра Linux, с помощью которых можно ограничить доступ процессов к ресурсам хоста:

- пространства имен служат для ограничения видимых процессу контейнера ресурсов — например, для изоляции предоставляемого контейнеру набора идентификаторов процессов;

- изменение корневого каталога дает возможность ограничить видимое контейнеру множество файлов и каталогов;
- контрольные группы позволяют управлять доступом контейнера к ресурсам.

Как вы видели в главе 1, изоляция рабочих заданий друг от друга — важный аспект безопасности контейнеров. Теперь вы уже должны отдавать себе отчет в том, что все контейнеры на конкретном хосте (неважно, на виртуальной машине или на реальном сервере) задействуют одно и то же ядро. Конечно, то же самое справедливо и в многопользовательской системе, где различные пользователи могут подключаться к одной машине и запускать приложения. Однако в такой системе администраторы обычно ограничивают полномочия всех пользователей; и уж конечно, не предоставляют им всех полномочий суперпользователя. Контейнеры же — по крайней мере, на момент написания данной книги — все по умолчанию запускаются от имени суперпользователя, и это предотвращает их взаимное влияние друг на друга за счет границ зон безопасности, обеспечиваемых пространствами имен, изменением корневого каталога и контрольными группами.

Теперь, когда вы уже представляете, как функционируют контейнеры, возможно, вам будет интересно заглянуть на сайт Джесс Фразель (Jess Frazelle) contained.af и посмотреть, насколько эффективными они могут быть. Возможно, именно вам удастся вырваться за границы контейнера?

В главе 8 мы обсудим варианты укрепления границ безопасности вокруг контейнеров, но пока что изучим более подробно, как работают виртуальные машины. Благодаря этому мы сможем сравнить прочность изоляции контейнеров и виртуальных машин, особенно с точки зрения безопасности.

ГЛАВА 5

Виртуальные машины

Контейнеры очень часто сравнивают с виртуальными машинами, особенно в плане изоляции. Убедимся, что вы хорошо понимаете, как функционируют виртуальные машины, а значит, можете с полным основанием рассуждать о различиях между ними и контейнерами. Это особенно важно для оценки границ зон безопасности запущенных в контейнерах или различных виртуальных машинах приложений. Оценивая преимущества контейнеров с точки зрения безопасности, отнюдь не помешает знать, чем они отличаются от виртуальных машин.

Впрочем, все не настолько однозначно. Как вы увидите в главе 8, существует несколько утилит-«песочниц», которые усиливают изоляцию контейнеров, приближая их к виртуальным машинам. Но чтобы понимать все за и против подобных подходов с точки зрения безопасности, лучше начать с четкого осознавания разницы между виртуальной машиной и «обычным» контейнером.

Принципиальное различие между ними состоит в том, что в виртуальной машине выполняется копия операционной системы целиком, включая ядро, в то время как контейнер использует одно ядро совместно с хост-компьютером. Чтобы понять, что это значит, необходимо сначала разобраться в том, как диспетчер виртуальных машин (Virtual Machine Monitor, VMM) создает их и управляет ими. Начнем с подготовки почвы для этой дискуссии, а именно задумаемся над тем, что происходит при загрузке компьютера.

Загрузка компьютера

Представьте физический сервер: с несколькими CPU, оперативной памятью и сетевыми интерфейсами. В начале загрузки компьютера запускается специальная программа — BIOS (Basic Input Output System — «базовая система ввода/вывода»). Она проверяет объем доступной оперативной памяти,

находит сетевые интерфейсы и все прочие устройства, например мониторы, клавиатуру, подключенные устройства хранения и т. д.

На самом деле значительная часть этой функциональности сегодня перешла в UEFI (Unified Extensible Firmware Interface — универсальный расширяемый интерфейс прошивки), но для простоты будем считать его современным BIOS.

«Переписав» аппаратное обеспечение, система запускает загрузчик, который загружает и запускает код ядра операционной системы — Linux, Windows или какой-либо другой. Как вы видели в главе 2, код ядра работает на более высоком уровне полномочий, чем код приложения. Такой уровень полномочий позволяет ему взаимодействовать с памятью, сетевыми интерфейсами и т. д., в то время как приложения, работающие в пользовательском пространстве, не могут делать этого непосредственно.

В процессорах x86 уровни полномочий организованы в *кольца* (rings), из которых кольцо 0 имеет наивысшие полномочиями, а кольцо 3 — наименьшие. В большинстве операционных систем в обычных условиях (без виртуальных машин) ядро работает на уровне кольца 0, а код пользовательского пространства — на уровне кольца 3, как показано на рис. 5.1.

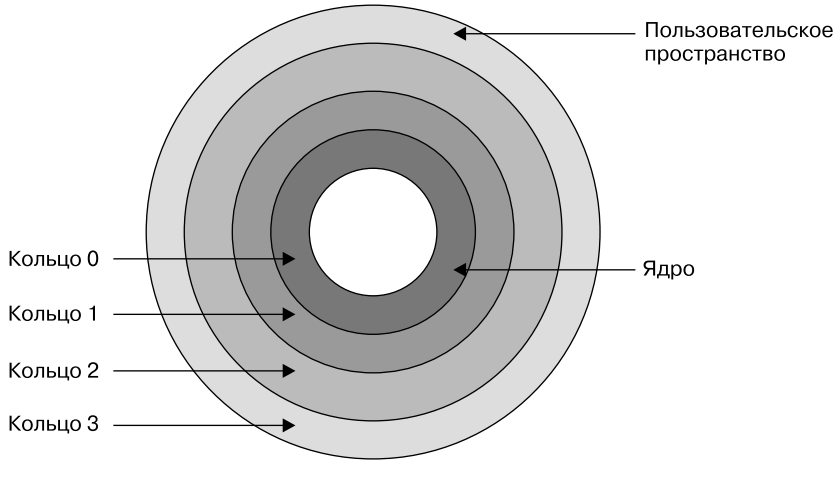


Рис. 5.1. Кольца полномочий

Код ядра (как и любой другой) выполняется на CPU в виде инструкций машинного кода, которые могут включать привилегированные инструк-

ции для обращения к памяти, запуска потоков выполнения CPU и т. д. Подробное изложение всего, что может и будет происходить при инициализации ядра, выходит за рамки данной книги, но фактически цель состоит в монтировании корневой файловой системы, настройке передачи данных по сети и запуске всех системных демонов. (Если вы хотели бы узнать больше, то на GitHub (https://oreil.ly/GPutF) можно найти множество информации о внутреннем устройстве ядра Linux, включая описание процесса начальной загрузки.)

Завершив инициализацию, ядро начинает запускать программы в пользовательском пространстве. Оно отвечает за контроль над всем, что необходимо этим программам. Ядро запускает, контролирует и планирует на выполнение потоки выполнения CPU, в которых работают эти программы, и отслеживает эти потоки в собственных структурах данных, соответствующих процессам. Одна из важных сторон функциональности ядра — управление памятью. Ядро выделяет процессам блоки памяти и обеспечивает недоступность блоков памяти одних процессов для других.

Знакомство с VMM

Как вы только что видели, в обычном случае ядро напрямую управляет ресурсами машины. В мире же виртуальных машин роль первого слоя управления ресурсами играет диспетчер виртуальных машин (Virtual Machine Monitor, VMM), распределяющий ресурсы по виртуальным машинам. У каждой виртуальной машины — свое ядро.

Каждой находящейся под его управлением виртуальной машине VMM выделяет некоторое количество оперативной памяти и ресурсов CPU, настраивает виртуальные сетевые интерфейсы и прочие виртуальные устройства, после чего запускает гостевое ядро с доступом к этим ресурсам.

В обычном сервере BIOS передает ядру подробную информацию о доступных ресурсах компьютера; в случае виртуальной машины VMM делит эти ресурсы и передает гостевым ядрам информацию лишь о той части ресурсов, которые им предоставляются. С точки зрения гостевой ОС ядро считает, что имеет непосредственный доступ к оперативной памяти и устройствам, хотя на самом деле получает доступ лишь к предоставляемой VMM абстракции.

VMM отвечает за то, чтобы гостевая операционная система и ее приложения не нарушали границы выделенных им ресурсов. Например, если такой системе выделен участок памяти на хост-компьютере, то в доступе к памяти вне этого участка будет отказано.

Существует два основных вида VMM, часто называемых (не слишком изобретательно) Type 1 и Type 2. И конечно, между ними есть небольшая «серая зона»!

VMM Type 1 (гипервизоры)

В обычных системах ядро операционной системы (например, Linux или Windows) запускает загрузчик. В чистой среде виртуальной машины Type 1 вместо него выполняется специальная программа VMM уровня ядра.

VMM Type 1 известны также под названием *гипервизоров* (hypervisors). В их числе, например, Hyper-V (https://oreil.ly/FsXVi), Xen (https://xenproject.org/) и ESX/ESXi (https://oreil.ly/ezG3t). Как видно из рис. 5.2, гипервизоры выполняются непосредственно на аппаратном обеспечении («пустой» машине), без какой-либо операционной системы в качестве прослойки.

Рис. 5.2. Диспетчер виртуальных машин Type 1 — гипервизор

Под «уровнем ядра» я подразумеваю, что гипервизор работает в кольце 0. (Это верно до тех пор, пока мы не начнем рассматривать аппаратную виртуализацию далее в этой главе, но сейчас просто предположим, что в кольце 0.) Ядро гостевой операционной системы работает в кольце 1, как показано на рис. 5.3, поэтому имеет меньше полномочий, чем гипервизор.

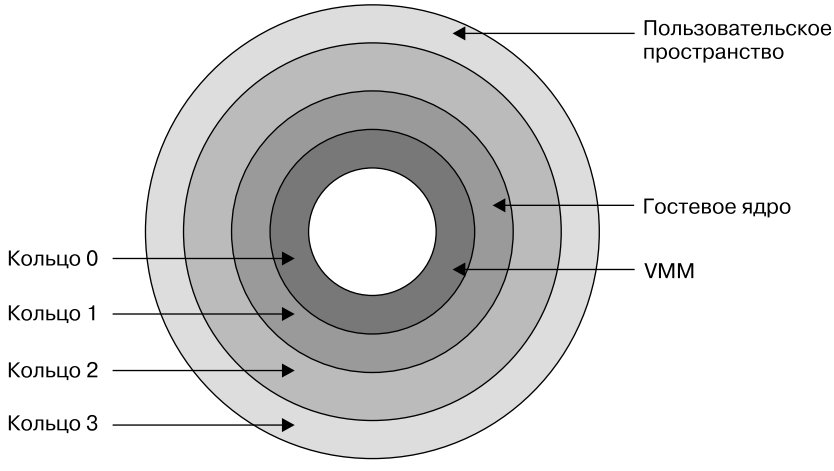


Рис. 5.3. Кольца полномочий при использовании гипервизора

VMM Type 2

Виртуальные машины, которые вы запускаете на своем ноутбуке или настольном компьютере, например, с помощью чего-то наподобие VirtualBox (https://www.virtualbox.org/), относятся к VMM Type 2 (хостируемым VM). Например, пусть на вашем ноутбуке установлена операционная система macOS, а значит, выполняется ядро macOS. VirtualBox устанавливается как отдельное приложение и управляет затем гостевыми виртуальными машинами, сосуществующими с операционной системой хоста. Эти машины могут работать под управлением Linux или Windows. На рис. 5.4 показано, как сосуществуют гостевая ОС и операционная система хоста.

Задумайтесь на минутку, что значит запустить, скажем, Linux внутри macOS. По определению это означает, что должно быть ядро Linux, которое должно отличаться от ядра macOS хоста.

У приложения VMM есть компоненты пользовательского пространства, с которыми вы можете взаимодействовать как пользователь, но оно также устанавливает привилегированные компоненты для виртуализации. В данной главе я расскажу вам более подробно, как это происходит.

Помимо VirtualBox, существуют и другие примеры VMM Type 2, например Parallels (https://parallels.com/) и QEMU (https://oreil.ly/LZmcn).

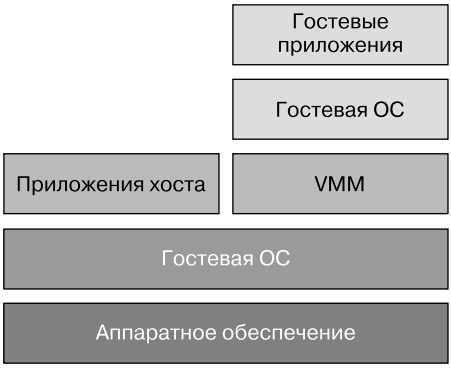


Рис. 5.4. Диспетчер виртуальных машин Type 2

Виртуальные машины, работающие в ядре

Я обещала, что границы между VMM Type 1 и Type 2 будут довольно размытыми. В Type 1 гипервизор запускается непосредственно на «пустой» машине; в Type 2 VMM запускается в пользовательском пространстве в операционной системе хоста. Но что, если запустить диспетчер виртуальных машин в ядре операционной системы хоста?

Именно это и происходит в модуле ядра Linux под названием KVM (Kernel-based Virtual Machine), как показано на рис. 5.5.

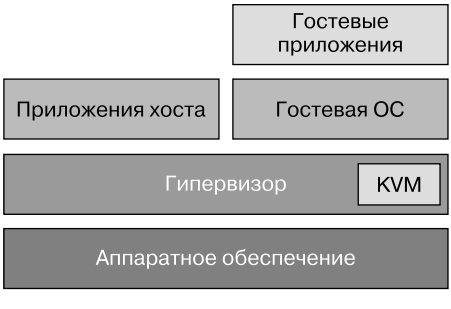


Рис. 5.5. KVM

Вообще говоря, KVM считается гипервизором Type 1, поскольку гостевая ОС не должна обращаться к операционной системе хоста, но мне подобная классификация кажется слишком упрощенной.

KVM часто используется совместно с QEMU, которая выше была отнесена к гипервизорам Type 2. QEMU динамически транслирует системные вызовы гостевой ОС в системные вызовы операционной системы хоста. Стоит упомянуть, что QEMU может полноценно задействовать возможности аппаратного ускорения, предоставляемые KVM.

Для виртуализации в VMM используются схожие технологии как в Type 1, так и Type 2 и промежуточных вариантах. Основная их идея называется trap-and-emulate («перехватывай и эмулируй»), хотя, как мы увидим, процессоры x86 требуют некоторой изобретательности при реализации этой идеи.

«Перехватывай и эмулируй»

Часть инструкций CPU — *привилегированные* (privileged) в том смысле, что могут выполняться только в кольце 0; попытка выполнения их в прочих кольцах приводит к *системному прерыванию* (trap). Системное прерывание можно считать своего рода исключением в ПО, которое вызывает срабатывание обработчика ошибок; оно приводит к тому, что процессор вызывает обработчик из кода в кольце 0.

Если VMM работает с полномочиями кольца 0, а код ядра гостевой операционной системы — с более низкими полномочиями, то привилегированная инструкция, выполняемая гостевой ОС, может привести к вызову обработчика из VMM для ее эмуляции. Благодаря этому VMM гарантирует, что гостевые ОС не вмешиваются в работу друг друга через привилегированные инструкции.

К сожалению, привилегированные инструкции — еще не все. Инструкции CPU, способные влиять на ресурсы машины, называются *инструкциями, чувствительными к виртуализации* (sensitive instructions). VMM приходится обрабатывать эти инструкции вместо гостевой операционной системы, поскольку только VMM действительно видит все ресурсы машины. Существует еще один класс чувствительных к виртуализации инструкций, ведущих себя по-разному при выполнении в кольце 0 или в кольцах с более низкими полномочиями. Обрабатывать эти инструкции приходится опять-таки VMM, поскольку код гостевой операционной системы рассчитан на полномочия кольца 0.

Жизнь разработчиков VMM весьма бы упростилась, если бы все чувствительные к виртуализации инструкции были привилегированными, ведь тогда достаточно было бы просто написать обработчики системных прерываний

для всех таких инструкций. К сожалению, не все чувствительные к виртуализации инструкции процессоров x86 еще и привилегированные, поэтому для их обработки диспетчерам виртуальных машин приходится использовать различные методики. Чувствительные к виртуализации, но непривилегированные инструкции считаются невиртуализируемыми.

Обработка невиртуализируемых инструкций

Существует несколько методик обработки подобных невиртуализируемых инструкций.

- Один вариант — *двоичная трансляция* (binary translation). VMM выявляет и переписывает в режиме реального времени все непривилегированные, чувствительные к виртуализации инструкции. Это сложная (ресурсоемкая) задача, и современные процессоры x86 поддерживают аппаратную виртуализацию, чтобы упростить двоичную трансляцию.
- Еще один вариант — *паравиртуализация* (paravirtualization). Вместо динамического изменения инструкций гостевой ОС эта система переписывается так, что не использует набор невиртуализируемых инструкций, фактически выполняя системные вызовы к гипервизору. Эта методика используется, в частности, в гипервизоре Xen.
- Аппаратная виртуализация (например, технология VT-x Intel) позволяет гипервизорам работать на новом уровне, отличающемся дополнительными полномочиями, — известном как режим VMX root, который, по сути, представляет собой кольцо –1. Благодаря этому ядра гостевых операционных систем виртуальной машины могут работать в кольце 0 (то есть в режиме VMX non-root), как если бы они были операционной системой хоста.

Если вам интересно узнать больше о том, как работает виртуализация, то Кит Адамс (Keith Adams) и Оле Агесен (Ole Agesen) в своей статье (https://oreil.ly/D1cZO) приводят подробное сравнение и описание улучшений производительности за счет аппаратных расширений.

Теперь, когда вы уже четко представляете себе, как создаются и управляются виртуальные машины, поговорим о том, что это означает в смысле изоляции процессов (приложений) друг от друга.

Изоляция процессов и безопасность

Первостепенная задача в сфере безопасности — обеспечение надежной изоляции приложений друг от друга. Если мое приложение может читать выделенную вашему приложению область памяти, значит, у меня есть доступ к вашим данным.

Самый надежный вид изоляции — физическая. Если наши приложения работают на совершенно отдельных компьютерах, то мой код никак не сможет получить доступ к памяти вашего приложения.

Как мы только что говорили, ядро отвечает за управление процессами его пользовательского пространства, включая выделение им памяти. Именно ядро отвечает за то, чтобы одни приложения не могли получить доступ к памяти других. Взломщик может воспользоваться ошибками в механизме управления памятью ядра при их наличии, чтобы получить доступ к памяти, к которой у него доступа быть не должно. И хотя ядро обычно исключительно хорошо проверено в боевых условиях, оно также очень велико и сложно устроено, к тому же постоянно развивается. На момент написания данной книги мне неизвестно о каких-либо серьезных изъянах в изоляции ядра, однако я не могу поручиться за то, что какие-нибудь проблемы не обнаружатся в будущем.

Возникнуть подобные изъяны могут вследствие усложнения нижележащего аппаратного обеспечения. В последние годы производители CPU разработали технологию «упреждающей обработки», при которой процессор «забегает вперед» инструкций, выполняемых в настоящий момент, и предугадывает результаты, чтобы понять, нужно ли на самом деле выполнять ветку кода. Это позволило добиться более высокого быстродействия и вместе с тем открыло дорогу для знаменитых уязвимостей Spectre и Meltdown.

Наверное, вам интересно, почему считается, что гипервизоры изолируют виртуальные машины лучше, чем ядро изолирует свои процессы; в конце концов, гипервизоры тоже управляют памятью и доступом к устройствам и отвечают за разделение виртуальных машин. Совершенно верно, что изъяны в гипервизоре могут привести к серьезным проблемам с изоляцией виртуальных машин. Разница в том, что стоящая перед гипервизорами задача намного, намного проще. В ядре процессы пользовательского пространства частично должны видеть друг друга; очень простой пример: выполните `ps` — и увидите запущенные на той же машине процессы. Вы можете (при наличии соответствующих прав) получить доступ к информации об этих процессах,

заглянув в каталог /proc. Вы можете намеренно использовать память совместно в нескольких процессах через механизм IPC и, собственно, с помощью совместно используемой памяти. Все эти механизмы, с помощью которых один процесс может на законных основаниях получать информацию о другом, ослабляют изоляцию. Так происходит вследствие возможных изъянов, из-за которых подобный доступ может предоставляться в неожиданных или непредусмотренных обстоятельствах.

В случае же виртуальных машин никакого эквивалента этому явлению нет; процессы одной (виртуальной) машины не видны из другой. Управление памятью требует меньшего объема кода, поскольку гипервизору не нужно учитывать ситуации, при которых машинам может потребоваться использовать память совместно — это просто нехарактерно для виртуальных машин. В результате гипервизоры обычно намного меньше и проще устроены, чем полноценные ядра. Ядро Linux включает более 20 миллионов строк кода (https://oreil.ly/FHKhp); а гипервизор Xen — всего 50 тысяч (https://oreil.ly/1MWub).

А где меньше кода и меньше степень сложности — меньше и поверхность атаки, а значит, и вероятность изъянов, которыми мог бы воспользоваться злоумышленник. Поэтому считается, что границы зон безопасности у виртуальных машин прочнее.

Тем не менее эксплойты в виртуальных машинах встречаются. Дархан Тан (Darshan Tank), Акшай Агарвал (Akshai Aggarwal) и Нирбхай Чобей (Nirbhay Chaubey) (https://oreil.ly/HCXBO) описали таксономию различных типов атак, а Национальный институт стандартов и технологий (NIST) опубликовал методические рекомендации по укреплению безопасности (https://oreil.ly/W_b7o) виртуализированных сред.

Недостатки виртуальных машин

Вероятно, я уже настолько убедила вас в преимуществах изоляции виртуальных машин, что вы недоумеваете: зачем вообще использовать контейнеры? У виртуальных машин, по сравнению с контейнерами, есть несколько недостатков.

- Время загрузки виртуальной машины на несколько порядков больше, чем у контейнера. В конце концов, запуск контейнера означает просто запуск

нового процесса Linux, а не полномасштабную загрузку и инициализацию виртуальной машины. Относительно медленная загрузка виртуальных машин означает медленную масштабируемость, не говоря уже о важности быстрой загрузки в случае частой поставки нового кода, например несколько раз в день. (Впрочем, обсуждаемая в разделе «Виртуальная машина Firecracker» на с. 144 технология виртуализации Firecracker от Amazon обеспечивает виртуальные машины с очень быстрой загрузкой, порядка 100 миллисекунд на момент написания данной книги.)

- Благодаря контейнерам разработчики могут «создать один раз, выполнять где угодно» быстро и эффективно. Можно, конечно, хотя это и займет немало времени, собрать образ виртуальной машины целиком и запускать его на своем ноутбуке, но данный подход не стал чрезмерно популярным среди разработчиков, в отличие от контейнеров.
- В современных облачных средах при аренде виртуальной машины выбирается CPU и объем оперативной памяти, за которые необходимо вносить арендную плату вне зависимости от того, какая часть этих ресурсов фактически используется работающим в ней кодом приложения.
- Каждая виртуальная машина влечет накладные расходы по выполнению всего ядра целиком. Благодаря совместному использованию ядра контейнеры гораздо рациональнее задействуют ресурсы и демонстрируют высокую производительность.

Выбор между виртуальными машинами или контейнерами означает выбор множества компромиссов относительно таких факторов, как быстродействие, цена, удобство, риски и прочность границ зон безопасности между приложениями.

Изоляция контейнеров по сравнению с изоляцией виртуальных машин

Как вы видели в главе 4, контейнеры представляют собой просто процессы Linux с ограничениями на доступные ресурсы. Ядро изолирует их с помощью механизмов пространств имен, контрольных групп и изменения корневого каталога, предназначенных специально для изоляции процессов. Однако сам факт совместного использования контейнерами ядра означает, что их изоляция слабее по сравнению с виртуальными машинами.

Однако еще не все потеряно! Для усиления изоляции можно применить дополнительные средства безопасности и различные «песочницы», о которых я расскажу в главе 8. Существуют также весьма эффективные утилиты обеспечения безопасности, базирующиеся на том, что контейнеры зачастую инкапсулируют микросервисы. Их мы рассмотрим в главе 13.

Резюме

Теперь вы уже хорошо представляете, что такое виртуальные машины. Вы узнали, почему изоляция виртуальных машин считается более надежной по сравнению с изоляцией контейнеров и почему контейнеры в целом не считаются достаточно безопасными для сред с жесткой мультиарендностью. Эту разницу очень важно понимать при обсуждении безопасности контейнеров.

Безопасность самих виртуальных машин выходит за рамки данной книги, хотя мы и затронули вкратце вопрос о том, как повысить надежность настроек хоста контейнера, в разделе «Хост-компьютеры контейнеров» на с. 82.

Далее в книге вы увидите примеры, в которых более слабая (по сравнению с виртуальными машинами) изоляция контейнеров легко нарушается из-за неправильных настроек. Но прежде убедимся, что вы полностью понимаете, как устроены образы контейнеров и то, почему это может существенно влиять на безопасность.

ГЛАВА 6

Образы контейнеров

Если вам случалось применять Docker или Kubernetes, то вы уже знакомы с идеей образов контейнеров, которые хранятся в реестре. В этой главе мы изучим данную идею, посмотрим на содержимое образов и обсудим их использование в таких средах выполнения, как Docker и `runc`.

Разобравшись, что такое образы контейнеров, вы сможете задуматься и о следствиях создания, хранения и извлечения образов для безопасности — а с этими шагами связано множество векторов атак. Кроме того, я расскажу вам о практических рекомендациях, позволяющих гарантировать, что сборки и образы не подорвут безопасность системы в целом.

Корневая файловая система и конфигурация образов контейнеров

Образ контейнера состоит из двух частей: корневой файловой системы и конфигурации.

Если вы внимательно следили за ходом примеров из главы 4, то скачали копию корневой файловой системы Alpine и использовали ее в качестве содержимого каталога `root` внутри вашего контейнера. Вообще говоря, при запуске контейнера на основе его образа создается экземпляр контейнера, а образ включает корневую файловую систему. Если выполнить команду `docker run -it alpine sh` и сравнить с тем, что находится внутри созданного вами вручную контейнера, то общая схема каталогов и файлов окажется той же, а если версия Alpine такая же, то они совпадут в точности.

Если ваше знакомство с контейнерами состоялось с помощью Docker, то вам уже привычна идея создания образов на основе инструкций из Dockerfile. Часть команд Dockerfile (например, `FROM`, `ADD`, `COPY` и `RUN`) изменяет содержимое корневой файловой системы, включенной в образ. Другие команды,

например USER, PORT и ENV, влияют на хранимую в образе параллельно с корневой файловой системой информацию о настройках. Просмотреть ее можно, выполнив для нужного образа команды docker inspect. Эти настройки задают Docker значения параметров времени выполнения, используемые по умолчанию при запуске образа. Например, переменные среды задаются для выполняемого процесса контейнера с помощью команды ENV в Dockerfile.

Переопределение настроек во время выполнения

В Docker можно переопределять конфигурацию во время выполнения, воспользовавшись параметрами командной строки. Например, изменить переменную среды или задать новую можно с помощью команды docker run -e <*имя_переменной*>=<*новое_значение*> ....

В Kubernetes это можно сделать с помощью определения env контейнера в YAML-описании модуля:

```
apiVersion: v1
kind: Pod
metadata:
  name: demo
spec:
  containers:
  - name: demo-container
    image: demo-reg.io/some-org/demo-image:1.0
    env:
    - name: DEMO_ENV
      value: "Переопределенное значение"
```

(Вымышленный) образ контейнера demo-image:1.0 был собран на основе Dockerfile, который мог содержать строку ENV DEMO_ENV="Исходное значение". В этом YAML переопределяется значение переменной DEMO_ENV, и если контейнер заносит значение данной переменной в журнал, то вы увидите там Переопределенное значение.

Если среда выполнения контейнера в случае Kubernetes представляет собой совместимую с OCI утилиту, например runc, то значения из YAML-описания попадут в итоге в совместимый с OCI файл config.json. Поговорим подробнее об этих файлах контейнеров и утилитах, совместимых со стандартом OCI.

Стандарты OCI

Целью основания Open Container Initiative (OCI) (https://opencontainers.org/) было описание стандартов образов контейнеров и сред выполнения. В основе этой инициативы лежит немалый объем работы, проведенной в Docker, поэтому между происходящим в Docker и описанным в спецификациях есть много общего. В частности, OCI стремилась к стандартам, которые бы гарантировали для пользователей те же возможности, которых ожидают от Docker его пользователи, например возможность запуска образов с набором настроек по умолчанию. Спецификации OCI охватывают формат образа, в том числе сборку и распространение образов контейнеров.

Работать и просматривать образы OCI удобно с помощью утилиты Skopeo (https://oreil.ly/Rxejf), которая умеет генерировать образы в формате OCI из образов Docker:

```
$ skopeo copy docker://alpine:latest oci:alpine:latest
$ ls alpine
blobs  index.json  oci-layout
```

Но совместимые с OCI среды выполнения, такие как `runc`, не умеют работать напрямую с образами в этом формате. Сначала их необходимо распаковать в *комплект файловой системы* (filesystem bundle) (https://oreil.ly/VtJ0F). Посмотрим на пример, в котором для распаковки образа используется umoci (https://oreil.ly/Rdfab):

```
$ sudo umoci unpack --image alpine:latest alpine-bundle
$ ls alpine-bundle
config.json
rootfs
sha256_3bf9de52f38aa287b5793bd2abca9bca62eb097ad06be660bfd78927c1395651.mtree
umoci.json
$ ls alpine-bundle/rootfs
bin  etc   lib    mnt  proc  run   srv  tmp  var
dev  home  media  opt  root  sbin  sys  usr
```

Как видите, этот комплект включает каталог rootfs, содержащий дистрибутив Alpine Linux. В него также входит файл `config.json`, содержащий описание настроек среды выполнения. Она создает экземпляр контейнера на основе этой корневой файловой системы и настроек.

В случае использования Docker у вас нет прямого доступа к информации о настройках в виде файла, который можно было бы просмотреть с помощью команды `cat` или текстового редактора, но можно убедиться в его наличии, применив команду `docker image inspect`.

Конфигурация образа

Поскольку вы уже знаете из глав 3 и 4, как создаются контейнеры, теперь можно взглянуть на один из файлов `config.json`, ведь многое в нем должно выглядеть знакомо. Ниже представлен фрагмент такого файла для примера:

```
...
  "linux": {
    "resources": {
      "memory": {
        "limit": 1000000
      },
      "devices": [
        {
          "allow": false,
          "access": "rwm"
        }
      ]
    },
    "namespaces": [
      {
        "type": "pid"
      },
      {
        "type": "network"
      },
      {
        "type": "ipc"
      },
      {
        "type": "uts"
      },
      {
        "type": "mount"
      }
    ]
  }
```

Как видите, конфигурация включает описание всего, что утилита `runc` должна сделать в целях создания контейнера, включая список всех ресурсов, ограничиваемых с помощью контрольных групп, и пространств имен, которые необходимо создать.

Как вы поняли, образ состоит из двух частей: корневой файловой системы и определенной информации о настройках. Теперь посмотрим на сборку образов.

Сборка образов

Большинство людей собирают образы контейнеров с помощью команды `docker build`, которая создает образ, следуя инструкциям из Dockerfile. Прежде чем обсудить собственно сборку, я хотела бы рассказать, почему к команде `docker build` нужно относиться с осторожностью с точки зрения безопасности.

Docker сейчас разрабатывает специальный режим, который не требует полномочий суперпользователя и предназначен для решения описанных в следующем разделе проблем. Однако на момент написания данной книги он все еще находится на стадии испытаний.

Опасности команды docker build

При выполнении команды `docker` сама по себе вызываемая утилита командной строки (`docker`) делает очень немного. Она просто преобразует команду в запрос API, отправляемый далее демону Docker с помощью сокета Docker. Любая программа, имеющая доступ к сокету Docker, может отправлять демону запросы API.

Демон Docker представляет собой выполняемый на протяжении длительного времени процесс, который фактически и производит в действительности всю работу по запуску и управлению контейнерами и их образами. Как вы видели в главе 4, чтобы создать контейнер, демону необходима возможность создавать пространства имен, поэтому он должен выполняться от имени суперпользователя.

Представьте, что вы хотели бы выделить машину (или виртуальную машину) для сборки образов контейнеров и сохранения их в реестре. В случае использования подхода Docker на вашей машине должен быть запущен демон, возможности которого выходят далеко за пределы создания образов и взаимодействия с реестром. При отсутствии дополнительных средств безопасности любой пользователь, который может выполнить на этой машине команду `docker build`, может выполнить и `docker run` для запуска на этой машине любых команд, которые только пожелает.

Причем дело не только в возможности выполнять любые команды, но и в том, что если он воспользуется этими полномочиями для злонамеренных действий, то выследить виновного будет нелегко. Можно поддерживать журнал аудита определенных выполняемых пользователями действий, но — как ясно демонстрирует Дэн Уолш (Dan Walsh) (https://oreil.ly/TszBl) — в данном журнале будет фиксироваться идентификатор процесса демона, а не пользователя.

Во избежание этих рисков для безопасности лучше воспользоваться одной из альтернативных утилит сборки образов контейнеров, не полагающихся на демон Docker.

Сборка без использования демона

Одна из таких утилит — BuildKit (https://oreil.ly/jefkr) из проекта Moby, работающая в режиме без полномочий суперпользователя. (Как вы, возможно, знаете, Docker переименовал свой открытый исходный код в Moby во избежание путаницы, которая возникла бы при совпадении названий проекта и компании.) BuildKit лежит в основе вышеупомянутого экспериментального режима сборки, не требующего полномочий суперпользователя.

В числе прочих непривилегированных средств сборки — `podman` (https://podman.io/) и `buildah` (https://buildah.io/) от Red Hat. Пуджа Аббасси (Puja Abbassi) в своем блоге (https://oreil.ly/FNYY_) описывает эти утилиты и сравнивает их с `docker build`.

Утилита Bazel (https://oreil.ly/Jz_30) от компании Google умеет собирать и множество прочих типов артефактов, а не только образы контейнеров. Причем мало того, что ей не требуется Docker, так вдобавок она отличается детерминистичной генерацией образов, что позволяет воспроизводить тот же самый образ на основе того же исходного кода.

Компания Google также разработала утилиту Kaniko (https://oreil.ly/_nRoM) для выполнения сборки в кластере Kubernetes, позволяющей обходиться без доступа к демону Docker.

В числе прочих утилит для сборки контейнеров без использования демона: утилита img (https://oreil.ly/RfXtm) от Джесс Фразель (Jess Frazelle) и orca-build (https://oreil.ly/kimqz) от Алексы Сарай (Aleksa Sarai).

На момент написания данной книги неясно, есть ли среди этих утилит однозначно лучшая.

Слои образов

Вне зависимости от используемой утилиты абсолютное большинство сборок образов контейнеров описывается в Dockerfile. Он позволяет описать ряд инструкций, в результате каждой из которых либо получается слой файловой системы, либо меняется конфигурация образа. Все это прекрасно описано в документации Docker (https://oreil.ly/zGJpP), но если вы хотите узнать больше подробностей, то, возможно, вас заинтересует мое сообщение в блоге, посвященное восстановлению Dockerfile на основе образа контейнера (https://oreil.ly/2SSdJ).

Конфиденциальные данные в слоях файловой системы. Любой, кто имеет доступ к образу контейнера, имеет и доступ ко всем включенным в этот образ файлам. С точки зрения безопасности нежелательно включать в образ конфиденциальные данные, например пароли или токены. (В главе 12 я расскажу, что делать с подобными данными.)

Все слои хранятся отдельно, поэтому нужно быть осторожными, чтобы не сохранить случайно конфиденциальные данные, даже если последующий слой их удаляет. Dockerfile, представленный ниже, демонстрирует, как делать *не* нужно:

```
FROM alpine
RUN echo "top-secret" > /password.txt
RUN rm /password.txt
```

Один слой создает файл, а следующий его удаляет. Если собрать этот образ и затем его запустить, то никаких следов файла `password.txt` вы не увидите:

```
vagrant@vagrant:~$ docker run --rm -it sensitive ls /password.txt
ls: /password.txt: No such file or directory
```

Однако не позволяйте ввести себя в заблуждение — конфиденциальные данные все равно попали в образ. Чтобы убедиться в этом, экспортируйте образ в файл `tar` с помощью команды `docker save` и разархивируйте его:

```
vagrant@vagrant:~$ docker save sensitive > sensitive.tar
vagrant@vagrant:~$ mkdir sensitive
vagrant@vagrant:~$ cd sensitive
vagrant@vagrant:~$ tar -xf ../sensitive.tar
vagrant@vagrant:~/sensitive$ ls
0c247e34f78415b03155dae3d2ec7ed941801aa8aeb3cb4301eab9519302a3b9.json
552e9f7172fe87f322d421aec2b124691cd80edc9ba3fef842b0564e7a86041e
818c5ec07b8ee1d0d3ed6e12875d9d597c210b488e74667a03a58cd43dc9be1a
8e635d6264340a45901f63d2a18ea5bc8c680919e07191e4ef276860952d0399
manifest.json
```

Из содержимого вполне понятно, для чего предназначен каждый из файлов и каталогов:

- `manifest.json` — файл верхнего уровня с описанием образа. В нем указывается, в каком файле находится конфигурация (в данном случае `0c24...json`), описываются все теги для данного образа и перечисляются все слои;
- `0c24...json` — конфигурация образа (как рассказывалось ранее в этой главе);
- каждый из каталогов соответствует одному из слоев, составляющих корневую файловую систему образа.

Конфигурация включает историю команд, выполнявшихся при формировании данного контейнера. Как можно видеть, в данном случае конфиденциальные данные отображаются на шаге выполнения команды `echo`:

```
vagrant@vagrant:~/sensitive$ cat 0c247*.json | jq '.history'
[
  {
    "created": "2019-10-21T17:21:42.078618181Z",
    "created_by": "/bin/sh -c #(nop) ADD
    file:fe1f09249227e2da2089afb4d07e16cbf832eeb804120074acd2b8192876cd28 in / "
  },
  {
    "created": "2019-10-21T17:21:42.387111039Z",
    "created_by": "/bin/sh -c #(nop)  CMD [\"/bin/sh\"]",
    "empty_layer": true
  },
  {
    "created": "2019-12-16T13:50:43.914972168Z",
    "created_by": "/bin/sh -c echo \"top-secret\" > /password.txt"
  },
  {
    "created": "2019-12-16T13:50:45.085349285Z",
    "created_by": "/bin/sh -c rm /password.txt"
  }
]
```

Внутри каталога каждого из слоев находится еще один файл `tar` с содержимым файловой системы на этом слое. Можно легко извлечь файл `password.txt` из каталога соответствующего слоя:

```
vagrant@vagrant:~/sensitive$ tar -xf 55*/layer.tar
vagrant@vagrant:~/sensitive$ cat password.txt
top-secret
```

Из этого ясно, что, даже если последующий слой и удаляет его, любой файл, когда-либо существовавший на любом из слоев, можно легко получить путем

распаковки образа. Не включайте в слои ничего, что не готовы продемонстрировать любому, у кого есть доступ к образу.

Ранее в этой главе вы видели, что можно найти внутри образа контейнера, совместимого с OCI. А теперь вы знаете, что происходит, когда эти образы собираются на основе Dockerfile. Обсудим, как образы хранятся.

Хранение образов

Образы сохраняются в реестрах контейнеров. Если вы используете Docker, то, вероятно, вам случалось использовать реестр Docker Hub (https://hub.docker.com/). Если же вы работаете с контейнерами с помощью облачных сервисов, то, возможно, знакомы с одним из их реестров — например, Amazon Elastic Container Registry или Google Container Registry. О сохранении образа в реестре обычно говорят как о *помещении* (буквально — «вталкивании», push), а о получении образа из реестра — как об *извлечении* (буквально — «вытаскивании», pull).

На момент написания данной книги OCI работала над проектом Distribution Spec (https://oreil.ly/3Jvl7) — спецификациями, содержащими описание интерфейсов для взаимодействия с реестром контейнеров, в котором хранятся образы. И хотя работа над ними еще не завершена, она основывается на технологиях уже существующих реестров контейнеров.

Все слои хранятся по отдельности, в виде больших двоичных объектов в реестре, идентифицируемых по хешу их содержимого. В целях экономии места желательно хранить только одну копию каждого объекта, на которую могут ссылаться различные образы. В реестре также хранится *манифест* образа, в котором указывается набор больших двоичных объектов слоев, составляющих образ. Хеш манифеста образа служит уникальным идентификатором всего образа и называется *дайджестом* (digest) образа. В случае повторной сборки образа и изменения любых его характеристик этот хеш также поменяется.

При использовании Docker просмотреть дайджесты образов, хранящихся локально на машине, можно с помощью следующей команды:

```
vagrant@vagrant:~$ docker image ls --digests
REPOSITORY   TAG      DIGEST              IMAGE ID       CREATED      SIZE
nginx        latest   sha256:50cf...8566  231d40e811cd   2 weeks ago  126MB
```

При помещении образа в реестр и извлечении из него можно использовать дайджест, чтобы ссылаться точно на конкретную сборку, но это не единственный способ сослаться на образ. Рассмотрим различные способы идентификации образов контейнеров.

Идентификация образов

Первая часть ссылки на образ — URL реестра, в котором он хранится. (Если адрес реестра опущен, значит, образ хранится либо на локальной машине, либо в Docker Hub, в зависимости от контекста команды.)

Следующая часть ссылки на образ представляет собой название учетной записи пользователя или организации — владельца этого образа. За ним следует имя образа, а далее — либо дайджест-идентификатор его содержимого, либо удобный для восприятия человека тег.

Вместе получаются адреса примерно следующего вида:

```
<URL реестра>/<Имя пользователя или название организации>/<репозиторий>
@sha256:<дайджест>
<URL реестра>/<Имя пользователя или название организации>/<репозиторий>:<тег>
```

Если URL реестра пропущен, то по умолчанию используется адрес Docker Hub, `docker.io`. На рис. 6.1 показан пример, как образ отображается в Docker Hub.

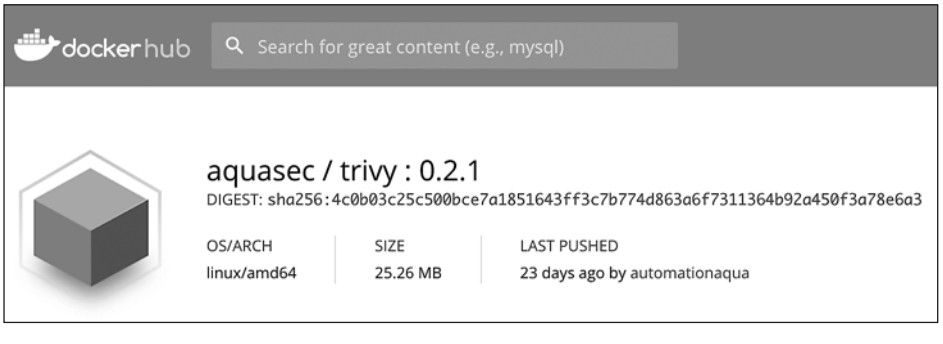


Рис. 6.1. Пример образа в Docker Hub

Извлечь образ можно с помощью одной из следующих команд:

```
vagrant@vagrant:~$ docker pull aquasec/trivy:0.2.1
vagrant@vagrant:~$ docker pull aquasec/
trivy:sha256:4c0b03c25c500bce7a1851643ff3c7b774d863a6f7311364b92a450f3a78e6a3
```

Для человека неудобно ссылаться на образ по хешу, поэтому часто используются *теги* — произвольные метки, присваиваемые образам. Одному образу можно присвоить произвольное число тегов, а любой тег можно перенести с одного образа на другой. С помощью тегов часто указывают версии содержащегося в образе ПО — как в приведенном выше примере с версией 0.2.1.

Теги можно переносить с образа на образ, поэтому обращение к образу по тегу не гарантирует одинаковый результат сегодня и завтра. А вот при ссылке по хешу вы всегда получаете один и тот же образ, поскольку хеш определяется содержимым образа. Любое изменение образа приводит к изменению хеша.

Именно такой эффект обычно и требуется. Например, можно ссылаться на образ с помощью тега, отражающего старший и младший номер версии в схеме семантического контроля версий. При выпуске новой версии специалисты, которые занимаются сопровождением, должны маркировать эту версию тегом с номером версии, соответствующим старшим и младшим, чтобы при следующем извлечении образа вы получили актуальную версию.

Однако бывают случаи, когда необходима уникальная ссылка на образ. Например, представьте сканирование образов на уязвимости (которое мы обсудим в главе 7). Контроллер допуска проверяет, чтобы развертываемые образы прошли этап сканирования на уязвимости, а потому должен сверяться с базой уже просканированных образов. В случае ссылки на образы по тегу на информацию полагаться нельзя, ведь точно не известно, был ли образ изменен и не требует ли повторного сканирования.

Теперь, когда вы знаете, как хранятся образы, обратимся к связанным с ними вопросам безопасности.

Безопасность образов

Основной вопрос безопасности образов — их целостность, то есть гарантия того, что используются именно те образы, которые должны применяться. Злоумышленник, которому удалось запустить какой-то нежелательный образ в развернутой системе, сможет выполнять любой нужный ему код. Технологический процесс содержит множество потенциально слабых звеньев, от сборки и сохранения образа до его запуска, как показано на рис. 6.2.

Разработчики приложений могут влиять на безопасность через создаваемый ими код. Статические и динамические средства анализа, проверка коллегами и тестирование помогают находить уязвимости, появившиеся во время

разработки. И к контейнеризованным приложениям это относится в той же степени, что и к обычным. Но поскольку данная книга посвящена контейнерам, то посмотрим на уязвимости, которые могут появиться на этапе сборки образа контейнера.

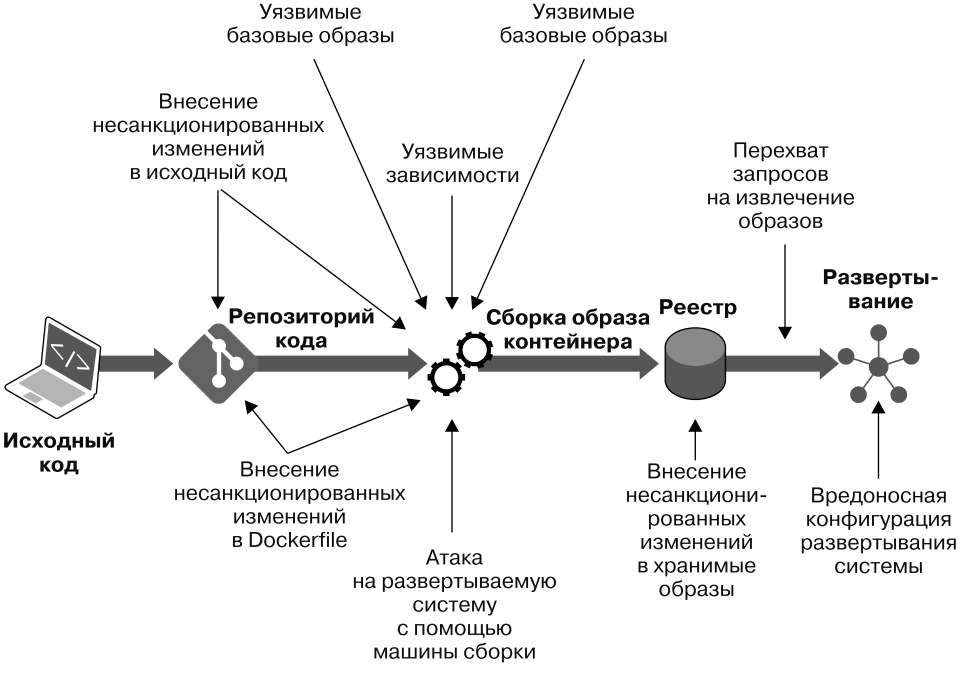


Рис. 6.2. Векторы атак на образы

Безопасность этапа сборки

На этапе сборки Dockerfile превращается в образ контейнера. Количество возможных рисков для безопасности на данном этапе весьма велико.

Происхождение Dockerfile

Инструкции для сборки образа контейнера берутся из Dockerfile. Каждый этап сборки включает выполнение одной из этих инструкций, и если злоумышленник сумеет изменить Dockerfile, то сможет предпринять вредоносные действия, в том числе:

- ❑ включить в образ вредоносное программное обеспечение или утилиты для майнинга криптовалют;
- ❑ получить доступ к секретным данным сборки;
- ❑ получить полную информацию о топологии сети, доступной из инфраструктуры сборки;
- ❑ атаковать хост-компьютер, на котором производится сборка.

Повторим очевидное: Dockerfile (как и любой другой исходный код) должен располагать средствами контроля доступа, чтобы иметь защиту от внесения злоумышленниками вредоносных шагов в сборку.

Содержимое Dockerfile также сильно влияет на безопасность полученного в результате сборки образа контейнера. Рассмотрим несколько мер, которые можно предпринять в Dockerfile на практике для повышения безопасности образа.

Практические рекомендации по безопасности Dockerfile

Все эти рекомендации повышают безопасность образа и сокращают шансы на то, что контейнеры, запущенные из этого образа, будут взломаны злоумышленником.

- ❑ *Базовые образы*. Первая строка Dockerfile представляет собой инструкцию `FROM`, в которой задается базовый образ для создания нового образа.
  - Должна ссылаться на образ из доверенного реестра (см. раздел «Безопасность хранилищ образов» на с. 112).
  - Произвольно выбранные сторонние базовые образы могут включать вредоносный код, поэтому некоторые организации требуют использования только заранее одобренных («золотых») базовых образов.
  - Чем меньше базовый образ, тем менее вероятно наличие в нем ненужного кода, а значит, меньше и поверхность атаки. Лучше создать образ с нуля (с совершенно пустого образа, подходящего для автономных исполняемых файлов) или использовать минимальный базовый образ, например distroless (https://oreil.ly/kaUEc). Преимущество меньших образов, помимо всего прочего, заключается еще и в более быстрой передаче по сети.
  - Задумайтесь, что использовать для ссылки на базовый образ: тег или дайджест. Во втором случае воспроизводимость сборки будет выше,

но снизится вероятность получения новых версий базового образа с обновлениями безопасности. (С другой стороны, вы должны получать недостающие обновления с помощью сканирования собранного образа на уязвимости.)

- *Многоэтапная сборка* (https://oreil.ly/k34z-) — способ исключения ненужного содержимого из итогового образа. На начальном этапе в образ могут включаться все пакеты и набор программных средств, необходимых для сборки образа, но во время выполнения многие из этих инструментов не нужны. Приведу такой пример: чтобы создать исполняемую программу, написанную на языке Go, нужен компилятор Go. Но контейнеру, в котором выполняется эта программа, доступ к компилятору Go не требуется. В данном примере имеет смысл разбить сборку на несколько этапов: на одном производится компиляция и создается двоичный исполняемый файл; а у следующего этапа просто есть доступ к этому автономному исполняемому файлу. В итоге поверхность атаки развертываемого образа значительно уменьшается; помимо пользы для безопасности, сам образ также будет меньше, а значит, сократится время, требуемое на его извлечение.

В блоге корпорации Capital One можно найти несколько примеров многоэтапной сборки (https://oreil.ly/CRMuY), демонстрирующих, что можно даже выполнять тесты на различных шагах многоэтапной сборки, не влияя при этом на содержимое итогового образа.

- *Несуперпользователь*. Инструкция `USER` в Dockerfile указывает, что учетная запись пользователя, с помощью которой по умолчанию запускаются контейнеры, основанные на этом образе, не является суперпользователем. Укажите эту опцию во всех своих Dockerfile, если не хотите, чтобы ваши контейнеры запускались от имени суперпользователя.
- *Команды RUN*. Отмечу для полной ясности: команда `RUN` Dockerfile позволяет выполнить любую произвольную команду. Если злоумышленнику удастся внести изменения в Dockerfile, содержащий настройки безопасности по умолчанию, то он сможет выполнять *любой код, какой только пожелает*. Если у вас есть хотя бы минимальные причины не доверять людям, запускающим сборки контейнеров в вашей системе, то скажу прямо: вы фактически предоставили им полномочия для удаленного выполнения кода. Убедитесь, что полномочия на редактирование Dockerfile

есть только у доверенных членов вашей команды, и внимательно следите за кодом, меняющим эти настройки. Возможно, имеет смысл даже производить проверку или заносить информацию в журнал аудита при внесении любых новых или изменении существующих команд `RUN` в Dockerfile.

- *Точки монтирования томов.* Каталоги хоста часто монтируются к контейнеру с помощью точек монтирования томов, особенно для демонстрации и тестов. Как вы увидите в главе 9, важно проверять, чтобы Dockerfile не монтировали к контейнеру каталоги с конфиденциальными данными, например /etc или /bin.
- *Не включайте в Dockerfile конфиденциальные данные.* Более подробно конфиденциальные данные и секреты мы обсудим в главе 12, а пока просто учтите, что включение учетных данных, паролей и прочих секретных данных в образ упрощает злоумышленникам доступ к ним.
- *Избегайте исполняемых файлов с установленным битом* `setuid`. Как обсуждалось в главе 2, желательно избегать включения в образ исполняемых файлов с установленным битом `setuid`, поскольку они потенциально могут открывать путь к повышению полномочий.
- *Избегайте ненужного кода.* Чем меньше в контейнере кода, тем меньше поверхность атаки. Избегайте включения в образ пакетов, библиотек и исполняемых файлов, за исключением абсолютно необходимых. По той же причине желательно использовать в качестве базового образа пустой образ или один из вариантов distroless, ведь это резко уменьшит объем кода — а значит, и уязвимого кода — в вашем образе.
- *Включайте все, что действительно нужно контейнеру.* Если в предыдущем пункте я призывала вас исключать из сборки лишний код, то здесь вывод иной: включайте в нее все, что требуется вашему приложению для работы. Если разрешить контейнеру устанавливать дополнительные пакеты во время выполнения, то как проверить, что все они не содержат вредоносного кода? Гораздо лучше выполнять установку и проверку всех пакетов при сборке образа контейнера и создать неизменяемый образ. Подробнее о том, почему этот вариант предпочтительнее, рассказывается в подразделе «Неизменяемые контейнеры» на с. 124.

Следуя этим рекомендациям, вы сможете создавать образы, которые сложнее взломать. А теперь обсудим риск того, что злоумышленник найдет слабые места в системе сборки контейнеров.

Атаки на машину сборки

Компьютер, на котором производится сборка, интересует нас по двум основным причинам.

- Сможет ли злоумышленник, взломавший машину сборки и имеющий возможность выполнять на ней код, добраться до прочих частей системы? Как вы видели в подразделе «Опасности команды docker build» на с. 101, имеет смысл использовать утилиту сборки, не требующую привилегированного процесса-демона.
- Может ли злоумышленник повлиять на результаты сборки так, что будут собраны, а потом и запущены образы с вредоносным кодом? Последствия любого несанкционированного доступа, который влияет на инструкции Dockerfile или инициирует сборку непредусмотренных образов, могут оказаться катастрофическими. Например, если злоумышленник может внести изменения в собираемый код, то может и включить лазейки в контейнеры, работающие в продакшене.

А поскольку машины сборки генерируют код, который в дальнейшем будет выполняться в кластере, работающем в промышленной эксплуатации, надежная защита их от атак столь же важна, как и для самого кластера. Старайтесь уменьшить поверхность атаки, удаляя ненужные утилиты из машин сборки. Ограничивайте прямой доступ пользователей к этим машинам и защищайте их от несанкционированного доступа по сети, задействуя VPC и брандмауэры.

Имеет смысл также выполнять сборку на отдельной (от работающего в промышленной эксплуатации кластера) машине или кластере машин, чтобы ограничить возможный эффект атаки на хост из сборки. Ограничьте доступ по сети и к облачным сервисам с этого хоста, чтобы не открыть злоумышленнику доступ к остальным элементам развертываемой системы.

Безопасность хранилищ образов

Собранный образ необходимо сохранить в реестре. Если злоумышленник сможет заменить или модифицировать образ, то сможет и выполнять любой код, какой пожелает.

Запуск собственного реестра

Многие организации поддерживают собственные реестры или используют управляемые реестры поставщиков облачных сервисов и позволяют применять лишь образы из этих доверенных реестров. Собственный реестр (или свой экземпляр управляемого реестра) предоставляет больше возможностей для контроля над тем, кто может помещать и извлекать образы. Кроме того, он сокращает вероятность DNS-атаки, позволяющей злоумышленнику подделать адрес реестра. Если же реестр располагается в виртуальном частном облаке (VPC), то тем более маловероятно, что злоумышленнику это удастся.

Желательно также позаботиться об ограничении прямого доступа к носителям информации, на которых хранится реестр. Например, в работающем в AWS реестре для хранения образов может использоваться S3, а права доступа в корзине (-ах) S3 должны быть ограничены так, чтобы злоумышленник не мог напрямую обращаться к данным хранимых образов.

Подписывание образов

Подписывание образов означает связывание с образом криптографической сущности (аналогично подписыванию сертификатов, о котором мы поговорим в главе 11).

Система подписывания образов весьма непроста, поэтому вряд ли вы захотите реализовывать ее самостоятельно. Многие реестры реализуют подписывание образов на основе реализации Notary спецификации TUF (The Update Framework) (https://oreil.ly/fMD6d). Считается, что использовать Notary непросто, и замечательно, что на момент написания данной книги большинство основных поставщиков облачных сервисов (если вообще не все) вовлечены в версию 2 этого проекта.

Еще один проект, предназначенный для решения проблем цепи поставок образов контейнеров, — in-toto (https://in-toto.io/). Этот фреймворк гарантирует полноценное выполнение всех ожидаемых шагов сборки в должном порядке и должными людьми и получение правильного результата при правильных входных данных. Весь набор шагов сборки соединяется в цепочку, а in-toto на протяжении всех шагов процесса отвечает за связанные с безопасностью метаданные. В результате ПО, работающее в промышленной эксплуатации, достоверно выполняет тот же код, который разработчик отправил со своего ноутбука.

А что, если вам понадобится сторонний образ контейнера либо непосредственно в качестве приложения, либо в качестве базового образа для сборки? Можно взять подписанный образ непосредственно от поставщика ПО или из другого доверенного источника, возможно протестировав образ перед сохранением в своем реестре.

Безопасность развертывания образов

Основной вопрос безопасности на этапе развертывания — как гарантировать извлечение и запуск того образа, который нужен, хотя в процессе так называемого *контроля допуска* (admission control) могут участвовать и дополнительные проверки.

Развертывание правильного образа

Как вы видели в разделе «Идентификация образов» на с. 106, теги образов контейнеров не являются неизменяемыми — они могут переноситься на другие версии того же образа. Если же ссылаться на образ по дайджесту, а не тегу, то можно быть уверенными, что версия образа именно та, которая нужна. Однако обычно достаточно, чтобы система сборки присваивала образам теги на основе семантического контроля версий, если это строго соблюдается, поскольку не нужно будет обновлять ссылки на образы при каждом мелком обновлении.

Если же ссылаться на образы по тегу, то необходимо всегда извлекать последнюю версию перед запуском, на случай возможного обновления. К счастью, это требует не слишком большого расхода ресурсов, поскольку сначала извлекается манифест, так что можно извлекать слои образа только в случае их изменений.

В Kubernetes это поведение задается параметром `imagePullPolicy`. Задавать стратегию извлечения образов всякий раз не нужно, если вы ссылаетесь на них по дайджесту, поскольку любое обновление означает необходимость изменения дайджеста.

В зависимости от профиля риска может понадобиться также проверить происхождение образа, проанализировав его подпись, за которую отвечает утилита наподобие вышеупомянутой Notary.

Вредоносная конфигурация развертывания системы

В случае использования механизма координации контейнеров составляющие каждое приложение контейнеры обычно описываются в файлах конфигурации — в YAML-файлах в случае Kubernetes, например. Проверять происхождение этих файлов не менее важно, чем проверять сами образы.

Очень тщательно проверяйте скачанные из Интернета YAML, прежде чем выполнять их в кластере, предназначенном для промышленной эксплуатации. Учтите, что даже крошечные отличия — например, замена одного символа в URL реестра — могут привести к запуску вредоносного образа в развернутой вами системе.

Контроль допуска

Это еще один вопрос, выходящий за рамки сугубо безопасности контейнеров, но мне хотелось бы познакомить вас с контролем допуска, поскольку он отлично подходит для обоснования многих идей, обсуждавшихся выше в данной главе.

Контроллер допуска может выполнять проверки непосредственно перед развертыванием ресурса в кластере. В Kubernetes контроль допуска позволяет проверять соответствие ресурсов любых типов заданным стратегиям, но в этой главе мы рассмотрим только контроллер допуска, проверяющий допустимость контейнера, в основе которого лежит конкретный образ контейнера. Если проверки контроля допуска не пройдены, то контейнер не запускается.

Контроллер допуска способен проверить несколько жизненно важных вопросов безопасности образа контейнера, прежде чем превратить последний в работающий контейнер.

- ❑ Был ли образ просканирован на предмет уязвимостей/вредоносного ПО/прохождения прочих проверок стратегий?
- ❑ Получен ли образ из доверенного реестра?
- ❑ Подписан ли образ?
- ❑ Одобрен ли образ для использования?
- ❑ Выполняется ли образ от имени суперпользователя?

Эти проверки гарантируют, что никто не сможет обойти проверки на более ранних этапах системы. Например, нет смысла вводить в конвейер сборки

образа сканирование на уязвимости, если оказывается, что у злоумышленников есть возможность указывать инструкции развертывания, ссылающиеся на непросканированные образы.

GitOps и безопасность развертывания

GitOps — методология, при которой вся конфигурационная информация о состоянии системы хранится в системе контроля исходного кода, как и исходный код приложения. Если пользователю нужно внести изменения в действующую систему, то он не выполняет команды напрямую, а вносит в репозиторий желаемое состояние в виде кода (например, YAML-файлов для Kubernetes). А автоматизированная система (так называемый GitOps-оператор) обеспечивает обновление системы в соответствии с последним состоянием, описанным в системе контроля исходного кода.

Это положительно влияет на безопасность. Пользователям больше не нужен прямой доступ к работающей системе, поскольку все производится опосредованно, с помощью системы контроля исходного кода (обычно Git, как понятно из названия). Как показано на рис. 6.3, учетные данные пользователя дают ему доступ к системе контроля исходного кода, но только у автоматизированного GitOps-оператора есть полномочия для модификации работающей системы. А поскольку Git фиксирует все изменения, то каждой операции соответствует запись в журнале аудита.

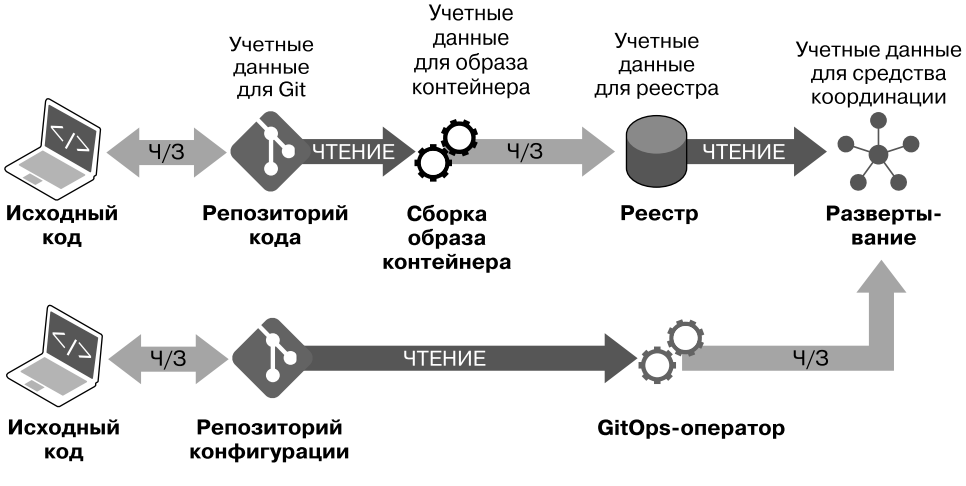


Рис. 6.3. GitOps

Резюме

Из этой главы вы узнали, что среде выполнения контейнера необходима корневая файловая система и определенная информация о настройках. Конфигурацию можно переопределить с помощью параметров, передаваемых в среду выполнения или указываемых в YAML Kubernetes. Некоторые из этих настроек конфигурации влияют на безопасность приложения. Кроме того, если не следовать практическим рекомендациям, представленным в подразделе «Практические рекомендации по безопасности Dockerfile» на с. 109, открывается масса путей для попадания в образы контейнеров вредоносного кода.

Стандартные широко используемые средства сборки образов контейнеров на момент написания данной книги обычно выполняются от имени суперпользователя и содержат множество слабых мест, которые необходимо защитить от атаки. Однако есть более безопасные альтернативные средства сборки образов, как уже существующие, так и разрабатываемые в настоящее время.

На этапе развертывания образа средства координации и утилиты безопасности позволяют производить контроль допуска, а это отличная возможность для выполнения проверок безопасности этих образов.

Образы контейнеров инкапсулируют код приложения, а также все зависимости от сторонних пакетов и библиотек. В следующей главе мы обсудим, что эти зависимости могут также включать уязвимости, и изучим инструменты для обнаружения и устранения этих уязвимостей.

ГЛАВА 7

Программные уязвимости в образах контейнеров

Исправление уязвимостей в ПО — с давних пор важный аспект сопровождения безопасности развернутого кода. В мире контейнеров эта проблема не теряет своей значимости, но, как вы увидите в данной главе, процесс исправления был полностью переосмыслен. Но сначала посмотрим, что такое программные уязвимости и как публикуется и отслеживается информация о них.

Исследования уязвимостей

Уязвимость — это известный изъян во фрагменте программного обеспечения, с помощью которого злоумышленник может произвести какие-либо вредоносные действия. В общем случае чем сложнее фрагмент ПО, тем вероятнее, что в нем есть изъяны, такие, что по крайней мере некоторыми из них может воспользоваться злоумышленник.

А когда в широко используемом программном обеспечении существует уязвимость, злоумышленники могут задействовать ее повсюду, где развернуто это ПО. Как следствие, существует целая сфера исследований, посвященная поиску и публикации информации об уязвимостях в общедоступном ПО, особенно в пакетах для операционных систем и библиотеках языков программирования. Наверное, вы слышали о некоторых наиболее катастрофических уязвимостях, таких как Shellshock, Meltdown и Heartbleed, имеющих не только название, но иногда даже и эмблему. Но эти рок-звезды в мире уязвимостей составляют лишь малую толику из тысяч проблем, о которых сообщается ежегодно.

После того как уязвимость будет выявлена, начинается «забег» по скорейшей публикации исправлений, чтобы пользователи могли задействовать их прежде, чем уязвимость задействуют злоумышленники. Если сразу же

оповещать широкую публику о новых проблемах в ПО, то перед злоумышленниками откроются широкие возможности по использованию этих проблем. Во избежание этого была принята методология ответственного подхода к раскрытию проблем безопасности. Обнаружив уязвимость, аналитик в области безопасности связывается с разработчиком или поставщиком соответствующего ПО и договаривается о сроке, по истечении которого может публиковать полученные результаты. Эти сроки давят на поставщика в положительном смысле, заставляя его как можно быстрее выпустить исправление, ведь как для него, так и для его пользователей будет лучше, если исправление будет доступно до публикации результатов.

Каждой новой проблеме присваивается уникальный идентификатор, начинающийся с букв CVE (Common Vulnerabilities and Exposures — распространенные уязвимости и дефекты), за которыми следует год. Например, уязвимость Shellshock была открыта в 2014 году и официально называется CVE-2014-6271. Организация, которая заведует этими идентификаторами, называется MITRE (https://mitre.org/). Она курирует несколько комитетов по нумерации CVE (CVE Numbering Authority, CNA), имеющих право присваивать идентификаторы CVE в определенных рамках. Некоторые крупные поставщики программного обеспечения — например, Microsoft, Red Hat и Oracle — являются CNA с правом присваивать идентификаторы уязвимостям в их собственных программных продуктах. GitHub стал CNA в конце 2019 года.

Эти идентификаторы CVE используются в Национальной базе уязвимостей (National Vulnerability Database, NVD) (https://nvd.nist.gov/) для отслеживания пакетов и версий программ, затронутых каждой из уязвимостей. На первый взгляд кажется, что на этом все и заканчивается — есть список всех затронутых версий пакетов, так что если у вас установлена одна из этих версий, значит, ваша система уязвима для атаки. К сожалению, не все так просто, поскольку в зависимости от используемого дистрибутива Linux может существовать исправленная версия соответствующего пакета.

Уязвимости, исправления и дистрибутивы

Возьмем для примера Shellshock — критически важную уязвимость, затронувшую пакет `bash` GNU. На странице NVD, посвященной CVE-2014-6271 (https://oreil.ly/XGgEb), перечислен длинный список уязвимых версий, от 1.14.0 до 4.3. Если вы работаете на очень старой версии Ubuntu, 12.04, и обнаружили, что версия bash на вашем сервере — 4.2-2ubuntu2.2, то можете подумать,

что она уязвима, поскольку основана на `bash` 4.2, включенной в список NVD для уязвимости Shellshock.

На самом же деле согласно инструкции по безопасности Ubuntu для этой уязвимости (https://oreil.ly/IEUqF) к данной конкретной версии уже применено исправление от данной уязвимости, и она безопасна. Дело в том, что специалисты по сопровождению Ubuntu решили, что лучше задействовать исправление уязвимости и опубликовать исправленную версию, чем заставлять всех работающих на 12.04 обновляться до совершенно новой младшей версии `bash`.

Чтобы составить полную картину того, уязвимы ли установленные на сервере пакеты, необходимо заглянуть не только в NVD, но и в инструкции по безопасности для конкретного дистрибутива.

До сих пор в этой главе шла речь о пакетах (например, `bash` в предыдущем примере), распространяемых в двоичном виде с помощью таких систем управления пакетами, как `apt`, `yum`, `rpm` или `apk`. Эти пакеты используются всеми приложениями, присутствующими в файловой системе, и данный факт приводит к бесчисленным проблемам на серверах и виртуальных машинах: приложение может зависеть от определенной версии пакета, несовместимой с другим приложением, которое должно работать на той же машине. Эта проблема управления зависимостями — одна из тех, в решении которых могут помочь контейнеры благодаря отдельной корневой файловой системе каждого из них.

Уязвимости уровня приложения

Встречаются и уязвимости на уровне приложений. Большинство приложений используют сторонние библиотеки, устанавливаемые обычно с помощью системы управления пакетами, ориентированной на конкретный язык. В Node.js применяется `npm`, в Python — `pip`, в Java — `Maven` и т. д. Сторонние библиотеки, установленные с помощью этих утилит, — еще один потенциальный источник уязвимостей.

В компилируемых языках, например в Go, C и Rust, сторонние зависимости либо устанавливаются в виде совместно используемых библиотек, либо подключаются к исполняемому файлу во время сборки.

У автономных двоичных исполняемых файлов по определению (в соответствии с эпитетом «автономный») нет внешних зависимостей. У них могут быть зависимости от сторонних библиотек, встроенные тем не менее в эти исполняемые файлы. В таком случае можно создать образ контейнера на основе пустого базового образа, содержащего только нужный двоичный исполняемый файл.

Если у приложения нет зависимостей, то его нельзя просканировать на предмет известных уязвимостей. Но оно все равно может включать изъяны, делающие его уязвимым для злоумышленников, которые мы обсудим в разделе «Уязвимости нулевого дня» на с. 132.

Управление рисками, связанными с уязвимостями

Решение проблемы уязвимостей программного обеспечения — один из важных аспектов управления рисками. Любое развертываемое нетривиальное ПО, скорее всего, содержит некоторое количество уязвимостей, и существует риск атаки на приложение с их помощью. Чтобы управлять этим риском, необходимо идентифицировать существующие уязвимости и оценить степень их серьезности, распределить по степени приоритетности и наладить процессы их исправления либо снижения негативных последствий.

Сканеры уязвимостей автоматизируют процесс идентификации уязвимостей и предоставляют информацию о степени серьезности проблем, а также версиях пакетов ПО, в которых они были исправлены (если исправление было опубликовано).

Сканирование на уязвимости

Если поискать в Интернете, то можно найти огромное множество утилит для сканирования на уязвимости, охватывающих разнообразные методики, включая такие утилиты сканирования портов, как `nmap` и `nessus`, предназначенные для внешнего поиска уязвимостей в системе, работающей в продакшене. Это ценный подход, но в данной главе нас больше интересуют

утилиты, с помощью которых можно найти уязвимости в ПО, установленном в корневой файловой системе.

Чтобы идентифицировать присутствующие в системе уязвимости, сначала необходимо установить, какое программное обеспечение в ней есть. Существует несколько различных механизмов установки ПО:

- корневая файловая система, основанная на корневой файловой системе дистрибутива Linux, в которой могут быть уязвимости;
- установленные через систему управления пакетами Linux (например, `rpm` или `apk`) системные пакеты, а также ориентированные на конкретный язык программирования пакеты, установленные с помощью таких утилит, как `pip` и `RubyGems`;
- программное обеспечение, установленное непосредственно с помощью `wget`, `curl` или даже FTP.

Некоторые сканеры зависимостей запрашивают у системы управления пакетами список установленного программного обеспечения. В случае использования одной из подобных утилит следует избегать установки ПО напрямую, поскольку оно не будет просканировано на предмет уязвимостей.

Установленные пакеты

Как вы видели в главе 6, любой образ контейнера может включать дистрибутив Linux, возможно, с какими-либо уже установленными пакетами, помимо кода приложения. Может быть запущено много экземпляров каждого контейнера, причем каждый — со своей копией файловой системы образа контейнера, включая все потенциально уязвимые пакеты, содержавшиеся в нем. На рис. 7.1 показана подобная ситуация: два экземпляра контейнера X и один экземпляр контейнера Y. Кроме того, показаны дополнительные пакеты, установленные непосредственно на хост-компьютере.

Установка пакетов непосредственно на хосте — обычное дело, на самом деле именно их исправления системные администраторы всегда должны были устанавливать из соображений безопасности. Для этого нередко администраторы подключались к каждому хосту по SSH и устанавливали исправленный пакет. В эпоху же ориентации на облачные сервисы это не приветствуется, поскольку изменение состояния машины вручную подобным образом озна-

чает невозможность автоматического восстановления ее в том же состоянии. Вместо этого лучше либо создать новый образ машины с обновленными пакетами, либо обновить сценарии автоматизации, служащие для подготовки образов, чтобы новые установки включали обновленные пакеты.

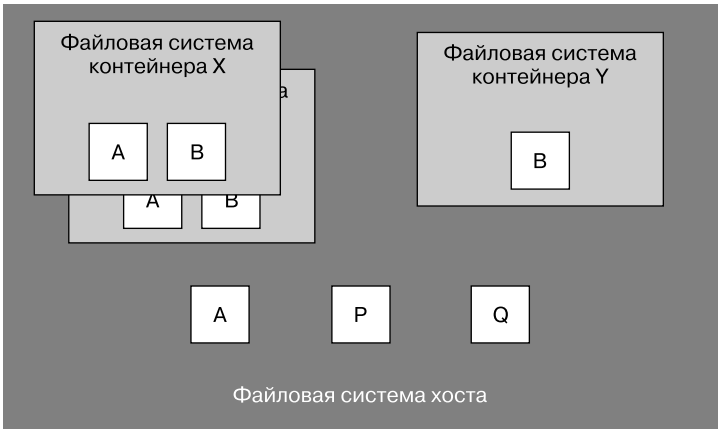


Рис. 7.1. Пакеты на хосте и в его контейнерах

Сканирование образов контейнеров

Чтобы знать, не запущены ли в развернутой системе контейнеры с уязвимым программным обеспечением, необходимо просканировать в них все зависимости. Для этого существует несколько различных подходов.

Представьте утилиту, сканирующую все запущенные на хосте (или в системе, развернутой на нескольких хостах) контейнеры. В современных системах, развертываемых в облаках, можно нередко видеть сотни экземпляров контейнеров, созданных на основе одного образа контейнера, и просматривать одни и те же зависимости сотни раз — очень неэкономично. Гораздо более рациональным будет просканировать образ контейнера, на котором основаны все эти контейнеры.

Однако данный подход требует, чтобы в контейнерах присутствовало только программное обеспечение из образа контейнера и ничего более. Работающий в каждом из контейнеров код должен быть *неизменяемым* (immutable). Посмотрим, каковы преимущества таких неизменяемых контейнеров.

Неизменяемые контейнеры

(Обычно) ничто не мешает контейнеру после запуска скачивать дополнительное программное обеспечение и размещать его в своей файловой системе. И действительно, на заре развития контейнеров подобный паттерн встречался довольно часто, поскольку считался удобным способом обновления ПО контейнера до последних версий, который не нуждался в повторной сборке образа контейнера. Если до сих пор эта идея не приходила вам в голову, то забудьте ее сразу же, она считается исключительно неудачной по многим причинам, включая представленные ниже.

- Если контейнер скачивает код во время выполнения, то в различных экземплярах контейнера могут работать разные версии этого кода, причем отследить, в каком экземпляре работает та или иная версия, будет непросто. В отсутствие сохраненной версии кода из данного контейнера будет сложно (или даже невозможно) воссоздать его идентичную копию, что создаст проблемы при попытках воспроизвести проблемы, возникшие при эксплуатации.
- Труднее контролировать и гарантировать происхождение программного обеспечения, работающего в каждом из контейнеров, если оно может скачиваться в любой момент времени и откуда угодно.
- Сборку образа контейнера и сохранение его в реестре можно с легкостью автоматизировать в конвейере CI/CD. Совсем нетрудно также добавить в тот же конвейер дополнительные проверки безопасности — например, сканирование на уязвимости или проверку цепочки поставок программного обеспечения.

Во многих системах, развертываемых в промышленной эксплуатации, контейнеры считаются неизменяемыми, просто в порядке практической рекомендации, но никаких мер для обеспечения этого не предпринимается. Существуют утилиты, способные автоматически обеспечить неизменяемость контейнеров с помощью запрета на запуск в контейнере отсутствовавших там на момент сканирования исполняемых файлов. Данная методика, обсуждаемая подробнее в главе 13, называется *предотвращением отклонений* (drift prevention).

Еще один способ реализовать неизменяемость — запустить контейнер с файловой системой, предназначенной только для чтения. Если коду приложения требуется доступ к открытому для записи локальному хранилищу, то можно смонтировать открытую для записи временную файловую систему. Здесь

может понадобиться вносить изменения в приложение, чтобы оно записывало данные лишь в эту временную файловую систему.

Если контейнеры неизменяемые, то достаточно просканировать только образ, чтобы найти все уязвимости, которые могут встретиться во всех контейнерах, основанных на нем. Но, к сожалению, просканировать их однократно недостаточно. Обсудим, почему необходимо регулярное сканирование.

Регулярное сканирование

Как обсуждалось в начале данной главы, по всему миру есть множество исследователей, занимающихся безопасностью, которые ищут неизвестные ранее уязвимости в существующем коде. Иногда эти люди находят проблемы, существовавшие уже многие годы. Один из самых ярких примеров — HeartBleed — критическая уязвимость в широко используемом пакете OpenSSL, основанная на проблеме в потоке heartbeat-запросов и ответов, поддерживающем TLS-соединение в активном состоянии. Эта уязвимость, обнаруженная в апреле 2014-го, позволяла злоумышленнику отправлять heartbeat-запрос, составленный специальным образом, на небольшое количество данных в большом буфере. Отсутствие проверки длины в коде OpenSSL приводило к тому, что в ответ отправлялось небольшое количество данных, а остальная часть буфера заполнялась содержимым используемой в текущий момент памяти, которое могло включать конфиденциальные данные, попадавшие таким образом к злоумышленнику. Было установлено, что эта уязвимость стала причиной серьезных утечек данных, связанных с потерей паролей, номеров социального страхования и медицинских данных.

Столь серьезные случаи, как HeartBleed, редки, но лучше всегда предполагать, что в любой сторонней зависимости рано или поздно может обнаружиться новая уязвимость. И, к сожалению, неизвестно, когда это произойдет. Даже если сам код не меняется, в его зависимостях могут обнаружиться новые уязвимости.

При регулярном сканировании образов контейнеров утилита сканирования может проверять их содержимое на актуальной базе знаний уязвимостей (из NVD и прочих источников инструкций по безопасности). Очень часто все развернутые образы сканируют каждые 24 часа, помимо сканирования новых образов при сборке, в качестве составной части автоматизированного конвейера CI/CD.

Средства сканирования

Существует множество утилит для сканирования образов контейнеров, начиная от реализаций с открытым исходным кодом, таких как Trivy (https://oreil.ly/SxKQT), Clair (https://oreil.ly/avK-2) и Anchore (https://oreil.ly/7rFFt), и до коммерческих решений от таких компаний, как JFrog, Palo Alto и Aqua. Во многих реестрах образов контейнеров, например Docker Trusted Registry (https://docs.docker.com/ee/dtr), и в проекте Harbor (https://goharbor.io/) от CNCF, а также в реестрах от основных поставщиков облачных сервисов есть встроенные возможности сканирования.

К сожалению, выдаваемые различными сканерами результаты сильно разнятся, и имеет смысл обсудить почему.

Источники информации

Как уже обсуждалось ранее в этой главе, есть множество источников информации об уязвимостях, включая инструкции по безопасности различных дистрибутивов. У Red Hat их даже несколько (https://oreil.ly/jE4ad) — их лента новостей OVAL включает только уязвимости, для которых существует исправление, и не включает уже опубликованные, но пока не исправленные.

Если сканер не использует данные из ленты новостей по безопасности дистрибутива и основывается лишь на данных NVD, то будет, вероятно, выдавать множество ложнопозитивных результатов для образов, в основе которых лежит этот дистрибутив. Если вы предпочитаете конкретный дистрибутив Linux в своих образах или программные решения наподобие distroless, то убедитесь, что сканер его поддерживает.

Устаревшие источники

Иногда персонал, отвечающий за сопровождение дистрибутивов, меняет способ извещать об уязвимостях. Относительно недавно это произошло с дистрибутивом Alpine: вместо информации об уязвимостях на alpine-secdb (https://oreil.ly/lVHll) теперь используется новая система на aports (https://oreil.ly/J1-YA). На момент написания данной книги есть сканеры, которые до

сих пор используют только данные из старой ленты новостей alpine-secdb, не обновлявшейся уже несколько месяцев.

Не все уязвимости исправляются

Иногда персонал, отвечающий за сопровождение дистрибутивов, решает не исправлять конкретную уязвимость (например, поскольку риск пренебрежимо мал, а исправить проблему непросто либо потому, что способ взаимодействия с прочими пакетами на их платформе делает невозможным использование этой уязвимости).

А раз уязвимость исправлять никто не собирается, то у разработчиков сканеров возникает философский вопрос: отображать ли уязвимость в списке результатов, если ничего все равно сделать нельзя? Мы, в Aqua, слышали от некоторых клиентов, что они не хотели бы видеть данную категорию результатов, поэтому мы предоставили пользователю возможность выбора. Все это свидетельствует о том, что не существует «правильных» наборов результатов, когда речь идет о сканировании на уязвимости.

Уязвимости подпакетов

Иногда система управления пакетами устанавливает пакет и сообщает о нем как о едином целом, однако на самом деле он состоит из нескольких подпакетов. Хороший пример: пакет `bind` в Ubuntu. Иногда он устанавливается с одним только подпакетом `docs`, включающим, как легко догадаться, лишь документацию. Некоторые сканеры предполагают, что, если система управления пакетами сообщает об установленном пакете, значит, установлен весь пакет (со всеми возможными подпакетами). Это может привести к ложнопозитивным результатам, когда сканер сообщает об уязвимостях, которых нет и быть не может, поскольку соответствующий подпакет вообще не установлен.

Различия названий пакетов

Имя источника для пакета может включать двоичные файлы с совершенно различными названиями. Например, в Debian пакет `shadow` (https://oreil.ly/SpPXQ) включает двоичные файлы `login`, `passwd` и `uidmap`. Если сканер это не учитывает, то может выдавать ложнонегативные результаты.

Дополнительные возможности сканирования

Некоторые сканеры образов способны выявлять и другие проблемы, помимо уязвимостей, например:

- известное вредоносное программное обеспечение, содержащееся в образе;
- исполняемые файлы с установленным битом `setuid` (что, как вы видели в главе 2, может позволить повысить полномочия);
- образы, конфигурация которых предусматривает выполнение от имени суперпользователя;
- секретные учетные данные, например токены или пароли;
- конфиденциальные данные, например номера платежных карт, номера социального страхования или что-то в этом роде.

Ошибки сканеров

Надеюсь, материал данного раздела позволяет понять, что поиск уязвимостей — не такая простая задача, как может показаться на первый взгляд. Поэтому весьма вероятно, что при использовании любого сканера будут встречаться ложнопозитивные или ложнонегативные результаты из-за ошибок в сканере или изъянов в информации об уязвимостях, которую он задействует.

Тем не менее со сканером лучше, чем без него. Если у вас нет сканера или вы не используете его регулярно, то никак не узнаете, не станет ли ваше программное обеспечение легкой добычей злоумышленников. Время в этом смысле плохой лекарь — критическая уязвимость Shellshock была обнаружена в коде, существовавшем уже десятилетия. Если использовать запутанные зависимости, то следует ожидать, что рано или поздно в них обнаружатся какие-либо уязвимости.

Ложнопозитивные результаты раздражают, но некоторые утилиты позволяют заносить отдельные отчеты об уязвимостях в белый список, поэтому вы можете сами решить, следует ли на них реагировать.

Будем считать, что мы убедили вас в преимуществах использования сканера, и рассмотрим возможные варианты включения его в ваш технологический процесс.

Сканирование в конвейере CI/CD

Пройдем по конвейеру CI/CD слева направо, где этап написания кода — крайний слева, а этап развертывания в промышленной эксплуатации — крайний справа, как показано на рис. 7.2. В данном конвейере желательно устранять проблемы как можно раньше — это проще и дешевле, точно так же, как поиск и исправление программных ошибок требует намного больше времени и затрат после развертывания, чем во время разработки.

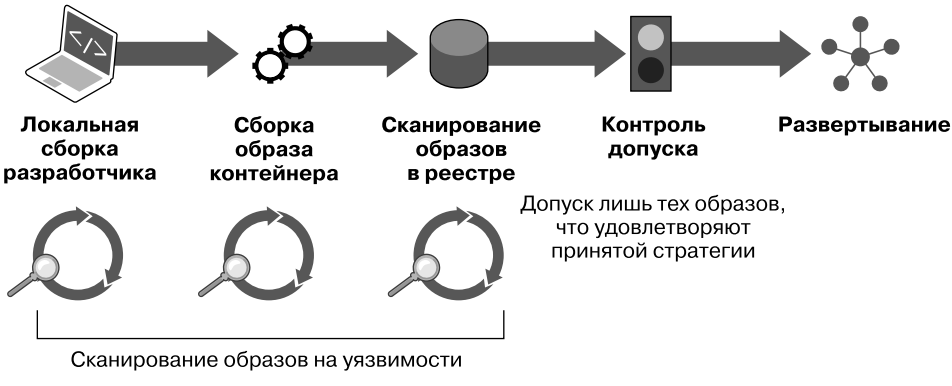


Рис. 7.2. Сканирование на уязвимости в конвейере CI/CD

При обычном развертывании на хосте все работающее на нем программное обеспечение использует одни и те же пакеты. За их регулярное обновление путем установки исправлений безопасности в организации обычно отвечает группа специалистов по компьютерной безопасности. Данная деятельность практически не зависит от этапов разработки и тестирования жизненного цикла приложения и располагается ближе к правому краю конвейера развертывания. Нередко возникают проблемы, связанные с совместным использованием одного и того же пакета различными приложениями, которым нужны разные его версии. Такие проблемы требуют продуманного управления зависимостями и в некоторых случаях изменений кода.

И напротив, как вы видели в главе 6, при развертывании на основе контейнеров все образы включают собственные зависимости, поэтому контейнеры для различных приложений могут включать различные версии пакетов, если это необходимо. Не нужно волноваться о совместимости кода приложения и набора используемых зависимостей. Это, с учетом существования утилит

для сканирования образов контейнеров, позволяет «сдвинуть» управление уязвимостями влево по конвейеру.

Сканирование на уязвимости можно автоматизировать. Исправить найденную уязвимость разработчики могут с помощью обновления и повторной сборки образа контейнера приложения с включением в него исправленной версии. Группам специалистов по компьютерной безопасности больше не нужно делать это вручную.

Количество мест, в которых можно ввести сканирование, невелико, как было показано выше на рис. 7.2.

- *Сканирование на стороне разработчика.* Если используемый сканер можно легко развернуть на настольном компьютере, то отдельные разработчики могут сканировать локальные сборки образов на предмет проблем, благодаря чему получают шанс внести исправления до отправки исходного кода в репозиторий.
- *Сканирование при сборке.* Взвесьте возможность включения шага сканирования сразу же после сборки образа контейнера в вашем конвейере. В случае обнаружения в результате этого уязвимостей, превышающих заданный порог серьезности, можно сообщить о неуспехе сборки, гарантируя тем самым, что она точно не будет развернута в промышленной эксплуатации. На рис. 7.3 приведена информация, выводимая проектом AWS CodeBuild при сборке образа на основе Dockerfile и дальнейшем его сканировании. В данном примере была обнаружена очень серьезная уязвимость, поэтому сборка завершилась неудачей.
- *Сканирование реестра.* После того как образ будет собран, конвейер обычно помещает его в реестр образов. Имеет смысл регулярно сканировать образы на случай обнаружения новой уязвимости в одном из пакетов, используемых в давно не пересобиравшемся образе.

В следующих статьях можно найти полезную информацию о включении разнообразных сканеров в различные конвейеры CI/CD:

- «Сканирование образов с помощью Trivy в AWS CodePipeline» (Scanning images with Trivy in an AWS CodePipeline) (https://oreil.ly/6ANm9);
- «Сканирование контейнеров» на GitLab (Container Scanning) (https://oreil.ly/okLcm);
- «Сканирование образов контейнеров Docker в конвейере Codefresh с помощью Aqua» (Docker Image Scanning in your Codefresh Pipeline with Aqua) (https://oreil.ly/P5_59).

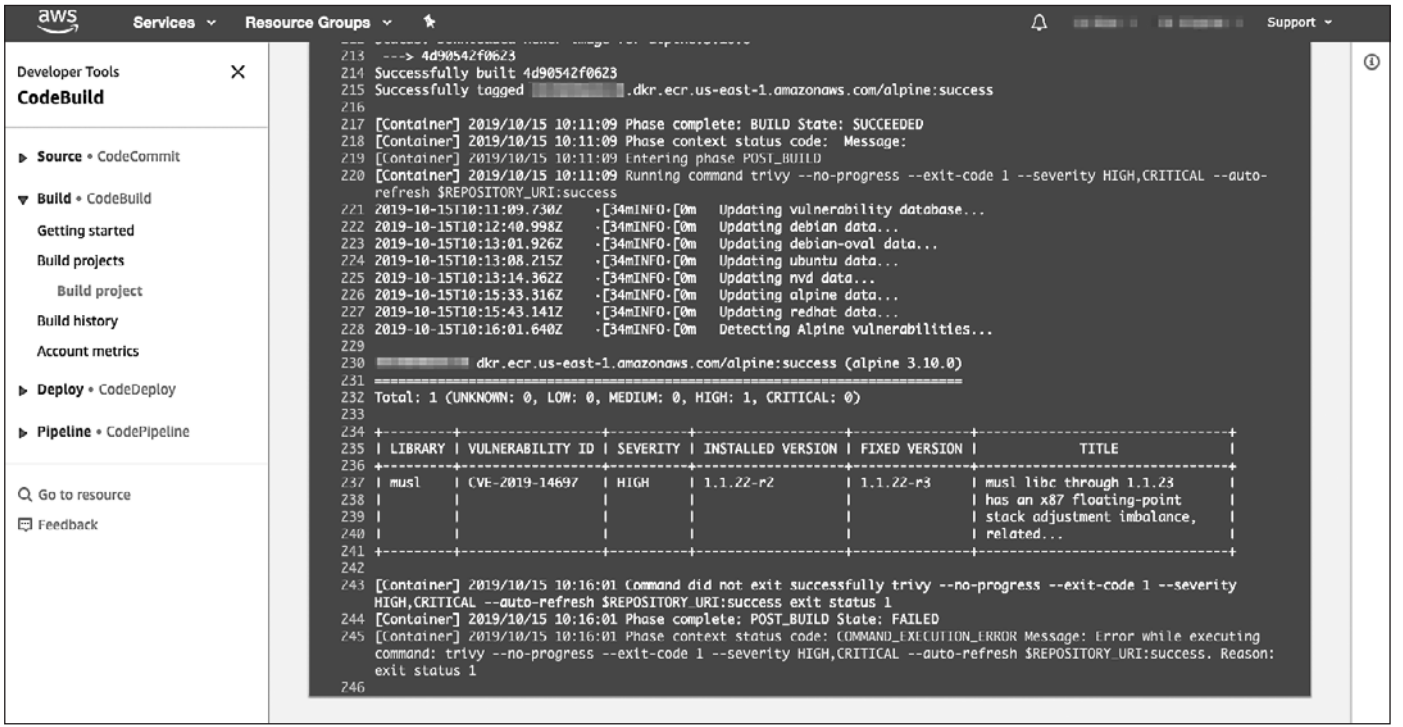


Рис. 7.3. Пример неуспеха сборки вследствие обнаружения очень серьезной уязвимости

Вероятно, лучше не откладывать шаг сканирования до момента развертывания по той простой причине, что вы станете сканировать каждый создаваемый экземпляр контейнера, хотя источником их всех является один и тот же образ контейнера. Если рассматривать контейнер как неизменяемый, то имеет смысл сканировать образ, а не контейнер.

Предотвращение запуска образов с уязвимостями

Проверять наличие существенных уязвимостей с помощью сканера — это одно, но надо еще и гарантировать, что образы с уязвимостями не попадут в промышленную эксплуатацию. Как было показано выше на рис. 7.2, это можно выполнить в виде части шага контроля допуска, который мы рассматривали в подразделе «Контроль допуска» на с. 115. Если не гарантируется развертывание лишь просканированных образов, то можно относительно легко обойти сканер уязвимостей.

В общем-то, коммерческие сканеры уязвимостей продаются в составе более обширной платформы, связывающей контроль допуска с результатами сканирования. В системах, развертываемых на основе Kubernetes, можно реализовывать пользовательские проверки при контроле допуска с помощью Open Policy Agent, включая проверку прохождения образами сканирования на уязвимости. Компания Google также работает над этой возможностью в рамках своего проекта Kritis (https://oreil.ly/PIhQu).

До сих пор в этой главе мы обсуждали лишь известные уязвимости в зависимостях, необходимых для кода приложения, но обошли вниманием такую важную категорию уязвимостей, как *уязвимости нулевого дня*.

Уязвимости нулевого дня

В разделе «Сканирование на уязвимости» на с. 121 рассказывалось, как исследователи в сфере безопасности по всему миру ищут новые способы взлома имеющегося программного обеспечения. Разумеется, при обнаружении новой уязвимости исправление публикуется не сразу, а спустя какое-то время. До тех пор пока не будет опубликовано исправление, такая уязвимость называется *уязвимостью нулевого дня* (zero-day vulnerability), поскольку с момента публикации еще не прошло ни одного дня. (Еще недавно считалось приемлемым применять исправления, скажем, раз в 30 дней.

Невозможно представить, сколько атак могло быть проведено за такой промежуток времени.)

И если вероятны программные ошибки в сторонних библиотеках, которыми может воспользоваться злоумышленник, то же самое справедливо для любого кода — включая создаваемые вами приложения. Обнаружить проблемы в коде помогают равноправный анализ, статический анализ и тестирование, но всегда есть вероятность, что некоторые проблемы замечены не будут. В зависимости от конкретной организации и ценности данных, всегда могут найтись злоумышленники, которые сочтут, что поиск этих изъянов стоит свеч.

Хорошая новость: если информация об уязвимости не была опубликована, то абсолютное большинство потенциальных злоумышленников знает о ней не больше, чем вы.

Плохая новость: у опытных злоумышленников и правительственных организаций есть библиотеки пока еще не опубликованных уязвимостей, как мы знаем из разоблачений Эдварда Сноудена (https://oreil.ly/Yz1tJ).

Сколько ни ищи в базе уязвимости, идентифицировать уязвимость, информация о которой пока еще не опубликована, это не поможет. В зависимости от типа и степени серьезности уязвимости защитить приложение и данные можно с помощью выполнения кода в «песочнице», как описывается в главе 8. Лучший способ защитить от уязвимостей нулевого дня — обнаружить и предотвратить аномальное поведение во время выполнения, о чем я расскажу в главе 13.

Резюме

В данной главе вы прочитали об исследованиях уязвимостей и идентификаторах CVE, присваиваемых различным уязвимостям. Вы узнали, почему так важна информация о безопасности, относящаяся к конкретному дистрибутиву, и почему не стоит полагаться на одну только базу NVD. Кроме того, я рассказала вам, почему различные сканеры могут выдавать разные результаты, поэтому теперь вы лучше подготовлены к выбору используемых утилит. Вне зависимости от того, на какой сканер падет ваш выбор, надеюсь, теперь вы понимаете, что сканирование образов контейнеров должно быть включено в ваш конвейер CI/CD.

ГЛАВА 8

Усиление изоляции контейнеров

В главах 3 и 4 вы видели, как контейнеры позволяют разделять рабочие задания, даже выполняемые на одном хост-компьютере. В этой главе я расскажу вам о некоторых более совершенных утилитах и методиках усиления изоляции выполняемых заданий.

Представьте, что у вас есть две части рабочей нагрузки, которые не должны мешать друг другу. Один из вариантов — изолировать их так, чтобы они не знали о существовании друг друга; фактически именно это делают контейнеры и виртуальные машины. Другой подход — ограничить доступные им действия, поэтому даже если одна из них знает о существовании другой, то просто ничем не может повлиять на нее. Изоляция приложений таким образом, что им доступна лишь часть ресурсов, называется *помещением в «песочницу»* (sandboxing).

Контейнер, в котором запускается приложение, играет, по сути, роль «песочницы». При каждом запуске контейнера известно, какой код приложения будет выполняться внутри него. В случае взлома приложения злоумышленник может попытаться запустить код, поведение которого выходит за нормальные для этого приложения рамки. Механизм «песочницы» позволяет ограничить разрешаемые данному коду действия, таким образом ограничивая возможности злоумышленника повлиять на систему. Сначала мы рассмотрим механизм *seccomp*.

Механизм seccomp

В разделе «Системные вызовы» на с. 36 вы видели, что системные вызовы предоставляют приложению интерфейс, с помощью которого оно может запрашивать выполнение ядром системы определенных действий. Seccomp — механизм ограничения набора системных вызовов, разрешенных приложению.

Вначале, после появления в ядре Linux в 2005 году, seccomp (сокращение от secure computing mode — «безопасный режим вычислений») означал, что переведенный в этот режим процесс может выполнять только несколько системных вызовов:

- `sigreturn` (возврат из обработчика сигнала);
- `exit` (завершение выполнения процесса);
- `read` и `write`, но лишь при использовании дескрипторов файлов, уже открытых до перехода в безопасный режим.

Ненадежный код можно спокойно выполнять в этом режиме, он не сможет совершить никаких вредоносных действий. К сожалению, есть побочный эффект — огромное количество кода тоже не сможет сделать ничего полезного в этом режиме. Такая «песочница» попросту слишком сильно ограничена.

В 2012 году в ядро был добавлен новый подход, *seccomp BFP*, определяющий допустимость конкретного системного вызова на основе фильтров пакетов Беркли (Berkeley Packet Filters, BPF) с учетом профиля seccomp данного процесса. У каждого процесса может быть свой профиль.

Фильтр seccomp BPF учитывает при решении о допустимости вызова данным профилем код операции системного вызова и его параметры. На самом деле все несколько сложнее: профиль указывает, что делать, если системный вызов соответствует конкретному фильтру, причем возможные действия включают возвращение ошибки, завершение процесса и вызов средства трассировки. Но в большинстве сценариев в мире контейнеров профиль либо разрешает вызов, либо возвращает ошибку, поэтому его можно считать своего рода белым списком (или черным списком) системных вызовов.

Все вышеописанное может весьма пригодиться в мире контейнеров, поскольку в действительности некоторые системные вызовы контейнеризованному приложению выполнять незачем, разве что в исключительно редких ситуациях. Например, нежелательно, чтобы контейнеризованное приложение могло менять системное время на хост-компьютере, поэтому имеет смысл запретить ему доступ к системным вызовам `clock_adjtime` и `clock_settime`. Маловероятно, что контейнерам понадобится менять модули ядра, и потому им не нужно вызывать `create_module`, `delete_module` или `init_module`. В ядре Linux существует сервис keyring, не ограниченный пространством имен, вследствие чего лучше запретить контейнерам обращаться к `request_key` и `keyctl`.

Профиль seccomp по умолчанию для Docker (https://oreil.ly/3sNNI) блокирует более 40 из 300 с лишним системных вызовов (включая все вышеперечисленные) без каких-либо отрицательных последствий для абсолютного большинства контейнеризованных приложений. Это отличный профиль для использования по умолчанию, разве что у вас есть особые причины не задействовать его.

К сожалению, хотя он и используется по умолчанию в Docker, при использовании Kubernetes по умолчанию профиль seccomp не применяется (даже при Docker в качестве среды выполнения контейнеров). По крайней мере, на момент написания данной книги поддержка seccomp в Kubernetes находится на стадии альфа-версии, и применять профили можно путем указания аннотаций для объектов PodSecurityPolicy (https://oreil.ly/6gIjt)[1].

Джесс Фразель (Jess Frazelle) использует на contained.af профиль seccomp для усиления эффекта при демонстрации возможностей изоляции seccomp + контейнеры — на момент написания данной книги она все еще не была взломана, несмотря на многолетние попытки.

Возможно, вы захотите ограничить контейнер еще меньшим кругом системных вызовов — в идеальном мире у каждого приложения был бы свой профиль, разрешающий ему только необходимый набор системных вызовов. Существует несколько различных подходов к созданию подобных профилей.

- Можно воспользоваться `strace` для трассировки всех системных вызовов, выполняемых приложением. В этом сообщении блога (https://oreil.ly/ROlHh) Джесс Фразель описала, как она сгенерировала на основе такого подхода и протестировала профиль seccomp по умолчанию для Docker.
- Более современный способ получить список системных вызовов — воспользоваться вспомогательной процедурой на основе eBPF. А поскольку seccomp ограничивает набор разрешенных системных вызовов с помощью BPF, то нет ничего удивительного в том, что eBPF (extended BPF — расширенный BPF) позволяет получить список используемых системных

[1] В версии 1.19 Kubernetes контролировать применение профилей seccomp можно с помощью поля seccompProfile в securityContext объектов PodSecurityPolicy или контейнеров.

вызовов. Для получения списка генерируемых контейнером системных вызовов можно применить такие утилиты, как falco2seccomp (https://oreil.ly/z5yyT) и Tracee (https://oreil.ly/iw-rL).

- Если вы не хотели бы тратить усилия на создание профилей seccomp самостоятельно, то, возможно, вам имеет смысл взглянуть на коммерческие утилиты, призванные обеспечить безопасность контейнеров. Некоторые из этих утилит умеют автоматически генерировать пользовательские профили seccomp на основе наблюдений за отдельными рабочими заданиями.

Если вам интересны технологии, лежащие в основе strace, то можете посмотреть доклад (https://oreil.ly/SV6d-), в котором я создала простейшую реализацию strace с помощью всего нескольких строк на языке Go.

Модуль AppArmor

AppArmor (сокращение от Application Armor — «броня приложения») — один из нескольких модулей безопасности Linux (Linux security modules, LSM), которые можно включить в ядре Linux. В AppArmor профиль можно связать с исполняемым файлом и описать, что этому файлу можно, а что — нет, на языке привилегий и прав доступа файла. Напомню, что о них мы говорили в главе 2. Чтобы проверить, включен ли модуль AppArmor в вашем ядре, посмотрите на файл `/sys/module/apparmor/parameters/enabled` — если найдете там `y`, значит, включен.

AppArmor и прочие LSM реализуют *мандатное управление доступом* (mandatory access control)[1]. Оно настраивается главным администратором, после чего остальные пользователи не могут никоим образом изменить права доступа или передавать их другим пользователям. Управление же доступом в Linux является *избирательным* (discretionary) в том смысле, что если моя учетная запись пользователя является владельцем файла, то я могу предоставить доступ к нему другому пользователю (если эта возможность не переопределена с помощью мандатного управления доступом) или запретить даже моей собственной учетной записи пользователя что-либо

[1] Иногда в русскоязычной литературе называется также принудительным контролем доступа.

записывать в этот файл, чтобы случайно не испортить что-то. Благодаря мандатному управлению доступом администратор может гораздо тоньше контролировать происходящее в системе, причем отдельные пользователи не смогут переопределить заданные им настройки.

AppArmor включает режим «претензий», при котором можно проверить исполняемый файл на соответствие профилю с занесением в журнал всех расхождений. Смысл в том, что можно обновлять профиль на основе этих журналов, чтобы в конечном итоге добиться отсутствия новых расхождений, после чего можно начинать обеспечивать соблюдение профиля.

Создавать профили AppArmor для контейнеров можно с помощью bane (https://oreil.ly/Xe7YZ).

Созданный профиль можно установить в каталог /etc/apparmor и запустить утилиту `apparmor_parser` для его загрузки. Просмотреть загруженные профили можно в `/sys/kernel/security/apparmor/profiles`.

Если запустить контейнер с помощью команды `docker run --security-opt="apparmor:<имя профиля>" ...`, то его поведение будет ограничено вариантами, разрешенными профилем. Среды выполнения контейнеров `Containerd` и `CRI-O` также поддерживают AppArmor.

Существует профиль AppArmor по умолчанию для Docker, но учтите, что аналогично seccomp Kubernetes не использует его по умолчанию. Чтобы применить какой-либо профиль AppArmor для контейнера в модуле Kubernetes, необходимо добавить аннотации (https://oreil.ly/_l1O8).

Модуль SELinux

SELinux (Security-Enhanced Linux — Linux с улучшенной безопасностью) — еще один тип LSM, но уже разработанный Red Hat, хотя «Википедия» утверждает (https://oreil.ly/gV7_e), что свои истоки он ведет от проектов Агентства национальной безопасности США. Если на вашем хосте запущен дистрибутив Red Hat (RHEL или Centos), то весьма вероятно, что SELinux уже активирован.

SELinux позволяет ограничивать доступные процессам действия в смысле взаимодействий с файлами и прочими процессами. Каждый процесс выполняется в каком-либо *домене* SELinux (можете считать его своего рода контекстом работы процесса), а у каждого файла есть свой тип. Просмотреть информацию SELinux для файла можно с помощью команды `ls -lZ`; вдобавок вы можете по аналогии добавить `-Z` к команде `ps`, чтобы получить подробную информацию для процессов, связанную с SELinux.

Ключевое отличие между правами доступа SELinux и обычными правами доступа DAC Linux (которые вы видели в главе 2) состоит в том, что в SELinux права никак не связаны с учетной записью пользователя — они описываются исключительно метками. Тем не менее эти системы прав делают одно дело, и потому любое действие должно быть разрешено как DAC, так SELinux.

Чтобы обеспечивать реализацию стратегий, необходимо сначала маркировать все файлы на машине метками SELinux. Эти стратегии указывают права доступа процесса конкретного домена к файлам конкретного типа. На практике это означает следующее: можно сделать так, что приложение будет иметь доступ только к собственным файлам, а у остальных процессов никакого доступа к ним не будет. Взлом приложения в этом случае повлияет лишь на ограниченное множество файлов, даже если обычные избирательные права доступа его разрешают. При включенном модуле SELinux существует режим, при котором нарушения стратегий просто заносятся в журнал, а не обеспечиваются принудительно (аналогично тому, что мы видели в AppArmor).

Чтобы создать удачный профиль SELinux приложения, нужно знать все файлы, к которым ему может потребоваться доступ в случае как нормальной работы, так и возникновения ошибок, поэтому данную задачу лучше возложить на разработчика приложения. Некоторые поставщики предоставляют профили для своих приложений.

Узнать больше о SELinux можно из прекрасного руководства от DigitalOcean (https://oreil.ly/2Hx6b), а также наглядного руководства от Дэна Уолша (Dan Walsh) (https://oreil.ly/jmhC-). Подробности о взаимодействии SELinux с Docker можно найти на сайте проекта Atomic (https://oreil.ly/msyOa).

Все обсуждавшиеся нами механизмы безопасности — seccomp, AppArmor и SELinux — предназначены для низкоуровневого управления поведением процессов. Генерация подробного профиля в смысле точного описания множества необходимых системных вызовов и привилегий — задача непростая, причем незначительное изменение приложения может потребовать существенных изменений профиля. Накладные расходы на хранение профилей в виде, согласованном с изменениями приложений, могут оказаться довольно большими, вдобавок люди по своей природе склонны либо использовать достаточно либеральные профили, либо вообще их отключать. Если у вас нет возможностей генерировать профили для всех своих приложений, то профили по умолчанию seccomp и AppArmor для Docker работают достаточно неплохо.

Стоит отметить, впрочем, что, хотя все эти механизмы и позволяют ограничить доступные приложениям пользовательского пространства действия, не следует забывать о совместно используемом ими ядре. От уязвимости в самом ядре, такой как Dirty COW (https://oreil.ly/qQiJL), вас не спасут никакие из этих утилит.

До сих пор в этой главе обсуждались применяемые к контейнерам механизмы безопасности, с помощью которых можно ограничить действия, разрешаемые контейнерам. Рассмотрим теперь методики «песочниц», обеспечивающие промежуточный уровень изоляции между контейнерами и виртуальными машинами, начиная с gVisor.

«Песочница» gVisor

«Песочница» gVisor от компании Google ограничивает контейнер, перехватывая системные вызовы примерно так же, как гипервизор перехватывает системные вызовы гостевой виртуальной машины.

Согласно ее документации (https://gvisor.dev/docs), gVisor представляет собой «ядро для пользовательского пространства», что кажется мне терминологически противоречивым, но описывает реализацию системных вызовов Linux в пользовательском пространстве с помощью паравиртуализации. Как вы видели в главе 5, паравиртуализация означает заместительную реализацию инструкций вместо выполнения их ядром хоста.

Для этого компонент Sentry «песочницы» gVisor перехватывает поступающие от приложения системные вызовы. Sentry жестко ограничен с помощью seccomp, вследствие чего сам не может обращаться к ресурсам файловой системы. Если возникает необходимость выполнить системные вызовы, связанные с доступом к файлам, он передает их для выполнения совершенно отдельному процессу — Gofer.

Даже системные вызовы, не имеющие отношения к доступу к файловой системе, не передаются непосредственно ядру хоста, а реализуются заново внутри Sentry. По сути, он представляет собой гостевое ядро, работающее в пользовательском пространстве.

Проект gVisor (https://oreil.ly/cMROh) содержит исполняемый файл `runsc`, совместимый с комплектами в формате OCI, очень похожий на обычную среду выполнения OCI `runc`, с которой мы сталкивались в главе 6. При выполнении контейнера с помощью `runsc` можно легко видеть процессы gVisor, но если у вас уже есть файл `config.json` для `runc`, то, вероятно, вам придется сгенерировать его версию, совместимую с `runsc`. В следующем примере я запускаю тот же комплект для Alpine Linux, который использовался в разделе «Стандарты OCI» на с. 99:

```
$ cd alpine-bundle
# Сохраняем уже имеющийся config.json, предназначенный для работы с runc
$ mv config.json config.json.runc
# Создаем файл config.json для runsc
$ runsc spec
$ sudo runsc run sh
```

В другом окне терминала можете воспользоваться командой `runsc list`, чтобы просмотреть контейнеры, созданные `runsc`:

```
$ runsc list
ID  PID    STATUS   BUNDLE                         CREATED              OWNER
sh  32258  running  /home/vagrant/alpine-bundle    2019-08-26T13:51:21  root
```

Запустите команду `sleep` внутри контейнера на достаточно длительное время, чтобы посмотреть на ее выполнение из другого окна терминала. Команда `runsc ps` <*ID контейнера*> отображает запущенные внутри контейнера процессы:

```
$ runsc ps sh
UID       PID       PPID      C         STIME     TIME      CMD
0         1         0         0         14:06     10ms      sh
0         15        1         0         14:15     0s        sleep
```

Пока все идет так, как ожидалось, но, если посмотреть на процессы с точки зрения хоста, ситуация сразу станет гораздо интереснее (вывод команды был отредактирован, чтобы подчеркнуть наиболее интересные нюансы):

```
$ ps fax
  PID TTY      STAT   TIME COMMAND
  ...
3226 pts/1    S+     0:00  |      \_ sudo runsc run sh
3227 pts/1    Sl+    0:00  |          \_ runsc run sh
3231 pts/1    Sl+    0:00  |              \_ runsc-gofer --root=/var/run/runsc
3234 ?        Ssl    0:00  |              \_ runsc-sandbox --root=/var/run/runsc
3248 ?        tsl    0:00  |                  \_ [exe]
3257 ?        tl     0:00  |                      \_ [exe]
3266 ?        tl     0:00  |                      \_ [exe]
3270 ?        tl     0:00  |                      \_ [exe]
  ...
```

Здесь виден процесс `runsc run`, породивший два процесса: один для Gofer, а второй — `runsc-sandbox`, в документации gVisor называемый Sentry. У «песочницы» есть дочерний процесс со своими тремя дочерними процессами. Если посмотреть на информацию о процессах для этих дочерних процессов двух уровней с точки зрения хоста, то обнаруживается нечто интересное: исполняемый файл во всех четырех — `runsc`. Для краткости в следующем примере показан только дочерний процесс первого уровня и один из дочерних процессов второго:

```
$ ls -l /proc/3248/exe
lrwxrwxrwx 1 nobody nogroup 0 Aug 26 14:11 /proc/3248/exe -> /usr/local/bin/runsc
$ ls -l /proc/3257/exe
lrwxrwxrwx 1 nobody nogroup 0 Aug 26 14:13 /proc/3257/exe -> /usr/local/bin/runsc
```

Что примечательно, ни один из этих процессов не ссылается на исполняемый файл `sleep`, который, как мы знаем (поскольку можем увидеть его с помощью команды `runsc ps`), работает внутри контейнера. Найти исполняемый файл `sleep` напрямую с хоста также не получается:

```
vagrant@vagrant:~$ sudo ps -eaf | grep sleep
vagrant  3554 3171  0 14:26 pts/2    00:00:00 grep --color=auto sleep
```

Подобная «невидимость» процессов, работающих внутри «песочницы» gVisor, намного ближе к поведению обычной виртуальной машины, чем контейнера. И обеспечивает дополнительную защиту для работающих внутри «песочницы» процессов: даже если злоумышленник получит права суперпользователя на хост-компьютере, граница между хостом и работающими процессами остается относительно непроницаемой. По крайней мере,

так должно было быть, если бы не сама команда `runsc`! Она предоставляет подкоманду `exec`, с помощью которой суперпользователь хоста может производить различные операции внутри работающего контейнера:

```
$ sudo runsc exec sh ps
PID   USER     TIME  COMMAND
    1 root      0:00 /bin/sh
   21 root      0:00 sleep 100
   22 root      0:00 ps
```

И хотя подобная изоляция выглядит достаточно прочной, ее существенно ограничивают два фактора.

- В gVisor реализованы далеко не все системные вызовы Linux (https://oreil.ly/PHsFm). Приложение, которому требуются какие-либо из нереализованных системных вызовов, не может работать в gVisor. На момент написания данной книги в gVisor были недоступны 97 системных вызовов, в то время как профилем Docker по умолчанию (https://oreil.ly/Lt5Ge) было заблокировано примерно 44 системных вызова.
- Производительность. Во многих случаях быстродействие примерно такое же, как и в случае `runc`, но большое количество системных вызовов, выполняемое приложением, может существенно повлиять на его производительность. Google опубликовал руководство по производительности (https://oreil.ly/zqC6i) для проекта gVisor, с помощью которого вы можете изучить этот вопрос более подробно.

Поскольку gVisor реализует заново функциональность ядра, то это большой и сложный проект, и данная сложность означает относительно высокую вероятность наличия в нем своих уязвимостей (таких как вот эта уязвимость повышения полномочий (https://oreil.ly/awCYt), обнаруженная Максом Юстицем (Max Justicz)).

Как вы видели в этом разделе, gVisor предоставляет механизм изоляции, напоминающий скорее виртуальную машину, чем обычный контейнер. Однако gVisor затрагивает только доступ приложения к системным вызовам. Для изоляции контейнера по-прежнему используются пространства имен, контрольные группы и изменение корневого каталога.

В оставшейся части данной главы мы обсудим подходы, в которых контейнеризованные приложения выполняются с помощью изоляции виртуальных машин.

Среда выполнения контейнеров Kata Containers

Как вы видели в главе 4, при работе обычного контейнера среда выполнения запускает новый процесс на хосте. Основная задумка Kata Containers (https://katacontainers.io/) состоит в выполнении контейнеров внутри отдельной виртуальной машины. Подобный подход позволяет выполнять приложения из обычных образов контейнеров в формате OCI при полной изоляции виртуальной машины.

Kata использует прокси между средой выполнения контейнеров и отдельным целевым хостом, на котором выполняется код приложения. Этот прокси среды выполнения с помощью QEMU создает отдельную виртуальную машину для выполнения контейнера.

Один из недостатков Kata Containers — необходимость дожидаться загрузки виртуальной машины. Сотрудники AWS создали облегченную виртуальную машину, которая специально предназначена для выполнения контейнеров и запускается намного быстрее, чем обычная виртуальная машина: Firecracker.

Виртуальная машина Firecracker

Как вы видели в разделе «Недостатки виртуальных машин» на с. 94, виртуальные машины медленно запускаются, вследствие чего плохо подходят для краткосрочных рабочих заданий, обычно выполняемых в контейнерах. Но что, если бы существовала чрезвычайно быстро запускающаяся виртуальная машина? Firecracker (https://oreil.ly/ZkPef) — виртуальная машина, предлагающая одновременно и надежную изоляцию с помощью гипервизора без совместно используемого ядра, и время загрузки порядка 100 миллисекунд, что подходит для контейнеров намного лучше. У нее есть еще одно преимущество: благодаря (насколько я понимаю, постепенному) принятию на вооружение в AWS, для сервисов Lambda и Fargate, она сейчас проходит полноценную проверку в условиях промышленной эксплуатации.

Firecracker запускается так быстро благодаря тому, что его создатели убрали из нее всю обычно включаемую в ядро функциональность, которая не требуется в контейнере. Один из самых медленных этапов загрузки системы — подсчет имеющихся устройств, но контейнеризованные приложения редко

используют большое количество устройств. В основном время экономится за счет минимальной модели устройств, из которой исключены все устройства, за исключением необходимых.

Firecracker работает в пользовательском пространстве и задействует API REST для настройки работы гостевых систем под управлением VMM Firecracker. Для гостевых операционных систем применяется аппаратная виртуализация на основе KVM, поэтому вы не сможете запустить ее в, скажем, гостевой ОС Type 2 на ноутбуке, если ваше сочетание аппаратного и программного обеспечения не поддерживает вложенную виртуализацию.

Я хотела бы рассмотреть в этой главе еще один, последний подход к изоляции с еще более радикальным вариантом сокращения размера гостевой операционной системы: Unikernels.

Unikernels

В образах виртуальных машин выполняются универсальные операционные системы, которые можно применять повторно для любого приложения. Само собой разумеется, что приложения вряд ли будут задействовать все функциональные возможности ОС. Исключение неиспользуемых частей позволяет сократить поверхность атаки.

Основная идея Unikernels состоит в создании специализированного образа машины, состоящего из приложения и необходимых ему частей операционной системы. Этот образ машины может выполняться непосредственно на гипервизоре с тем же уровнем изоляции, что и у обычных виртуальных машин, однако с более коротким временем начальной загрузки по сравнению с Firecracker.

Каждое приложение необходимо скомпилировать в образ Unikernels, включающий все необходимое ему для работы. А затем гипервизор сможет выполнять загрузку этой машины точно так же, как и обычного образа виртуальной машины Linux.

Методики Unikernels для контейнеров применяются в проекте Nabla (https://oreil.ly/W_BRY) компании IBM. В контейнерах Nabla используется крайне ограниченный набор всего из семи системных вызовов, что обеспечивается с помощью профиля seccomp. Все прочие системные вызовы приложения

обрабатываются в библиотечном компоненте OS Unikernels. Благодаря обращению лишь к небольшой доле ядра контейнеры Nabla сокращают поверхность атаки. У такого подхода есть недостаток: необходима повторная сборка приложений в виде контейнера Nabla.

Резюме

В этой главе вы увидели, что существует множество способов изолировать код экземпляров приложения друг от друга, которые в известной степени можно рассматривать как контейнер.

- Один из вариантов — использовать обычные контейнеры с дополнительными механизмами безопасности для усиления исходной изоляции контейнеров: seccomp, AppArmor, SELinux. Все они проверены на практике, но также печально известны тем, насколько сложно добиться их эффективной работы.
- Существует несколько решений, обеспечивающих изоляцию уровня виртуальной машины: Firecracker и Unikernels.
- Наконец, есть третья категория методик реализации «песочниц»: gVisor и ей подобные, с уровнем изоляции примерно посередине между контейнерами и виртуальными машинами.

Какой вариант больше подходит для вашего приложения — зависит от профиля рисков, причем на ваш выбор могут влиять опции, предлагаемые вашим поставщиком облачных сервисов и/или управляемым решением. Вне зависимости от используемой вами среды выполнения контейнеров и степени обеспечиваемой ею изоляции, всегда существуют способы с легкостью ее взломать. В главе 9 вы узнаете, как именно.

ГЛАВА 9

Нарушение изоляции контейнеров

Из главы 4 вы узнали, как устроен контейнер и что он «видит» только часть ресурсов машины, на которой работает. В этой главе вы увидите, насколько просто задать настройки контейнера, приводящие к прорехам в его изоляции.

Иногда это делается умышленно для достижения определенных целей, например, чтобы частично делегировать сетевую функциональность вспомогательному контейнеру (sidecar container). В остальных же случаях обсуждаемые в данной главе идеи серьезно подорвут безопасность ваших приложений!

Для начала поговорим о поведении в мире контейнеров, вероятно, по умолчанию наиболее опасном: о выполнении контейнеров от имени суперпользователя.

Выполнение контейнеров по умолчанию от имени суперпользователя

Если в образе контейнера не прописан не суперпользователь или не указано, что нужно запускать не от имени пользователя по умолчанию, то по умолчанию контейнер будет выполняться от имени суперпользователя. И, как легко убедиться (если вы не настроили пространства имен пользователей), он окажется суперпользователем не только внутри контейнера, но и на хост-компьютере.

В данном примере предполагается, что вы используете команду docker от Docker. Если вы устанавливали podman (https://podman.io/), то, возможно, последовали совету создать псевдоним для docker, так что на самом деле при этом запускается podman. Поведение podman относительно суперпользователей довольно сильно отличается от docker. Далее в главе мы поговорим об этих различиях, а пока просто учтите, что с podman следующий пример работать не будет.

Запустите командную оболочку от имени не суперпользователя внутри контейнера Alpine, задействуя `docker`, и посмотрите на идентификатор пользователя:

```
$ whoami
vagrant
$ docker run -it alpine sh
/ $ whoami
root
```

Хотя контейнер создавался с помощью команды `docker`, выполнявшейся от имени несуперпользователя, внутри контейнера пользователь — `root`. Теперь убедимся, что это `root` и на хосте, открыв на той же машине второе окно терминала. Выполните внутри контейнера команду `sleep`:

```
/ $ sleep 100
```

И посмотрите во втором окне имя пользователя, который ее запустил:

```
$ ps -fC sleep
UID        PID  PPID  C STIME TTY          TIME CMD
root     30619 30557  0 16:44 pts/0    00:00:00 sleep 100
```

С точки зрения хоста владельцем этого процесса является `root`. `Root` внутри контейнера является пользователем `root` и на хосте.

Если для запуска используется `runc`, а не `docker`, то аналогичная демонстрация будет менее убедительной (за исключением контейнеров, не требующих полномочий суперпользователя, о которых мы поговорим чуть позже), поскольку для запуска контейнера необходимо быть суперпользователем на хосте. Дело в том, что только у суперпользователя есть достаточные привилегии для создания пространств имен. В Docker контейнеры для вас создает демон Docker, выполняемый от имени суперпользователя.

То, что в Docker контейнеры выполняются от имени суперпользователя, даже когда запускаются изначально несуперпользователем, — одна из форм повышения полномочий. Сама по себе работа контейнера от имени суперпользователя не является проблемой, но вызывает вопросы, связанные с безопасностью. Если злоумышленник сможет выйти за рамки контейнера, работающего от имени суперпользователя, то получит все возможности суперпользователя по доступу к хост-компьютеру, то есть свободный доступ ко всему, что есть на машине. Вы же не хотите оказаться всего в одном шаге от захвата злоумышленником хоста?

К счастью, контейнеры можно выполнять и от имени несуперпользователей. Для этого можно или задать его идентификатор, или воспользоваться вышеупомянутыми контейнерами, не требующими полномочий суперпользователя. Рассмотрим оба варианта.

Переопределение идентификатора пользователя

Переопределить идентификатор пользователя в среде выполнения можно, указав идентификатор пользователя для контейнера.

В `runc` для этого необходимо модифицировать файл `config.json` внутри комплекта. Попробуйте, например, поменять `process.user.uid` следующим образом:

```
  ...
  "process": {
    "terminal": true,
    "user": {
      "uid": 5000,
      ...
    }
    ...
  }
```

Теперь среда выполнения подхватит указанный идентификатор пользователя и будет применять его для процесса контейнера:

```
$ sudo runc run sh
/ $ whoami
whoami: unknown uid 5000
/ $ sleep 100
```

И хотя мы задействуем `sudo` для запуска команды от имени `root`, идентификатор пользователя контейнера — 5000, в чем можно убедиться со стороны хоста:

```
$ ps -fC sleep
UID        PID  PPID  C STIME TTY          TIME CMD
5000     26909 26893  0 16:16 pts/0    00:00:00 sleep 50
```

Как вы видели в главе 6, в комплект совместимого с OCI образа входит как корневая файловая система образа, так и информация о настройках среды выполнения. Та же информация упаковывается в образы Docker. Переопределить настройки пользователя можно с помощью опции `--user`, вот так:

```
$ docker run -it --user 5000 ubuntu bash
I have no name!@b7ca6ec82aa4:/$
```

Поменять встроенный в образ Docker идентификатор пользователя можно с помощью команды `USER` в соответствующем Dockerfile. Но абсолютное большинство образов контейнеров в общедоступных репозиториях настроены на использование `root`, поскольку параметр конфигурации `USER` в них не задан. А если идентификатор пользователя не задан, то по умолчанию контейнер выполняется от имени суперпользователя.

Требование выполнения от имени суперпользователя внутри контейнера

Многие часто используемые образы контейнеров инкапсулируют распространенное программное обеспечение, которое изначально было предназначено для запуска непосредственно на сервере. Возьмем, например, обратный прокси-сервер и балансировщик нагрузки Nginx; он появился задолго до того, как Docker обрел популярность, но сейчас доступен в виде образа контейнера на Docker Hub. Стандартный контейнер Nginx был настроен для выполнения по умолчанию от имени суперпользователя, по крайней мере по состоянию на момент написания данной книги. Если вы запустите контейнер `nginx` и посмотрите на работающие в нем процессы, то увидите, что главный процесс запущен от имени суперпользователя:

```
$ docker run -d --name nginx nginx
4562ab6630747983e6d9c59d839aef95728b22a48f7aff3ad6b466dd70ebd0fe
$ docker top nginx
PID      USER    TIME                COMMAND
91413    root    0:00                nginx: master process nginx -g daemon off;
91458    101     0:00                nginx: worker process
```

Вполне логично, что при выполнении на сервере код `nginx` запускается от имени суперпользователя. По умолчанию он принимает запросы на обычном веб-порте 80. А для открытия портов с номером меньше 1024 необходима привилегия `CAP_NET_BIND_SERVICE` (см. главу 2), и самый простой способ гарантировать ее наличие — запускать `nginx` от имени суперпользователя. Но в контейнере подобное требование особого смысла не имеет, ведь там проброс портов означает, что код `nginx` может прослушивать на любом порте с пробросом на порт 80 (при необходимости) на хосте.

Многие поставщики признали, что выполнение от имени суперпользователя — это проблема с точки зрения безопасности, и предоставляют образы Docker, выполняемые от имени обычных, непривилегированных пользователей. Например, репозиторий Dockerfile для Nginx можно найти по адресу https://github.com/nginxinc/docker-nginx-unprivileged.

Сборка образа Nginx, выполняемого от имени несуперпользователя, не является чем-то тяжелым (см. простой пример на https://oreil.ly/UFmcG). Для других приложений это может оказаться задачей посложнее и потребовать изменений кода, более обширных, чем несколько исправлений в Dockerfile и конфигурации. К счастью, компания Bitnami (https://oreil.ly/W4nV2) взяла на себя труд по созданию и поддержанию в актуальном состоянии набора образов контейнеров, не требующих `root`, для многих популярных приложений.

Еще одна причина, по которой образы контейнеров иногда требуют запуска от имени суперпользователя: им иногда нужно устанавливать программное обеспечение с помощью таких систем управления пакетами, как `yum` и `apt`. Логично выполнять это *во время сборки образа контейнера*, но в Dockerfile можно включить вслед за установкой шаг с командой `USER`, чтобы запускать образ от имени несуперпользователя.

Я настоятельно рекомендую вам не разрешать контейнерам устанавливать дополнительное программное обеспечение во время выполнения. И тому есть несколько причин.

- Нерациональность: в случае установки всего необходимого ПО во время сборки достаточно сделать это один раз, а не повторять всю процедуру при каждом создании нового экземпляра контейнера.
- Пакеты, устанавливаемые во время выполнения, не были просканированы на уязвимости (см. главу 7).
- Вероятно, еще худший, чем отсутствие сканирования на уязвимости, хотя в чем-то родственный факт: при установке во время выполнения сложно отследить, какие точно версии пакетов установлены во всех различных запущенных экземплярах контейнеров. Поэтому, даже если у вас появится информация о какой-то уязвимости, вы не будете знать, какие контейнеры необходимо удалить и развернуть заново с исправлениями.
- В зависимости от приложения в случае установки во время сборки контейнер можно запустить с доступом только для чтения (с помощью опции `--read-only` команды `docker run` или путем установки в значение `true` поля `ReadOnlyRootFileSystem` объекта PodSecurityPolicy Kubernetes), что значительно усложнит для злоумышленника установку кода.
- Добавление пакетов во время выполнения означает, что контейнер не рассматривается как неизменяемый. О преимуществах неизменяемых контейнеров с точки зрения безопасности рассказывается в подразделе «Неизменяемые контейнеры» на с. 124.

Еще одно действие, которое можно произвести только от имени суперпользователя, — изменение ядра. Если вы хотите разрешить своим контейнерам делать это — пеняйте на себя!

Если вам интересно, какие опасности несет выполнение контейнера от имени суперпользователя в случае применения Kubernetes, то на https://github.com/lizrice/running-with-scissors вы можете найти несколько примеров.

Везде, где возможно, старайтесь запускать код своих приложений от имени не суперпользователей или задействовать для него пространства имен пользователей (как показано в разделе «Пространство имен пользователей» на с. 74), чтобы суперпользователь внутри контейнера отличался от суперпользователя хоста. Один из удобных на практике способов применения пространств имен — *контейнеры, не требующие полномочий суперпользователя*, если ваша система их поддерживает.

Контейнеры, не требующие полномочий суперпользователя

Если вы внимательно изучали примеры в главе 4, то знаете, что выполнение некоторых действий, участвующих в создании контейнера, требует полномочий суперпользователя. Такой вариант часто не подходит в обычной среде совместно используемой машины, когда на одну и ту же машину может входить несколько пользователей. Примером этого может служить компьютерная система университета, в которой на совместно используемой машине или кластере машин создаются учетные записи для студентов и преподавателей. Системный администратор, само собой, откажется предоставлять всем подряд полномочия суперпользователя для создания контейнеров и будет прав, ведь такие полномочия позволят делать все что угодно (случайно или намеренно) с кодом и данными других пользователей.

В последние годы участники инициативы Rootless Containers (https://rootlesscontaine.rs/) активно работают над изменениями ядра, которые позволили бы несуперпользователям запускать контейнеры.

В системе Docker запускать контейнер можно, будучи не суперпользователем, а лишь членом группы docker, имеющей права на отправку команд через сокеты Docker демону Docker. Стоит отметить, что эти возможности эквивалентны полномочиям суперпользователя на хосте. Любой член данной группы может запускать контейнеры, и, как вы уже знаете, по умолчанию они будут запускаться от имени суперпользователя. А если он смонтирует корневой каталог хоста с помощью команды docker run -v /:/host <образ>, то получит и полный доступ к корневой файловой системе хоста.

Контейнеры, не требующие полномочий суперпользователя, активно применяют функциональность пространств имен пользователей, которую вы

видели в разделе «Пространство имен пользователей» на с. 74. Обычный идентификатор несуперпользователя на хосте можно связать с суперпользователем в контейнере. Если злоумышленнику и удастся неким образом выйти за рамки контейнера, то не получит автоматически полномочий суперпользователя, так что с точки зрения безопасности это значительный прогресс.

Реализация контейнеров `podman` поддерживает контейнеры, не требующие полномочий суперпользователя, и не использует привилегированный процесс-демон, как Docker. Именно поэтому примеры в начале данной главы ведут себя иначе в случае применения `docker` в качестве псевдонима для `podman`.

Узнать больше о root внутри и снаружи контейнера podman можно в сообщении из блога Скотта Маккарти (Scott McCarty) (https://oreil.ly/ISuFf).

Впрочем, контейнеры, не требующие полномочий суперпользователя, не панацея. Не всякий образ, успешно работающий при запуске от имени суперпользователя в обычном контейнере, будет вести себя так же в контейнере, не требующем полномочий суперпользователя, хотя с точки зрения контейнера он запускается от имени суперпользователя. Причина — в некоторых тонких нюансах работы системы привилегий Linux.

Как утверждает документация (https://oreil.ly/iZiaw) по пространствам имен пользователей, они изолируют не только идентификаторы пользователей и групп, но и другие атрибуты, включая привилегии. Другими словами, можно добавлять или убирать привилегии процесса в пространстве имен пользователя, причем они будут применяться лишь в пределах данного пространства имен. Поэтому добавленная привилегия для контейнера, не требующего полномочий суперпользователя, будет применяться исключительно в этом контейнере, но не в том случае, если у контейнера должен быть доступ к другим ресурсам хоста.

Дэн Уолш (Dan Walsh) привел в блоге (https://oreil.ly/1fwZP) несколько неплохих примеров этого. Один из них относится к привязке к портам с номером меньше 1024, требующей привилегии `CAP_NET_BIND_SERVICE`. Обычный контейнер, который запущен с данной привилегией (присутствующей по умолчанию, если контейнер запущен от имени суперпользователя) и совместно

с хостом использует пространство имен сети, сможет привязаться к любому порту хоста. Контейнер, не требующий полномочий суперпользователя, тоже с привилегией `CAP_NET_BIND_SERVICE`, использующий сеть совместно с хостом, не сможет привязаться к портам с номером меньше 1024, поскольку привилегия неприменима вне пространства имен пользователя контейнера.

По большому счету, ограничение привилегий пространствами имен — хорошая вещь, ведь она позволяет контейнеризованным процессам выполняться вроде бы от имени суперпользователя, но без возможности производить действия, требующие привилегий на системном уровне, например менять время или перезагружать машину. Абсолютное большинство приложений, которые могут работать в обычном контейнере, смогут успешно работать и в контейнере, не требующем полномочий суперпользователя.

В случае применения контейнеров, не требующих полномочий суперпользователя, процесс, который с точки зрения контейнера выполняется от имени суперпользователя, с точки зрения хоста запущен от имени обычного пользователя. Что любопытно, вследствие этого, в частности, права доступа к файлам у контейнера, не требующего полномочий суперпользователя, не обязательно такие же, какими были бы без такой подмены пользователей. Чтобы обойти данную проблему, файловая система должна поддерживать подмену владельцев файлов и групп в пространстве имен пользователя. (На момент написания книги не все файловые системы могут похвастаться такой поддержкой.)

На момент написания книги контейнеры, не требующие полномочий суперпользователя, все еще находятся в стадии становления. Их поддерживают некоторые среды выполнения, включая `runc` и `podman`; вдобавок есть определенная экспериментальная поддержка в Docker (https://oreil.ly/GnOoq). Вне зависимости от среды выполнения, в Kubernetes использовать контейнеры, не требующие полномочий суперпользователя, пока не получится, хотя Акихиро Суда (Akihiro Suda) и другие подтвердили практическую возможность этого в проекте Usernetes (https://oreil.ly/42RRY).

Сама по себе работа от имени суперпользователя внутри контейнера — не проблема, ведь злоумышленнику еще надо как-то выйти за рамки контейнера. Но уязвимости выхода за рамки контейнера время от времени находят и, вероятно, будут находить в будущем. Но уязвимость среды выполнения — не единственный способ выйти за рамки контейнера. Далее в этой главе вы увидите, как рискованные настройки контейнеров сильно упрощают выход за рамки контейнера без всяких уязвимостей. В сочетании с контейнерами,

запущенными от имени суперпользователя, подобные неудачные конфигурации — готовый рецепт для катастрофы.

Благодаря переопределению идентификаторов пользователей и контейнерам, не требующим полномочий суперпользователя, можно избежать выполнения контейнеров от имени суперпользователя. Неважно, как вы этого добьетесь, но лучше попытаться избежать запуска контейнеров от имени `root`.

Флаг --privileged и привилегии

Docker и прочие среды выполнения контейнеров позволяют указывать опцию `--privileged` при запуске контейнера. Эндрю Мартин (Andrew Martin) назвал эту опцию «наиболее опасным флагом в истории вычислительной техники», и не без оснований: возможности ее чрезвычайно широки, вдобавок ее очень часто используют неправильно.

Многие считают, что флаг `--privileged` эквивалентен выполнению контейнера от имени суперпользователя, но вы ведь уже знаете, что контейнеры выполняются от имени суперпользователя по умолчанию. Так какими же еще полномочиями этот флаг наделяет контейнер?

Ответ: хотя в Docker процесс выполняется от имени идентификатора пользователя `root` по умолчанию, значительная группа обычных привилегий пользователя `root` Linux не предоставляется ему автоматически. (Чтобы освежить в памяти, что такое привилегии, загляните в раздел «Привилегии Linux» на с. 44.)

С помощью утилиты `capsh` можно легко посмотреть, какие привилегии получает контейнер, сначала в контейнере без флага `--privileged`, а затем — с ним (я опустила часть выводимой информации для большей ясности):

```
vagrant@vagrant:~$ docker run --rm -it alpine sh -c 'apk add -U libcap; capsh
--print | grep Current'
...
Current: = cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,
cap_setuid,cap_setpcap,cap_net_bind_service,cap_net_raw,cap_sys_chroot,cap_mknod,
cap_audit_write,cap_setfcap+eip

vagrant@vagrant:~$ docker run --rm -it --privileged alpine sh -c 'apk add -U
libcap; capsh --print | grep Current'
...
Current: = cap_chown,cap_dac_override,cap_dac_read_search,cap_fowner,cap_fsetid,
cap_kill,cap_setgid,cap_setuid,cap_setpcap,cap_linux_immutable,
```

```
cap_net_bind_service,cap_net_broadcast,cap_net_admin,cap_net_raw,cap_ipc_lock,
cap_ipc_owner,cap_sys_module,cap_sys_rawio,cap_sys_chroot,cap_sys_ptrace,
cap_sys_pacct,cap_sys_admin,cap_sys_boot,cap_sys_nice,cap_sys_resource,
cap_sys_time,cap_sys_tty_config,cap_mknod,cap_lease,cap_audit_write,
cap_audit_control,cap_setfcap,cap_mac_override,cap_mac_admin,cap_syslog,
cap_wake_alarm,cap_block_suspend,cap_audit_read+eip
```

Точный список предоставляемых без флага `--privileged` привилегий зависит от конкретной реализации. OCI определяет набор по умолчанию (https://oreil.ly/ryVjj), предоставляемый `runc`.

Данный набор по умолчанию включает `CAP_SYS_ADMIN`, и один этот флаг привилегии предоставляет доступ к огромному множеству привилегированных действий, включая операции с пространствами имен и монтирование файловых систем.

Эрик Чан (Eric Chiang) написал в блоге (https://oreil.ly/4f4QO) сообщение об опасностях флага --privileged и привел пример выхода из контейнера в файловую систему хоста с помощью монтирования устройства из /dev к файловой системе контейнера.

Флаг `--privileged` появился в Docker ради функциональности «Docker в Docker», широко используемой для сборки утилит и систем CI/CD, которые работают в виде контейнеров, требующих доступа к демону Docker для сборки с помощью Docker образов контейнеров. Но, как описано в этом сообщении блога (https://oreil.ly/-ULQo), функциональность «Docker в Docker», да и вообще флаг `--privileged`, следует использовать осмотрительно.

Есть и менее очевидная причина, по которой флаг `--privileged` столь опасен, — поскольку люди часто считают, что контейнерам необходимо предоставлять полномочия суперпользователя, то считают верным и обратное: что запущенный без данного флага контейнер не является процессом, работающим от имени суперпользователя. Если вы не вполне убеждены в этом, то загляните еще раз в раздел «Выполнение контейнеров по умолчанию от имени суперпользователя» на с. 147.

Даже если у вас есть серьезные причины запускать контейнеры с флагом `--privileged`, я рекомендую вам описать средства контроля или по крайней мере аудита использования его только для тех контейнеров, которым он действительно нужен. Вместо него лучше при возможности задавать отдельные привилегии.

Для трассировки событий `cap_capable` можно воспользоваться утилитой Tracee (https://oreil.ly/1dQof) (см. главу 8), которая отображает привилегии, требуемые у ядра заданным контейнером.

Ниже приведен пример нескольких первых событий из трассировки контейнера с запущенным `nginx` (часть вывода была убрана ради большей понятности).

Терминал 1:

```
$ docker run -it --rm nginx
```

Терминал 2:

```
root@vagrant$ ./tracee.py -c -e cap_capable
TIME(s)  UTS_NAME      UID  EVENT         COMM    PID   PPID   RET  ARGS
125.000  c8520fe719e5  0    cap_capable   nginx   6     1      0    CAP_SETGID
125.000  c8520fe719e5  0    cap_capable   nginx   6     1      0    CAP_SETGID
125.000  c8520fe719e5  0    cap_capable   nginx   6     1      0    CAP_SETUID
124.964  c8520fe719e5  0    cap_capable   nginx   1     3500   0    CAP_SYS_ADMIN
124.964  c8520fe719e5  0    cap_capable   nginx   1     3500   0    CAP_SYS_ADMIN
```

Выяснив, какие привилегии необходимы контейнеру, вы можете, следуя принципу минимума полномочий, указать в среде выполнения лишь необходимые. Рекомендуемый подход: убрать все привилегии, а затем добавить обратно только нужные, вот так:

```
$ docker run --cap-drop=all --cap-add=<привил> --cap-add=<привил2> <образ> ...
```

Теперь вы предупреждены обо всех опасностях флага `--privileged` и возможностях, позволяющих точно задать привилегии для контейнеров. Рассмотрим теперь другой способ обойти изоляцию контейнеров — монтирование каталогов с конфиденциальными данными с хоста.

Монтирование каталогов с конфиденциальными данными

С помощью опции `-v` можно примонтировать каталог хоста к контейнеру и получить из контейнера доступ к нему. При этом ничто не мешает вам примонтировать к контейнеру корневой каталог хоста, вот так:

```
$ touch /ROOT_FOR_HOST
$ docker run -it -v /:/hostroot ubuntu bash
root@91083a4eca7d:/$ ls /
bin   dev  home      lib    media  opt   root  sbin  sys  usr
```

```
boot  etc  hostroot  lib64  mnt    proc  run   srv   tmp  var
root@91083a4eca7d:/$ ls /hostroot/
ROOT_FOR_HOST  etc             lib         media  root  srv  vagrant
bin            home            lib64       mnt    run   sys  var
...
```

Взломавший этот контейнер злоумышленник оказывается суперпользователем на хосте и получает полный доступ ко всей файловой системе хоста.

Монтирование всей файловой системы — пример скорее патологический, но существует и множество других примеров разной степени изощренности.

- Монтирование /etc позволит модифицировать файл `/etc/passwd` хоста изнутри контейнера, а также нарушать работу заданий `cron`, `init` и `systemd`.
- Монтирование /bin и тому подобных каталогов, например /usr/bin или /usr/sbin, позволит контейнеру записывать исполняемые файлы в каталог хоста — в том числе перезаписывать уже существующие исполняемые файлы.
- Монтирование каталогов журналов хоста к контейнеру может открыть для злоумышленника возможность изменять журналы, а значит, и ликвидировать следы его мерзких деяний на этом хосте.
- В среде Kubernetes монтирование /var/log открывает доступ ко всей файловой системе хоста любому пользователю, у которого есть доступ к команде `kubectl logs`. А все потому, что файлы журналов контейнера представляют собой символические ссылки из каталога /var/log на прочие места файловой системы, но ничто не мешает контейнеру создать символическую ссылку на любой другой файл. Больше информации об этой интересной разновидности выхода за рамки контейнера можно найти в следующем сообщении блога: https://oreil.ly/gN7no.

Монтирование сокета Docker

В среде Docker всю работу, по существу, выполняет процесс-демон Docker. При запуске утилиты командной строки `docker` он отправляет демону инструкции через сокет Docker, расположенный в `/var/run/docker.sock`. Любой, кто может записывать данные в этот сокет, может и отправлять инструкции демону Docker. Демон выполняется от имени суперпользователя и легко может произвести для вас сборку и запуск любого программного обеспечения,

включая — как вы видели — запуск контейнера от имени суперпользователя на хосте. Следовательно, доступ к сокету Docker, по существу, эквивалентен доступу с полномочиями суперпользователя на хосте.

Сокет Docker очень часто монтируют в утилитах CI наподобие Jenkins, где он необходим для отправки Docker инструкций по запуску сборки образов, в виде составной части конвейера. Это вполне допустимое его применение, которое, однако, создает потенциальное уязвимое место на радость злоумышленнику. Пользователь, у которого есть возможность изменять Jenkinsfile, может заставить Docker выполнять нужные ему команды, в том числе предоставляющие пользователю доступ к кластеру с полномочиями суперпользователя. Поэтому крайне не рекомендуется задействовать при промышленной эксплуатации конвейер CI/CD, монтирующий сокет Docker.

Совместное использование пространств имен контейнером и его хостом

Иногда возникают основания для того, чтобы контейнер использовал некоторые из тех же пространств имен, что и его хост. Например, допустим, процессу в контейнере Docker нужен доступ к информации о процессе с хоста. В Docker подобную возможность можно получить с помощью параметра `--pid=host`.

Напомню, что все контейнеризованные процессы видны на хосте; следовательно, совместное использование пространства имен процессов с контейнером позволяет ему видеть другие контейнеризованные процессы. Следующий пример начинается с выполнения длительной операции `sleep` внутри одного контейнера; этот процесс оказывается виден из другого контейнера, запущенного с опцией `--pid=host`:

```
vagrant@vagrant$ docker run --name sleep --rm -d alpine sleep 1000
fa19f51fe07fca8d60454cf8ee32b7e8b7b60b73128e13f6a01751c601280110
vagrant@vagrant$ docker run --pid=host --name alpine --rm -it alpine sh
/ $ ps | grep sleep
30575 root      0:00 sleep 1000
30738 root      0:00 grep sleep
/ $
```

Еще более поразительный факт: выполнение `kill -9 <pid>` из второго контейнера позволяет прерывать выполнение процесса `sleep` в первом!

Я показала выше несколько вариантов того, как совместное использование пространств имен или томов контейнерами либо контейнером и его хостом может ослабить изоляцию контейнера и нарушить безопасность. Но это никоим образом не означает, что обмен информацией с контейнерами — *всегда* плохая идея. В завершение главы рассмотрим вспомогательные контейнеры, которые не случайно используются так широко.

Вспомогательные контейнеры

Вспомогательному контейнеру (sidecar container) умышленно предоставляется доступ к одному из пространств имен контейнера приложения в целях делегирования ему какой-либо функциональности данного приложения. В микросервисной архитектуре встречается функциональность, которую необходимо повторно использовать во всех микросервисах системы, поэтому нередко подобную функциональность помещают в образы вспомогательных контейнеров для удобства повторного использования. Ниже представлены несколько распространенных примеров.

- Вспомогательные контейнеры типа service mesh берут на себя сетевую функциональность контейнера приложения. Service mesh может, например, обеспечить использование во всех сетевых подключениях взаимного протокола TLS. Делегирование этой функциональности вспомогательному контейнеру означает, что всегда, когда контейнер развертывается вместе со вспомогательным, он будет устанавливать защищенные TLS-соединения; и каждой команде разработчиков не нужно тратить время и включать эту возможность в код приложения. (Дальнейшее обсуждение service mesh вы найдете в следующей главе — см. раздел «Service mesh» на с. 177.)
- Вспомогательные контейнеры наблюдения позволяют задавать цели и настройки журналирования, трассировки и сбора метрик. Например, Prometheus (https://oreil.ly/Jn10W) и OpenTelemetry (https://oreil.ly/0HwpE) поддерживают вспомогательные контейнеры для экспорта данных наблюдений.
- Вспомогательные контейнеры безопасности позволяют контролировать, какие исполняемые файлы и сетевые подключения разрешаются внутри контейнера приложения. (Например, см. мое сообщение в блоге (https://oreil.ly/oHAEk) про обеспечение безопасности контейнеров AWS Fargate с помощью MicroEnforcer от компании Aqua во вспомогательных контейнерах или аналогичного решения от Twistlock (https://oreil.ly/5YHQk)).

Это лишь часть приложений вспомогательных контейнеров, на законных основаниях совместно использующих пространства имен с контейнерами приложений.

Резюме

В этой главе рассматривается несколько вариантов нарушения обеспечиваемой контейнерами изоляции вследствие неудачных настроек.

Все опции конфигурации существуют не просто так. Например, монтирование каталогов хоста к контейнеру может оказаться исключительно полезным, и иногда может пригодиться возможность запуска контейнера от имени суперпользователя или даже с дополнительными привилегиями, предоставляемыми флагом `--privileged`. Однако если вас волнует безопасность, то лучше прибегать к этим потенциально небезопасным возможностям как можно реже и применять утилиты для выявления случаев их использования.

При работе в мультиарендной среде следует еще внимательнее отслеживать контейнеры с этими потенциально небезопасными настройками. У любого контейнера с флагом `--privileged` есть полный доступ ко всем прочим контейнерам на том же хосте, вне зависимости от довольно несерьезных средств контроля, например запуска их в одном или разных пространствах имен Kubernetes.

В разделе «Вспомогательные контейнеры» на с. 160 я упоминала контейнеры типа service mesh, которым можно делегировать часть сетевой функциональности. А значит, самое время обсудить передачу данных по сети в контейнерах.

ГЛАВА 10

Сетевая безопасность контейнеров

Внешние атаки на развертываемую систему производятся по сети, поэтому, чтобы обеспечить безопасность приложений и данных, необходимо иметь хотя бы базовые представления о передаче данных по сети. Мы не станем обсуждать эту сложную тему во всех подробностях (иначе книга оказалась бы намного больше), но опишем главные элементы продуманной ментальной модели, которую можно использовать для организации передачи данных по сети при развертывании контейнеров.

Я начну с обзора брандмауэров для контейнеров, позволяющих реализовывать сетевую безопасность на основе намного более детализированного подхода, по сравнению с традиционными брандмауэрами.

Далее вас ждет обзор семиуровневой сетевой модели, о которой необходимо знать, чтобы понимать, на каком уровне действуют конкретные элементы сетевой безопасности. После этого мы обсудим реализацию брандмауэров для контейнеров и практические рекомендации для стратегий безопасности сети. Завершается глава рассказом о возможностях обеспечения сетевой безопасности с помощью service mesh.

Брандмауэры для контейнеров

Контейнеры часто сочетаются с микросервисной архитектурой, при которой приложения разбиваются на маленькие компоненты, развертываемые независимо друг от друга. С точки зрения безопасности преимущества такого подхода несомненны, ведь для небольшого компонента намного легче определить «нормальное» поведение. Каждый контейнер при этом, вероятно, будет взаимодействовать лишь с ограниченным количеством других контейнеров, и только некоторое множество контейнеров должно будет взаимодействовать с внешним миром.

Например, рассмотрим разбитое на микросервисы приложение для электронной коммерции. Один из его микросервисов, допустим, может отвечать за поиск товаров: получать поисковые запросы от конечных пользователей и производить поиск в базе товаров. Контейнерам, составляющим этот сервис, вовсе не требуется обмениваться информацией, скажем, с платежным шлюзом. Рисунок 10.1 иллюстрирует данный пример.

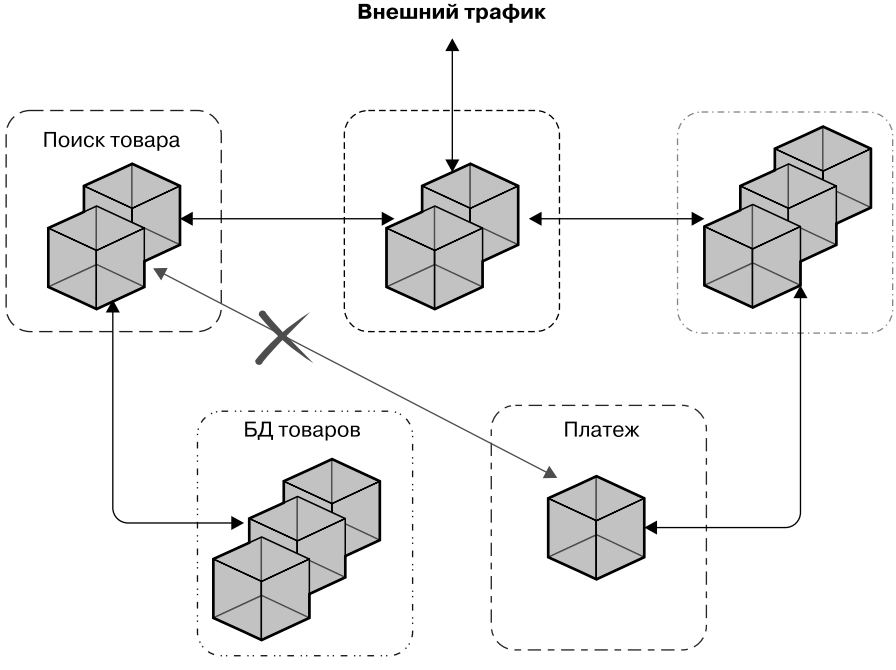


Рис. 10.1. Защита контейнеров брандмауэром

Брандмауэры для контейнеров могут ограничивать трафик в набор контейнеров и из него. В средствах координации, например Kubernetes, термин «брандмауэр для контейнера» используется редко, обычно речь идет о соблюдении стратегий безопасности сети, обеспечиваемой сетевым плагином. В обоих случаях идея заключается в ограничении входящего/исходящего сетевого трафика контейнера только заданными пунктами назначения. Брандмауэр для контейнера (и его более традиционный аналог) также обычно уведомляет о попытках неразрешенных правилами соединений, тем самым предоставляя полезную информацию для дальнейшего расследования возможных атак.

Брандмауэры для контейнеров можно использовать совместно с прочими утилитами сетевой безопасности, встречающимися и при обычном варианте развертывания системы. Например:

- среду выполнения контейнеров нередко развертывают в виртуальных частных облаках (VPC), изолирующих хосты от остального мира;
- с помощью брандмауэра можно контролировать входящий/исходящий трафик всего кластера;
- ограничивать трафик на уровне 7 можно с помощью API-брандмауэров, известных также под названием брандмауэров веб-приложений (Web Application Firewall, WAF).

Ни один из этих подходов не нов. В сочетании со средствами безопасности, учитывающими специфику контейнеров, они обеспечивают дополнительный уровень защиты.

Прежде чем заниматься нюансами того, как реализуется защита контейнеров брандмауэрами, вспомним устройство семиуровневой сетевой модели и проследим путь IP-пакетов по сети.

Сетевая модель OSI

Сетевая модель взаимодействия открытых систем (Open Systems Interconnection, OSI), опубликованная в 1984 году, описывает многоуровневую модель обмена информацией по сети, которую и сегодня часто упоминают, хотя, как вы видите на рис. 10.2, не у всех из ее семи уровней есть эквиваленты в IP-сетях.

Уровень 7	Прикладной	HTTP
Уровень 6	Представления	
Уровень 5	Сеансовый	
Уровень 4	Транспортный	TCP
Уровень 3	Сетевой	IP
Уровень 2	Канальный	Например, Ethernet
Уровень 1	Физический	

Рис. 10.2. Модель OSI

- Уровень 7 — прикладной (уровень приложений). Представьте приложение, которое выполняет веб-запрос или отправляет запрос к воплощающему REST API, — и поймете, что происходит на уровне 7. Такой запрос обычно направляется по URL, и в целях его маршрутизации к месту назначения доменное имя преобразуется в IP-адрес с помощью протокола разрешения доменных имен (Domain Name Resolution), реализуемого сервисом доменных имен (Domain Name Service, DNS).
- Уровень 4 — транспортный, обычно данные передаются по нему в виде TCP или UDP-пакетов. На этом уровне используются номера портов.
- Уровень 3 — уровень, на котором перемещаются IP-пакеты и работают IP-маршрутизаторы. Каждой IP-сети выделяется множество IP-адресов, и когда контейнер присоединяется к ней, ему назначается один из этих IP-адресов. В данной главе нам неважно, используется ли в IP-сети протокол IP v4 или IP v6, — можете считать это несущественным нюансом реализации.
- Уровень 2 — здесь пакеты данных направляются к конечным точкам, соединенным с физическим или виртуальным интерфейсом (которые мы обсудим чуть позже). Существует несколько протоколов уровня 2, включая Ethernet, Wi-Fi и, если мысленно перенестись в прошлое, Token Ring. (Wi-Fi здесь немного все запутывает, поскольку охватывает как уровень 2, так и уровень 1.) В этой главе я буду рассматривать только протокол Ethernet — именно он в основном используется для передачи данных по сети в контейнерах.
- Уровень 1 называется физическим, хотя учтите, что интерфейсы на этом уровне могут быть виртуальными. У физического компьютера есть физическое сетевое устройство, подключенное к кабелю или беспроводному передатчику. Если вы мысленно вернетесь к главе 5, то вспомните, что VMM предоставляет гостевому ядру доступ к виртуальным устройствам, соответствующим этим реальным устройствам. При наличии сетевого интерфейса, скажем, на виртуальном узле EC2 в AWS вы получаете доступ к одному из этих виртуальных интерфейсов. Сетевые интерфейсы контейнеров на уровне 1 тоже обычно виртуальные. Присоединяясь к сети, контейнер получает интерфейс уровня 1 для нее.

Посмотрим, что происходит на каждом из этих уровней, когда приложение хочет отправить сообщение.

Отправка IP-пакета

Представьте приложение, которое хочет отправить сообщение на целевой URL. Поскольку речь идет о приложении, то из предыдущего определения следует, что это происходит на уровне 7.

Первый шаг — поиск по DNS IP-адреса, соответствующего имени хоста в этом URL. DNS может быть задан локально (например, в файле `/etc/hosts` на вашем ноутбуке) или разрешаться путем DNS-запроса к удаленному сервису, расположенному по заданному в настройках IP-адресу. (Если приложению известен IP-адрес, по которому нужно отправить сообщение, а не URL, то шаг разрешения DNS пропускается.)

Когда сетевой стек знает целевой IP-адрес пакета, производится выбор маршрутизации на уровне 3, состоящий из двух частей.

1. От целевого пункта нас может отделять несколько транзитных участков IP-сети. При заданном целевом IP-адресе каков IP-адрес следующего транзитного участка?
2. Какой интерфейс соответствует этому IP-адресу следующего транзитного участка?

Далее пакет преобразуется в кадры Ethernet, а IP-адрес следующего транзитного участка сопоставляется с соответствующим MAC-адресом. Для этого используется протокол разрешения адресов (Address Resolution Protocol, ARP), который сопоставляет IP-адреса с MAC-адресами. Если сетевому стеку пока еще не известен MAC-адрес, который соответствует следующему IP-адресу (он может уже содержаться в кэше ARP), то его можно выяснить, применив ARP.

После того как сетевой стек получит MAC-адрес следующего транзитного участка, сообщение отправляется через интерфейс. В зависимости от реализации сети для этого может использоваться прямое соединение либо интерфейс может подключаться к *мосту* (bridge).

Чтобы понять, что такое мост, представьте физическое устройство, в которое включены несколько кабелей Ethernet. Вторые концы всех кабелей включены в сетевые карты устройств — например, компьютеров. Каждой физической сетевой карте соответствует уникальный MAC-адрес, зашитый в нее производителем. Мост выясняет MAC-адреса всех устройств, расположенных на

вторых концах кабелей, включенных в его интерфейс. Все подключенные к мосту устройства могут отправлять через него друг другу пакеты. При передаче данных по сети между контейнерами мосты реализованы программно, а не аппаратно, а роль кабелей Ethernet играют виртуальные интерфейсы Ethernet. Таким образом, сообщение поступает в мост, который на основе MAC-адреса следующего транзитного участка определяет, в какой интерфейс его направить.

Когда сообщение достигает другого конца Ethernet-соединения, IP-пакет извлекается и переводится обратно на уровень 3. Данные инкапсулируются с заголовками на различных уровнях сетевого стека, как показано на рис. 10.3.

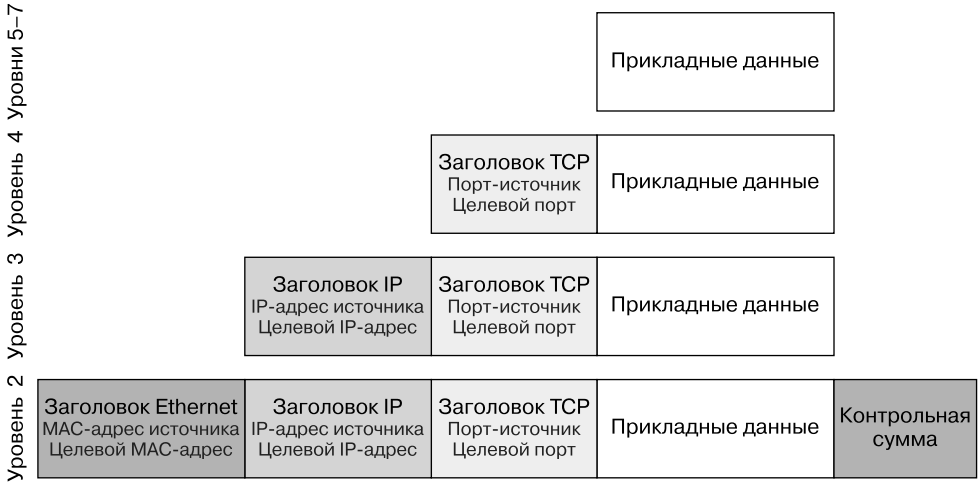


Рис. 10.3. Сетевые заголовки

Если это конечный пункт назначения пакета, то данные передаются приложению-получателю. Впрочем, если для пакета это всего лишь очередной транзитный участок, то сетевому стеку приходится снова принимать решение о том, куда отправлять пакет далее.

В данном пояснении опущены некоторые детали (например, механизм работы ARP и способ определения IP-адреса следующего транзитного участка), но этого должно быть достаточно для нашего обсуждения передачи данных по сети в контейнерах.

IP-адреса контейнеров

В предыдущем разделе мы говорили о том, как трафик достигает места назначения на основе IP-адреса. IP-адреса контейнеров могут совпадать с IP-адресами их хостов, но для них может применяться и отдельный сетевой стек, оперирующий в их собственном пространстве имен сети. О таких пространствах мы говорили в главе 4. А поскольку весьма вероятно, что вы запускаете свои контейнеры под управлением Kubernetes, посмотрим, как IP-адреса используются в Kubernetes.

У каждого модуля в Kubernetes — свой IP-адрес. Если модуль содержит более одного контейнера, то, как вы понимаете, все они используют один и тот же IP-адрес, благодаря тому что делят одно и то же пространство имен. Каждому из узлов выделяется свой диапазон адресов (блок CIDR), а когда модуль распределяется на узел, то ему присваивается один из адресов этого диапазона.

Строго говоря, узлам далеко не всегда заранее выделяются диапазоны адресов. Например, в AWS подключаемый модуль управления IP-адресами динамически распределяет IP-адреса по модулям Kubernetes из соответствующего VPC диапазона.

Kubernetes требует, чтобы модули в кластере могли связываться друг с другом напрямую, без какого-либо преобразования сетевых адресов (Network Address Translation, NAT). В некоторых случаях NAT отображает IP-адреса таким образом, что пункт назначения представляется IP-адресом, хотя на самом деле не является фактическим адресом целевого устройства. (Именно благодаря этому IPv4 используется гораздо дольше, чем планировалось изначально. Хотя количество адресуемых по протоколу IP устройств намного превысило количество адресов, доступных в пространстве адресов IPv4, NAT позволяет повторно задействовать большинство IP-адресов в частных сетях.) В Kubernetes стратегии безопасности сети и разбиение на сегменты могут мешать модулям взаимодействовать друг с другом, но если уж два модуля могут обмениваться информацией, то видят IP-адреса друг друга в открытую, без какого-либо преобразования NAT. Однако NAT между модулями и внешним миром вполне возможен.

Сервисы Kubernetes представляют собой некую разновидность NAT. Сервис Kubernetes — самостоятельный ресурс, которому присваивается свой

IP-адрес. Впрочем, это всего лишь IP-адрес — у такого сервиса нет никаких интерфейсов, он не прослушивает на предмет трафика и не отправляет собственный трафик. Его адрес используется только для маршрутизации. Каждому сервису соответствует несколько модулей, фактически выполняющих вместо него всю работу, вследствие чего пакеты, посылаемые на IP-адрес данного сервиса, приходится перенаправлять одному из этих модулей. Вскоре мы посмотрим, как это происходит с помощью правил на уровне 3.

Сетевая изоляция

На всякий случай уточню, что компоненты могут взаимодействовать друг с другом, только если подключены к одной сети. В обычных средах на основе хостов можно было изолировать приложения друг от друга, размещая их в разных VLAN.

В мире контейнеров Docker значительно упрощает настройку большого количества сетей благодаря команде `docker network`, но в модель Kubernetes, где любой модуль может (если не учитывать сетевые стратегии и настройки безопасности) обращаться по IP-адресу к любому другому модулю, подобное вписывается плохо.

Кроме того, обратите внимание: в Kubernetes управляющие компоненты запускаются в модулях и все подключаются к той же сети, что и прикладные модули. Если вы ранее занимались средствами связи, то вышесказанное может вас удивить, поскольку в телефонных сетях было приложено немало усилий для разделения плоскости управления с плоскостью данных, в основном из соображений безопасности.

В Kubernetes же передача данных по сети между контейнерами основывается на сетевых стратегиях (https://oreil.ly/sHh4p), работающих на уровнях 3/4.

Маршрутизация на уровнях 3/4 и правила

Как вы уже знаете, основная задача маршрутизации на уровне 3 — выбор следующего транзитного участка для IP-пакета. Это решение основывается на наборе правил, которые описывают, какой интерфейс следует использовать для достижения тех или иных адресов. Но это лишь малая часть возможностей

правил уровня 3: на нем можно производить много других интересных вещей, например игнорировать пакеты или выполнять различные операции с IP-адресами, скажем, в целях реализации балансировки нагрузки, NAT, брандмауэров и стратегий безопасности сети. Правила также могут применяться на уровне 4, для учета номеров портов. В основе этих правил лежит возможность netfilter (https://netfilter.org/) ядра.

Средство `netfilter` служит для фильтрации пакетов; впервые появилось в версии 2.4 ядра Linux. В нем используется набор правил, описывающий, что делать с пакетом в зависимости от его адреса-источника и целевого адреса.

Настраивать правила `netfilter` в пользовательском пространстве можно несколькими способами. Рассмотрим два наиболее распространенных варианта: `iptables` и `IPVS`.

Утилита iptables

Утилита `iptables` — один из способов настройки правил обработки IP-пакетов, работающий с ядром с помощью `netfilter`. Существует несколько типов таблиц. Применительно к передаче данных по сети между контейнерами наиболее интересны две из них — `filter` и `nat`:

- таблица `filter` предназначена для того, чтобы определить, игнорировать пакет или передать его далее;
- таблица `nat` предназначена для преобразования адресов.

Суперпользователь может просмотреть текущий набор правил нужного вида, воспользовавшись командой `iptables -t <`*`имя_таблицы`*`> -L`.

Правила `netfilter`, задаваемые с помощью `iptables`, очень удобны в деле обеспечения безопасности. Утилита `iptables` используется в нескольких брандмауэрах для контейнеров, а также в сетевых плагинах Kubernetes для формирования хитроумных правил сетевых стратегий, реализуемых с помощью правил `netfilter`. Я вернусь к этому вопросу в разделе «Сетевые стратегии» на с. 173. Но сначала займемся правилами, которые задаются с применением `iptables`.

В Kubernetes сетевой прокси kube-proxy использует правила `iptables` для балансировки объемов трафика, направляемого к сервисам. Как упомина-

лось ранее, сервис — это абстракция, которая сопоставляет название сервиса с набором модулей Kubernetes. С помощью DNS название сервиса разрешается (преобразуется) в IP-адрес. Когда поступает пакет, предназначенный для этого IP-адреса сервиса, находится соответствующее данному целевому адресу правило `iptables` и он меняется на целевой адрес одного из модулей. Если набор модулей для этого сервиса меняется, то правила `iptables` перезаписываются соответствующим образом на всех хостах.

Можно легко посмотреть правила `iptables` для сервиса. Для примера возьмем кластер Kubernetes, в котором для сервиса `nginx` используется две реплики:

```
$ kubectl get svc,pods -o wide
NAME                 TYPE        CLUSTER-IP      PORT(S)
service/kubernetes   ClusterIP   10.96.0.1       443/TCP
service/my-nginx     NodePort    10.100.132.10   8080:32144/TCP

NAME                              READY   STATUS    IP
pod/my-nginx-75897978cd-n5rdv   1/1     Running   10.32.0.4
pod/my-nginx-75897978cd-ncnfk   1/1     Running   10.32.0.3
```

Просмотреть текущие правила преобразования адресов можно с помощью команды `iptables -t nat -L`. Вероятно, при этом будет выведено немало информации, но найти в ней интересующие нас части, которые относятся к `nginx`, не так уж сложно. Для начала видим правило, соответствующее сервису `my-nginx`, который запущен по IP-адресу 10.100.132.10. Как видите, оно входит в цепочку KUBE-SERVICES, что логично, поскольку оно относится к сервису:

```
Chain KUBE-SERVICES (2 references)
target                     prot opt source     destination
...
KUBE-SVC-SV7AMNAGZFKZEMQ4  tcp  --  anywhere   10.100.132.10   /* default/my-
nginx:http cluster IP */ tcp dpt:http-alt
...
```

Это правило задает целевую цепочку, встречающуюся далее в правилах `iptables`:

```
Chain KUBE-SVC-SV7AMNAGZFKZEMQ4 (2 references)
target                     prot opt source     destination
KUBE-SEP-XZGVVMRRSKK6PWWN  all  --  anywhere   anywhere      statistic mode
random probability 0.50000000000
KUBE-SEP-PUXUHBP3DTPPX72C  all  --  anywhere   anywhere
```

Из вышесказанного можно сделать разумный вывод о том, что трафик поровну разбит между этими двумя пунктами назначения. Логика становится

понятной, когда мы видим, что для этих пунктов назначения есть правила, соответствующие IP-адресам модулей (10.32.0.3 и 10.32.0.4):

```
Chain KUBE-SEP-XZGVVMRRSKK6PWWN (1 references)
target          prot opt source               destination
KUBE-MARK-MASQ  all  --  10.32.0.3            anywhere
DNAT            tcp  --  anywhere             anywhere        tcp
to:10.32.0.3:80
...
Chain KUBE-SEP-PUXUHBP3DTPPX72C (1 references)
target          prot opt source               destination
KUBE-MARK-MASQ  all  --  10.32.0.4            anywhere
DNAT            tcp  --  anywhere             anywhere        tcp
to:10.32.0.4:80
```

Основная проблема при работе с `iptables` заключается в падении производительности при большом количестве сложных правил на каждом хосте. На самом деле то, как kube-proxy использует `iptables`, — известное узкое место в смысле производительности при работе Kubernetes в крупных системах. В сообщении блога на https://oreil.ly/xOyqb отмечается, что 2000 сервисов по десять модулей каждый означает 20 000 дополнительных правил `iptables` на каждом узле. Чтобы решить эту проблему, в Kubernetes была добавлена возможность использования IPVS для сервисов балансировки нагрузки.

IPVS

Виртуальный сервер IP (IP Virtual Server, IPVS) иногда называют балансировщиком нагрузки уровня 4 или коммутатором LAN уровня 4. Он представляет собой еще одну реализацию сетевых правил, подобную `iptables`, оптимизированную для балансировки нагрузки за счет хранения правил пересылки в хеш-таблицах.

Подобная оптимизация значительно повышает производительность в случае использования kube-proxy, но это не повод делать далеко идущие выводы относительно производительности сетевых плагинов, которые реализуют сетевые стратегии.

Проект Calico опубликовал сравнение производительности iptables и IPVS (https://oreil.ly/xGO0N).

Неважно, с помощью чего строится работа с правилами `netfilter` — `iptables` или IPVS, все равно они функционируют внутри ядра. Оно используется совместно всеми расположенными на хосте контейнерами, поэтому обеспечение соблюдения сетевых стратегий происходит на уровне хоста, а не внутри контейнеров.

Теперь, когда вы уже знаете, как функционируют правила `netfilter`, посмотрим, как с их помощью реализуются сетевые стратегии безопасности.

Сетевые стратегии

Существует множество программных решений для применения сетевых стратегий, как в Kubernetes, так и при других вариантах развертывания контейнеров. За рамками Kubernetes их могут называть брандмауэрами для контейнеров, микросегментацией сети, но основные принципы от этого не меняются.

Сетевые стратегии в Kubernetes применяются к входящему/исходящему трафику различных модулей. Стратегии описываются в разрезе портов, IP-адресов, сервисов или маркированных модулей. Если стратегия запрещает отправку/получение конкретного сообщения, то сети приходится либо отказать в установлении соединения, либо игнорировать пакеты, которые соответствуют данному сообщению. В примере приложения для электронной коммерции из начала данной главы может использоваться, например, стратегия, которая запрещает исходящий трафик контейнера поиска товаров, целевой адрес которого — сервис платежей.

Во многих реализациях сетевых стратегий используются правила `netfilter`. Посмотрим на правило сетевой стратегии Kubernetes, реализованное в утилите `iptables`. Ниже представлен простой объект NetworkPolicy, который разрешает модулям доступ к сервису `my-nginx` только в том случае, когда они помечены `access=true`:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: access-nginx
spec:
  podSelector:
    matchLabels:
      app: my-nginx
```

```
  ingress:
  - from:
    - podSelector:
        matchLabels:
          access: "true"
```

В результате создания подобной сетевой стратегии в таблице `filter` возникает дополнительное правило `iptables`:

```
Chain WEAVE-NPC-INGRESS (1 references)
target     prot opt source       destination
ACCEPT     all  --  anywhere     anywhere             match-set weave-{U;]
TI.l|MdRzDhN7$NRn[t)d src match-set weave-vC070kAfB$if8}PFMX{V9Mv2m dst /* pods:
namespace: default, selector: access=true -> pods: namespace: default, selector:
app=my-nginx (ingress) */
```

Правила `iptables` для сопоставления с сетевой стратегией создает сетевой плагин (https://oreil.ly/VmVCc), а не один из базовых компонентов Kubernetes. В предыдущем примере, как вы, наверное, догадались из названия цепочки, в качестве сетевого плагина я использовала Weave. Правило match-set не предназначено для чтения человеком, но комментарий вполне соответствует нашим ожиданиям относительно того, что это правило разрешает трафик из модулей в пространстве имен по умолчанию с меткой `access=true`, направляемый к модулям в пространстве имен по умолчанию с меткой `app=my-nginx`.

Теперь, когда вы уже посмотрели, как Kubernetes использует правила `iptables` для обеспечения соблюдения сетевых стратегий, попробуем описать собственное правило. Я сделаю это в только что установленной системе Ubuntu, чтобы список правил был пуст:

```
$ sudo iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

Настроим `netcat` отвечать на запросы, поступающие на порт 8000:

```
$ while true; do  echo "hello world" | nc -l 8000 -N; done
```

В другом окне терминала можем теперь отправить запросы на этот порт:

```
$ curl localhost:8000
hello world
```

Теперь можно создать правило `iptables`, которое будет отклонять трафик, поступающий на порт 8000:

```
$ sudo iptables -I INPUT -j REJECT -p tcp --dport=8000
$ sudo iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source       destination
REJECT     tcp  --  anywhere     anywhere         tcp dpt:8000 reject-with icmp-
port-unreachable

Chain FORWARD (policy ACCEPT)
target     prot opt source       destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source       destination
```

Как вы, наверное, понимаете, команде `curl` теперь не удается получить ответ:

```
$ curl localhost:8000
curl: (7) Failed to connect to localhost port 8000: Connection refused
```

Этот пример демонстрирует использование `iptables` для ограничения трафика. Вы можете представить создание множества подобных правил для ограничения трафика между контейнерами, но я не рекомендую вам делать это вручную. На практике удобнее задействовать уже существующие реализации, а не писать собственные стратегии сетевой безопасности непосредственно в виде правил `iptables`. Вы можете с помощью предоставляемого более удобного интерфейса настроить стратегии, вместо того чтобы писать их с нуля, а во многоузловой системе — еще и можете задать различные правила для каждого узла. А правил будет очень много — чтобы вы понимали масштаб, у меня есть один узел Kubernetes с сетевым плагином Calico и всего несколькими запущенными прикладными модулями, без каких-либо сетевых стратегий, и выполнение команды `iptables -L` дает на этой машине более 300 строк правил таблицы filter. Сами правила выполняются быстро, но их написание — непростая задача. Кроме того, время существования контейнеров часто невелико, вследствие чего правила приходится переписывать по мере создания и уничтожения контейнеров. Все это возможно только при автоматизации их создания, а не при написании вручную.

Программные решения для сетевых стратегий

Так чем же можно воспользоваться для подобной автоматизации? В Kubernetes существуют объекты NetworkPolicy, хотя, как уже упоминалось ранее, сам Kubernetes не обеспечивает их соблюдения. Они работают только при

наличии поддерживающего их сетевого плагина (https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/). Отдельные сетевые плагины позволяют перейти на платную версию, которая отличается большей гибкостью или более удобным управлением.

Некоторые коммерческие платформы безопасности контейнеров включают брандмауэры для контейнеров, позволяющие добиться практически тех же результатов, но не устанавливаемые непосредственно в виде сетевых плагинов Kubernetes. Иногда они самостоятельно обучаются распознавать обычный трафик, что позволяет создавать стратегии автоматически.

Практические рекомендации для сетевых стратегий

Какие бы утилиты вы ни использовали для создания, управления и обеспечения соблюдения сетевых стратегий, существует несколько рекомендаций.

- *Отказ по умолчанию*. Следуя принципу минимума полномочий, задавайте стратегии для всех пространств имен, которые бы отклоняли входящий трафик по умолчанию (https://oreil.ly/L6OjC), и лишь затем добавьте стратегии, которые будут позволять прохождение ожидаемого вами трафика.
- *Отказ по умолчанию для исходящего трафика*. Стратегии исходящего трафика относятся к трафику, который покидает ваш модуль. В случае взлома контейнера злоумышленник может начать зондировать окружающую среду по сети. Настройте стратегии для всех пространств имен, которые по умолчанию отклоняют исходящий трафик (https://oreil.ly/RmeUT), а затем добавьте стратегии для ожидаемого вами исходящего трафика.
- *Ограничение трафика между модулями*. Модули обычно маркируются в соответствии с приложением. Ограничивайте трафик с помощью стратегий, чтобы он был возможен лишь между разрешенными приложениями, а также используйте стратегии для разрешения трафика только из модулей с соответствующими метками.
- *Ограничение списка портов*. Ограничивайте прием трафика для каждого из приложений только определенными портами.

Ахмет Алп Балкан (Ahmet Alp Balkan) приводит список полезных рецептов создания сетевых стратегий (https://oreil.ly/JogsQ).

Обсуждавшиеся выше сетевые стратегии работают на нижних уровнях сетевого стека (вплоть до уровня 4). Теперь посмотрим на service mesh, которые работают на прикладных уровнях.

Service mesh

Service mesh[1] предоставляет дополнительный набор привилегий и средств контроля за связью приложений друг с другом, реализованный на прикладном уровне (уровнях 5–7 в модели OSI, которую вы видели в начале данной главы).

В экосистеме нативных облачных сервисов существует несколько различных проектов service mesh, включая Istio, Envoy и Linkerd, а также управляемых вариантов от поставщиков облачных сервисов, например AWS App Mesh. Service mesh предоставляет пользователям различные функциональные возможности и преимущества, часть которых связана с безопасностью.

В Kubernetes service mesh обычно внедряется в модули для каждого из приложений в виде вспомогательного контейнера, после чего он передает данные по сети вместо остальных контейнеров в модуле. Весь входящий/исходящий трафик модуля проходит через этот вспомогательный контейнер. Обеспечение соблюдения правил производится в пользовательском пространстве внутри прокси.

Service mesh можно настроить для использования взаимного TLS в этих вспомогательных контейнерах-прокси, что обеспечивает защищенные, зашифрованные соединения внутри развернутой системы. Благодаря этому злоумышленнику оказывается намного сложнее перехватить трафик, даже если ему и удастся закрепиться в системе. Если вы еще не знакомы с концепцией взаимного TLS, то я расскажу о ней в главе 11.

Service mesh обычно предоставляет возможности для обеспечения соблюдения сетевых стратегий на прикладном уровне, чтобы модули сервиса могли взаимодействовать с другими модулями (внешними или внутренними), только если это разрешено стратегией. А поскольку они работают на прикладном уровне, то существует четкое разделение ответственности между

[1] В русскоязычной литературе встречается несколько не слишком устоявшихся вариантов перевода этого термина, в частности «сервисная сетка», «сервисное сито» и др.

этими стратегиями и сетевыми стратегиями уровней 1–4, которые обсуждались ранее в данной главе.

В документации проекта Istio можно найти пример стратегии изоляции прикладного слоя (https://oreil.ly/6bUHM), которая разрешает только исходящий от конкретного приложения трафик, направленный в конкретные порты модулей.

Взаимный протокол TLS и поддержка стратегий — огромные плюсы service mesh с точки зрения безопасности, но нужно учесть два нюанса:

- service mesh может обеспечить безопасность только тех модулей, в которые он был внедрен в качестве вспомогательного контейнера. В противном случае он ничего сделать не может;
- поскольку сетевые стратегии service mesh описываются на уровне сервиса, то не смогут защитить нижележащую инфраструктуру от взломанного модуля.

Крупным предприятиям рекомендуется использовать принцип многослойной защиты. Помимо service mesh, не помешают инструменты для подтверждения наличия вспомогательного контейнера во всех контейнерах. Кроме того, будут кстати дополнительные программные решения по обеспечению сетевой безопасности, способные запрещать/ограничивать трафик непосредственно между контейнерами, или между контейнерами и внешними адресами, за исключением IP-адреса сервиса.

Service mesh может предложить и другие возможности, например канареечное развертывание, не связанное с передачей данных по сети или безопасностью. Больше информации можно найти в следующей статье: https://oreil.ly/0Fq5A — на сайте DigitalOcean.

Вспомогательные контейнеры service mesh сосуществуют внутри модуля с контейнерами приложений. В случае взлома такого контейнера злоумышленник может попытаться обойти или изменить правила, соблюдение которых обеспечивает вспомогательный контейнер. Вспомогательный контейнер и контейнер приложения используют одно и то же пространство имен, поэтому имеет смысл не предоставлять контейнерам приложений привилегии `CAP_NET_ADMIN`, чтобы в случае взлома они не могли изменить общий сетевой стек.

Резюме

В этой главе вы видели, как контейнеры позволяют весьма скрупулезно настроить брандмауэр внутри развернутой системы, что помогает обеспечить соблюдение нескольких принципов безопасности, таких как:

- разделение обязанностей/принцип минимума полномочий благодаря ограничению для контейнеров возможностей обмена информацией;
- ограничение радиуса поражения путем предотвращения возможности атаки его соседей со стороны взломанного контейнера;
- многослойная защита за счет сочетания брандмауэров контейнеров с service mesh и обычными брандмауэрами в масштабах всего кластера.

Я уже упоминала, что service mesh способен автоматически устанавливать взаимные TLS-соединения между контейнерами. В следующей главе я поясню, каким образом TLS повышает безопасность обмена информацией, и попытаюсь пролить свет на роль ключей и сертификатов в установлении этих безопасных соединений.

ГЛАВА 11

Защищенное соединение компонентов с помощью TLS

Все распределенные системы включают компоненты, которым нужно обмениваться информацией друг с другом. В мире нативных облачных сервисов этими компонентами могут вполне быть контейнеры, которые обмениваются сообщениями друг с другом или с другими внутренними или внешними компонентами. В данной главе вы увидите, как, используя защищенные соединения, компоненты могут безопасно отправлять друг другу зашифрованные сообщения. Вы узнаете, как компоненты идентифицируют друг друга и устанавливают защищенные соединения между собой, чтобы исключить вредоносные компоненты из обмена информацией.

Если вы знакомы с механизмом работы ключей и сертификатов, то можете спокойно пропустить эту главу, поскольку в ней нет ничего относящегося именно к контейнерам. Я включила ее в книгу лишь потому, что, по моему опыту, эти вопросы часто вызывают затруднения у многих, кто при изучении контейнеров и нативных облачных инструментов столкнулся с ними впервые.

Если вы отвечаете за администрирование нативной облачной системы, то, вероятно, вам придется настраивать сертификаты, ключи и центры сертификации для Kubernetes, etcd и других элементов инфраструктуры. Эти задачи печально известны своей запутанностью, и инструкции по установке обычно рассказывают только то, *что* нужно сделать, не объясняя, *почему* или *как*. Данная глава поможет вам понять роли, которые играют все эти элементы.

Начнем с обсуждения того, что понимается под защищенными соединениями.

Защищенные соединения

В повседневной жизни защищенные соединения можно наблюдать, например, в браузерах. Если вы зашли, скажем, в свою систему онлайн-банкинга и не увидели маленький зеленый замочек, значит, соединение не защищено и лучше не вводить там учетные данные для входа в систему. Формирование защищенного соединения с сайтом банка состоит из двух частей.

- Во-первых, необходимо убедиться, что сайт, на который вы зашли, действительно принадлежит вашему банку. Браузер убеждается в подлинности сайта путем проверки его сертификата.
- Вторая часть — шифрование. При доступе к банковской информации нужно гарантировать, что никакие сторонние лица не смогут перехватить (или, что еще хуже, изменить) передаваемую по этому каналу информацию.

Вероятно, вы знаете, что для защищенных соединений с сайтами используется протокол *HTTPS* (что означает HTTP-Secure — «защищенный HTTP»). Как ясно из названия, это обычное HTTP-соединение, только защищенное на транспортном уровне с помощью протокола, изобретательно названного *безопасностью транспортного уровня* (transport layer security, TLS).

Если вы думали, что S обозначает SSL, то есть secure sockets layer — «уровень защищенных сокетов», то не слишком ошиблись. Транспортный уровень представляет собой уровень обмена информацией между парами сетевых сокетов, а TLS — новое название протокола, который когда-то назывался SSL. Первые спецификации SSL были опубликованы компанией Netscape в 1995 году (версия 2, в первоначальной версии 1 содержалось слишком много изъянов, поэтому она так и не была опубликована[1]). К 1999 году IETF (Internet Engineering Task Force — Инженерный совет Интернета) создал стандарт TLS v1.0, в значительной степени основанный на SSL v3.0 от Netscape, а сейчас в общем и целом применяется TLS v1.3.

Неважно, называть ли его SSL или TLS, суть остается одной: защищенные соединения создаются в этом протоколе на основе сертификатов. Непонятно, почему 20 лет спустя после перехода на TLS их все еще называют SSL-сертификатами. Правильнее всего называть их сертификатами X.509.

---

[1] Собственно, версия 2 тоже содержала много недостатков, поэтому версия 3 была выпущена уже в начале 1996 года.

С помощью сертификатов X.509 можно обмениваться информацией как о владельце, так и о ключах шифрования. Посмотрим, что представляют собой эти сертификаты и как работают.

Существует несколько утилит для генерации ключей, сертификатов и создания центров сертификации, в числе которых ssh-keygen, openssl keygen и minica. В докладе «Руководство по защищенным соединениям для программиста на Go (A Go Programmer's Guide to Secure Connections) (https://youtu.be/OF3TM-b890E) я продемонстрировала, как пользоваться minica, а также пошагово описала, что происходит, когда клиент устанавливает TLS-соединение с сервером.

Сертификаты X.509

Термин X.509 представляет собой название стандарта Международного союза электросвязи (International Telecommunication Union, ITU), в котором описаны эти сертификаты. Сертификат представляет собой элемент структурированных данных, содержащий информацию о его владельце, а также открытый ключ шифрования, предназначенный для обмена информацией с его владельцем. Этот открытый ключ входит в пару «открытый/секретный ключ».

Как демонстрирует рис. 11.1, неотъемлемыми элементами информации, содержащейся в сертификате, являются:

- название сущности, которую идентифицирует данный сертификат, то есть *субъекта* (subject). Название субъекта обычно представляет собой доменное имя. На практике, в сертификатах обычно есть поле «Другие названия субъекта», благодаря которому сертификат может идентифицировать субъект по нескольким названиям;
- открытый ключ субъекта;

Рис. 11.1. Сертификат

- название центра сертификации, выдавшего сертификат. Я вернусь к этому вопросу позднее, в подразделе «Центры сертификации» далее;
- действительность сертификата, то есть дата и время истечения его срока действия.

Пары «открытый/секретный ключ»

Как ясно из названия, открытым ключом можно делиться с кем угодно. Каждому открытому ключу соответствует секретный, который владелец никогда не должен обнародовать.

Формальное математическое описание способов шифрования и дешифровки выходит за рамки данной книги, но в сообщении на сайте Medium (https://oreil.ly/Tbhvd) я привожу несколько рекомендуемых источников информации по этому вопросу.

Сначала генерируется секретный ключ, а затем уже вычисляется соответствующий открытый. Пару «открытый/секретный ключ» можно использовать для двух полезных целей.

- Как показано на рис. 11.2, открытый ключ можно применять для шифрования сообщения, которое затем сможет расшифровать только тот, у кого есть соответствующий секретный ключ.

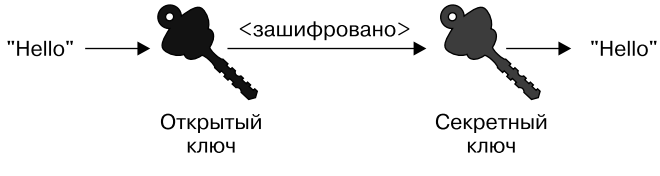


Рис. 11.2. Шифрование

- Секретный ключ можно использовать для создания цифровой подписи сообщения, с помощью которой любой владелец соответствующего открытого ключа сможет убедиться, что сообщение поступило от владельца секретного (рис. 11.3).

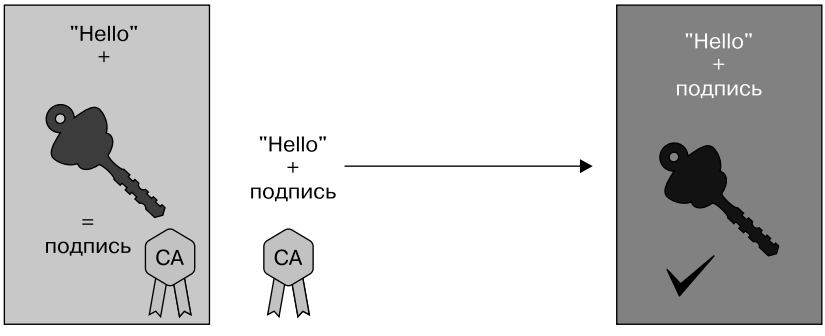


Рис. 11.3. Цифровая подпись

При установлении защищенных соединений возможности пар «открытый/секретный ключ» используются как для шифрования, так и для создания подписей.

Представьте, что вы хотите обмениваться со мной сообщениями. Я генерирую пару ключей и передаю вам открытый ключ, поэтому вы можете отправлять мне зашифрованные сообщения. Но если я отправлю вам открытый ключ, то как вы узнаете, что он действительно пришел от меня, а не от самозванца? Чтобы вы могли удостовериться, что я — та, за кого себя выдаю, необходима третья сторона, которой вы доверяете и которая за меня поручится. Эти функции выполняет *центр сертификации* (certificate authority, CA).

Центры сертификации

Центр сертификации (ЦС) — доверенный орган, подписывающий сертификат и тем самым подтверждающий правильность информации о владельце, которая содержится в этом сертификате. Следует доверять только сертификатам, подписанным центрами, которым вы доверяете.

Открывая TLS-соединение с конкретным адресатом, инициирующий соединение клиент получает сертификат с другого конца соединения и проверяет его, чтобы убедиться, что его собеседник — тот, за кого себя выдает. Например, при открытии веб-соединения с банком браузер проверяет, что сертификат соответствует URL банка, а также проверяет, какой ЦС подписал этот сертификат.

Другие компоненты нуждаются в том, чтобы неким образом безопасно идентифицировать ЦС, поэтому ему соответствует свой сертификат. Но

этот сертификат также должен быть подписан каким-то центром, «личность» которого тоже нужно проверить, для чего требуется еще один сертификат и так далее до бесконечности. Похоже, получается бесконечная цепочка сертификатов! В конце концов, должен быть какой-то сертификат, которому мы можем просто доверять.

На практике цепочка заканчивается так называемым самоподписанным сертификатом: сертификатом X.509, который центр подписал для себя. Другими словами, этот сертификат представляет того же, чей секретный ключ был использован для подписания сертификата. Если вы доверяете данному лицу, то можете доверять и сертификату. На рис. 11.4 приведена цепочка сертификатов, в которой сертификат Энн подписан Бобом, а сертификат Боба подписан Кэрол. Цепочка оканчивается на самоподписанном сертификате Кэрол.

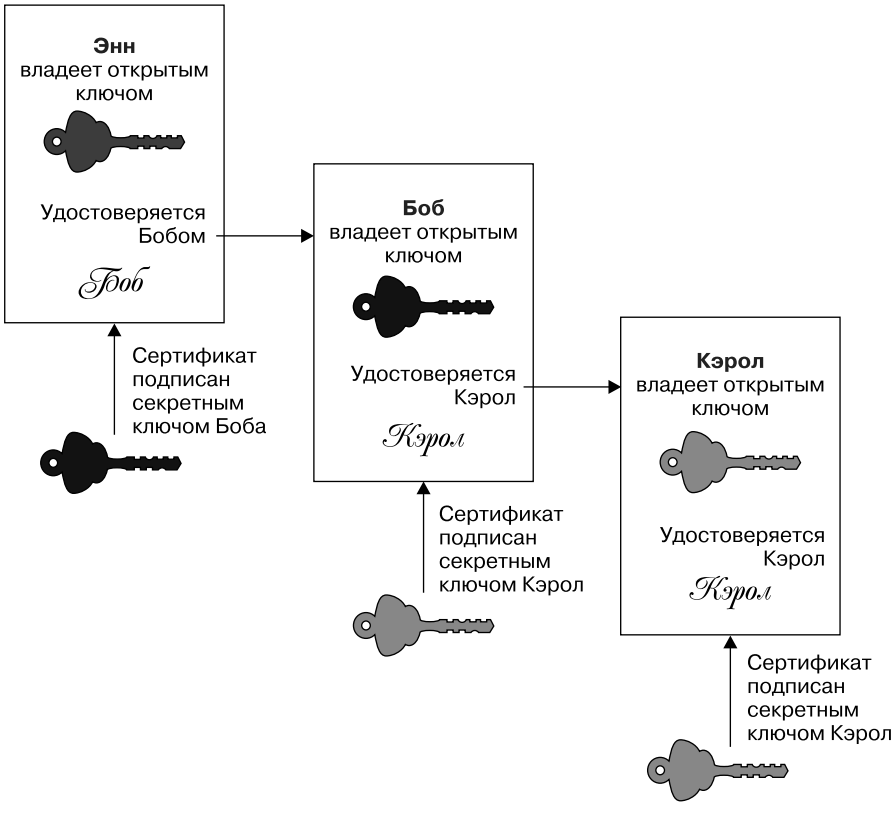


Рис. 11.4. Цепочка сертификатов

Браузеры поставляются с набором сертификатов известных, доверенных центров сертификации — их называют корневыми ЦС. Ваш браузер будет доверять любому сертификату (или цепочке сертификатов), подписанному одним из этих корневых центров. Если же сертификат сайта не подписан одним из доверенных ЦС браузера, то он отметит сайт как небезопасный (в большинстве современных браузеров при этом пользователю будет выдано предупреждение или сообщение об ошибке).

При создании сайта, к которому пользователи должны подключаться через Интернет, вам понадобится сертификат, подписанный доверенным, общедоступным ЦС. Существует несколько поставщиков сертификатов, выступающих в роли таких центров и генерирующих сертификаты за определенную плату. Кроме того, можно получить сертификат бесплатно от Let's Encrypt (https://letsencrypt.org/).

При настройке компонентов распределенных систем наподобие Kubernetes или etcd необходимо указать набор ЦС, используемых для проверки сертификатов. Если систему контролируете только вы, то вам неважно, доверяет ли этим компонентам широкая общественность (или их браузеры), — главное, чтобы компоненты доверяли друг другу. А поскольку это частная система, задействовать общедоступные доверенные центры сертификации не обязательно, можно просто использовать собственные с самоподписанными сертификатами.

Вне зависимости от того, какой ЦС вы используете: собственный или общедоступный, необходимо сообщить ему, какие сертификаты вы хотели бы сгенерировать. Это делается с помощью запроса на подписание сертификата.

Запросы на подписание сертификатов

Запрос на подписание сертификата (certificate signing request, CSR) представляет собой файл, включающий такую информацию:

- открытый ключ, который нужно включить в сертификат;
- доменное имя (имена), с которым (-и) должен работать этот сертификат;
- информацию о владельце данного сертификата (например, название компании или организации).

Вы создаете CSR и отправляете его в ЦС, чтобы запросить сертификат X.509. Как вы знаете из изложенного ранее в данной главе, сертификат включает

нужную информацию плюс подпись центра, поэтому его как раз и имеет смысл включить в CSR.

Создать новую пару ключей и CSR за один шаг можно с помощью таких утилит, как `openssl`. Некоторую путаницу, возможно, вызывает то, что `openssl` может использовать секретный ключ в качестве входных данных для генерации CSR. Соответствующий этому сертификату компонент задействует секретный ключ для расшифровки и подписывания сообщений (как вы скоро увидите), но никогда не использует сам открытый ключ. Последний нужен прочим компонентам, с которыми он обменивается информацией, и они получают этот открытый ключ из сертификата. А этому компоненту нужен секретный ключ, а также сертификат для отправки другим компонентам.

Теперь, когда вы уже понимаете, что такое сертификаты, обсудим, как они используются для TLS-соединений.

TLS-соединения

Организатором соединений транспортного уровня является компонент, называемый клиентом. А компонент, с которым он обменивается информацией, называется сервером. Вполне возможно, что такая взаимосвязь «клиент — сервер» имеет место только на транспортном уровне, а на более высоких уровнях ранг компонентов, которые обмениваются информацией, будет одинаковым.

Клиент открывает сокет и запрашивает сетевое соединение с сервером. Если соединение защищено, то клиент запрашивает у сервера сертификат. Как вы уже знаете из этой главы, сертификат содержит два важных элемента информации: название сервера и его открытый ключ.

Смысл в том, чтобы клиент мог выяснить, стоит ли доверять серверу. Клиент проверяет, подписан ли сертификат сервера доверенным ЦС, что служит подтверждением: серверу можно доверять, клиент может использовать открытый ключ сервера для шифрования отправляемых серверу сообщений. Клиент и сервер приходят к соглашению относительно симметричного ключа, который будет применяться для дальнейших сообщений, передаваемых по этому соединению, — быстродействие при этом выше, чем при использовании асимметричной пары «открытый/секретный ключ». Данный поток сообщений показан на рис. 11.5.

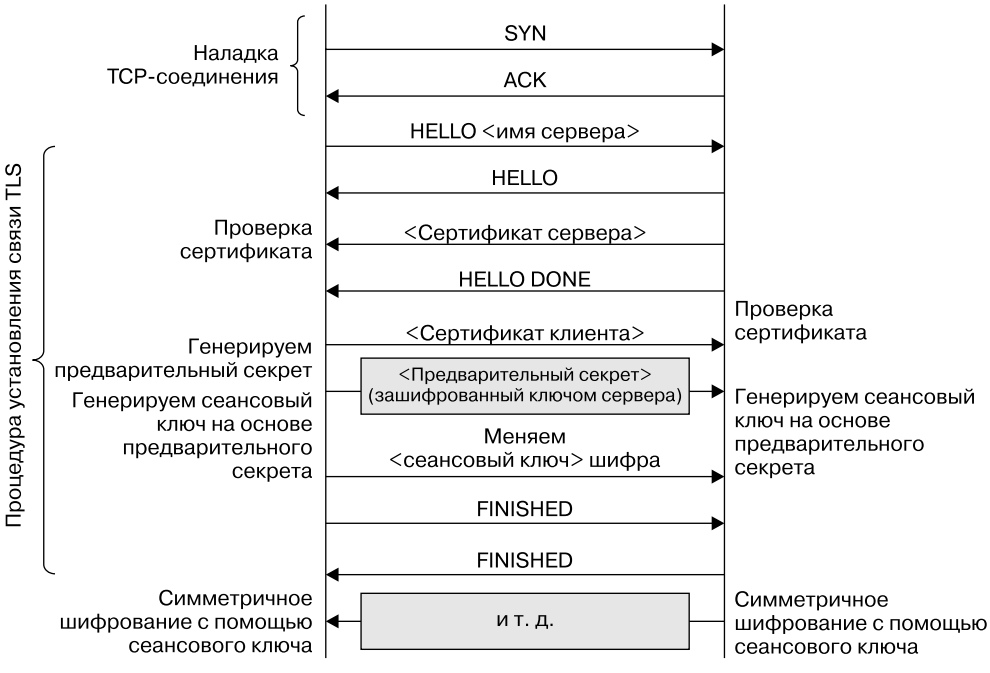


Рис. 11.5. Процедура установления связи TLS

Вероятно, вам встречался термин «пропуск проверки» (skip verify). Он описывает вариант, когда клиент на транспортном уровне пропускает проверку, был ли сертификат подписан известным ему ЦС. Клиент при этом просто предполагает, что сервер — тот, за кого себя выдает. Это удобно при разработке, поскольку можно не тратить усилия на то, чтобы настроить клиент, задать на нем информацию о центрах сертификации, и можно просто использовать самоподписанные сертификаты. При этом обмен информацией между компонентами все равно остается зашифрованным, но нет уверенности, что эти компоненты не «самозванцы», поэтому не используйте вариант пропуска проверки сертификатов в промышленной эксплуатации!

Проверив подлинность сервера, клиент может ему доверять. Но может ли этот сервер доверять данному клиенту?

Если речь идет о сайте, на котором у вас есть учетная запись, например о сайте банка, то важно, чтобы сервер проверял, с кем имеет дело, перед выдачей, скажем, информации о состоянии банковского счета. При настоящей

связи «клиент — сервер» наподобие входа на сайт банка эта задача решается с помощью аутентификации на уровне 7. Вы указываете имя пользователя и пароль, возможно, с многофакторной аутентификацией в виде ввода кода, отправленного в текстовом сообщении, или одноразового пароля, сгенерированного Yubikey или мобильным приложением — Authy, 1Password или Google Auth.

Еще один способ подтвердить «личность» клиента — воспользоваться еще одним сертификатом X.509. Поток сообщений на рис. 11.5 (см. выше) демонстрирует обмен сертификатами как клиента, так и сервера — это опция, настраиваемая на стороне сервера. С помощью сертификата сервер подтверждает клиенту свою «личность», так почему же нельзя сделать обратное? Это называется взаимным TLS (mutual TLS, mTLS).

Защищенные соединения между контейнерами

До сих пор в этой главе не обсуждалась специфика контейнеров, теперь же поговорим об относящихся к ним обстоятельствах, когда может пригодиться понимание ключей, сертификатов и центров сертификации.

- При установке или администрировании Kubernetes или других компонентов распределенных систем, вероятно, вы столкнетесь с различными вариантами использования защищенных соединений. Сегодня существуют такие установочные утилиты, как `kubeadm`, которые упрощают применение TLS между компонентами управляющей плоскости и автоматически настраивают сертификаты. Но трафик между контейнерами и внешним миром они никак не защищают.
- Вам как разработчику может понадобиться написать прикладной код для организации защищенных соединений с другими компонентами (работающий в контейнере или нет). В этом случае коду вашего приложения понадобится доступ к сертификатам, которые вам придется создать.
- Вместо того чтобы переписывать собственный код для организации защищенных соединений, вы можете воспользоваться service mesh.

Сертификаты предназначены для распространения, но их использование требует от каждого из компонентов наличия доступа к собственному

секретному ключу, соответствующему этому сертификату. В следующей главе мы обсудим способы передачи в контейнеры таких конфиденциальных данных, как секретные ключи.

Отзыв сертификатов

Представьте, что злоумышленник каким-то образом раздобыл секретный ключ. Теперь он может выдавать себя за того, кому принадлежит этот ключ, поскольку может расшифровывать сообщения, зашифрованные открытым ключом, вложенным в соответствующие сертификаты. Чтобы это предотвратить, необходимо иметь способ немедленно аннулировать сертификаты, а не ждать истечения срока их действия.

Подобное аннулирование называется отзывом сертификатов (certificate revocation), для него используется специальный *список отозванных сертификатов* (certificate revocation list, CRL), содержащий сертификаты, которые более не следует принимать.

Старайтесь не задействовать субъекты (и их сертификаты) сразу для нескольких компонентов и пользователей. Возможно, создание субъектов и сертификатов для каждого компонента требует дополнительных усилий, но позволяет отзывать сертификаты субъектов по отдельности, не генерируя заново сертификаты для всех остальных (нормальных) пользователей. Вдобавок дает возможность разделять ответственность, предоставляя каждому субъекту свой набор прав.

В Kubernetes сертификаты используются компонентом kubelet на всех узлах для аутентификации его в сервере API и подтверждения, что он действительно является авторизованным kubelet. Эти сертификаты можно чередовать.

Сертификаты также служат одним из механизмов аутентификации клиентов в сервере API Kubernetes (https://oreil.ly/0DuGQ).

На момент написания данной книги Kubernetes не поддерживает отзыв сертификатов (https://oreil.ly/RU3ga). Можете воспользоваться возможностями RBAC, чтобы запретить доступ к API клиентов, связанных с такими сертификатами.

Резюме

Чтобы защититься от атаки типа «злоумышленник посередине», необходимо удостовериться в защищенности сетевых соединений между отдельными программными компонентами. Проверенный способ убедиться в этом — прибегнуть к защищенным соединениям на основе mTLS. Настроить mTLS-соединения между контейнерами приложений — хорошая идея, а если вы занимаетесь администрированием компонентов распределенной системы, то вам понадобятся защищенные соединения и между ними.

Для каждого контейнера или другого компонента, использующего для аутентификации сертификаты X.509, вам понадобится три элемента:

- секретный ключ, который следует хранить в секрете и никогда никому не давать;
- сертификат для свободного распространения, с помощью которого другие компоненты могут проверить его «личность»;
- сертификаты от одного или нескольких доверенных ЦС для проверки сертификатов, полученных от других компонентов.

Прочитав данную главу, вы должны уже хорошо представлять роли, которые играют ключи, сертификаты и ЦС. Эти знания пригодятся вам при настройке компонентов.

Если вы доверяете соединениям между контейнерами и умеете идентифицировать компоненты на другом конце соединения, то можете начать передавать между контейнерами *конфиденциальные данные*. Но для этого вам нужно научиться безопасно передавать в контейнеры конфиденциальные значения — о чем мы и поговорим в следующей главе.

ГЛАВА 12

Передача в контейнеры секретных данных

Коду приложения для работы нередко требуются какие-либо учетные данные, например пароль для доступа к базе данных или токен для доступа к определенному API. Учетные (секретные) данные существуют специально для того, чтобы ограничить доступ к ресурсам — в этих примерах к базе данных и API. Важно обеспечить секретность этих данных и в соответствии с принципом минимума полномочий предоставлять доступ к ним лишь тем людям/компонентам, которым они действительно нужны.

Сначала в этой главе мы обсудим желаемые свойства секретных данных, а затем варианты передачи их в контейнеры. А завершится она обсуждением нативной поддержки их в Kubernetes.

Свойства секретных данных

Самое очевидное свойство секретных данных состоит в том, что они должны оставаться секретными, то есть доступными только для тех людей (или сущностей), которые должны иметь к ним доступ. Обычно секретность достигается с помощью шифрования секретных данных и передачи ключа дешифровки только тем, у кого есть право их знать.

Секретные данные необходимо хранить в зашифрованном виде, чтобы они не были доступны всем пользователям/сущностям, имеющим доступ к хранилищу данных. При перемещении секретных данных из хранилища в какое-либо место их применения их также необходимо шифровать, чтобы никто их не «подсмотрел» по пути. Желательно никогда не записывать секретные данные на диск в незашифрованном виде. Если приложению понадобится незашифрованная версия, то ее лучше хранить только в оперативной памяти.

Заманчиво считать, что достаточно зашифровать секретные данные один раз — и все, теперь можно безопасно передавать их в другие компоненты. Однако получателю нужно знать, как расшифровать полученную информацию, а значит, должен существовать ключ расшифровки. Он сам по себе является секретной информацией, и получателю нужно как-то его передать, что возвращает нас к исходному вопросу о безопасной передаче секретных данных теперь уже на следующем уровне.

Кроме того, необходимо иметь возможность *отзывать* (revoke) секретные данные, то есть аннулировать их в том случае, если доверия к их безопасности больше нет. Такое может происходить, если вы узнали (или заподозрили), что доступ к этим секретным данным получил кто-то посторонний. Вдобавок секретные данные иногда приходится отзывать исключительно по производственной необходимости, например, когда увольняется кто-то из разработчиков.

Необходимо иметь также возможность *заменять* (rotate) секретные данные. Совсем не факт, что вы узнаете об их утечке, поэтому необходимо менять их время от времени, чтобы злоумышленник, получивший доступ к учетным данным, рано или поздно обнаружил, что они уже недействительны. Общепризнано, что заставлять людей часто менять пароли — плохая идея (https://oreil.ly/ETKEZ), но программные компоненты вполне могут справиться с задачей частой смены учетных данных.

Желательно, чтобы жизненный цикл секретных данных не зависел от жизненного цикла использующего их компонента, поскольку тогда не нужно будет заново собирать и распространять компонент при смене секретных данных.

Круг людей, которые должны иметь доступ к секретным данным, часто намного у́же, чем круг людей, которым требуется доступ к исходному коду приложения, использующего эти данные, либо которые могут развертывать/администрировать систему или ее части. Например, в банке разработчикам вряд ли нужно предоставлять доступ к секретным данным, используемым во время эксплуатации, дающим доступ к информации о банковских счетах. Нередко учетные данные доступны людям только для записи: после того как секретные данные будут сгенерированы (зачастую автоматически, случайным образом), никаких законных оснований для их чтения может и не быть.

Доступ к секретным данным следует ограничить не только для людей. Желательно, чтобы доступ к ним имели только те компоненты, которым он

действительно нужен. А поскольку мы говорим о контейнерах, это значит, что секретные данные должны видеть лишь те контейнеры, которым они нужны для работы.

Теперь, когда мы обсудили желаемые свойства секретных данных, посмотрим на возможные механизмы передачи таких данных в код приложения, запущенный в контейнере.

Передача информации в контейнер

Поскольку контейнеры умышленно изолируются от окружающего мира, неудивительно, что возможностей передачи информации — в том числе секретных данных — в работающий контейнер не так уж и много:

- данные можно включить в образ контейнера в виде файла в корневой файловой системе образа;
- можно описать в качестве части сопутствующей образу конфигурации переменные среды (см. в главе 6 сведения о том, что образ состоит из корневой файловой системы и информации о настройках);
- контейнер может получать информацию через сетевой интерфейс;
- можно задавать или переопределять переменные среды в момент запуска контейнера (например, включая параметры `-e` в команду `docker run`);
- контейнер может монтировать тома с хоста и читать из них информацию.

Рассмотрим все эти варианты по очереди.

Хранение секретных данных в образе контейнера

Первые два из упомянутых вариантов для секретных данных не подходят, поскольку требуют включения секретных данных в образ во время сборки. Хотя так сделать, конечно, можно, но это не слишком хорошая идея.

- Секретные данные будут видны всем, у кого есть доступ к исходному коду образа. Конечно, можно хранить эти данные в коде в зашифрованном виде, а не открытым текстом, но тогда придется передать как-то другие секретные данные, чтобы контейнер мог расшифровать первые. И какой же механизм использовать для передачи этих вторых секретных данных?

- Секретные данные нельзя будет изменить, не собрав повторно образ контейнера, а эти две задачи лучше разделить. Более того, централизованная, автоматизированная система управления секретными данными (например, CyberArk или Hashicorp Vault) не сможет контролировать жизненный цикл секретных данных, жестко «зашитых» в код.

К сожалению, встраивание секретных данных в исходный код — очень распространенная практика. Во-первых, поскольку не все разработчики знают, что это плохая идея; во-вторых, ведь так легко вставить секретные данные непосредственно в код, чтобы упростить разработку или тестирование, искренне собираясь потом их убрать оттуда — а затем просто забыть сделать это.

А раз передача секретных данных во время сборки даже не обсуждается, то все остальные варианты связаны с передачей их во время запуска или работы контейнера.

Передача секретных данных по сети

Третий вариант, передача секретных данных через сетевой интерфейс, требует от кода приложения выполнения соответствующих сетевых вызовов для извлечения или получения информации, и потому этот вариант реже всего можно встретить на практике.

Кроме того, остается вопрос шифрования сетевого трафика, содержащего секретные данные, требующий передачи других секретных данных в виде сертификата X.509 (см. главу 11). Эту часть задачи можно делегировать service mesh, который можно настроить на использование взаимного TLS при сетевых соединениях в целях безопасности.

Передача секретных данных в переменных среды

Четвертый вариант, передача секретных данных с помощью переменных среды, тоже обычно не приветствуется по нескольким причинам.

- Во многих языках программирования и фреймворках фатальный сбой приводит к сбросу системой отладочной информации, которая вполне может включать все переменные среды. И если эта информация передается в систему журналирования, то все, у кого есть доступ к журналам, могут увидеть секретные данные, передаваемые в переменных среды.

- Если выполнить для контейнера команду `docker inspect` (или аналогичную ей), то можно увидеть все переменные среды, заданные для него как во время сборки, так и в процессе выполнения. А администраторам, которым по вполне уважительным причинам может понадобиться просмотреть свойства контейнера, вовсе не обязательно иметь доступ к секретным данным.

Ниже представлен пример извлечения переменных среды из образа контейнера:

```
vagrant@vagrant:~$ docker image inspect --format '{{.Config.Env}}' nginx
[PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin NGINX_
VERSION=1.17.6 NJS_VERSION=0.3.7 PKG_RELEASE=1~buster]
```

Можно также легко просмотреть переменные среды во время выполнения. Следующий пример демонстрирует, что выводимые результаты включают все определения переменных, указанные в команде `run` (в данном случае `EXTRA_ENV`).

```
vagrant@vagrant:~$ docker run -e EXTRA_ENV=HELLO --rm -d nginx
13bcf3c571268f697f1e562a49e8d545d78aae65b0a102d2da78596b655e2f9a
vagrant@vagrant:~$ docker container inspect --format '{{.Config.Env}}' 13bcf
[EXTRA_ENV=HELLO PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
NGINX_VERSION=1.17.6 NJS_VERSION=0.3.7 PKG_RELEASE=1~buster]
```

Манифест 12-факторного приложения (https://12factor.net/ru/config) поощряет передачу разработчиками параметров настроек через переменные среды, поэтому на практике можно встретить сторонние приложения, которые ожидают именно такого способа конфигурации, включающего и некие секретные значения. Снизить риски при передаче секретных данных через переменные среды можно несколькими способами (которые могут стоить затраченных усилий или нет, в зависимости от профиля рисков):

- обработав выходные журналы: удалив или выполнив обфускацию секретных значений;
- изменив контейнер приложения (или использовав вспомогательный контейнер), чтобы извлечь секретные данные из защищенного хранилища (наподобие Hashicorp Vault, CyberArk Conjur или систем управления секретными данными/криптографическими ключами поставщиков облачных сервисов). Возможности подобной интеграции предлагают некоторые коммерческие решения по обеспечению безопасности.

И последнее, что стоит отметить по поводу передачи секретных данных через переменные среды: среда для процесса формируется однократно, а именно во время его создания. И вам не удастся заново задать параметры среды контейнера снаружи, если вы захотите изменить секретные данные.

Передача секретных данных через файлы

Рекомендуемый вариант передачи секретных данных — запись их в файлы, к которым у контейнера есть доступ через смонтированный том. Желательно, чтобы этот смонтированный том был временным каталогом, который хранится в оперативной памяти, а не записывается на диск. В сочетании с защищенным хранилищем секретных данных это гарантирует, что такие данные никогда не будут храниться в незашифрованном виде.

А поскольку файл монтируется с хоста в контейнер, его можно в любой момент изменить на стороне хоста, не перезапуская контейнер. И если приложение знает, что нужно извлечь новые секретные данные из файла, когда старые перестали работать, то можно заменять секретные данные, не прибегая к перезапуску контейнеров.

Секретные данные в Kubernetes

Хорошая новость, для тех, кто использует Kubernetes, — его встроенная поддержка секретных данных удовлетворяет многим критериям, описанным в начале этой главы:

- секретные данные Kubernetes создаются в виде независимых ресурсов, так что никак не привязаны к жизненному циклу использующего их приложения;
- секретные данные можно хранить в зашифрованном виде, хотя (по крайней мере на момент написания данной книги) необходимо включить соответствующий параметр, так как *по умолчанию эта опция отключена*;
- секретные данные передаются между контейнерами в зашифрованном виде. Для этого необходимо установить защищенные соединения между компонентами Kubernetes, хотя этот вариант используется по умолчанию в большинстве случаев;
- Kubernetes поддерживает передачу секретных данных как через файлы, так и через переменные среды, с монтированием секретных данных в виде файлов во временную файловую систему, которая хранится исключительно в оперативной памяти и не записывается на диск;
- при желании можно включить RBAC (role-based access control — управление доступом на основе ролей) Kubernetes, поэтому пользователи смогут

создавать секретные данные, но не смогут затем читать их, то есть у них будут права только для записи.

По умолчанию в Kubernetes эти секретные значения хранятся в хранилище данных etcd в виде незашифрованных значений в кодировке base64. Можно настроить etcd так, что хранилище данных будет шифроваться, хотя желательно при этом не хранить ключ дешифровки на хосте.

Как показывает мой опыт, большинство предприятий выбирает для хранения секретных данных стороннее коммерческое решение, либо предоставляемое их поставщиком облачных сервисов (например, AWS Key Management System или ее эквиваленты на платформах Azure и GCP), либо решения от таких компаний, как Hashicorp или CyberArk. Эти решения имеют ряд преимуществ.

- Во-первых, можно автоматически заменять сертификаты. При такой замене сертификатов, используемых самими компонентами Kubernetes, необходимо обновлять все секретные данные Kubernetes (https://landing.app.cloud.gov/docs/ops/runbook/rotating-kubernetes/). Этого можно избежать, если обратиться к специализированному решению по управлению секретными данными.
- Еще одно преимущество: специализированную систему управления секретными данными можно использовать сразу для нескольких кластеров. Секретные значения можно заменять независимо от жизненного цикла кластера (-ов) приложений.
- Эти программные решения упрощают для организаций стандартизацию хранения секретных данных и использование единых практических рекомендаций по управлению и согласованному журналированию/аудиту секретных данных.

Документация Kubernetes охватывает множество относящихся к безопасности свойств (https://oreil.ly/XmzCc) его нативной поддержки секретных данных.

Документация Rancher включает пример использования KMS AWS для шифрования секретных данных Kubernetes (https://oreil.ly/qi6yC) при хранении.

В блоге Hashicorp также можно найти описание внедрения секретных данных Vault (https://oreil.ly/CyN1J).

Секретные данные доступны для суперпользователя хоста

Вне зависимости от того, как секретные данные передаются в контейнер: в виде смонтированного файла или переменной среды, — суперпользователь хоста может при желании получить к ним доступ.

Если секретные данные хранятся в файле, то он хранится где-то в файловой системе хоста. Даже если он хранится во временном каталоге, суперпользователь хоста сможет получить к нему доступ. Чтобы на это посмотреть, можете вывести список временных файловых систем, смонтированных на узле Kubernetes, и вы увидите нечто наподобие следующего:

```
root@vagrant:/$ mount -t tmpfs
...
tmpfs on /var/lib/kubelet/pods/f02a9901-8214-4751-b157-d2e90bc6a98c/volumes/kuber
netes.io~secret/coredns-token-gxsqd type tmpfs (rw,relatime)
tmpfs on /var/lib/kubelet/pods/074d762f-00ed-48e6-a22f-43fc673df0e6/volumes/kuber
netes.io~secret/kube-proxy-token-bvktc type tmpfs (rw,relatime)
tmpfs on /var/lib/kubelet/pods/e1bad0db-8c0b-4d7b-8867-9fc019de258f/volumes/kuber
netes.io~secret/default-token-h2x8p type tmpfs (rw,relatime)
...
```

Исходя из названий каталогов, приведенных в этом выводе, суперпользователь без труда сможет получить доступ к хранящимся в них файлам с секретными данными.

Извлечь секретные данные из переменных среды для суперпользователя ничуть не сложнее. Продемонстрируем это, запустив контейнер с помощью Docker в командной строке и передав в него переменную среды:

```
vagrant@vagrant:~$ docker run --rm -it -e SECRET=mysecret ubuntu sh
$ env
...
SECRET=mysecret
...
```

В данном контейнере запущен `sh`, и из другого терминала можно узнать идентификатор процесса для этого исполняемого файла:

```
vagrant@vagrant:~$ ps -C sh
  PID TTY          TIME CMD
17322 pts/0    00:00:00 sh
```

В главе 4 вы видели, что в каталоге /proc содержится много интересной информации о процессах, включая все переменные среды, находящиеся в /proc/<ID процесса>/environ:

```
vagrant@vagrant:~$ sudo cat /proc/17322/environ
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/
binHOSTNAME=2cc99c9
8ba5aTERM=xtermSECRET=mysecretHOME=/root
```

Как видите, этот способ позволяет прочитать любые секретные данные, передаваемые через переменные среды. А если вам кажется, что лучше будет сначала зашифровать секретные данные, то задумайтесь, как передать ключ дешифровки — который также необходимо хранить в секрете — в контейнер?

Совсем не лишним будет еще раз подчеркнуть: полномочия суперпользователя на хост-компьютере дают ему полную свободу доступа ко всему на хосте, включая все контейнеры и их секретные данные. Именно поэтому так важно не допускать несанкционированного доступа с полномочиями суперпользователя в развернутой системе и именно поэтому работа от имени суперпользователя в контейнере настолько опасна: суперпользователь в контейнере = суперпользователь на хосте, следовательно, от этого до несанкционированного доступа к тому, что есть на данном хосте, — всего один шаг.

Резюме

Если вы внимательно прочитали все изложенное в этой книге ранее, то уже неплохо представляете, как работают контейнеры и как передавать между ними секретную информацию. Вы уже видели множество способов эксплуатации контейнеров и немало средств их защиты.

Последняя группа механизмов защиты, которые мы обсудим в следующей главе, — *защита во время выполнения* (runtime protection).

ГЛАВА 13

Защита контейнеров во время выполнения

Как мы видели в главе 10, один из отличительных признаков контейнеров — они хорошо подходят для *микросервисных* архитектур. Разработчик приложения может разбить сложную программную систему на маленькие самодостаточные фрагменты кода, масштабируемые независимо друг от друга, каждый — в виде образа контейнера.

Разбиение большой системы на маленькие компоненты с четко очерченными интерфейсами упрощает проектирование, написание кода и тестирование отдельных компонентов. Оказывается, обеспечить безопасность при этом тоже проще.

Профили образов контейнеров

Если образ контейнера содержит код микросервиса, выполняющего ровно одну небольшую функцию, то относительно легко разобраться, что этот микросервис должен делать. Код микросервиса встроен в образ контейнера, и можно сформировать соответствующий этому образу профиль времени выполнения, который описывает, какие возможности у него должны быть.

Все контейнеры, созданные на основе данного образа, должны вести себя схожим образом, поэтому логично будет описывать профиль желаемого поведения для образа, а затем использовать его для контроля входящего/исходящего трафика всех контейнеров, основанных на этом образе.

В системах, развертываемых на основе Kubernetes, безопасность во время выполнения можно контролировать помодульно — например, с помощью объекта PodSecurityPolicy или средств обеспечения безопасности, также работающих на данном уровне. Модуль, по сути, представляет собой набор контейнеров с общим пространством имен сети, поэтому и механизмы обеспечения безопасности во время выполнения тоже одни.

В качестве примера я воспользуюсь платформой электронной коммерции, представленной в разделе «Брандмауэры для контейнеров» на с. 162. Она имеет микросервис поиска товаров, который принимает веб-запросы с ключевым словом для поиска (или первыми несколькими символами такого ключевого слова), вводимые покупателем на сайте данного интернет-магазина. Задача поискового микросервиса состоит в поиске в базе товаров тех элементов, которые соответствуют ключевому слову, и возврате результатов. Начнем с ожидаемого сетевого трафика для этого микросервиса.

Профили сетевого трафика

Из описания микросервиса поиска товаров понятно, что его контейнеры должны получать веб-запросы от балансировщика нагрузки или входящего трафика и отвечать на них, а также открывать соединение к базе данных товаров. Помимо распространенных функций платформы, например журналирования или проверок состояния, не существует причин, по которым этот сервис мог бы обрабатывать или порождать какой-либо прочий сетевой трафик.

Не так уж сложно сделать эскиз профиля, описывающего допустимый для этого сервиса трафик, а затем на его основе описать правила, соблюдение которых обеспечивается на сетевом уровне, как вы видели в главе 10. Существуют сетевые утилиты, способные работать в режиме «записи», при котором они производят мониторинг входящих/исходящих сообщений сервиса за некий период времени и автоматически формируют профиль обычного для него потока трафика. Такой профиль можно затем преобразовать в правила брандмауэра контейнера или сетевые стратегии.

Сетевой трафик — не единственный вид поведения, подходящий для наблюдения и создания профилей.

Профили исполняемых файлов

Сколько программ должно работать в этом микросервисе поиска товаров? В качестве примера допустим, что его код написан на языке Go в виде одного исполняемого файла `productsearch`. Если вы посмотрите список исполняемых файлов, работающих внутри соответствующих контейнеров поиска товаров, то вы увидите только `productsearch`. Если увидите что-то еще — это аномалия и, вероятно, признак атаки.

Даже если сервис написан на языках сценариев, например Python или Ruby, можно делать определенные выводы о том, какие варианты допустимы, а какие — нет. Приходится ли сервису «подключаться наружу контейнера», чтобы выполнить другие команды? Если нет, то запущенные в контейнере поиска товаров исполняемые файлы `bash`, `sh` и `zsh` должны вас насторожить.

Все вышеописанное предполагает, что контейнеры считаются неизменяемыми, а вы не открываете подключения непосредственно в контейнеры системы, находящейся в промышленной эксплуатации. С точки зрения безопасности практически нет разницы между открытием злоумышленником обратного подключения с помощью уязвимости приложения и открытием администратором консольного подключения в целях выполнения какого-либо технического обслуживания системы. Как обсуждалось в подразделе «Неизменяемые контейнеры» на с. 124, подобное считается порочной практикой!

Как же обнаружить запуск исполняемых файлов внутри контейнера? Одна из технологий, позволяющих это, — eBPF.

Наблюдение за исполняемыми файлами с помощью eBPF. Возьмем для примера контейнер `nginx`. В обычных условиях процессы `nginx` — единственные, которые можно встретить в таком контейнере. Проект Tracee, с которым мы познакомились в главе 8, позволяет с легкостью наблюдать за процессами, запущенными внутри контейнера `nginx`.

Tracee использует технологию eBPF (extended Berkeley Packet Filter, расширенный фильтр пакетов университета Беркли). Она позволяет Tracee внедрять код в ядро, так что его необходимо запускать от имени суперпользователя.

Чтобы узнать больше о технологии eBPF, взгляните сначала на слайды с моего доклада «Сверхспособности eBPF» (eBPF Superpowers) (https://oreil.ly/-ERmb). А далее вы можете найти гораздо больше информации и литературы на сайте Брендана Грегга (Brendan Gregg) (http://brendangregg.com/).

Запустив Tracee в одном окне терминала, я запустила контейнер `nginx` во втором:

```
$ docker run --rm -d --name nginx nginx
```

Tracee показывает, как запускается исполняемый файл `nginx`, как и можно было ожидать (часть выводимой информации я опустила, чтобы было более понятно):

```
EVENT  ARGS
execve /usr/sbin/nginx
```

Теперь если выполнить еще одну команду внутри контейнера — например, `docker exec -it nginx ls`, то она появится в выводимой Tracee информации.

```
EVENT ARGS
execve /usr/sbin/nginx
execve /bin/ls
```

Допустим, злоумышленник поместил внутрь контейнера программу для добычи криптовалюты. Утилиты наподобие Tracee позволяют заметить запуск подобного исполняемого файла. Отслеживать такие события (запуск исполняемых файлов со сравнением имени исполняемого файла с белым/черным списком) могут и утилиты обеспечения безопасности во время выполнения, задействуя eBPF или собственные технологии. Чуть ниже мы поговорим о недостатках современных утилит на основе eBPF.

Для начала рассмотрим ряд других свойств, для которых можно создать профили для конкретного образа контейнера.

Профили доступа к файлам

Подобно тому как eBPF или другие технологии позволяют отслеживать системные вызовы, предназначенные для запуска исполняемых файлов, можно и отслеживать системные вызовы, с помощью которых можно получить доступ к файлам. Вообще говоря, набор местоположений файлов, к которым может потенциально обращаться конкретный микросервис, довольно ограничен. В качестве примера я получила с помощью Tracee следующий список файлов для контейнера `nginx`:

```
openat  /etc/ld.so.cache
openat  /lib/x86_64-linux-gnu/libdl.so.2
openat  /lib/x86_64-linux-gnu/libpthread.so.0
openat  /lib/x86_64-linux-gnu/libcrypt.so.1
openat  /lib/x86_64-linux-gnu/libpcre.so.3
openat  /usr/lib/x86_64-linux-gnu/libssl.so.1.1
openat  /usr/lib/x86_64-linux-gnu/libcrypto.so.1.1
openat  /lib/x86_64-linux-gnu/libz.so.1
openat  /lib/x86_64-linux-gnu/libc.so.6
openat  /etc/localtime
openat  /var/log/nginx/error.log
openat  /usr/lib/ssl/openssl.cnf
openat  /sys/devices/system/cpu/online
openat  /etc/nginx/nginx.conf
openat  /etc/nsswitch.conf
openat  /etc/ld.so.cache
```

```
openat  /lib/x86_64-linux-gnu/libnss_files.so.2
openat  /etc/passwd
openat  /etc/group
openat  /etc/nginx/mime.types
openat  /etc/nginx/conf.d
openat  /etc/nginx/conf.d/default.conf
openat  /var/log/nginx/error.log
openat  /var/log/nginx/access.log
openat  /var/run/nginx.pid
openat  /proc/sys/kernel/ngroups_max
openat  /etc/group
```

Данный список файлов довольно длинен, поэтому даже опытный программист мог бы случайно пропустить часть из них, если бы создавал профиль вручную, но благодаря таким утилитам, как Tracee, создать полный список файлов, к которым потенциально может обращаться контейнер, совсем не сложно. Опять же ряд инструментов обеспечения безопасности позволяют автоматически создавать профили работающих контейнеров, а затем оповещать (блокировать) при попытке открытия файлов, которые выходят за рамки этого профиля.

Профили идентификаторов пользователей

Как обсуждалось в главе 6, можно задать идентификатор пользователя, от имени которого будут запускаться процессы внутри контейнера, и это еще один аспект, который позволяет контролировать утилиты обеспечения безопасности во время выполнения. (Надеюсь, вы используете не пользователей в профилях своих приложений — см. раздел «Выполнение контейнеров по умолчанию от имени суперпользователя» на с. 147.)

Как правило, контейнеру, выполняющему только одну задачу, достаточно работать от имени одного пользователя. Если этот контейнер задействует другую учетную запись, то это сразу должно вызвать подозрения. Если же какой-либо процесс неожиданно запускается от имени суперпользователя, то подобное повышение полномочий — еще больший повод для беспокойства.

Другие профили времени выполнения

Можно перейти на еще более низкий уровень и создать профиль `productsearch` в разрезе необходимых ему системных вызовов и привилегий. А на основе этого можно создать сжатый профиль seccomp или AppArmor (см. главу 8),

специально для контейнеров, в которых работает `productsearch`. Утилита `bane` (https://oreil.ly/aQy3Q) Джесс Фразель как раз и предназначена для генерации подобных профилей AppArmor.

Просмотрев системные вызовы `cap_capable` с помощью Tracee, я получила вот такой список необходимых для контейнера `nginx` привилегий:

```
CAP_CHOWN
CAP_DAC_OVERRIDE
CAP_DAC_READ_SEARCH
CAP_NET_BIND_SERVICE
CAP_SETGID
CAP_SETUID
CAP_SYS_ADMIN
```

Аналогичным образом можно получить список системных вызовов, используемых контейнером, для преобразования в профиль seccomp.

Создавать такие профили для микросервисных приложений намного проще, чем для монолитных, поскольку количество возможных путей выполнения кода микросервиса намного меньше. Поэтому относительно несложно пройти все пути, приводящие как к ошибке, так и к успешному выполнению, чтобы проверить, не пропустили ли мы какие-либо события доступа к файлам, системные вызовы и исполняемые файлы.

Итак, вы узнали, что можно создать профиль, описывающий «обычное» поведение микросервиса в смысле того, какие исполняемые файлы, идентификаторы пользователей и сетевой трафик можно от него ожидать. Существует несколько утилит на основе подобных профилей, обеспечивающих безопасность во время выполнения.

Утилиты обеспечения безопасности контейнеров

Вы уже встречали некоторые из этих утилит в предыдущих главах:

- в главе 8 мы говорили о настройке для каждого контейнера своего профиля AppArmor, SELinux или seccomp;
- контролировать сетевой трафик во время выполнения можно с помощью сетевых стратегий или service mesh, как описывается в главе 10.

Существуют и другие утилиты для контроля исполняемых файлов, доступа к файлам и идентификаторов пользователей во время выполнения. Большинство из них — платные, но существует и один вариант с открытым исходным кодом: проект Falco от CNCF. Он отслеживает поведение контейнеров с помощью eBPF и вызывает срабатывание предупреждений в случае аномалий,

например запуска исполняемого файла, непредусмотренного профилем (как в предыдущем примере). Этот способ выявления аномального поведения очень действенный, но имеет свои ограничения, когда дело доходит до исполнения, поскольку eBPF может обнаруживать, но не изменять системные вызовы. Для наблюдения и оповещения о потенциальных атаках этот механизм подходит отлично, однако остановка таких атак требует чего-то другого. Выдаваемые Falco предупреждения можно использовать для того, чтобы автоматически перенастроить работающую систему или обратиться за помощью к сопровождающему персонала.

Предотвращение или оповещение. Какую бы утилиту вы ни использовали для обеспечения безопасности во время выполнения, возникает один последний вопрос: что эта утилита должна сделать при обнаружении аномального поведения? Лучше всего, чтобы она предотвращала неожиданное поведение, что и происходит в случае применения сетевых стратегий и профилей seccomp/SELinux/AppArmor. Но как насчет прочих, обсуждавшихся в этой главе видов профилей времени выполнения?

Коммерческие утилиты, которые предоставляют настоящую защиту во время выполнения, задействуют проприетарные методики подключения к контейнерам и предотвращения запуска аномальных исполняемых файлов, применения неверных идентификаторов пользователей или непредвиденных вариантов доступа к файлам/сети.

Обычно у таких утилит есть режим, в котором при обнаружении аномального поведения просто выдается предупреждение, а не предпринимаются какие-либо упреждающие действия. Он удобен во время пробного периода, поскольку позволяет убедиться в правильности настройки профилей времени выполнения.

Если используемая вами утилита и не умеет предотвращать выходящее за рамки профиля поведение, то хотя бы выдаст предупреждение. То есть оповестит вас в случае какого-либо нежелательного поведения контейнера. Что делать с подобными предупреждениями — вопрос непростой.

- Если автоматически удалять контейнер в случае предупреждения, то не повлияет ли это на обслуживание пользователей?
- Более того, если удалить контейнер, то не потеряются ли при этом улики, которые могли бы пригодиться при последующем расследовании?
- Можно завершать выполнение только вызвавшего предупреждение процесса, но что, если это «хороший» процесс, который заставили сделать что-то неожиданное (например, с помощью атаки типа «внедрение кода»)?

- Что будет в случае, когда за создание нового экземпляра отвечает средство координации, если этот новый экземпляр тоже подвержен этой же атаке? Может возникнуть порочный круг, в котором запускается контейнер, обнаруживается неправильное поведение, утилита безопасности прерывает выполнение контейнера, а все для того, чтобы его снова создал механизм координации (например, представьте, как Kubernetes создает/уничтожает модули, чтобы обеспечить нужное количество реплик).
- Если версия контейнера — новая, то можно ли откатиться к предыдущей?
- Следует ли ждать вмешательства человека, который будет расследовать неожиданное поведение и примет решение, как на него реагировать? Подобный подход неизбежно означает значительную задержку реагирования на атаку, возможно длящуюся столько, что за это время злоумышленник успеет украсть данные или повредить систему.

Не существует однозначного ответа на все случаи жизни, когда речь идет об автоматической обработке предупреждений безопасности. Какая бы задержка ни потребовалась, злоумышленнику *может* хватить времени, чтобы нанести вред. В этом отношении профилактика лучше лечения.

Если используемые вами утилиты безопасности действительно смогут предотвратить нежелательное поведение внутри контейнера, то есть надежда, что контейнер сможет продолжить работу. Например, представьте, что злоумышленник взломал контейнер поиска товаров и пытается запустить там программу для добычи криптовалюты. Соответствующий исполняемый файл не упоминается в профиле, поэтому утилита обеспечения безопасности во время выполнения вообще предотвращает его запуск. «Хорошие» процессы продолжают работать как ни в чем не бывало, но атака с добычей криптовалюты предотвращена.

Лучший вариант — утилиты, способные предотвратить аномальное поведение и вдобавок генерирующие предупреждения или журналы, которые можно потом изучить и определить, была ли это действительно атака, и если да, то решить, что делать дальше.

Предотвращение отклонений

Как вы помните из подраздела «Неизменяемые контейнеры» на с. 124, рекомендуется работать с контейнерами как с неизменяемыми. Контейнер создается на основе образа, после чего содержимое этого контейнера

не должно меняться. В образ должны быть включены все исполняемые файлы и зависимости, в которых нуждается код приложения. Мы уже обсуждали это ранее через призму выявления уязвимостей: не включенный в образ код нельзя просканировать на уязвимости, поэтому позаботьтесь, чтобы все, что вы хотите просканировать, уже там было.

Неизменяемость контейнеров открывает замечательную возможность, позволяющую выявить внедрение кода во время выполнения: *предотвращение отклонений* (drift prevention). Данную возможность должно предоставлять любое уважающее себя программное решение по обеспечению безопасности контейнеров во время выполнения. Это требует согласования шагов сканирования и выполнения:

- сканер в ходе сканирования вычисляет контрольные суммы содержащихся в образе файлов;
- во время выполнения контролирующая утилита добавляет проверку при каждом запуске нового исполняемого процесса в контейнере и сравнивает контрольную сумму исполняемого файла с полученной во время сканирования. Если они отличаются, то запуск исполняемого файла блокируется (и внутри контейнера генерируется ошибка «доступ запрещен»).

Использование контрольных сумм, а не списка имен файлов позволяет предотвратить возможную подмену злоумышленником настоящего исполняемого файла на внедренный.

Резюме

Возможности тонко настраиваемой защиты во время выполнения, описанные в этой главе, делают специализированные утилиты по обеспечению безопасности контейнеров весьма привлекательными, особенно для организаций, которым есть что терять, например банков или медицинских учреждений.

Мы приближаемся к концу книги. В последней главе мы рассмотрим список из десяти главных рисков для безопасности, составленный OWASP, и обсудим уменьшение последствий от них применительно к развертыванию контейнеров.

ГЛАВА 14

Контейнеры и десять главных рисков по версии OWASP

Если вы занимаетесь безопасностью, то почти наверняка слышали об OWASP — открытом проекте обеспечения безопасности веб-приложений; возможно, вы даже состоите в их местном отделении. Эта организация периодически публикует список десяти главных рисков для безопасности веб-приложений.

И хотя не все приложения, контейнеризованные или нет, относятся к веб-приложениям, данный источник прекрасно подходит в качестве отправной точки выяснения наиболее опасных атак. Вдобавок на сайте OWASP вы найдете прекрасные пояснения к перечисленным атакам и советы по их предотвращению. В этой главе я расскажу о десяти текущих главных рисках, имеющих отношение к безопасности контейнеров.

Внедрение кода

Если в вашем коде есть «дыра», открывающая путь для внедрения кода, то злоумышленник может воспользоваться ею, чтобы выполнить команды, замаскированные под данные. Лучше всего этот сценарий иллюстрирует нетленный персонаж xkcd Little Bobby Tables (https://xkcd.com/327).

Ничего относящегося исключительно к контейнерам тут нет, хотя сканирование образов контейнеров позволяет обнаружить в зависимостях известные уязвимости подобного типа. Анализируйте и тестируйте код своих приложений, как советует OWASP.

Взлом аутентификации

К этой категории относится взлом аутентификации и похищение учетных данных. На прикладном уровне одни и те же советы применимы как к контейнеризованным приложениям, так и к монолитным в обычных

системах, но здесь есть некоторые соображения, связанные именно с контейнерами:

- с необходимыми для каждого контейнера учетными данными следует обращаться как с секретными. Их необходимо хранить со всеми мерами предосторожности и передавать в контейнеры во время выполнения, как обсуждалось в главе 12;
- разбиение приложения на множество контейнеризованных компонентов означает, что им нужно будет как-то опознавать друг друга, обычно с помощью сертификатов, и обмениваться информацией по защищенным соединениям. Эту задачу могут брать на себя сами контейнеры, либо можно делегировать ее выполнение service mesh. См. главу 11.

Раскрытие конфиденциальных данных

Чрезвычайно важно надежно защищать любые личные, финансовые и прочие конфиденциальные данные, к которым ваше приложение имеет доступ.

Неважно, контейнеризована она или нет, конфиденциальная информация должна всегда храниться и передаваться в зашифрованном с помощью криптостойкого алгоритма виде. По мере роста вычислительных возможностей старые алгоритмы начинают поддаваться взлому путем прямого подбора, а значит, теряют надежность.

А поскольку конфиденциальные данные зашифрованы, приложению понадобятся учетные данные для доступа к ним. Следуя принципам минимума полномочий и разделения обязанностей, давайте доступ к учетным данным только тем контейнерам, которым они действительно нужны. См. обсуждение безопасной передачи секретных данных в контейнеры в главе 12.

Сканируйте образы контейнеров на предмет ключей шифрования, паролей и прочих конфиденциальных данных.

Внешние сущности XML

Ничего относящегося исключительно к контейнерам в этой категории уязвимостей обработчиков XML нет. Как и в случае уязвимостей внедрения кода, просто следуйте советам OWASP в части анализа кода приложений на предмет изъянов, а также используйте сканер образов контейнеров, чтобы обнаружить уязвимости в зависимостях.

Взлом управления доступом

Эта категория относится к злоупотреблению излишними полномочиями, предоставленными пользователям или компонентам. Существует несколько специальных методик, позволяющих применять принцип минимума полномочий к контейнерам, как обсуждалось в главе 9:

- не запускайте контейнеры от имени суперпользователя;
- ограничивайте предоставляемые контейнерам привилегии;
- задействуйте seccomp, AppArmor или SELinux;
- по возможности используйте контейнеры, не требующие полномочий суперпользователя.

Указанные подходы позволяют ограничить радиус поражения, однако ни один из них не относится к полномочиям пользователей *на прикладном уровне*, поэтому все равно нужно следовать тем же советам, действующим и при развертывании обычных систем.

Неправильные настройки безопасности

В основе многих атак лежат ошибки, допущенные при настройке систем. В списке OWASP особо упомянуты такие примеры: не обеспечивающие должную безопасность или неполные настройки, общедоступное облачное хранилище и слишком пространные сообщения об ошибках, которые содержат конфиденциальную информацию. Последствия всего этого применительно к контейнерам и развертываниям в облачной среде можно смягчить следующим образом.

- Используйте положения, такие как тесты Центра интернет-безопасности (Center for Internet Security (CIS) Benchmarks), чтобы оценить, настроена ли ваша система в соответствии с практическими рекомендациями. Там есть тесты для Docker и Kubernetes, а также для хостов под управлением Linux. Возможно, в вашей конкретной среде не имеет смысла следовать всем рекомендациям, но это прекрасная отправная точка для оценки системы.
- Для проверки настроек и поиска таких проблем, как общедоступные корзины хранилища или неудачные стратегии паролей, при использовании общедоступных облачных сервисов существуют утилиты наподобие

CloudSploit (https://cloudsploit.com/) и DivvyCloud (https://divvycloud.com/). Гартнер (Gartner) называет это управлением концепцией облачной безопасности (Cloud Security Posture Management, CSPM). (Полное разоблачение: CloudSploit создан моим работодателем, компанией Aqua Security.)

- Как обсуждалось в главе 12, передача секретных данных через переменные среды чревата их раскрытием в журналах, поэтому я рекомендую использовать переменные среды только для неконфиденциальной информации.

К этой категории, возможно, следует отнести также информацию о конфигурации, входящую в каждый образ контейнера. Этот вопрос мы обсуждали в главе 6, вместе с практическими рекомендациями по безопасному созданию образов.

Межсайтовое выполнение сценариев (XSS)

Это еще одна категория, относящаяся к прикладному уровню, но выполнение приложений в контейнерах никак особенно на эти риски не влияет. Для обнаружения уязвимых зависимостей следует использовать сканер образов контейнеров.

Небезопасная десериализация

При подобных атаках злоумышленник «подсовывает» приложению для интерпретации специальный объект, в результате чего оно предоставляет пользователю дополнительные полномочия или меняет свое поведение каким-либо образом. (Я была свидетелем подобной атаки в 2011 году в качестве клиента Citibank, когда у банка обнаружилась уязвимость (https://oreil.ly/EsgO7), которая с помощью простого изменения URL открывала вошедшему в систему пользователю доступ к информации о счетах других клиентов.)

Опять же выполняется ли приложение в контейнере или нет — никак особенно на это не влияет, хотя существуют некоторые связанные именно с контейнерами способы ограничить последствия подобных атак.

- Советы OWASP по предотвращению такой атаки включают рекомендацию изолировать выполняющий десериализацию код и запускать его в среде с низким уровнем полномочий. Подобной изоляции можно достичь за счет выполнения шага десериализации в выделенном для микросервиса

контейнере, особенно при использовании Firecracker, gVisor, Unikernels и прочих подходов, которые мы видели в главе 8. Кроме того, ограничить полномочия, которые злоумышленник может попытаться задействовать при подобной атаке, позволяет выполнение контейнера от имени отличного от `root` пользователя, с минимальным количеством привилегий и сжатым профилем seccomp/AppArmor/SELinux.

- Еще одна рекомендация от OWASP — ограничить входящий и исходящий трафик контейнеров, которые выполняют десериализацию. Способы ограничения сетевого трафика мы обсуждали в главе 10.

Использование компонентов, содержащих известные уязвимости

Надеюсь, к этому времени вы уже догадываетесь, что я посоветую: используйте сканер образов для поиска известных уязвимостей в образах контейнеров. Вам также пригодится процесс/набор инструментов для:

- повторной сборки образов контейнеров с включением в них актуальных версий пакетов;
- выявления и замены запущенных контейнеров, лежащие в основе которых образы содержали уязвимости.

Недостаток журналирования и мониторинга

На сайте OWASP приводится шокирующая статистика, которая показывает, что в среднем выявление несанкционированного доступа занимает почти 200 дней. Этот срок можно значительно сократить, если проводить достаточно наблюдений за системой в сочетании с оповещением про неожиданное поведение.

В промышленной эксплуатации следует заносить в журналы события контейнеров, в том числе:

- события запуска/останова контейнера, включая идентификаторы образа и пользователя;
- доступ к секретным данным;
- любую модификацию полномочий;

- изменение содержимого контейнера (может указывать на внедрение кода, см. раздел «Предотвращение отклонений» на с. 208);
- входящие/исходящие сетевые соединения;
- монтирование томов (для анализа смонтированных томов, в которых могут оказаться конфиденциальные данные, как рассказывается в разделе «Монтирование каталогов с конфиденциальными данными» на с. 157);
- действия, завершившиеся неудачей, например попытки открыть сетевые соединения, записать в файлы или изменить права доступа пользователя, поскольку они могут указывать на то, что злоумышленник зондирует систему.

Наиболее серьезные коммерческие утилиты обеспечения безопасности контейнеров интегрированы со средствами корпоративного SIEM (security information and event management — управление относящимися к безопасности информацией и событиями) и позволяют получать относящуюся к безопасности аналитику и предупреждения через одну централизованную систему. Эти утилиты способны не только отслеживать атаки и сообщать о них постфактум, но и, что еще лучше, предотвращать их на основе профилей времени выполнения, как обсуждалось в главе 13.

Резюме

Список десяти главных рисков по версии OWASP — полезный источник информации, позволяющий повысить защиту подключенных к Интернету приложений от наиболее распространенных типов атак.

Вероятно, вы обратили внимание на то, что относящаяся к контейнерам рекомендация, которая чаще всего встречается в этой главе, звучит так: просканировать образы контейнеров на известные уязвимости в сторонних зависимостях. И хотя некоторые вещи обнаружить при этом не получится (в частности, «дыры» в коде приложения, пригодные для использования злоумышленниками), но отдача при этом получится больше, чем у любого другого средства предотвращения атак, которое вы только можете применять для контейнеризованной системы.

Заключение

Поздравляем, вы дочитали книгу до конца!

Прежде всего, я надеюсь, теперь вы хорошо представляете, что такое контейнеры, что пригодится вам при обеспечении безопасности систем, развертываемых на основе контейнеров. Кроме того, вы теперь вооружены знаниями о доступных вариантах изоляции, если изоляции обычных контейнеров между рабочими заданиями для ваших целей недостаточно.

Я также надеюсь, что вам теперь понятно, как контейнеры обмениваются информацией друг с другом и с окружающим миром. Передача данных по сети — обширная тема, но главное, что вы должны для себя уяснить: контейнеры — единица не только развертывания, но и безопасности. Существует множество вариантов, как ограничить трафик так, чтобы между контейнерами, а также во внешний мир и из него передавался лишь тот трафик, который и должен.

Думаю, вы понимаете теперь, насколько полезна многослойная защита в случае несанкционированного доступа. Если даже злоумышленнику удалось воспользоваться уязвимостью в системе, его ждут далее другие преграды. Чем больше слоев защиты, тем меньше вероятность успешной атаки.

Как вы видели в главе 14, для контейнеров существуют специфические меры предотвращения, применимые к распространенным атакам на веб-приложения. Список из десяти наиболее частых, который я привела выше, охватывает далеко не все возможные слабые места ваших систем. Теперь, после прочтения всей книги, рекомендую вам освежить в памяти список векторов атак на контейнеры из раздела «Модель угроз для контейнеров» на с. 23. В приложении вы также найдете список вопросов, с помощью которых удобно анализировать, какие места развернутой системы наиболее уязвимы и где следует усилить меры безопасности.

Надеюсь, информация из этой книги поможет вам защитить развертываемые вами системы при любых обстоятельствах. Если вы подверглись атаке — удалось ли злоумышленнику взломать вашу систему, или вы смогли сохранить приложение и данные в безопасности, — расскажите мне об этом. Я буду рада любой обратной связи, комментариям, рассказам об атаках, а также проблемах, найденных вами в данной книге, вы можете отправлять свои письма по адресу containersecurity.tech. В Twitter меня можно найти под именем @lizrice (https://twitter.com/lizrice).

Приложение. Контрольный список по безопасности

В этом приложении перечислены важные нюансы, которые желательно учесть при выборе оптимального способа обеспечения безопасности контейнеризованной системы. Возможно, в вашей среде не имеет смысла применять *все* эти рекомендации, но хотя бы задуматься о них вовсе не помешает. Разумеется, данный список отнюдь не исчерпывающий!

- ❑ Все ли ваши контейнеры запущены от имени несуперпользователя? См. раздел «Выполнение контейнеров по умолчанию от имени суперпользователя» на с. 147.
- ❑ Запускаете ли вы какие-нибудь контейнеры с флагом `--privileged`? См. раздел «Флаг --privileged и привилегии» на с. 155.
- ❑ Запускаете ли вы контейнеры с доступом только для чтения, если есть такая возможность? См. подраздел «Неизменяемые контейнеры» на с. 124.
- ❑ Проверяете ли вы, не смонтированы ли в контейнере какие-либо каталоги с хоста, содержащие конфиденциальную информацию? А как насчет сокета Docker? См. разделы «Монтирование каталогов с конфиденциальными данными» на с. 157 и «Монтирование сокета Docker» на с. 158.
- ❑ Выполняете ли вы конвейер CI/CD в кластере, предназначенном для промышленной эксплуатации? Располагает ли он привилегированными возможностями доступа, использует ли сокет Docker? См. подраздел «Опасности команды docker build» на с. 101.
- ❑ Сканируете ли вы свои образы контейнеров на предмет уязвимостей? Налажен ли у вас процесс и есть ли инструментарий для повторной сборки и развертывания контейнеров, в базовом образе которых были найдены уязвимости? См. главу 7.

- ❑ Используете ли вы профили seccomp или AppArmor? Для начала подойдут профили Docker по умолчанию; еще лучше будет создать «сжатый» профиль для каждого приложения. См. главу 8.
- ❑ Включен ли SELinux, если операционная система хоста его поддерживает? Есть ли у вашего приложения нужные профили SELinux? См. раздел «Модуль SELinux» на с. 138.
- ❑ Какой базовый образ вы используете? Можете ли вы воспользоваться пустым образом или образом distroless, дистрибутивом Alpine или минимальным RHEL? Можете ли вы сократить содержимое своих образов, чтобы уменьшить поверхность атаки? См. подраздел «Практические рекомендации по безопасности Dockerfile» на с. 109.
- ❑ Требуете ли вы использования неизменяемых контейнеров? Другими словами, есть ли у вас гарантии, что весь исполняемый код добавляется в образ контейнера во время сборки, а не во время выполнения? См. подраздел «Неизменяемые контейнеры» на с. 124.
- ❑ Установили ли вы ограничения ресурсов для своих контейнеров? См. раздел «Установка ограничений на ресурсы» на с. 52.
- ❑ Используете ли вы контроль допуска, чтобы гарантировать, что в промышленной эксплуатации контейнеры будут создаваться только на основе проверенных образов? См. подраздел «Контроль допуска» на с. 115.
- ❑ Используете ли вы mTLS-соединения между компонентами? Реализовать их можно в коде приложения или с помощью service mesh. См. главу 11.
- ❑ Есть ли у вас сетевая стратегия ограничения трафика между компонентами? См. главу 10.
- ❑ Используете ли вы для передачи секретных данных между контейнерами временную файловую систему? Зашифрованы ли ваши секретные данные во время хранения и передачи? Используете ли вы систему управления секретными данными для их хранения и замены? См. главу 12.
- ❑ Используете ли вы утилиту обеспечения безопасности во время выполнения, чтобы гарантировать, что внутри контейнеров будут запускаться только нужные исполняемые файлы? См. главу 13.
- ❑ Применяете ли вы программное решение по обеспечению безопасности контейнеров во время выполнения для предотвращения отклонений? См. раздел «Предотвращение отклонений» на с. 208.

- Используете ли вы хост-компьютеры исключительно для выполнения контейнеров, отдельно от прочих приложений? Установлены ли на операционных системах ваших хостов все последние выпуски обновлений безопасности? Взвесьте возможность установки на них операционной системы, специально предназначенной для хостов. См. раздел «Хост-компьютеры контейнеров» на с. 82.
- Выполняете ли вы регулярные проверки настроек безопасности нижележащей облачной инфраструктуры с помощью утилиты CSPM? Настроены ли ваши хосты и контейнеры в соответствии с практическими рекомендациями по обеспечению безопасности, например тестами Центра интернет-безопасности для Linux, Docker и Kubernetes? См. раздел «Неправильные настройки безопасности» на с. 212.

Об авторе

Лиз Райс (Liz Rice) — вице-президент компании Aqua Security, занимающейся созданием ПО с открытым исходным кодом и ориентированной на безопасность контейнеров. Лиз отвечает за несколько проектов, включая Trivy, Tracee, kubehunter и kube-bench. Кроме того, она была председателем группы по техническому надзору CNCF и сопредседателем конференций KubeCon + CloudNativeCon 2018 в Копенгагене, Шанхае и Сиэтле.

Лиз имеет огромный опыт в сфере разработки программного обеспечения, командной работы и управления программным продуктом, полученный при работе над сетевыми протоколами и распределенными системами, а также в сфере цифровых технологий, в частности устройств с голосовым управлением, создания музыки и VoIP. В свободное от написания кода и соответствующих докладов время Лиз катается на велосипеде в местах с более приятной, чем в ее родном Лондоне, погодой, а также соревнуется в виртуальных гонках на Zwift.

Об иллюстрации на обложке

На обложке изображен представитель семейства кольчужных сомов, называемых также лорикариевыми *(Loricariidae)* или просто лорикой. Эти рыбы населяют пресноводные реки и ручьи Коста-Рики, Панамы и Южной Америки. Кольчужные сомы легко приспосабливаются к самым разным средам обитания: медленным и быстрым потокам, каналам, прудам, озерам, эстуариям и даже домашним аквариумам.

Существует более 680 видов кольчужных сомов самых разных расцветок, форм и размеров. Их характерные особенности: гибкие костяные пластины, отличающие их от других сомов, сплюснутое тело и направленный вниз рот-присоска, с помощью которого они питаются, дышат и прикрепляются к различным поверхностям. Их рот и губы приспособлены для того, чтобы одновременно прикрепляться к чему-либо и дышать, а тело и плавники покрыты вкусовыми рецепторами. В зависимости от внешних условий их размер может составлять от десяти сантиметров до метра. Кольчужные сомы практически всеядны, в их рацион входят водоросли, беспозвоночные, мелкие двустворчатые моллюски, дафнии, черви, личинки насекомых и детрит; известен даже один вид, питающийся деревом. Лорикариевые обычно заботятся о потомстве, многие виды выкапывают длинные норы у берега, куда самки откладывают яйца. Самцы охраняют яйца до того момента, когда вылупятся мальки.

Кольчужные сомы ведут ночной оседлый образ жизни, но при попадании в новую среду склонны распространяться по территории и вытеснять местные популяции рыб. Помимо бронированного тела, кольчужные сомы в процессе эволюции развили пищеварительную систему, способную служить дополнительным органом дыхания. При необходимости они способны дышать воздухом и существовать вне воды в течение более чем 20 часов!

Автор иллюстрации на обложке — Карен Монтгомери, за основу взята черно-белая гравюра из «Зоологии» Джорджа Шоу.

Лиз Райс

Безопасность контейнеров. Фундаментальный подход к защите контейнеризированных приложений

Перевел с английского И. Пальти

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*А. Питиримов*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Н. Хлебина*
Художественный редактор	*В. Мостипан*
Корректоры	*О. Андриевич, Е. Павлович*
Верстка	*Г. Блинов*

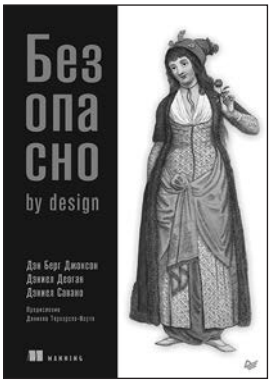
Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 06.2021. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —
Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 28.05.21. Формат 70×100/16. Бумага офсетная. Усл. п. л. 18,060. Тираж 500. Заказ 0000.



Дэн Берг Джонсон,
Дэниел Деоган,
Дэниел Савано

БЕЗОПАСНО BY DESIGN

«Безопасно by Design» не похожа на другие книги по безопасности. В ней нет дискуссий на такие классические темы, как переполнение буфера или слабые места в криптографических хэш-функциях. Вместо собственно безопасности она концентрируется на подходах к разработке ПО. Поначалу это может показаться немного странным, но вы поймете, что недостатки безопасности часто вызваны плохим дизайном. Значительного количества уязвимостей можно избежать, используя передовые методы проектирования. Изучение того, как дизайн программного обеспечения соотносится с безопасностью, является целью этой книги. Вы узнаете, почему дизайн важен для безопасности и как его использовать для создания безопасного программного обеспечения.

Джейсон Андресс

ЗАЩИТА ДАННЫХ. ОТ АВТОРИЗАЦИИ ДО АУДИТА

Чем авторизация отличается от аутентификации? Как сохранить конфиденциальность и провести тестирование на проникновение? Автор отвечает на все базовые вопросы и на примерах реальных инцидентов рассматривает операционную безопасность, защиту ОС и мобильных устройств, а также проблемы проектирования сетей. Книга подойдет для новичков в области информационной безопасности, сетевых администраторов и всех интересующихся. Она станет отправной точкой для карьеры в области защиты данных.