O'REILLY®

BPF для мониторинга Linux

Дэвид Калавера
Лоренцо Фонтана

Linux Observability with BPF

Advanced Programming for Performance Analysis and Networking

David Calavera and Lorenzo Fontana

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

BPF для мониторинга Linux

Дэвид Калавера
Лоренцо Фонтана

ПИТЕР®

Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону
Самара • Минск

2021

ББК 32.973.2-018.2
УДК 004.43
К17

Калавера Дэвид, Фонтана Лоренцо

К17 BPF для мониторинга Linux. — СПб.: Питер, 2021. — 208 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-1624-9

Виртуальная машина BPF — один из важнейших компонентов ядра Linux. Ее грамотное применение позволит системным инженерам находить сбои и решать даже самые сложные проблемы.

Вы научитесь создавать программы, отслеживающие и модифицирующие поведение ядра, сможете безопасно внедрять код для наблюдения событий в ядре и многое другое.

Дэвид Калавера и Лоренцо Фонтана помогут вам раскрыть возможности BPF. Расширьте свои знания об оптимизации производительности, сетях, безопасности.

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

ББК 32.973.2-018.2
УДК 004.43

ISBN 978-1492050209 англ. Authorized Russian translation of the English edition of Linux Observability with BPF
ISBN 9781492050229 © 2020 David Calavera and Lorenzo Fontana
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 978-5-4461-1624-9 © Перевод на русский язык ООО Издательство «Питер», 2021
© Издание на русском языке, оформление ООО Издательство «Питер», 2021
© Серия «Бестселлеры O'Reilly», 2021

Краткое содержание

Оглавление

Вступление

Как программисту и веб-разработчику, мне нравится узнавать о последних дополнениях к различным ядрам и исследованиях в области вычислительной техники. Когда я впервые попробовала поработать с Berkeley Packet Filter (BPF) и Express Data Path (XDP) в Linux, я в них влюбилась. Это очень хорошие инструменты, и я рада, что данная книга привлекает к ним внимание, чтобы все больше разработчиков использовали их в своих проектах.

Позвольте подробнее рассказать о моей работе и о том, почему я так полюбила эти интерфейсы ядра. В свое время я выступала в качестве основного сопровождающего Docker вместе с Дэвидом. Docker, если вы не в курсе, посылает `iptables` бóльшую часть логики фильтрации и маршрутизации для контейнеров. Первым делом я выпустила для Docker патч, решающий такую проблему: `iptables` не предоставляла в CentOS одинаковые флаги командной строки, поэтому запись в `iptables` не удавалась. Было еще много загадочных проблем, похожих на озвученную, — опытные разработчики меня поймут. Кроме того, `iptables` не предназначена для соблюдения тысяч правил на хосте, к тому же это ухудшает производительность.

Затем я узнала о BPF и XDP. Познакомившись с ними поближе, я забыла о проблемах с `iptables`! Сообщество разработчиков ядра даже работает над заменой `iptables` на BPF (oreil.ly/cuqTy). Аллилуйя! Cilium (cilium.io) — инструмент для создания контейнерных сетей — также использует BPF и XDP для внутренних компонентов своего проекта.

Но это не все! BPF может сделать гораздо больше, чем просто выполнить сценарий `iptables`. С BPF вы можете отследить любую функцию `syscall` или

ядра, а также любую программу пользовательского пространства. `bpftrace` (github.com/iovisor/bpftrace) предоставляет пользователям DTrace-подобные возможности в Linux из их командной строки. Вы можете отследить все открываемые файлы и процесс, вызывающий уже открытые, подсчитать системные вызовы, выполненные программой, отследить OOM killer и многое другое... BPF и XDP используются также в Cloudflare (oreil.ly/OZdmj) и балансировщике нагрузки Facebook (oreil.ly/wrM5-) для предотвращения распределенных атак типа «отказ в обслуживании». Не буду сейчас говорить о том, почему XDP так хорош в отбрасывании пакетов, потому что вы узнаете об этом в главах, посвященных XDP и сетевым технологиям.

Я познакомилась с Лоренцо в сообществе Kubernetes. Созданный им инструмент `kubectl-trace` (oreil.ly/Ot7kq) позволяет пользователям легко запускать собственные программы трассировки в кластерах Kubernetes.

Мой любимый сценарий использования BPF — написание пользовательских трассировщиков, демонстрирующих людям, что производительность их программного обеспечения невысока или программа выполняет слишком много обращений к системным вызовам. Никогда не стоит недооценивать возможность доказать чью-то неправоту на основании объективных данных. Не волнуйтесь: эта книга поможет вам написать первую программу отслеживания. Существовавшие ранее инструменты использовали очереди потерь для отправки наборов образцов (sample set) в пространство пользователя для агрегации, а BPF — более удобный инструмент, поскольку позволяет создавать гистограммы и применять фильтры непосредственно в источнике событий. И в этом его прелесть.

По долгу службы я работаю над инструментами для разработчиков. Лучшие инструменты обеспечивают автономность интерфейсов, что позволяет реализовывать любые возможности. Процитирую Ричарда Фейнмана: «Я очень рано понял разницу между знанием названия предмета и знанием предмета»[1]. До сих пор вы могли знать только название BPF и то, что он может быть полезен для вас.

Что мне нравится в этой книге — она дает знания, которые пригодятся вам для создания новых инструментов с помощью BPF. Прочитав книгу и вы-

[1] Оригинальная цитата: I learned very early the difference between knowing the name of something and knowing something.

полнив упражнения, вы сможете использовать BPF как свою секретную суперсилу. Можете сохранить BPF в своем инструментарии на тот случай, когда он окажется наиболее необходим и максимально полезен. Вы не просто изучите BPF, но и поймете, как он работает. Прочитайте эту книгу, и вы узнаете, на что способен BPF.

Эта развивающаяся экосистема очень интересна! Я надеюсь, что она еще вырастет, так как все больше людей начинают пользоваться BPF. Я с радостью прочитаю о том, что в итоге создадут читатели этой книги, будь то сценарий для отслеживания глупой ошибки в программном обеспечении, настраиваемый брандмауэр или инфракрасное декодирование (lwn.net/Articles/759188). Обязательно сообщайте мне обо всем, что вы разрабатываете!

Джесси Фразелль

Предисловие

В 2015 году Дэвид был разработчиком ядра в Docker — компании, которая популяризировала контейнеры. Его повседневная деятельность заключалась в помощи сообществу и развитии проекта. В его обязанности входило также рассмотрение срочных запросов, отправленных членами сообщества. А еще он должен был убедиться в том, что Docker работает для всех видов сценариев, включая высокопроизводительные рабочие нагрузки, которые поступают с тысяч контейнеров, в любой момент времени.

Для диагностики проблем с производительностью в Docker мы использовали *флейм-графы*, которые представляют собой инструмент для расширенной визуализации, облегчающей навигацию по этим данным. Язык программирования Go позволяет легко измерять производительность приложений и извлекать сведения о ней с помощью встроенной конечной точки HTTP и создавать графики на их основе. Дэвид написал статью о возможностях профилировщика Go и о том, как можно использовать выдаваемые им показатели для создания флейм-графов. Существенный недостаток того, как Docker собирает данные о производительности, состоит в том, что профилировщик по умолчанию отключен. Поэтому, если вы пытаетесь отладить проблему, связанную с производительностью, первое, что необходимо предпринять, — перезапустить Docker. Основная проблема этой стратегии в том, что при перезапуске службы вы, вероятно, потеряете соответствующие собираемые данные, а затем вам придется ждать, пока вновь не произойдет

событие, которое вы пытаетесь отследить. В своей статье о флейм-графах Дэвид упомянул это как необходимый шаг для измерения производительности Docker, но не обязательно все должно происходить именно так. Эта реализация заставила его начать исследовать различные технологии для сбора и анализа показателей производительности любого приложения, в результате чего и произошло его знакомство с BPF.

Параллельно, но без участия Дэвида, Лоренцо искал возможность лучше разобраться во внутренних компонентах ядра Linux. Он обнаружил, что можно легко изучить многие подсистемы ядра, обратившись к ним с помощью BPF. Пару лет спустя, работая в компании InfluxData, он смог применить BPF для того, чтобы ускорить прием данных в InfluxCloud. Сейчас Лоренцо состоит в сообществах BPF и IOVisor и трудится в компании Sysdig над Falco.

За последние несколько лет мы использовали BPF в нескольких сценариях — от сбора данных о применении кластеров Kubernetes до управления политиками сетевого трафика. Мы узнали о его плюсах и минусах, поработав с ним и прочитав множество постов в блогах таких лидеров технологии, как Брендан Грегг и Алексей Старовойтов, и таких компаний, как Cilium и Facebook. Эти публикации очень помогли нам в свое время, а также послужили превосходным справочным материалом для создания данной книги.

Изучив множество ресурсов, мы поняли, что каждый раз, когда нужно было что-то узнать о BPF, нам приходилось перечитывать множество постов в блогах, страницы руководств и прочих материалов в Интернете. Написание этой книги — попытка собрать все знания, разбросанные в Сети, под одной обложкой, чтобы рассказать об этой фантастической технологии следующему поколению энтузиастов BPF.

Мы разделили нашу работу на девять глав, в которых демонстрируем, чего вы можете достичь, используя BPF. Можно читать отдельные главы как справочное руководство, но если вы новичок в BPF, рекомендуем знакомиться с ними по порядку. Так вы поймете основные концепции BPF и узнаете, каков его потенциал.

Мы надеемся: даже если вы эксперт в области анализа наблюдаемости и производительности или исследуете новые возможности производственных систем, то тоже почерпнете что-то для себя из этой книги.

Условные обозначения

В этой книге используются следующие условные обозначения.

Курсив

Применяется для обозначения новых понятий и терминов, которые авторы хотят особо выделить.

Шрифт без засечек

Используется для обозначения адресов электронной почты и URL-адресов.

`Моноширинный`

Используется для текста (листингов) программ, а также внутри абзацев для выделения элементов программ: имен переменных или функций, названий баз данных, типов данных, имен переменных среды, инструкций и ключевых слов, имен файлов и каталогов.

Этот значок означает совет или предложение.

Этот значок указывает на примечание общего характера.

Этот значок обозначает предупреждение.

Использование примеров кода

Сопутствующий материал (примеры кода, упражнения и т. д.) вы можете найти по адресу github.com/bpftools/linux-observability-with-bpf.

Эта книга призвана помочь вам в работе. Примеры кода из нее вы можете использовать в своих программах и документации. Если объем кода не-

существенный, связываться с нами для получения разрешения не нужно. Например, для написания программы, использующей несколько фрагментов кода из этой книги, разрешения не требуется. А вот для продажи или распространения компакт-диска с примерами из книг издательства O'Reilly нужно получить разрешение. Ответы на вопросы с использованием цитат из этой книги и примеров кода разрешения не требуют. Но для включения объемных примеров кода из этой книги в документацию по вашему программному продукту разрешение понадобится.

Мы приветствуем указание ссылки на источник, но не делаем это обязательным требованием. Такая ссылка обычно включает название книги, имя автора, название издательства и ISBN. Например: «BPF для мониторинга Linux, Дэвид Калавера, Лоренцо Фонтана (Питер), 2020. 978-5-4461-1624-9».

Если вам покажется, что использование кода примеров выходит за рамки оговоренных выше условий и разрешений, свяжитесь с нами по адресу permissions@oreilly.com.

Благодарности

Написать книгу оказалось сложнее, чем мы думали, но это было одно из самых полезных занятий в нашей жизни. Книга создавалась на протяжении многих дней и ночей, и это было бы невозможно без помощи наших партнеров, семей, друзей и домашних питомцев. Мы хотели бы поблагодарить Дебору Пейс, подругу Лоренцо, и его сына Рикардо за терпение, которое они проявили, когда он бесконечно сидел за компьютером. Спасибо другу Лоренцо Леонардо Ди Донато за советы и написание статей о XDP и тестировании.

Мы бесконечно благодарны Робин Минс, жене Дэвида, за то, что она вычитала первые наброски, с которых началась эта книга, и черновики нескольких глав, помогла ему написать много статей за те годы, что он был занят сочинительством, и смеялась над выдуманными им английскими словами.

Мы оба хотим поблагодарить всех тех, кто разработал eBPF и BPF. Особая благодарность Дэвиду Миллеру и Алексею Старовойтову за их постоянный вклад в улучшение ядра Linux и в конечном счете eBPF, а также в организацию сообщества. Спасибо Брендану Греггу за желание поделиться, энтузиазм и работу над инструментами, которые делают eBPF доступным для всех.

Команде IOVisor — за то, что они делают, и за тот вклад, который они внесли в создание `bpftrace`, `gobpf`, `kubectl-trace` и BCC. Дэниелу Боркману — за его огромную работу, в частности, над `libbpf` и инфраструктурой инструментов. Джесси Фразелль — за то, что она написала вступление и вдохновила нас и тысячи других разработчиков. Жерому Петаццони — за то, что он был лучшим научным редактором, которого мы только могли пожелать: его вопросы заставили нас переосмыслить многие фрагменты этой книги и пересмотреть свой выбор примеров кода.

Благодарим всех причастных к разработке ядра Linux, особенно тех, кто активно участвует в сопровождении BPF, за ответы на вопросы, исправления и новшества. Наконец, спасибо всем, кто участвовал в издании этой книги, включая редакторов Джона Девинса и Мелиссу Поттер, создателей обложки и корректоров, которые сделали эту книгу более читабельной.

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.

1 Введение

За последние несколько десятилетий вычислительные системы стали не проще — наоборот, сложнее. Рассуждения о том, как программное обеспечение само себя обеспечивает, привели к созданию нескольких бизнес-категорий, и все они пытаются решить проблемы, связанные с необходимостью понять сложные системы. Один из подходов — анализ журналов данных, генерируемых приложениями, работающими в вычислительной системе. Журналы являются отличным источником информации. Они предоставляют точные данные о том, как себя ведет приложение. Тем не менее они ограничивают вас, потому что вы получаете только ту информацию, на сбор которой инженеры, создавшие приложение, запрограммировали эти журналы. Сбор любой дополнительной информации в формате журнала из любой системы может быть такой же сложной задачей, как декомпиляция программы и наблюдение за ходом выполнения. Другой популярный подход заключается в использовании метрик, позволяющих понять, почему программа ведет себя именно так и как она это делает. Метрики отличаются от журналов форматом данных: журналы предоставляют явные данные, а метрики объединяют данные, чтобы выяснить, как программа ведет себя в определенный момент времени.

Наблюдаемость — это новая практика, которая позволяет подойти к данной проблеме с другой стороны. Мы определяем наблюдаемость как способность задавать произвольные вопросы и получать сложные ответы от любой системы. Основное различие между наблюдаемостью, ведением журналов и группированием метрик — это данные, которые вы собираете. Учитывая то, что, практикуя наблюдаемость, вы должны отвечать на любой произвольный

вопрос в любой момент, единственный способ рассуждать о данных — собрать все данные, которые ваша система может сгенерировать, и скомпоновать их только тогда, когда необходимо ответить на ваши вопросы.

Нассим Николас Талеб, автор такого бестселлера, как *Antifragile: Things That Gain From Disorder*[1], популяризировал термин *«черный лебедь»*, назвав так неожиданные события, имеющие серьезные последствия, которых можно было ожидать, если бы их наблюдали до того, как они случились. В своей книге «Черный лебедь»[2] он объясняет, как наличие соответствующих данных может помочь снизить риск возникновения этих редких событий. При разработке программного обеспечения «черные лебеди» встречаются чаще, чем мы думаем, и они неизбежны. Поскольку можно предположить, что предотвратить такого рода события нельзя, то получить как можно больше информации о них — единственная возможность решить проблему так, чтобы она не оказала критического воздействия на бизнес-системы. Наблюдаемость помогает создавать надежные системы и предотвращать будущие «черные лебеди», поскольку она основана на той предпосылке, что вы собираете любые данные, которые помогут ответить на любой заданный в дальнейшем вопрос. Изучение «черного лебедя» и практика наблюдаемости сходятся в центральной точке — данных, которые вы собираете из своих систем.

Контейнеры Linux — это абстракция в дополнение к набору функций ядра Linux для изоляции компьютерных процессов и управления ими. Ядро, традиционно отвечающее за управление ресурсами, обеспечивает также изоляцию задач и безопасность. В Linux основными функциями, на которых основаны контейнеры, являются пространства имен и контрольные группы. Пространства имен — это компоненты, которые изолируют задачи друг от друга. В некотором смысле, когда вы находитесь внутри пространства имен, вам кажется, что операционная система не выполняет никаких других задач на компьютере. Контрольные группы — это компоненты, которые обеспечивают управление ресурсами. С точки зрения эксплуатации они дают вам детальный контроль над любым использованием ресурсов, таких как ЦП, дисковый ввод-вывод, сеть и т. д. В последнее десятилетие с ростом популярности контейнеров Linux

---

1 *Taleb N. N.* Antifragile: Things That Gain from Disorder. — N. Y.: Random House, 2012.

2 *Талеб Н. Н.* Черный лебедь. Под знаком предсказуемости. 2-е изд., доп. — М.: КоЛибри, 2020.

произошли изменения в том, как разработчики программного обеспечения проектируют большие распределенные системы и вычислительные платформы. Многоклиентские вычисления полностью зависят от этих функций в ядре.

Мы настолько полагаемся на низкоуровневые возможности ядра Linux, что у нас появляются новые источники информации, которые необходимо учитывать при разработке наблюдаемых систем. Ядро представляет собой систему, в которой вся работа описывается и реализуется на основе событий. Открытие файла — это своего рода событие, выполнение некоторой инструкции процессором — событие, получение сетевого пакета — событие и т. д. Berkeley Packet Filter (BPF) — это подсистема ядра, которая может проверять новые источники информации. BPF позволяет писать программы, которые выполняются безопасно, когда ядро реагирует на какое-либо событие. BPF обеспечивает безопасность, чтобы предотвратить системные сбои и вредоносное поведение каких-либо программ. BPF предоставляет новое поколение инструментов, которые помогают разработчикам систем наблюдать за новыми платформами и работать с ними.

В этой книге мы продемонстрируем возможности BPF, позволяющие сделать любую вычислительную систему более наблюдаемой. А также покажем, как писать программы BPF с использованием нескольких языков программирования. Мы поместили код для ваших программ в GitHub, поэтому не нужно копировать и вставлять его — вы найдете его в Git-репозитории к этой книге (github.com/bpftools/linux-observability-with-bpf).

Но прежде, чем начать разбирать технические аспекты BPF, посмотрим, как все начиналось.

История BPF

В 1992 году Стивен Маккейн и Ван Якобсон опубликовали статью *The BSD Packet Filter: A New Architecture for User-Level Packet Capture* («Пакетный фильтр BSD: новая архитектура для захвата пакетов на уровне пользователя»). В ней они описали способ реализации фильтра сетевых пакетов для ядра Unix, который работал в 20 раз быстрее, чем все остальные, имеющиеся на то время в области фильтрации пакетов. Пакетные фильтры имеют конкретную цель: предоставлять приложениям, которые отслеживают сетевую активность, прямую информацию из ядра. Обладая этой информацией, при-

ложения могут решить, что делать с пакетами. BPF представил два серьезных нововведения в области фильтрации пакетов:

- новую виртуальную машину (ВМ), предназначенную для эффективной работы с ЦП на основе регистров;
- возможность использования буферов для каждого приложения, способных фильтровать пакеты без копирования всей информации о них. Это минимизировало количество данных BPF, необходимых для принятия решений.

Такие радикальные улучшения заставили все системы Unix принять BPF в качестве технологии выбора для фильтрации сетевых пакетов, отказавшись от прежних реализаций, которые потребляли больше памяти и были менее производительными. Хотя они все еще присутствуют во многих производных ядра Unix, включая ядро Linux.

В начале 2014 года Алексей Старовойтов разработал расширенную реализацию BPF. Новый подход был оптимизирован для современного оборудования, благодаря чему его результирующий набор команд работает быстрее, чем машинный код, сгенерированный старым интерпретатором BPF. Расширенная версия также увеличила число регистров в виртуальной машине BPF с двух 32-битных регистров до десяти 64-битных. Увеличение количества регистров и их глубины позволило писать более сложные программы, поскольку разработчики могли свободно обмениваться дополнительной информацией, используя параметры функций. Эти изменения наряду с прочими улучшениями привели к тому, что расширенная версия BPF стала в четыре раза быстрее оригинальной реализации BPF.

Первоначальная цель создания новой реализации состояла в том, чтобы оптимизировать внутренний набор команд BPF, обрабатывающих сетевые фильтры. На этом этапе BPF все еще был ограничен пространством ядра, и только несколько программ в пользовательском пространстве могли задавать фильтры BPF для обработки ядром, такие как Tcpdump и Seccomp, о которых мы поговорим в следующих главах. Сегодня эти программы все еще генерируют байт-код для старого интерпретатора BPF, но ядро преобразует данные инструкции в значительно улучшенное внутреннее представление.

В июне 2014 года расширенная версия BPF стала доступной для пользователей. Это был переломный момент для будущего BPF. Как написал Алексей в патче с изменениями, «новый набор патчей демонстрирует потенциал eBPF».

BPF стал подсистемой ядра верхнего уровня и перестал ограничиваться только сетевым стеком. BPF-программы стали больше походить на модули ядра с сильным акцентом на безопасности и стабильности. В отличие от обычных модулей ядра, BPF-программы не требуют его перекомпиляции и гарантированно завершаются без сбоев.

Верификатор BPF, о котором мы поговорим в следующей главе, добавил необходимые гарантии безопасности. Здесь подразумевается, что любая BPF-программа завершится без сбоев и программы не будут пытаться получить доступ к памяти вне области их деятельности. Но эти преимущества сопровождаются определенными ограничениями: программы имеют максимально допустимый размер, и циклы должны быть ограничены, чтобы гарантировать, что память системы никогда не будет исчерпана неправильно написанной программой BPF.

Одновременно с изменениями, сделавшими BPF доступным из пространства пользователя, разработчики ядра добавили новый системный вызов (`syscall`) — `bpf`. Он станет центральным элементом связи между пользовательским пространством и ядром. Мы обсудим, как применять этот системный вызов для работы с программами и картами BPF, в главах 2 и 3.

Карты BPF станут основным механизмом обмена данными между ядром и пользовательским пространством. В главе 2 говорится о том, как применять эти специализированные структуры для сбора информации из ядра, а также отправки информации в программы BPF, которые уже работают в нем.

В книге в первую очередь рассматривается расширенная версия BPF. За пять лет, прошедших с момента появления расширенной версии, BPF получил значительное развитие, и мы подробно обсудим эволюцию программ BPF, карт BPF и подсистем ядра.

Архитектура

Архитектура BPF в ядре впечатляет. Далее мы еще рассмотрим конкретные детали, а сейчас, в этой главе, хотим дать краткий обзор того, как все работает.

Как мы говорили ранее, BPF — это высокоразвитая виртуальная машина, выполняющая инструкции кода в изолированной среде. В определенном смысле

вы можете думать о BPF как о виртуальной машине Java (JVM) — специализированной программе, которая выполняет машинный код, скомпилированный из исходного кода на языке программирования высокого уровня. Компиляторы, такие как LLVM и GNU Compiler Collection (GCC), в ближайшем будущем обеспечат поддержку BPF, что позволит вам компилировать код C в инструкции BPF. После того как код скомпилирован, BPF использует верификатор, чтобы убедиться, что программа безопасна для запуска в пространстве ядра. Он не позволяет запускать код, который может представлять угрозу для вашей системы из-за сбоя ядра. Если код безопасен, программа BPF будет загружена в ядро. Ядро Linux также включает в себя JIT-компилятор для команд BPF. JIT преобразует байт-код BPF в машинный код непосредственно после проверки программы, убирая все ненужное во время выполнения. Одним из интересных аспектов этой архитектуры является то, что вам не нужно перезагружать систему для загрузки BPF-программ — вы можете загрузить их по мере надобности, а также написать собственные сценарии инициализации, которые загружают программы BPF при запуске вашей системы.

Прежде чем запустить какую-либо BPF-программу, ядро должно знать, к какой точке выполнения она прикреплена. В ядре несколько точек подключения, и их список постоянно растет. Точки выполнения определяются типами программ BPF, мы обсудим их в следующей главе. Когда вы выбираете точку выполнения, ядро предоставляет также специальные помощники функций: их можно использовать для работы с данными, которые получает ваша программа, что связывает точки выполнения и программы BPF.

Последний компонент в архитектуре BPF отвечает за обмен данными между ядром и пользовательским пространством. Он называется *картой* BPF (см. главу 3). Карты BPF представляют собой двунаправленные структуры для обмена данными. Это означает, что вы можете записывать в них и читать из них с обеих сторон — из ядра и из пользовательского пространства. Существует несколько типов структур: от простых массивов и хеш-карт до специализированных карт, в которых можно сохранять целые BPF-программы.

Далее в книге мы более подробно рассмотрим каждый компонент архитектуры BPF. Вы также научитесь пользоваться преимуществами расширяемости и обмена данными BPF, изучая конкретные примеры, охватывающие различные темы: от анализа трассировки стека до сетевой фильтрации и изоляции во время выполнения.

Резюме

Мы написали эту книгу, чтобы познакомить вас с основными концепциями BPF, которые вам понадобятся в повседневной работе с данной подсистемой Linux. BPF все еще находится в процессе разработки, и каждый день совершенствуются новые концепции и парадигмы. В идеале эта книга поможет вам расширить свои знания, обеспечив прочную базу для понимания основных компонентов BPF.

В следующей главе будет подробно рассказано о структуре программ BPF и о том, как их запускает ядро. Мы также рассмотрим точки в ядре, в которых вы можете прикрепить указанные программы. Это поможет вам в деталях узнать, какие данные могут понадобиться вашим программам и как их использовать.

2 Запуск программ BPF

Виртуальная машина BPF способна выполнять инструкции в ответ на события, выдаваемые ядром. Однако не все программы BPF имеют доступ ко всем событиям, запускаемым ядром. Загружая программу в виртуальную машину BPF, вы должны решить, какой тип программы запускаете. Это информирует ядро о том, где программа будет запущена. При этом верификатор BPF будет знать, какие помощники разрешены в вашей программе. Выбирая тип программы, вы также выбираете интерфейс, который она реализует. Этот интерфейс гарантирует, что у вас есть доступ к соответствующему типу данных, и показывает, может ли программа получать доступ к сетевым пакетам напрямую.

В этой главе мы продемонстрируем, как написать ваши первые программы BPF. А также познакомим вас с различными типами программ BPF (существующими на момент написания этой книги), которые вы можете создать. На протяжении многих лет разработчики ядра добавляли различные точки входа, к которым вы можете присоединять программы BPF. Эта работа еще не завершена, и каждый день появляются новые способы использования BPF. В данной главе мы хотим сосредоточиться на некоторых наиболее полезных типах программ, чтобы дать вам представление о том, чего вы можете достичь с помощью BPF. В следующих главах также будет приведено множество примеров написания программ BPF.

Кроме того, мы рассмотрим роль, которую верификатор BPF играет в запуске ваших программ. Этот компонент проверяет, что ваш код безопасен для

выполнения, и помогает вам писать программы, которые не приведут к неожиданным результатам, таким как исчерпание памяти или внезапные сбои ядра. Но начнем мы с основ написания ваших собственных BPF-программ с нуля.

Написание программ BPF

Самый распространенный способ написания BPF-программ — использование подмножества C, скомпилированного с помощью LLVM. LLVM — это компилятор общего назначения, который может генерировать различные типы байт-кода. В нашем случае LLVM выведет код ассемблера BPF, который мы позже загрузим в ядро. Мы не собираемся создавать большую сборку BPF. После долгого обсуждения мы решили, что лучше привести примеры того, как применять его при определенных обстоятельствах, хотя вы можете легко найти несколько ссылок на другие примеры в Интернете или на справочных страницах BPF. В следующих главах мы покажем короткие примеры сборки BPF.

Ядро предоставляет системный вызов `bpf` для загрузки программ в виртуальную машину BPF после их компиляции. Этот вызов используется не только для загрузки программ, но и для других операций (больше примеров приведено в последующих главах). Ядро также предоставляет несколько утилит, которые избавляют вас от загрузки BPF-программ. В первом примере кода мы используем эти помощники, чтобы показать пример Hello, BPF World:

```
#include <linux/bpf.h>
#define SEC(NAME) __attribute__((section(NAME), used))

SEC("tracepoint/syscalls/sys_enter_execve")
  int bpf_prog(void *ctx) {
  char msg[] = "Hello, BPF World!";
  bpf_trace_printk(msg, sizeof(msg));
  return 0;
}

char _license[] SEC("license") = "GPL";
```

Эта первая программа содержит несколько интересных элементов. Мы используем атрибут `SEC`, чтобы сообщить виртуальной машине BPF, когда

хотим запустить программу. В данном случае мы запустим программу BPF, когда будет обнаружена точка трассировки в системном вызове `execve`. Точки трассировки — это статические метки в двоичном коде ядра, которые позволяют разработчикам вводить код для проверки работы ядра. Мы подробно поговорим о них в главе 4, а сейчас вам нужно знать только то, что `execve` — это инструкция, выполняющая другие программы. Итак, мы увидим сообщение `Hello, BPF World!` каждый раз, когда ядро обнаружит, что программа выполняет другую программу.

В конце примера мы указываем лицензию для этой программы. Поскольку ядро Linux лицензируется как GPL, оно может загружать только программы с такой же лицензией. Если установить какую-либо другую лицензию, ядро откажется загружать нашу программу. Мы применим `bpf_trace_printk` для печати сообщений в журнале трассировки ядра, который находится в папке `/sys/kernel/debug/tracing/trace_pipe`.

Используем `clang` для компиляции нашей первой программы в правильный двоичный файл ELF. Именно этот формат ядро ожидает для загрузки. Сохраним нашу первую программу в файле с именем `bpf_program.c`, чтобы затем скомпилировать:

```
clang -O2 -target bpf -c bpf_program.c -o bpf_program.o
```

Вы найдете несколько сценариев для компиляции этих программ в репозитории GitHub (oreil.ly/lbpf-repo), поэтому не нужно дословно запоминать синтаксис команды `clang`.

Теперь, когда мы скомпилировали нашу первую программу BPF, нужно загрузить ее в ядро. Как уже упоминалось, для этого используем специальный помощник, который предоставляется ядром, чтобы освободить стандартный шаблон компиляции и загрузки программы. Этот помощник называется `load_bpf_file`, он берет двоичный файл и пытается загрузить его в ядро. Вы можете найти помощник в репозитории GitHub, как и все примеры из книги. Все это показано в файле `bpf_load.h`:

```
#include <stdio.h>
#include <uapi/linux/bpf.h>
#include "bpf_load.h"

int main(int argc, char **argv) {
  if (load_bpf_file("hello_world_kern.o") != 0) {
    printf("The kernel didn't load the BPF program\n");
```

```
    return -1;
  }

  read_trace_pipe();

  return 0;
}
```

Мы собираемся использовать сценарий, чтобы скомпилировать программу и скомпоновать ее как двоичный файл ELF. В данном случае не нужно указывать цель, потому что эта программа не будет загружена в виртуальную машину BPF. Следует подключить внешнюю библиотеку, и написание сценария облегчит задачу объединения:

```
TOOLS=../../../tools
INCLUDE=../../../libbpf/include
HEADERS=../../../libbpf/src
clang -o loader -l elf \
  -I${INCLUDE} \
  -I${HEADERS} \
  -I${TOOLS} \
  ${TOOLS}/bpf_load.c \
  loader.c
```

Если вы захотите запустить эту программу, то можете выполнить двоичный файл с помощью `sudo ./loader`. `sudo` — это команда Linux, которая даст вам права root для выполнения определенной команды на вашем компьютере. Если вы не воспользуетесь `sudo`, то получите сообщение об ошибке, поскольку большинство BPF-программ могут быть загружены в ядро только пользователем с правами root.

Запустив эту программу, через несколько секунд вы увидите сообщение `Hello, BPF World!`, даже если ничего не делаете на компьютере. Это связано с тем, что программы, работающие в фоновом режиме, могут сейчас запускать другие программы.

Когда вы останавливаете данную программу, сообщение перестает отображаться в вашем терминале. BPF-программы выгружаются из ВМ, как только завершаются программы, которые их загрузили. В следующих главах мы рассмотрим, как сделать работу BPF-программы постоянной даже после завершения работы загрузчика, но пока не будем слишком углубляться. Это важная концепция, которую нужно учитывать, потому что во многих ситуациях важно, чтобы программы BPF запускались в фоновом режиме и собирали данные из системы независимо от того, запущены ли другие процессы.

Теперь, когда вы узнали о базовой структуре программ BPF, рассмотрим, какие программы вы можете написать, чтобы получить доступ к различным подсистемам ядра Linux.

Типы программ BPF

Хотя четкого деления программ на категории не существует, вы скоро поймете, что все их типы, описанные в этом разделе, распределены на две категории в зависимости от основного назначения.

Первая категория — это программы *трассировки*. Многие написанные вами программы помогут лучше понять, что происходит в вашей системе. Они обеспечивают информацию о поведении системы и оборудовании, на котором она работает. Могут получать доступ к областям памяти, используемым конкретными программами, и трассировать выполнение запущенных процессов. Они также дают прямой доступ к ресурсам, выделенным для каждого конкретного процесса, от применения файловых дескрипторов до использования процессора и памяти.

Вторая категория — это программы для *работы в сети*. Они позволяют вам контролировать сетевой трафик в своей системе, фильтровать пакеты, поступающие от сетевого интерфейса, и даже полностью отбрасывать их. Разные типы программ могут быть связаны с различными этапами сетевой обработки в ядре. В этом есть и преимущества, и недостатки. Например, вы можете привязать программы BPF к сетевым событиям, как только ваш сетевой драйвер получает пакет, но такая программа будет иметь доступ к меньшему количеству сведений о пакете, поскольку ядро еще не обладает достаточной информацией. В то же время можно привязать программы BPF к сетевым событиям непосредственно перед их передачей в пространство пользователя. В этом случае у вас будет гораздо больше информации о пакете, что поможет принимать более обоснованные решения, однако придется затратить ресурсы для полной обработки пакета.

Список типов программ, о которых мы поговорим далее, не разделен на категории — представим их в хронологическом порядке, в котором они были добавлены в ядро. Мы переместили наиболее редко используемые программы в конец раздела и пока сосредоточимся на наиболее полезных. Если вас интересует какая-либо программа, о которой здесь подробно не говорится, можете узнать о ней больше, введя команду `man 2 bpf` (oreil.ly/qXl0F).

Программы сокетной фильтрации

Программы типа `BPF_PROG_TYPE_SOCKET_FILTER` были добавлены в ядро Linux в числе первых. Привязывая программу BPF к открытому сокету, вы получаете доступ ко всем пакетам, проходящим через него. Программы с сокетными фильтрами не позволяют вам изменять содержимое этих пакетов или их назначение — они дают доступ к ним только для наблюдения. Метаданные, которые получает ваша программа, содержат относящуюся к сетевому стеку информацию, такую как тип протокола, используемого для доставки пакета.

Мы рассмотрим сокетную фильтрацию и другие сетевые программы более подробно в главе 6.

Программы kprobe

Как вы увидите в главе 4, в которой рассматривается трассировка, kprobes — это функции, которые можно динамически подключать к определенным точкам вызова в ядре. Программы типа BPF kprobe позволяют использовать программы BPF в качестве обработчиков kprobe. Они определены как тип `BPF_PROG_TYPE_KPROBE`. Виртуальная машина BPF гарантирует, что программы kprobe всегда безопасны при запуске, что является преимуществом по сравнению с традиционными модулями kprobe. Однако нужно помнить: kprobes не считаются стабильными точками входа в ядро, поэтому необходимо убедиться, что ваши программы kprobe совместимы с версиями ядра, которые вы используете.

Создавая программу BPF, которая связана с kprobe, вы должны решить, как она станет выполняться — в качестве первой инструкции в вызове функции или когда вызов завершится. Следует указать это в разделе заголовка программы BPF. Например, если вы хотите проверить аргументы, когда ядро создает системный вызов `exec`, то подключаете программу в начале вызова. В этом случае нужно прописать заголовок раздела `SEC("kprobe/sys_exec")`. Если же хотите проверить возвращаемое значение системного вызова `exec`, требуется написать в заголовке раздела `SEC("kretprobe/sys_exec")`.

В последующих главах книги мы гораздо подробнее обсудим kprobes. Они станут фундаментом для понимания трассировки с помощью BPF.

Программы трассировки

Программы этого типа позволяют вам подсоединить BPF-программы к обработчику трассировки, предоставляемому ядром. Программы трассировки определяются как тип `BPF_PROG_TYPE_TRACEPOINT`. Как вы увидите в главе 4, точки трассировки — это статические метки в кодовой базе ядра, которые позволяют вводить произвольный код для выполнения трассировки и отладки. Они менее гибки, чем kprobes, потому что должны быть определены ядром заранее, но гарантированно стабильны после введения в ядро соответствующей точки отладки или модуля. Это обеспечивает гораздо более высокий уровень предсказуемости при отладке системы.

Все точки трассировки в вашей системе определены в каталоге `/sys/kernel/debug/tracing/events`. Там вы найдете все подсистемы, включающие в себя любые точки трассировки, к которым вы можете подключить программу BPF. Еще один интересный факт состоит в том, что BPF объявляет собственные точки трассировки, поэтому вы можете писать программы BPF, которые проверяют поведение других программ BPF. Точки трассировки BPF определены в `/sys/kernel/debug/tracing/events/bpf`. Там можно найти, например, определение точки трассировки для `bpf_prog_load`. Это означает, что вы можете написать программу BPF, которая срабатывает при загрузке других программ BPF.

Как и kprobes, точки трассировки являются основой для понимания того, как работает трассировка с помощью BPF. В следующих главах мы поговорим о них подробнее и покажем, как писать действительно полезные программы.

Программы XDP

Программы XDP позволяют вам писать код, который выполняется на самом первом этапе, как только сетевой пакет поступает в ядро. Они определены как тип `BPF_PROG_TYPE_XDP`. Здесь предоставляется ограниченный набор информации из пакета, учитывая, что у ядра было немного времени для ее обработки. Поскольку пакет исследуется и выполняется на ранней стадии, вы можете намного лучше контролировать его обработку.

Программы XDP реализуют несколько действий, которыми вы можете управлять и которые позволяют решать, что делать с пакетом. Можете вернуть

XDP_PASS из своей программы XDP, что означает: пакет должен быть передан следующей подсистеме в ядре. Вы также можете вернуть XDP_DROP, что означает: ядро должно полностью игнорировать этот пакет и больше ничего с ним не делать. А еще можете вернуть XDP_TX, что означает: пакет должен быть направлен обратно в сетевую интерфейсную карту (NIC), через которую был получен.

Такая степень контроля дает возможность использовать много интересных программ на сетевом уровне. XDP стал одним из основных компонентов BPF, поэтому мы включили в книгу главу о нем. В главе 7 мы рассмотрим сценарии использования XDP, такие как реализация программ для защиты вашей сети от атак распределенного отказа в обслуживании (DDoS).

Программы Perf Event

Такие программы BPF позволяют вашим программам понять и воспринять события Perf. Они определены как тип BPF_PROG_TYPE_PERF_EVENT. Perf — это внутренний профилировщик ядра, который генерирует события, предоставляющие данные о производительности для аппаратного и программного обеспечения. Вы можете использовать его для мониторинга многих компонентов: от процессора вашего компьютера до любого программного обеспечения, работающего в вашей системе. После связывания программы BPF с событиями Perf ваш код будет выполняться каждый раз, когда Perf генерирует данные для анализа.

Программы для сокетов контрольных групп

Эти программы позволяют вам связать логику BPF с контрольными группами (cgroups). Они имеют тип BPF_PROG_TYPE_CGROUP_SKB. Это дает возможность механизму контрольных групп (cgroups) контролировать сетевой трафик процессов, которые к ним относятся. С помощью программ для сокетов контрольных групп вы можете решить, что делать с сетевым пакетом до его доставки процессу в контрольной группе. Любой пакет, который ядро пытается доставить любому процессу в той же группе, пройдет через один из этих фильтров. В то же время вы можете решить, что предпринять, когда процесс в контрольной группе отправляет сетевой пакет через данный интерфейс.

Как видите, их поведение аналогично действиям программ BPF_PROG_TYPE_SOCKET_FILTER. Основное различие состоит в том, что программы BPF_PROG_TYPE_CGROUP_SKB привязаны ко всем процессам внутри группы, а не к конкретным процессам. Такое поведение применяется ко всем существующим и будущим сокетам, созданным в данной группе. BPF-программы, связанные с контрольными группами, особенно полезны в контейнерных средах, где группы процессов ограничены контрольными группами и где вы можете применять одинаковые политики к ним всем, не идентифицируя каждую по отдельности. Cillium (github.com/cilium/cilium) — популярный проект с открытым исходным кодом, который предоставляет возможности Kubernetes для балансировки нагрузки и обеспечения безопасности, — широко использует программы сокетов контрольных групп для применения своих политик в группах, а не в изолированных контейнерах.

Программы Cgroup Open Socket

Программы этого типа позволяют вам выполнять код, когда какой-либо процесс в контрольной группе открывает сетевой сокет. Такое поведение подобно поведению программ, подключенных к буферам сокетов контрольных групп, но вместо того, чтобы предоставлять доступ к пакетам, когда они поступают через сеть, они позволяют вам контролировать то, что происходит, когда процесс открывает новый сокет. Они определены как тип BPF_PROG_TYPE_CGROUP_SOCK. Это полезно для обеспечения безопасности и контроля доступа к группам программ, которые могут открывать сокеты, не ограничивая возможности каждого процесса в отдельности.

Дополнительные программы для сокетов

Программы этого типа позволяют изменять параметры подключения к сокету во время выполнения, пока пакет проходит через несколько этапов в сетевом стеке ядра. Они привязаны к контрольным группам, во многом как BPF_PROG_TYPE_CGROUP_SOCK и BPF_PROG_TYPE_CGROUP_SKB, но, в отличие от них, могут вызываться несколько раз в течение жизненного цикла соединения. Эти программы определены как тип BPF_PROG_TYPE_SOCK_OPS.

Когда вы создаете такую программу BPF, вызов функции получает аргумент op, представляющий операцию, которую ядро собирается выполнить

с соединением в сокете, следовательно, вы знаете, в какой момент программа вызывается в жизненном цикле соединения. Имея эту информацию, можно получить доступ к данным, например к сетевым IP-адресам и портам подключения, а также изменить параметры подключения, чтобы установить тайм-ауты и задать другое время задержки приема-передачи для данного пакета.

Например, Facebook использует такую методику, чтобы установить краткие сроки восстановления (RTO) для соединений в одном и том же центре данных. RTO — это время восстановления системы или сетевого соединения после сбоя. Оно также может дать представление о том, как долго система может быть недоступна, прежде чем все станет окончательно не таким, каким должно быть. В Facebook предполагается, что машины в одном центре обработки данных должны иметь короткое RTO, и Facebook изменяет этот порог с помощью программы BPF.

Программы карт в сокете

Программы `BPF_PROG_TYPE_SK_SKB` предоставляют вам доступ к картам сокетов и перенаправлениям сокетов. В следующей главе вы узнаете, как карты сокетов позволяют хранить ссылки на несколько сокетов. Имея эти ссылки, вы можете задействовать специальные помощники для перенаправления входящего пакета из одного сокета в другой. Это полезно, если вы хотите реализовать возможности балансировки нагрузки с помощью BPF. Отслеживая несколько сокетов, можно пересылать сетевые пакеты между ними, не покидая пространства ядра. Такие проекты, как Cillium и Facebook Katran (oreil.ly/wDtfR), широко используют программы данного типа для управления сетевым трафиком.

Программы для устройств контрольных групп

Программы данного типа позволяют решить, может ли операция в пределах контрольной группы быть выполнена для данного устройства. Эти программы имеют тип `BPF_PROG_TYPE_CGROUP_DEVICE`. Первая реализация cgroups (v1) имеет механизм, который позволяет вам устанавливать разрешения для определенных устройств, у второй версии контрольной группы такой возможности нет. Этот тип программ был разработан для обеспечения безопасности. В то же время возможность написать программу BPF дает вам больше свободы действий для установки таких разрешений, когда они нужны.

Программы доставки сообщений через сокет

Программы таких типов позволяют вам контролировать, должно ли доставляться сообщение, отправленное в сокет. Они относятся к `BPF_PROG_TYPE_SK_MSG`. Создав сокет, ядро сохраняет его в карте сокетов. Она обеспечивает ядру быстрый доступ к определенным группам сокетов. Когда вы с помощью сокетной программы сообщений BPF обращаетесь к карте сокетов, все сообщения, отправленные в них, отфильтровываются программой перед их доставкой. Перед фильтрацией сообщений ядро копирует данные, содержащиеся в сообщении, чтобы вы могли прочитать их и решить, что с ними делать. Эти программы имеют два возможных возвращаемых значения: `SK_PASS` и `SK_DROP`. Первое используется, если вы хотите, чтобы ядро отправило сообщение в сокет, а второе — если ядро игнорирует сообщение и не доставляет его в сокет, а просто отбрасывает.

Программы для доступа к необработанным точкам трассировки

Ранее мы говорили о программе, которая обращается к точкам трассировки в ядре. Разработчики ядра добавили новую программу трассировки, чтобы удовлетворить потребность в доступе к аргументам трассировки в необработанном формате, хранящемся в ядре. Этот формат дает вам доступ к более подробной информации о задаче, которую выполняет ядро, однако в таком случае произойдет небольшой спад производительности. В большинстве случаев вы захотите применять в своих программах обычные точки трассировки, чтобы избежать снижения производительности, но помните, что также можете получить доступ к необработанным аргументам, когда это необходимо, используя необработанные точки трассировки. Такие программы имеют тип `BPF_PROG_TYPE_RAW_TRACEPOINT`.

Адресные программы сокетов контрольных групп

С помощью этих программ вы сможете манипулировать IP-адресами и номерами портов, к которым подключаются программы пользовательского пространства, когда ими управляют определенные контрольные группы. Бывают случаи, когда система использует несколько IP-адресов, если нужно, чтобы определенный набор программ пользовательского пространства задействовал один и тот же IP-адрес и порт (это прокси). Эти BPF-

программы позволяют манипулировать привязками, помещая пользовательские программы в одну и ту же группу. При этом все входящие и исходящие соединения данных приложений используют IP-адрес и порт, которые предоставляет программа BPF. Такие программы определены с типом `BPF_PROG_TYPE_CGROUP_SOCK_ADDR`.

Сокетные программы повторного использования портов

`SO_REUSEPORT` — это опция ядра, которая позволяет нескольким процессам на одном хосте применять один и тот же порт. Этот параметр дает возможность повысить производительность сетевой работы, когда требуется распределить нагрузку между несколькими потоками.

Программы такого типа, как `BPF_PROG_TYPE_SK_REUSEPORT`, позволяют указать, будет ли ваша программа BPF повторно использовать порт. Вы можете запретить программам повторно задействовать один и тот же порт, если ваша BPF-программа возвращает `SK_DROP`, а также можете сообщить ядру, чтобы оно следовало собственной процедуре повторного применения, когда вы возвращаете `SK_PASS` в результате выполнения BPF-программ.

Программы разделения потока

Анализатор потока — это компонент ядра, который отслеживает различные уровни, через которые должен пройти сетевой пакет с момента поступления в вашу систему до доставки в программу пользовательского пространства. Это позволяет контролировать поток пакетов с помощью различных методов классификации. Встроенный в ядро диссектор называется *классификатором Flower* и используется брандмауэрами и другими фильтрующими устройствами для определения того, что делать с конкретными пакетами.

Программы `BPF_PROG_TYPE_FLOW_DISSECTOR` предназначены для перехвата по пути анализатора потока. Они дают гарантии безопасности, которые не может обеспечить встроенный диссектор, например гарантируют завершение программы, что не обязательно происходит во встроенном диссекторе. Эти BPF-программы могут изменять поток, по которому в ядре идут сетевые пакеты.

Другие программы BPF

Мы говорили о типах программ, которые используются в разных средах, но стоит отметить, что есть еще несколько типов программ BPF, которые мы не рассмотрели. Здесь лишь упомянем их.

- *Программы классификатора трафика*. `BPF_PROG_TYPE_SCHED_CLS` и `BPF_PROG_TYPE_SCHED_ACT` — два типа программ BPF, которые позволяют классифицировать сетевой трафик и изменять некоторые свойства пакетов в буфере сокета.
- *Упрощенные туннельные программы*. Программы BPF типов `BPF_PROG_TYPE_LWT_IN`, `BPF_PROG_TYPE_LWT_OUT`, `BPF_PROG_TYPE_LWT_XMIT` и `BPF_PROG_TYPE_LWT_SEG6LOCAL` позволяют вам связывать код с упрощенной туннельной инфраструктурой ядра.
- *Программы инфракрасных устройств*. Программы `BPF_PROG_TYPE_LIRC_MODE2` позволяют использовать программы BPF в соединениях с инфракрасными устройствами, такими как удаленные контроллеры, просто для развлечения.

Все это специализированные программы, члены сообщества редко применяют их.

Далее поговорим о том, как BPF гарантирует, что ваши программы не вызовут катастрофического сбоя в системе после того, как ядро их загрузит. Это важная тема, ведь понимание того, как загружается программа, влияет на правильность ее написания.

Верификатор BPF

Позволять всем исполнять произвольный код внутри ядра Linux поначалу кажется ужасной идеей. Риск запуска программ BPF в производственных системах был бы слишком высоким, если бы не было верификатора BPF. Верификатор BPF — это тоже программа, работающая в вашей системе, и ее следует тщательно изучить, чтобы убедиться, что она хорошо выполняет свою работу. В последние годы исследователи безопасности обнаружили в верификаторе определенные уязвимости, которые позволяли злоумышленнику получать доступ к памяти в ядре, даже будучи непривилегированным пользователем. Вы можете подробнее прочитать об уязвимостях, подобных этой, в каталоге Common Vulnerabilities and Exposures

(CVE) — списке известных тем безопасности, курируемом Министерством внутренней безопасности США. Например, CVE-2017-16995 описывает, как любой пользователь может читать из памяти ядра и записывать в нее, обходя верификатор BPF.

В этом разделе мы расскажем о мерах, предпринимаемых верификатором для предотвращения подобных проблем.

Первая проверка, выполняемая верификатором, — это статический анализ кода, который собирается загрузить виртуальная машина. Целью проверки является обеспечение ожидаемого завершения программы. Для этого верификатор создает прямой ациклический граф (DAG) с кодом. Каждая инструкция, которую проверяет верификатор, становится узлом графа, и каждый узел связан со следующей инструкцией. Сгенерировав этот граф, верификатор выполнит первый глубокий поиск (DFS), чтобы убедиться, что программа завершает работу и код не содержит опасных путей. Это означает, что он будет проходить каждую ветвь графа до конца, чтобы гарантировать отсутствие рекурсивных циклов.

Вот условия, при которых верификатор может отклонить ваш код в ходе данной проверки.

- В программе нет контрольных циклов. Чтобы гарантировать, что программа не войдет в бесконечный цикл, верификатор отклоняет цикл управления любого вида. Были предложения разрешить петли в программах BPF, но на момент написания этой статьи ни одно из них не было принято.
- Программа не пытается выполнить больше инструкций, чем максимально допустимо ядром. Сейчас максимальное количество выполняемых инструкций — 4096. Это ограничение должно предотвратить вечную работу BPF. В главе 3 мы обсудим, как безопасно использовать вложенные программы BPF, чтобы обойти его.
- Программа не содержит недоступных инструкций, таких как условия или функции, которые никогда не выполняются. Это предотвращает загрузку мертвого кода в ВМ и позволяет избежать задержки завершения программы BPF.
- Программа не пытается выйти за свои границы.

Вторая проверка, которую выполняет верификатор, — пробный запуск программы BPF. Это означает, что верификатор будет пытаться про-

анализировать каждую инструкцию, выполняемую программой, чтобы убедиться, что среди них нет недопустимых. При этом также проверяется, что все указатели памяти доступны и разыменованы правильно. Наконец, пробный прогон информирует верификатор о потоках управления в программе, чтобы гарантировать: независимо от того, какой путь управления выберет программа, она попадет в инструкцию `BPF_EXIT`. Верификатор отслеживает все пути, пройденные по ветвям в стеке, чтобы убедиться: при выборе нового пути программа не будет проходить его более одного раза. По окончании обеих проверок верификатор считает программу безопасной для выполнения.

Системный вызов `bpf` позволяет отлаживать проверки верификатора, если вам интересно посмотреть, как проходит анализ программ. Загружая программу с помощью следующего системного вызова, вы можете установить несколько атрибутов, которые позволят верификатору распечатать журнал своих действий:

```
union bpf_attr attr = {
  .prog_type = type,
  .insns     = ptr_to_u64(insns),
  .insn_cnt  = insn_cnt,
  .license   = ptr_to_u64(license),
  .log_buf   = ptr_to_u64(bpf_log_buf),
  .log_size  = LOG_BUF_SIZE,
  .log_level = 1,
};

bpf(BPF_PROG_LOAD, &attr, sizeof(attr));
```

Поле `log_level` сообщает верификатору, печатать ли какой-либо журнал. Журнал будет выведен, если вы установите значение поля равным `1`, и ничего не будет выводиться, если значение поля — `0`. Если вы хотите распечатать журнал верификатора, необходимо также указать буфер журнала и его размер. Этот буфер представляет собой несколько строк, которые вы можете распечатать, чтобы проверить решения, принятые верификатором.

Верификатор BPF играет важную роль в обеспечении безопасности и доступности вашей системы, пока вы запускаете какие-либо программы в ядре, хотя иногда может быть трудно понять, почему он принимает какие-то решения. Не отчаивайтесь, если столкнетесь с проблемами верификации, пытаясь загрузить свои программы. Далее в этой книге мы рассмотрим примеры, которые помогут вам понять, как писать безопасные программы.

В следующем разделе говорится, как BPF структурирует в памяти информацию программы. Зная, как структурирована программа, проще выяснить, как получить доступ к внутренним компонентам BPF, а также разобраться в отладке и понять поведение программ.

Формат типа BPF

Формат типа BPF (BTF) представляет собой набор структур метаданных, которые расширяют информацию для отладки программ, карт и функций BPF. BTF включает в себя сведения об источниках, поэтому такие инструменты, как BPFTool, о котором мы поговорим в главе 5, могут обеспечить богатую интерпретацию данных BPF. Эти метаданные хранятся в бинарной программе, в специальном разделе метаданных `.BFT`. Информация BTF помогает облегчить отладку программ, но значительно увеличивает размер двоичных файлов, потому что содержит информацию о типах всего, что объявлено в вашей программе. Верификатор BPF также использует эту информацию, чтобы убедиться, что типы структуры правильно определены вашей программой.

BTF применяется исключительно для аннотирования типов C. Компиляторы BPF, такие как LLVM, знают, как включить эту информацию, поэтому вам не нужно заниматься сложной задачей ее добавления в каждую структуру. Однако в некоторых случаях инструментальным средствам все еще нужны аннотации для расширения ваших программ. В последующих главах мы опишем, как эти аннотации вступают в игру и как инструменты наподобие BPFTool отображают данную информацию.

Оконечные вызовы BPF

Программы BPF могут вызывать другие программы BPF с помощью *оконечных вызовов*. Это мощная функция, поскольку она позволяет создавать более сложные программы, комбинируя меньшие функции BPF. Версии ядра до 5.2 имеют жесткое ограничение количества машинных инструкций, которые может генерировать программа BPF. Был установлен лимит 4096, чтобы гарантировать, что программы будут завершаться в течение разумного времени. Но поскольку люди создавали все более сложные программы BPF, им нужен

был способ увеличить количество команд, ограниченное ядром, и именно здесь вступили в игру оконечные вызовы. Начиная с версии ядра 5.2, лимит инструкций увеличился до 1 млн. Вложенность оконечных вызовов в этом случае ограничена 32. Это означает, что вы можете объединить в цепочку до 32 программ, чтобы создать более сложное решение.

Когда вы вызываете одну программу BPF из другой, ядро полностью сбрасывает контекст программы. Важно помнить об этом, потому что вам, вероятно, понадобится обеспечить обмен информацией между этими программами. Объект контекста, который каждая программа BPF получает в качестве аргумента, не поможет нам решить проблему обмена данными. В следующей главе мы поговорим о картах BPF как о средстве обмена информацией между программами. Там же покажем вам, как использовать оконечные вызовы для перехода от одной BPF-программы к другой.

Резюме

В этой главе мы рассмотрели несколько примеров кода, чтобы понять, как работают программы BPF. Мы также описали все типы программ, которые вы можете создать с помощью BPF. Не беспокойтесь, если некоторые из представленных здесь концепций не совсем понятны, — в дальнейшем будут приведены дополнительные примеры. Мы также перечислили важные шаги проверки, которые BPF предпринимает для обеспечения безопасности ваших программ.

В следующей главе еще немного углубимся в написание программ и приведем больше примеров. А также поговорим о том, как программы BPF общаются со своими визави в пространстве пользователя и обмениваются информацией.

3 Карты BPF

Передача сообщений для другой программы широко используется в разработке программного обеспечения. Программа может изменять поведение другой программы, отправляя сообщения. Кроме того, это позволяет обеим программам обмениваться информацией. Одним из наиболее интересных аспектов BPF является то, что код, работающий в ядре, и программа, которая его загрузила, во время выполнения могут взаимодействовать друг с другом с помощью сообщений.

В этой главе мы рассмотрим, как программы BPF и программы в пространстве пользователя могут общаться друг с другом. Мы опишем различные каналы связи между ядром и пользовательским пространством и способы сохранения ими информации. Кроме того, покажем варианты применения этих каналов и расскажем, как сделать так, чтобы данные в них оставались постоянными при инициализации программ.

Карты BPF — это хранилища ключей/значений, которые находятся в ядре. Доступ к ним может получить любая программа BPF, знающая о них. Программы, работающие в пространстве пользователя, также могут обращаться к картам с помощью файловых дескрипторов. Вы можете хранить в карте любые данные, если заранее правильно укажете их размер. Ядро обрабатывает ключи и значения как двоичные объекты и не заботится о том, что вы храните в карте.

Верификатор BPF включает в себя несколько предохранителей для обеспечения безопасности при создании карт и получении доступа к ним. Мы должны учитывать это, если хотим иметь возможность пользоваться данными, имеющимися в картах.

Создание карт BPF

Самый простой способ создать карту BPF — использовать системный вызов `bpf`. Когда первым аргументом в вызове является `BPF_MAP_CREATE`, вы сообщаете ядру, что хотите создать новую карту. Этот вызов вернет идентификатор дескриптора файла, связанный с только что созданной картой. Второй аргумент в системном вызове — конфигурация данной карты:

```
union bpf_attr {
  struct {
    __u32 map_type;     /* одно из значений bpf_map_type */
    __u32 key_size;     /* размер ключей в байтах */
    __u32 value_size;   /* размер значений в байтах */
    __u32 max_entries;  /* максимальное количество записей в карте */
    __u32 map_flags;    /* флаги для модификации того, как создать карту */
  };
}
```

Третий аргумент в системном вызове — это размер атрибута конфигурации.

Например, вы можете создать карту хеш-таблицы для хранения целых чисел без знака в виде ключей и значений следующим образом:

```
union bpf_attr my_map {
  .map_type    = BPF_MAP_TYPE_HASH,
  .key_size    = sizeof(int),
  .value_size  = sizeof(int),
  .max_entries = 100,
  .map_flags   = BPF_F_NO_PREALLOC,
};

int fd = bpf(BPF_MAP_CREATE, &my_map, sizeof(my_map));
```

Если вызов не удался, ядро возвращает значение `-1`. Это может произойти по одной из трех причин. Если один из атрибутов недействителен, ядро устанавливает для переменной `errno` значение `EINVAL`. Если пользователь, выполняющий операцию, не имеет достаточных привилегий, ядро устанавливает для переменной `errno` значение `EPERM`. Наконец, если для хранения карты недостаточно памяти, ядро устанавливает для переменной `errno` значение `ENOMEM`.

В следующих разделах мы рассмотрим примеры более сложных операций с картами BPF. Начнем с довольно простого способа создания карты любого типа.

Соглашения ELF для создания карт BPF

Ядро включает в себя несколько соглашений и помощников для создания карт BPF и работы с ними. Вы, вероятно, посчитаете использование этих соглашений более удобным, чем выполнение системных вызовов напрямую, потому что они более читабельны и их легче придерживаться. Имейте в виду, что соглашения по-прежнему используют системный вызов `bpf` для создания карт, даже если они запускаются непосредственно в ядре. Здесь вы обнаружите, что применение системного вызова напрямую более полезно, если заранее не знаете, с каким типом карт собираетесь работать.

Вспомогательная функция `bpf_map_create` — обертка для кода, который вы только что видели, предназначенная для упрощения инициализации карт по требованию. Мы можем использовать ее для создания предыдущей карты с помощью одной строки кода:

```
int fd;
fd = bpf_create_map(BPF_MAP_TYPE_HASH, sizeof(int), sizeof(int), 100,
    BPF_F_NO_PREALOC);
```

Если вы знаете, какая карта вам понадобится в вашей программе, то можете предопределить ее. Это полезно, чтобы заранее было понятно, какие карты использует программа:

```
struct bpf_map_def SEC("maps") my_map = {
      .type        = BPF_MAP_TYPE_HASH,
      .key_size    = sizeof(int),
      .value_size  = sizeof(int),
      .max_entries = 100,
      .map_flags   = BPF_F_NO_PREALLOC,
};
```

Когда вы определяете карту таким образом, вы применяете то, что называется атрибутом раздела, — в данном случае `SEC("maps")`. Этот макрос сообщает ядру, что данная структура является картой BPF и должна быть создана соответствующим образом.

Возможно, вы заметили, что в новом примере нет идентификатора дескриптора файла, связанного с картой. В данном случае ядро использует глобальную переменную `map_data` для хранения информации о картах в вашей программе. Эта переменная является массивом структур, она упорядочена в соответствии

с тем, как вы указали каждую карту в своем коде. Например, если предыдущая карта была первой указанной в вашем коде, вы получите идентификатор дескриптора файла из первого элемента в массиве:

```
fd = map_data[0].fd;
```

Из этой структуры вы также можете получить доступ к названию карты и ее определению. Такая информация иногда бывает полезна при отладке и трассировке.

Инициализировав карту, можете начать обмен сообщениями между ядром и пользовательским пространством. Теперь посмотрим, как работать с данными, которые хранятся в этих картах.

Работа с картами BPF

Связь между ядром и пользовательским пространством станет фундаментальной частью каждой созданной вами BPF-программы. API для доступа к картам различаются в коде для ядра и коде для программы пользовательского пространства. В этом разделе представлены семантика и конкретные детали обеих реализаций.

Обновление элементов в карте BPF

После создания любой карты вы, вероятно, захотите наполнить ее информацией. Для этой цели помощники ядра предоставляют функцию `bpf_map_update_elem`. Сигнатуры функции различаются в зависимости от того, загружаете вы ее из `bpf/bpf_helpers.h` в программе, работающей в ядре, или из `tools/lib/bpf/bpf.h` в программе, действующей в пространстве пользователя. Так происходит потому, что вы можете обращаться к картам напрямую, когда работаете в ядре, но ссылаетесь на них с помощью файловых дескрипторов, когда работаете в пространстве пользователя. Поведение функции тоже немного различается: код, работающий в ядре, может напрямую обращаться к карте в памяти и атомарно обновлять элементы на месте. Однако код, действующий в пользовательском пространстве, должен отправить сообщение ядру, которое копирует предоставленное значение перед обновлением карты, что делает операцию обновления неатомарной.

Эта функция возвращает `0`, когда операция завершается успешно, и отрицательное значение — в случае неудачи. При сбое в глобальную переменную `errno` записывается код сбоя. Позже в этой главе мы перечислим случаи сбоев и дадим их описание.

Функция `bpf_map_update_elem` в ядре принимает четыре аргумента. Первый — указатель на карту, которую мы уже определили. Второй — указатель на ключ, который хотим обновить. Поскольку ядро не знает тип обновляемого ключа, этот метод определяется как непрозрачный указатель типа `void`, что означает: можно передавать любые данные. Третий аргумент — значение, которое мы хотим установить. У этого аргумента та же семантика, что и у ключевого аргумента. В книге мы приведем несколько усложненных примеров применения непрозрачных указателей. В этой функции можете использовать четвертый аргумент, чтобы изменить способ обновления карты. Он может принимать три значения:

- передав `0`, вы сообщите ядру, что хотите обновить элемент, если он существует, или что оно должно создать элемент в карте, если его не существует;
- передав `1`, вы скажете ядру, что нужно создать элемент только тогда, когда его не существует;
- если вы передадите `2`, ядро обновит элемент, только если он существует.

Эти значения определены как константы, которые вы можете использовать вместо того, чтобы запоминать семантику целых чисел. Значения: `BPF_ANY` для `0`, `BPF_NOEXIST` для `1` и `BPF_EXIST` для `2`.

Воспользуемся картой, которую мы определили в предыдущем разделе, чтобы написать несколько примеров. В первом примере добавим в карту новое значение. Поскольку она пуста, можно предположить, что любое поведение при обновлении приведет к успеху:

```
int key, value, result;
key = 1, value = 1234;
result = bpf_map_update_elem(&my_map, &key, &value, BPF_ANY);
if (result == 0)
  printf("Map updated with new element\n");
else
  printf("Failed to update map with new value: %d (%s)\n",
      result, strerror(errno));
```

В этом примере мы используем strerror для описания ошибки, установленной в переменной errno. Больше об этой функции можете узнать на страницах справки, введя man strerror.

Теперь посмотрим, какой результат мы получим при попытке создать элемент с тем же ключом:

```
int key, value, result;
key = 1, value = 5678;

result = bpf_map_update_elem(&my_map, &key, &value, BPF_NOEXIST);
if (result == 0)
  printf("Map updated with new element\n");
else
  printf("Failed to update map with new value: %d (%s)\n",
      result, strerror(errno));
```

Поскольку мы уже создали в карте элемент с ключом 1, результатом вызова bpf_map_update_elem станет -1, а значение errno будет равно EEXIST. Эта программа выведет на экран следующее:

```
Failed to update map with new value: -1 (File exists)
```

А теперь изменим программу, чтобы попытаться обновить элемент, которого еще не существует:

```
int key, value, result;
key = 1234, value = 5678;

result = bpf_map_update_elem(&my_map, &key, &value, BPF_EXIST);
if (result == 0)
  printf("Map updated with new element\n");
else
  printf("Failed to update map with new value: %d (%s)\n",
      result, strerror(errno));
```

При использовании флага BPF_EXIST результат этой операции снова будет равен -1. Ядро установит для переменной errno значение ENOENT, и программа выведет следующее:

```
Failed to update map with new value: -1 (No such file or directory)
```

Эти примеры показывают, как можно обновлять карты в программе ядра. Вы также можете обновить карты из программ пользовательского пространства. Помощники для этого похожи на те, которые мы только что

видели, единственное отличие состоит в том, что для доступа к карте они используют файловый дескриптор, а не указатель на карту напрямую. Как вы помните, программы пользовательского пространства всегда получают доступ к картам с помощью файловых дескрипторов. Поэтому здесь в примерах аргумент `my_map` заменен глобальным идентификатором файлового дескриптора `map_data[0].fd`. Вот как выглядит оригинальный код в этом случае:

```
int key, value, result;
key = 1, value = 1234;

result = bpf_map_update_elem(map_data[0].fd, &key, &value, BPF_ANY);
if (result == 0)
  printf("Map updated with new element\n");
else
  printf("Failed to update map with new value: %d (%s)\n",
      result, strerror(errno));
```

Хотя тип информации, которую вы можете хранить в карте, напрямую связан с типом карты, с которой вы работаете, метод заполнения информации останется таким же, как в предыдущем примере. Позже мы обсудим типы ключей и значений, принятых для каждого типа карты. А сначала посмотрим, как манипулировать сохраненными данными.

Считывание элементов из карты BPF

Теперь, когда карта наполнена новыми элементами, мы можем читать их из других точек программы. API чтения станет понятен после изучения `bpf_map_update_element`.

Для чтения с карты BPF предоставляет два различных помощника, в зависимости от того, где работает код. Оба они называются `bpf_map_lookup_elem`. Как и помощники по обновлению, они различаются своим первым аргументом: метод ядра получает ссылку на карту, тогда как помощник пользовательского пространства принимает идентификатор дескриптора файла карты в качестве первого аргумента. Оба метода возвращают целое число, говорящее о том, была операция успешной или нет. Третий аргумент в этих помощниках — указатель на переменную в вашем коде, которая будет хранить значение,

считанное с карты. Приведем два примера на основе кода, который вы видели в предыдущем разделе.

В первом примере считывается значение, записанное в карту, когда программа BPF работает в ядре:

```
int key, value, result; // будет хранить значение ожидаемого элемента
key = 1;

result = bpf_map_lookup_elem(&my_map, &key, &value);
if (result == 0)
  printf("Value read from the map: '%d'\n", value);
else
  printf("Failed to read value from the map: %d (%s)\n",
      result, strerror(errno));
```

Если бы ключ `bpf_map_lookup_elem`, который мы пытались прочитать, возвратил отрицательное число, он установил бы ошибку в переменной `errno`. Например, если бы мы не вставили значение до того, как пытались его прочитать, ядро вернуло бы ошибку *not found* — `ENOENT`.

Второй пример похож на тот, который вы только что видели, но на этот раз мы читаем карту из программы, работающей в пространстве пользователя:

```
int key, value, result; // будет хранить значение ожидаемого элемента
key = 1;

result = bpf_map_lookup_elem(map_data[0].fd, &key, &value);
if (result == 0)
  printf("Value read from the map: '%d'\n", value);
else
  printf("Failed to read value from the map: %d (%s)\n",
      result, strerror(errno));
```

Как видите, мы заменили первый аргумент в `bpf_map_lookup_elem` идентификатором файлового дескриптора на карте. Поведение помощника такое же, как и в предыдущем примере.

Это все, что нам нужно, чтобы информация в карте BPF стала доступной. В последующих главах рассмотрим, как можно добиться этого гораздо проще, используя различные инструменты. А сейчас поговорим об удалении данных из карт.

Удаление элемента из карты BPF

Третья операция — удаление элементов из карты. Так же, как и для записи и считывания элементов, BPF для удаления элементов предоставляет два различных помощника, которые называются `bpf_map_delete_element`. Как и в предыдущих примерах, они используют прямую ссылку на карту, когда вы задействуете их в программе, работающей в ядре, и дескриптор файлового идентификатора карты, когда применяете их в программе, работающей в пользовательском пространстве.

В первом примере удаляется значение, записанное в карту, когда программа BPF работает в ядре:

```
int key, result;
key = 1;

result = bpf_map_delete_element(&my_map, &key);
if (result == 0)
  printf("Element deleted from the map\n");
else
  printf("Failed to delete element from the map: %d (%s)\n",
      result, strerror(errno));
```

Если элемента, который вы пытаетесь удалить, не существует, ядро возвращает отрицательное число. В этом случае оно также заполняет переменную `errno` ошибкой *not found* — `ENOENT`.

В этом примере удаляется значение, если программа BPF выполняется в пространстве пользователя:

```
int key, result;
key = 1;

result = bpf_map_delete_element(map_data[0].fd, &key);
if (result == 0)
  printf("Element deleted from the map\n");
else
  printf("Failed to delete element from the map: %d (%s)\n",
      result, strerror(errno));
```

Как видите, мы снова изменили первый аргумент, чтобы использовать идентификатор дескриптора файла. Это подскажет помощнику ядра, как действовать.

Собственно, это все, что можно считать операциями создания, чтения, обновления данных в карте BPF, а также их удаления из нее (CRUD). Ядро

предоставляет некоторые дополнительные функции, чтобы помочь вам с другими общими операциями (о части из них мы поговорим в следующих двух разделах).

Перебор элементов в карте BPF

Последняя операция, которую мы рассмотрим в этом разделе, нужна для того, чтобы найти произвольные элементы в программе BPF. Бывает, что вы не знаете точно ключ для элемента, который ищете, или просто хотите посмотреть, что находится внутри карты. BPF предоставляет для этого инструкцию `bpf_map_get_next_key`. В отличие от помощников, которые вы видели до сих пор, эта инструкция доступна только для программ, работающих в пространстве пользователя.

Этот помощник обеспечивает детерминированный способ перебора элементов в карте, но его поведение менее интуитивно понятно, чем поведение итераторов в большинстве языков программирования. Требуется три аргумента. Первый — это идентификатор файлового дескриптора карты, как и другие помощники пользовательского пространства, которые вы уже видели. Следующие два аргумента... ну, здесь все сложнее. Согласно официальной документации второй аргумент `key` — это идентификатор, который вы ищете, а третий, `next_key`, — следующий ключ в карте. Мы предпочитаем вызывать первый аргумент `lookup_key` — почему, станет ясно буквально через секунду. Когда вы вызываете данный помощник, BPF пытается найти в этой карте элемент с ключом, который вы передаете в качестве ключа поиска, затем устанавливает аргумент `next_key` со смежным ключом в карте. Поэтому, если вы хотите узнать, какой ключ следует за ключом `1`, установите `1` в качестве ключа поиска, и, если у карты есть смежный с ним ключ, BPF задаст его в качестве значения для аргумента `next_key`.

Прежде чем смотреть, как `bpf_map_get_next_key` работает на реальном примере, добавим еще несколько элементов в нашу карту:

```
int new_key, new_value, it;

for (it = 2; it < 6 ; it++) {
  new_key = it;
  new_value = 1234 + it;
  bpf_map_update_elem(map_data[0].fd, &new_key, &new_value, BPF_NOEXIST);
}
```

Если вы хотите распечатать все значения, сохраненные в карте, можете использовать `bpf_map_get_next_key` с ключом поиска, которого в ней нет. Это заставляет BPF начать с начала карты:

```
int next_key, lookup_key;
lookup_key = -1;

while(bpf_map_get_next_key(map_data[0].fd, &lookup_key, &next_key) == 0) {
  printf("The next key in the map is: '%d'\n", next_key);
  lookup_key = next_key;
}
```

Данный код выведет что-то вроде следующего:

```
The next key in the map is: '1'
The next key in the map is: '2'
The next key in the map is: '3'
The next key in the map is: '4'
The next key in the map is: '5'
```

Вы видите, что мы назначаем следующий ключ для `lookup_key` в конце цикла, таким образом, мы продолжаем перебирать карту, пока не дойдем до конца. Когда `bpf_map_get_next_key` достигает конца карты, возвращаемое значение становится отрицательным числом, а переменная `errno` устанавливается в `ENOENT`. Это прервет выполнение цикла.

Можно увидеть, что `bpf_map_get_next_key` ищет ключи, начиная с любой позиции в карте. Вам не нужно стартовать с начала карты, если требуется только следующий ключ для другого конкретного ключа.

Хитрости `bpf_map_get_next_key` на этом не заканчиваются — есть еще кое-что, о чем вам нужно знать. Многие языки программирования копируют значения, имеющиеся в карте, прежде чем выполнять итерации по ее элементам. Это предотвращает непредсказуемое поведение, если какой-то другой код в вашей программе захочет изменить карту. Это особенно опасно, если код удаляет элементы из карты. BPF не копирует значения в карте перед прохождением по ним с помощью `bpf_map_get_next_key`. Если другая часть вашей программы удалит элемент из карты, пока вы проходите по значениям, `bpf_map_get_next_key` запустится заново, когда попытается найти следующее значение для ключа элемента, который был удален. Рассмотрим это на примере:

```
int next_key, lookup_key;
lookup_key = -1;
```

```
while(bpf_map_get_next_key(map_data[0].fd, &lookup_key, &next_key) == 0) {
  printf("The next key in the map is: '%d'\n", next_key);
  if (next_key == 2) {
    printf("Deleting key '2'\n");
    bpf_map_delete_element(map_data[0].fd &next_key);
  }
  lookup_key = next_key;
}
```

Эта программа напечатает следующее:

```
The next key in the map is: '1'
The next key in the map is: '2'
Deleteing key '2'
The next key in the map is: '1'
The next key in the map is: '3'
The next key in the map is: '4'
The next key in the map is: '5'
```

Такое поведение не очень интуитивно понятно, имейте это в виду, когда используете `bpf_map_get_next_key`.

Поскольку большинство типов карт, которые мы рассматриваем в этой главе, ведут себя как массивы, итерирование по ним будет ключевой операцией, если вы захотите получить доступ к хранящейся в них информации. Однако есть дополнительные функции для доступа к данным. Об этом поговорим чуть позже.

Поиск и удаление элементов

Еще одна интересная функция, которую ядро предоставляет для работы с картами, — `bpf_map_lookup_and_delete_elem`. Она ищет конкретный ключ в карте, удаляет элемент из нее и в то же время записывает значение элемента в переменную для использования вашей программой. Эта функция удобна, когда вы задействуете карты очередей и стеков, о которых мы поговорим в следующем разделе. Однако ее применение не ограничено типами карт. Рассмотрим, как использовать ее с картой, с которой мы работали в предыдущих примерах:

```
int key, value, result, it;
key = 1;

for (it = 0; it < 2; it++) {
  result = bpf_map_lookup_and_delete_element(map_data[0].fd, &key, &value);
```

```
  if (result == 0)
    printf("Value read from the map: '%d'\n", value);
  else
    printf("Failed to read value from the map: %d (%s)\n",
        result, strerror(errno));
}
```

В этом примере мы пытаемся извлечь один и тот же элемент из карты дважды. На первой итерации код будет печатать значение элемента в карте. Однако, поскольку мы используем `bpf_map_lookup_and_delete_element`, эта итерация также удалит элемент из карты. Во второй раз, когда цикл попытается извлечь элемент, код завершится ошибкой и заполнит переменную `errno` ошибкой *not found* — `ENOENT`.

Есть такое понятие — конкуренция. До сих пор мы не обращали особого внимания на то, что происходит, когда параллельные операции пытаются получить доступ к одной и той же части информации в карте BPF. Давайте поговорим об этом.

Конкурентный доступ к элементам карты

Одна из проблем работы с картами BPF заключается в том, что многие программы могут одновременно обращаться к одним и тем же картам. Это может привести к гонкам в сфере доступа программ BPF и сделать доступ к ресурсам в картах непредсказуемым. Чтобы предотвратить условия гонки, в BPF была введена концепция спин-блокировки BPF, позволяющей вам блокировать доступ к элементу карты во время работы с ней. Спин-блокировки работают только в картах хранения массивов, хешей и контрольных групп.

Есть две вспомогательные функции BPF для работы со спин-блокировками: `bpf_spin_lock` блокирует элемент, а `bpf_spin_unlock` разблокирует. Эти помощники работают со структурой, которая действует как семафор, чтобы получить доступ к элементу, включающему этот семафор. Когда семафор закрыт, другие программы не могут получить доступ к значению элемента и ждут, пока он будет открыт. В то же время спин-блокировки BPF вводят новый флаг, который программы пользовательского пространства могут использовать для изменения состояния блокировки. Флаг называется `BPF_F_LOCK`.

Первое, что нам нужно сделать для работы со спин-блокировками, — создать элемент, доступ к которому мы хотим заблокировать, а затем добавить семафор:

```
struct concurrent_element {
  struct bpf_spin_lock semaphore;
  int count;
}
```

Мы будем хранить эту структуру в нашей карте BPF и использовать семафор внутри элемента, чтобы предотвратить нежелательный доступ к нему. Теперь мы можем объявить карту, которая будет содержать эти элементы. Она должна быть аннотирована с помощью BPF Type Format (BTF), чтобы верификатор знал, как интерпретировать структуру. Формат типа позволяет ядру и другим инструментам глубже понять структуры данных BPF, добавляя отладочную информацию в двоичные объекты. Поскольку этот код будет работать внутри ядра, мы можем задействовать макросы ядра, которые предоставляет `libbpf`, для аннотирования карты с конкурентным доступом:

```
struct bpf_map_def SEC("maps") concurrent_map = {
      .type        = BPF_MAP_TYPE_HASH,
      .key_size    = sizeof(int),
      .value_size  = sizeof(struct concurrent_element),
      .max_entries = 100,
};

BPF_ANNOTATE_KV_PAIR(concurrent_map, int, struct concurrent_element);
```

В рамках программы BPF мы можем использовать два помощника блокировки для защиты от состояния гонки. Несмотря на то что семафор закрыт, наша программа гарантированно сможет безопасно изменять значение элемента (позже):

```
int bpf_program(struct pt_regs *ctx) {
  int key = 0;
  struct concurrent_element init_value = {};
  struct concurrent_element *read_value;

  bpf_map_create_elem(&concurrent_map, &key, &init_value, BPF_NOEXIST);

  read_value = bpf_map_lookup_elem(&concurrent_map, &key);
  bpf_spin_lock(&read_value->semaphore);
  read_value->count += 100;
  bpf_spin_unlock(&read_value->semaphore);
}
```

В этом примере запись в нашу карту конкурентна, здесь может блокироваться доступ к ее значению. Затем помощник выбирает это значение из карты и блокирует его семафор, чтобы он мог хранить значение счетчика, предотвращая изменения данных. Когда это сделано и есть определенное значение, блокировка снимается, чтобы другие карты могли безопасно получить доступ к элементу.

Из пользовательского пространства мы можем сохранить ссылку на элемент в параллельной карте, используя флаг `BPF_F_LOCK`. Вы можете применить этот флаг с помощью вспомогательных функций `bpf_map_update_elem` и `bpf_map_lookup_elem_flags`. Он позволяет обновлять элементы на месте, не беспокоясь о гонках данных.

BPF_F_LOCK немного по-разному ведет себя при обновлении хеш-карты и обновлении массивов и карт хранения контрольных групп. Последние два обновления происходят на месте, и элементы, которые вы обновляете, должны существовать в карте перед выполнением этой процедуры. В случае хеш-карт, если элемент еще не существует, программа блокирует область в карте и вставляет в нее новый элемент.

Спин-блокировки требуются не всегда. Например, они не нужны, если вы всего лишь агрегируете значения в карте. Однако они полезны, если вы хотите, чтобы взаимодействующие программы не изменяли элементы в карте, когда вы выполняете над ними несколько операций, сохраняя атомарность.

В этом разделе мы рассмотрели операции, которые можно выполнять с картами BPF. Однако до сих пор мы работали только с одним типом карты. Но в есть много других типов карт, которые используются в различных ситуациях. Мы рассмотрим все типы карт, которые определяет BPF, и приведем конкретные примеры того, как их применять.

Типы карт BPF

Документация Linux (https://oreil.ly/XfoqK) определяет карты как общие структуры данных, где можно хранить различные типы данных. За прошедшие годы разработчики ядра добавили много специализированных структур данных, более эффективных в конкретных случаях. В этом разделе рассматриваются все типы карт и способы их использования.

Карты хеш-таблиц

Карты хеш-таблиц были первыми картами, добавленными в BPF. Они определены с типом `BPF_MAP_TYPE_HASH`. Реализуются и применяются они аналогично другим хеш-таблицам, с которыми вы, возможно, знакомы. Вы можете брать ключи и значения любого размера, а ядро позаботится о том, как распределять и освобождать их для вас по мере необходимости. Когда вы используете `bpf_map_update_elem` в карте хеш-таблицы, ядро заменяет элементы атомарно.

Карты хеш-таблиц оптимизированы для быстрого поиска. Лучше всего хранить в них структурированные данные, к которым обращаются наиболее часто. Рассмотрим пример программы, которая использует их для отслеживания сетевых IP-адресов и их ограничений по скорости:

```
#define IPV4_FAMILY 1
struct ip_key {
  union {
    __u32 v4_addr;
    __u8 v6_addr[16];
  };
__u8 family;
};

struct bpf_map_def SEC("maps") counters = {
      .type        = BPF_MAP_TYPE_HASH,
      .key_size    = sizeof(struct ip_key),
      .value_size  = sizeof(uint64_t),
      .max_entries = 100,
      .map_flags   = BPF_F_NO_PREALLOC
};
```

В этом коде мы объявили структурированный ключ и собираемся применять его для хранения информации об IP-адресах. Определяем карту, которую программа будет задействовать для отслеживания ограничений скорости. Можно видеть, что IP-адреса используются в качестве ключей в этой карте. Значения будут представлять собой количество получений BPF-программой сетевого пакета с определенного IP-адреса.

Напишем небольшой фрагмент кода, который обновляет эти счетчики в ядре:

```
uint64_t update_counter(uint32_t ipv4) {
  uint64_t value;
```

```
  struct ip_key key = {};
  key.v4_addr = ip4;
  key.family = IPV4_FAMILY;

  bpf_map_lookup_elem(counters, &key, &value);
  (*value) += 1;
}
```

Данная функция берет IP-адрес, извлеченный из сетевого пакета, и выполняет поиск карты с помощью составного ключа, который мы объявляем. В этом случае предполагаем, что ранее инициализировали счетчик с нулевым значением, в противном случае вызов `bpf_map_lookup_elem` вернет отрицательное число.

Карты массивов

Карты массива были вторым типом карт BPF, добавленных в ядро. Они определены с типом `BPF_MAP_TYPE_ARRAY`. Когда вы инициализируете карту массива, все ее элементы предварительно распределяются в памяти и устанавливаются в ноль. Поскольку эти карты поддерживаются определенным количеством элементов, ключи являются индексами в массиве и их размер должен составлять ровно 4 байта.

Недостатком использования карт массива является то, что содержащиеся в них элементы нельзя удалить и вы не можете уменьшить массив. Если попытаетесь применить `map_delete_elem` в карте массива, вызов не удастся и в результате вы получите ошибку `EINVAL`.

В картах массивов обычно хранится информация, значение которой может меняться, а поведение остается неизменным. Многие используют их для хранения глобальных переменных с предопределенным предназначением. Поскольку элементы удалять нельзя, то можно предположить, что элемент в определенной позиции — это всегда один и тот же элемент.

Следует также помнить, что `map_update_elem` не является атомарной, как вы видели в картах хеш-таблиц. Одна и та же программа может одновременно считывать из одной и той же позиции разные значения, если происходит обновление. Если вы храните счетчики в карте массива, то можете использовать встроенную в ядро функцию `__sync_fetch_and_add` для выполнения атомарных операций со значениями карты.

Карты программных массивов

Карты массивов программ были первыми специализированными картами, добавленными в ядро. Они определены с типом `BPF_MAP_TYPE_PROG_ARRAY`. Вы можете использовать такие карты для хранения ссылок на программы BPF, задействуя их идентификаторы файловых дескрипторов. Вместе с помощником `bpf_tail_call` такая карта позволяет вам переключаться между программами, чтобы обойти ограничение максимального количества команд для отдельных программ BPF и уменьшить сложность реализации.

Есть несколько условий, которые необходимо учитывать, работая с этой довольно специфичной картой. Первое, о чем нужно помнить, — то, что размеры ключа и значения должны составлять 4 байта. Второй аспект заключается в том, что при переходе к новой программе последняя будет повторно использовать тот же стек памяти, поэтому ваша программа не задействует всю доступную память. Наконец, если вы попытаетесь перейти к программе, которой нет в карте, оконечный вызов завершится неудачно и продолжит выполняться текущая программа.

Рассмотрим подробный пример, чтобы понять, как лучше применять этот тип карты:

```
struct bpf_map_def SEC("maps") programs = {
  .type = BPF_MAP_TYPE_PROG_ARRAY,
  .key_size = 4,
  .value_size = 4,
  .max_entries = 1024,
};
```

Во-первых, нам нужно объявить новую карту программы (как мы уже упоминали, размеры ключа и значения всегда равны 4 байтам):

```
int key = 1;
struct bpf_insn prog[] = {
  BPF_MOV64_IMM(BPF_REG_0, 0), // присваиваем r0 = 0
  BPF_EXIT_INSN(),  // возвращаем r0
};

prog_fd = bpf_prog_load(BPF_PROG_TYPE_KPROBE, prog, sizeof(prog), "GPL");
bpf_map_update_elem(&programs, &key, &prog_fd, BPF_ANY);
```

Следует объявить программу, на которую мы собираемся переключиться. В этом случае пишем программу BPF, единственная цель которой — вернуть `0`.

Мы используем bpf_prog_load, чтобы загрузить ее в ядро, а затем добавляем идентификатор ее файлового дескриптора в карту нашей программы.

Теперь, когда наша программа сохранена, мы можем написать другую программу BPF, к которой первая будет переходить, то есть вызывать ее из себя. BPF-программы могут переходить к другим программам, только если они одного типа, в этом случае мы присоединяем программу к трассировке kprobe (это рассматривалось в главе 2):

```
SEC("kprobe/seccomp_phase1")
int bpf_kprobe_program(struct pt_regs *ctx) {
  int key = 1;
  /* отправка в следующую программу BPF */
  bpf_tail_call(ctx, &programs, &key);

  /* попадаем сюда, когда дескриптора программы нет в карте */
  char fmt[] = "missing program in prog_array map\n";
  bpf_trace_printk(fmt, sizeof(fmt));
  return 0;
}
```

С bpf_tail_call и BPF_MAP_TYPE_PROG_ARRAY можно связать до 32 вложенных вызовов. Это строгое ограничение для предотвращения появления бесконечных циклов и исчерпания памяти.

Карты массивов событий производительности

Эти типы карт хранят данные perf_events в кольце буфера, с помощью которого программы BPF и программы пользовательского пространства обмениваются данными в режиме реального времени. Они определены с типом BPF_MAP_TYPE_PERF_EVENT_ARRAY и предназначены для пересылки событий, которые инструменты трассировки ядра передают программам пользовательского пространства для дальнейшей обработки. Это один из самых интересных типов карт и основа для многих инструментов наблюдения, о которых мы поговорим в следующих главах. Программа пользовательского пространства действует как прослушиватель, который ожидает события, поступающие от ядра, поэтому вам необходимо убедиться, что ваш код начинает прослушивание до инициализации программы BPF в ядре.

Посмотрим, как мы можем отслеживать все программы, которые выполняет наш компьютер. Прежде чем перейти к программному коду BPF, нужно объявить структуру событий, которую мы собираемся отправить из ядра в пространство пользователя:

```
struct data_t {
  u32 pid;
  char program_name[16];
};
```

Теперь нам нужно создать карту, которая будет отправлять события в пространство пользователя:

```
struct bpf_map_def SEC("maps") events = {
  .type        = BPF_MAP_TYPE_PERF_EVENT_ARRAY,
  .key_size    = sizeof(int),
  .value_size  = sizeof(u32),
  .max_entries = 2,
};
```

После создания карты и получения данных мы можем создать программу BPF, которая будет захватывать данные и отсылать в пространство пользователя:

```
SEC("kprobe/sys_exec")
int bpf_capture_exec(struct pt_regs *ctx) {
  data_t data;
  // bpf_get_current_pid_tgid возвращает идентификатор текущего процесса
  data.pid = bpf_get_current_pid_tgid() >> 32;
  // bpf_get_current_comm загружает текущее имя программы
  bpf_get_current_comm(&data.program_name, sizeof(data.program_name));
  bpf_perf_event_output(ctx, &events, 0, &data, sizeof(data));
  return 0;
}
```

В этом фрагменте мы используем `bpf_perf_event_output`, чтобы добавлять данные в карту. Поскольку это буфер реального времени, вам не нужно беспокоиться о ключах для элементов в карте — ядро позаботится о добавлении нового элемента в карту и ее очистке после того, как программа из пользовательского пространства обработает его.

В главе 4 обсуждается более продвинутое использование этих типов карт и приводятся примеры работы программ в пространстве пользователя.

Хеш-карты для каждого процессора

Карты этого типа являются улучшенной версией BPF_MAP_TYPE_HASH. Карты определены с типом BPF_MAP_TYPE_PERCPU_HASH. Когда вы определяете одну из них, каждый ЦП видит собственную изолированную ее версию, что делает ее намного более эффективной для высокопроизводительных поисков и объединения данных. Этот тип карты полезен, если ваша программа BPF собирает показатели и объединяет их в карты хеш-таблиц.

Карты массивов для каждого процессора

Такие карты также являются улучшенной версией BPF_MAP_TYPE_ARRAY. Они определены с типом BPF_MAP_TYPE_PERCPU_ARRAY. При выделении одной из этих карт (как и предыдущей) каждый ЦП видит собственную изолированную версию карты, что делает ее намного более эффективной для высокопроизводительных поисков и объединения данных.

Карты трассировки стека

Карты этого типа хранят трассировку стека от запущенного процесса. Они определены с типом BPF_MAP_TYPE_STACK_TRACE. Наряду с этой картой разработчики ядра добавили помощник bpf_get_stackid, который поможет вам заполнить карту трассировкой стека. Помощник принимает карту и ряд флагов в качестве аргументов, так что можете указать, хотите ли вы, чтобы трассировки были только из ядра, только из пользовательского пространства или из них обоих. Помощник возвращает ключ, связанный с элементом, добавленным в карту.

Карты массива контрольной группы

Карты такого типа хранят ссылки на контрольные группы. Карты массива контрольной группы определяются с типом BPF_MAP_TYPE_CGROUP_ARRAY. По сути, их поведение аналогично поведению BPF_MAP_TYPE_PROG_ARRAY, но они хранят идентификаторы файловых дескрипторов, которые указывают на контрольные группы.

Эта карта полезна, когда вы хотите поделиться ссылками контрольной группы (cgroup) между картами BPF для контроля трафика, отладки и тестирования. Рассмотрим, как заполнить эту карту. Начнем с ее определения:

```
struct bpf_map_def SEC("maps") cgroups_map = {
  .type        = BPF_MAP_TYPE_CGROUP_ARRAY,
  .key_size    = sizeof(uint32_t),
  .value_size  = sizeof(uint32_t),
  .max_entries = 1,
};
```

Можно получить дескриптор файла контрольной группы, открыв файл, содержащий информацию о ней. Мы собираемся открыть группу, которая контролирует базовые ресурсы ЦП для контейнеров Docker, и сохранить информацию о контрольной группе в нашей карте:

```
int cgroup_fd, key = 0;
cgroup_fd = open("/sys/fs/cgroup/cpu/docker/cpu.shares", O_RDONLY);

bpf_update_elem(&cgroups_map, &key, &cgroup_fd, 0);
```

Хеш-карты LRU и хеш-карты отдельных процессоров

Карты этих двух типов являются картами хеш-таблиц подобно тем, которые вы видели ранее, но они также реализуют внутренний кэш LRU. LRU очень редко используется в последнее время, что означает: если карта заполнена, она будет удалять элементы, которые применяются нечасто, чтобы освободить место для новых элементов. Поэтому вы можете использовать эти карты для вставки элементов, превышающих максимальный предел, если не возражаете против потери элементов, не используемых в последнее время. Они определяются с типами `BPF_MAP_TYPE_LRU_HASH` и `BPF_MAP_TYPE_LRU_PERCPU_HASH`.

Версия этой карты `per cpu` немного отличается от других карт для процессора, которые вы видели ранее. Она содержит только одну хеш-таблицу для хранения всех элементов и применяет разные кэши LRU для каждого ЦП. Таким образом, она гарантирует, что наиболее часто используемые элементы в каждом ЦП остаются в карте.

Карты LPM Trie

Древовидные карты LPM — это тип карт, которые используют поиск по самому длинному префиксу (LPM) для поиска элементов в карте. LPM — это алгоритм, который выбирает элемент дерева, совпадающий с самым длинным ключом поиска из любого другого соответствия в дереве. Этот алгоритм применяется в маршрутизаторах и других устройствах, которые хранят таблицы пересылки трафика для сопоставления IP-адресов с конкретными маршрутами. Карты определены с типом `BPF_MAP_TYPE_LPM_TRIE`.

Эти карты требуют, чтобы размеры их ключей были кратны восьми и находились в диапазоне от 8 до 2048. Если вы не хотите реализовывать собственный ключ, ядро предоставляет структуру, которую можно использовать для этого. Она называется `bpf_lpm_trie_key`.

В следующем примере мы добавляем два маршрута пересылки в карту и пытаемся сопоставить IP-адрес с правильным маршрутом. Сначала нам нужно создать карту:

```
struct bpf_map_def SEC("maps") routing_map = {
  .type = BPF_MAP_TYPE_LPM_TRIE,
  .key_size = 8,
  .value_size = sizeof(uint64_t),
  .max_entries = 10000,
  .map_flags = BPF_F_NO_PREALLOC,
};
```

Мы собираемся заполнить ее тремя маршрутами для передачи: 192.168.0.0/16, 192.168.0.0/24 и 192.168.1.0/24:

```
uint64_t value_1 = 1;
struct bpf_lpm_trie_key route_1 = {.data = {192, 168, 0, 0}, .prefixlen = 16};
uint64_t value_2 = 2;
struct bpf_lpm_trie_key route_2 = {.data = {192, 168, 0, 0}, .prefixlen = 24};
uint64_t value_3 = 3;
struct bpf_lpm_trie_key route_3 = {.data = {192, 168, 1, 0}, .prefixlen = 24};

bpf_map_update_elem(&routing_map, &route_1, &value_1, BPF_ANY);
bpf_map_update_elem(&routing_map, &route_2, &value_2, BPF_ANY);
bpf_map_update_elem(&routing_map, &route_3, &value_3, BPF_ANY);
```

Теперь мы используем ту же структуру ключей, чтобы найти правильное соответствие для IP 192.168.1.1/32:

```
uint64_t result;
struct bpf_lpm_trie_key lookup = {.data = {192, 168, 1, 1}, .prefixlen = 32};

int ret = bpf_map_lookup_elem(&routing_map, &lookup, &result);
if (ret == 0)
  printf("Value read from the map: '%d'\n", result);
```

В этом примере и 192.168.0.0/24, и 192.168.1.0/24 могут соответствовать IP-адресу поиска, поскольку маска 16 (см. в коде) покрывает оба диапазона. Но, поскольку эта карта использует алгоритм LPM, результатом будет значение для ключа 192.168.1.0/24.

Массив карт и хеш-карт

`BPF_MAP_TYPE_ARRAY_OF_MAPS` и `BPF_MAP_TYPE_HASH_OF_MAPS` — два типа карт, в которых хранятся ссылки на другие карты. Они поддерживают только один уровень косвенной адресации, поэтому их нельзя применять для хранения карты из карты и т. д. Это гарантирует, что вы не займете всю память, случайно сохраняя бесконечные цепочки карт.

Такие карты полезны, если вы хотите иметь возможность заменить полноценные карты во время выполнения. Можете создавать снимки полного состояния, если все ваши карты являются дочерними элементами глобальной карты. Ядро гарантирует, что любая операция обновления в родительской карте ожидает, пока все ссылки на старые дочерние карты не будут удалены перед ее завершением.

Карты устройств

Этот специализированный тип карт хранит ссылки на сетевые устройства. Карты определены с типом `BPF_MAP_TYPE_DEVMAP`. Они полезны для сетевых приложений, которые хотят манипулировать трафиком на уровне ядра. Вы можете создать виртуальную карту портов, которые указывают на определенные сетевые устройства, а затем перенаправить пакеты с помощью помощника `bpf_redirect_map`.

Карты процессоров

`BPF_MAP_TYPE_CPUMAP` — это другой тип карт, который позволяет перенаправлять сетевой трафик. В таком случае карта хранит ссылки на разные процессоры вашего хоста. Вы можете использовать этот тип карт, как и предыдущий, с помощником `bpf_redirect_map` для перенаправления пакетов. Однако данная карта отправляет пакеты на другой процессор. Это позволяет назначать конкретные процессоры сетевым стекам для обеспечения масштабируемости и изоляции.

Карты открытого сокета

`BPF_MAP_TYPE_XSKMAP` — это тип карт, в которых хранятся ссылки на открытые сокеты. Как и предыдущие карты, они могут служить для пересылки пакетов между сокетами.

Карты массива и хеша сокета

`BPF_MAP_TYPE_SOCKMAP` и `BPF_MAP_TYPE_SOCKHASH` — это две специализированные карты, в которых хранятся ссылки на открытые сокеты в ядре. Как и предыдущие карты, эти типы карт используются вместе с помощником `bpf_redirect_map` для пересылки буферов сокетов из текущей программы XDP в другой сокет.

Их основное различие состоит в том, что один из них хранит сокеты в массиве, а другой — в хеш-таблице. Преимущество хеш-таблицы заключается в том, что вы можете получить доступ к сокету напрямую по его ключу, не просматривая полную карту, чтобы найти сокет. Каждый сокет в ядре идентифицируется ключом из пяти кортежей, которые содержат информацию, необходимую для установления двунаправленных сетевых подключений. Вы можете использовать этот ключ в качестве ключа поиска для своей карты, если применяете ее версию в хеш-таблице.

Карты сохранения cgroup и сохранения по ЦПУ

Эти два типа карт были введены, чтобы помочь работать с программами BPF, прикрепленными к контрольной группе. Как говорилось в главе 2, можно присоединять программы BPF к контрольным группам и отсоединять их, а также изолировать их среду выполнения от конкретных контрольных групп

с помощью BPF_PROG_TYPE_CGROUP_SKB. Эти две карты определены с типами BPF_MAP_TYPE_CGROUP_STORAGE и BPF_MAP_TYPE_PERCPU_CGROUP_STORAGE.

С точки зрения разработчика, данные типы карт похожи на карты хеш-таблиц. Ядро предоставляет структурный помощник для генерации ключей для этой карты, bpf_cgroup_storage_key, который содержит информацию об идентификаторе узла контрольной группы и типе вложения. Вы можете добавить в эту карту любое значение, но доступ будет ограничен программой BPF, находящейся внутри присоединенной группы.

У этих карт есть два ограничения. Во-первых, вы не можете создавать новые элементы в карте из пространства пользователя. Программа BPF в ядре способна создавать элементы с помощью bpf_map_update_elem, но если вы используете этот метод из пользовательского пространства, а ключа еще не существует, bpf_map_update_elem завершится с ошибкой и для errno будет задано значение ENOENT. Второе ограничение заключается в том, что вы не можете удалять элементы из этой карты. bpf_map_delete_elem всегда завершается неуспехом и устанавливает errno в EINVAL.

Основное различие между этими двумя типами карт, как вы видели ранее на подобных картах, заключается в том, что BPF_MAP_TYPE_PERCPU_CGROUP_STORAGE хранит разные хеш-таблицы для каждого процессора.

Карты переиспользования сокетного порта

Этот специализированный тип карты хранит ссылки на сокеты, которые может повторно пользовать открытый порт в системе. Они определены с типом BPF_MAP_TYPE_REUSE_PORT_SOCKARRAY. Эти карты в основном применяются с типами программ BPF_PROG_TYPE_SK_REUSEPORT. В совокупности они дают вам контроль над тем, как фильтровать и обслуживать входящие пакеты с сетевого устройства. Например, вы можете решить, какие пакеты в какой сокет будут отправляться, даже если оба сокета подключены к одному порту.

Карты очередей

Карты очередей для хранения элементов используют принцип «первым пришел — первым вышел» (FIFO). Они определены с типом BPF_MAP_TYPE_QUEUE. FIFO означает, что, когда вы выбираете элемент из карты, это будет элемент, который находился в ней дольше всех.

Помощники для карт bpf работают предсказуемо для этой структуры данных. Когда вы используете bpf_map_lookup_elem, карта всегда ищет самый старый элемент. Когда применяете bpf_map_update_elem, она всегда добавляет элемент в конец очереди, поэтому вам нужно прочитать остальные элементы в карте, прежде чем вы сможете получить этот элемент. Вы также можете задействовать помощник bpf_map_lookup_and_delete для выбора более старого элемента и удаления его из карты атомарно. Данная карта не поддерживает помощники bpf_map_delete_elem и bpf_map_get_next_key. Если вы попытаетесь использовать их, они не сработают и в результате для переменной errno будет установлено значение EINVAL.

Что еще следует сказать об этих типах карт: они не используют ключи карт для поиска, а размер ключа при инициализации карт всегда должен быть равен 0. Когда вы помещаете элементы в эти карты, ключ должен иметь нулевое значение.

Посмотрим на пример использования карты такого типа:

```
struct bpf_map_def SEC("maps") queue_map = {
  .type = BPF_MAP_TYPE_QUEUE,
  .key_size = 0,
  .value_size = sizeof(int),
  .max_entries = 100,
  .map_flags = 0,
};
```

Поместим несколько элементов в карту и получим их в том же порядке, в котором вставляли:

```
int i;
for (i = 0; i < 5; i++)
  bpf_map_update_elem(&queue_map, NULL, &i, BPF_ANY);

int value;
for (i = 0; i < 5; i++) {
  bpf_map_lookup_and_delete(&queue_map, NULL, &value);
  printf("Value read from the map: '%d'\n", value);
}
```

Эта программа выведет следующее:

```
Value read from the map: '0'
Value read from the map: '1'
Value read from the map: '2'
Value read from the map: '3'
Value read from the map: '4'
```

Если мы попытаемся извлечь новый элемент из карты, `bpf_map_lookup_and_delete` вернет отрицательное число, а для переменной `errno` будет задано значение `ENOENT`.

Карты стека

Карты стеков для хранения элементов используют хранилище, организованное по принципу «последним пришел — первым вышел» (LIFO). Они определены с типом `BPF_MAP_TYPE_STACK`. LIFO означает: выбирая элемент из карты, вы получите элемент, добавленный в карту последним.

Помощники карты `bpf` также работают предсказуемо для этой структуры данных. Когда вы берете `bpf_map_lookup_elem`, карта всегда ищет самый новый элемент. Когда применяете `bpf_map_update_elem`, она всегда добавляет элемент в верхнюю часть стека, поэтому и выбирает его первым. Вы также можете задействовать помощник `bpf_map_lookup_and_delete`, чтобы получить самый новый элемент и удалить его из карты атомарным способом. Эта карта не поддерживает помощники `bpf_map_delete_elem` и `bpf_map_get_next_key`. Если вы попытаетесь использовать их, они не сработают и в результате для переменной `errno` будет установлено значение `EINVAL`.

Рассмотрим пример применения карты:

```
struct bpf_map_def SEC("maps") stack_map = {
  .type = BPF_MAP_TYPE_STACK,
  .key_size = 0,
  .value_size = sizeof(int),
  .max_entries = 100,
  .map_flags = 0,
};
```

Поместим несколько элементов в эту карту и получим их в том же порядке, в котором вставили:

```
int i;
for (i = 0; i < 5; i++)
  bpf_map_update_elem(&stack_map, NULL, &i, BPF_ANY);

int value;
for (i = 0; i < 5; i++) {
  bpf_map_lookup_and_delete(&stack_map, NULL, &value);
  printf("Value read from the map: '%d'\n", value);
}
```

Программа выведет следующее:

```
Value read from the map: '4'
Value read from the map: '3'
Value read from the map: '2'
Value read from the map: '1'
Value read from the map: '0'
```

Если мы попытаемся извлечь новый элемент из карты, `bpf_map_lookup_and_delete` вернет отрицательное число, а для переменной `errno` будет задано значение `ENOENT`.

Вот такие типы карт вы можете использовать в программах BPF. Некоторые окажутся полезнее, чем другие, — это будет зависеть от типа программы, которую вы пишете. В дальнейшем мы углубим новоприобретенные знания.

Как упоминалось ранее, карты BPF хранятся в операционной системе как обычные файлы. Мы не говорили о специфических характеристиках файловой системы, которую ядро использует для сохранения карт и программ. В следующем разделе расскажем о файловой системе BPF, о ее свойствах и, в частности, о том, какой уровень безопасности она способна обеспечить.

Виртуальная файловая система BPF

Фундаментальная характеристика карт BPF заключается в том, что использование файловых дескрипторов означает следующее: при закрытии дескриптора карта и вся содержащаяся в ней информация исчезают. Первоначально карты BPF были реализованы для краткосрочных изолированных программ, которые не делятся никакой информацией друг с другом. В таких сценариях стирание всех данных при закрытии файлового дескриптора имело большой смысл. Однако с введением в ядро более сложных карт и иных новшеств разработчики поняли, что им нужен способ сохранить информацию, находящуюся в картах, даже после того, как программа завершилась и закрыла файловый дескриптор карты. Версия 4.4 ядра Linux представила два новых системных вызова, позволяющих сохранять карты и программы BPF в виртуальной файловой системе и извлекать их из нее. Карты и программы BPF,

прикрепленные к файловой системе, останутся в памяти после завершения работы программы, которая их создала. В этом разделе мы объясним, как работать с виртуальной файловой системой.

Каталог по умолчанию, в котором BPF ожидает найти виртуальную файловую систему, — это `/sys/fs/bpf`. Некоторые дистрибутивы Linux не монтируют ее по умолчанию, поскольку не предполагают, что ядро поддерживает BPF. Вы можете смонтировать ее самостоятельно, используя команду `mount`:

```
# mount -t bpf /sys/fs/bpf /sys/fs/bpf
```

Как и любая другая файловая иерархия, постоянные объекты BPF в файловой системе идентифицируются путями. Вы можете организовать эти пути любым способом, который имеет смысл для ваших программ. Например, если хотите, чтобы программы делились между собой определенной картой с информацией об IP, можете сохранить ее в `/sys/fs/bpf/shared/ips`. Как упоминалось ранее, есть два типа объектов, которые вы можете сохранить в этой файловой системе: карты BPF и полные программы BPF. Оба они идентифицируются дескрипторами файлов, поэтому интерфейс для работы с ними одинаков. Управлять этими объектами можно только системным вызовом `bpf`. Хотя ядро предоставляет высокоуровневые помощники, которые помогут взаимодействовать с ними, вы не сможете открыть файлы с помощью системного вызова `open`.

`BPF_PIN_FD` — это команда для сохранения объектов BPF в файловой системе. После успешного выполнения команды объект будет располагаться в ней по указанному вами пути. Если команда не выполняется, она возвращает отрицательное число и глобальная переменная `errno` получает код ошибки.

`BPF_OBJ_GET` — это команда для извлечения объектов BPF, которые были закреплены в файловой системе. Команда использует путь, который вы назначили объекту для загрузки. Если она выполняется успешно, то возвращает идентификатор дескриптора файла, связанный с объектом. При сбое возвращается отрицательное число и в глобальную переменную `errno` записывается конкретный код ошибки.

Рассмотрим пример применения этих двух команд в разных программах с помощью вспомогательных функций, которые предоставляет ядро.

Во-первых, напишем программу, которая создает карту, заполняет ее несколькими элементами и сохраняет в файловой системе:

```
static const char * file_path = "/sys/fs/bpf/my_array";

int main(int argc, char **argv) {
  int key, value, fd, added, pinned;

  fd = bpf_create_map(BPF_MAP_TYPE_ARRAY, sizeof(int), sizeof(int), 100, 0); ❶
  if (fd < 0) {
    printf("Failed to create map: %d (%s)\n", fd, strerror(errno));
    return -1;
  }

  key = 1, value = 1234;
  added = bpf_map_update_elem(fd, &key, &value, BPF_ANY);
  if (added < 0) {
    printf("Failed to update map: %d (%s)\n", added, strerror(errno));
    return -1;
  }

  pinned = bpf_obj_pin(fd, file_path);
  if (pinned < 0) {
    printf("Failed to pin map to the file system: %d (%s)\n",
        pinned, strerror(errno));
    return -1;
  }

  return 0;
}
```

❶ Этот раздел кода уже должен быть вам знаком из предыдущих примеров. Сначала мы создаем карту хеш-таблицы с фиксированным размером одного элемента. Затем обновляем ее, чтобы добавить только этот элемент. Если бы мы попытались добавить больше элементов, `bpf_map_update_elem` не сработал бы, потому что карта была бы переполнена.

Используем вспомогательную функцию `pbf_obj_pin` для сохранения карты в файловой системе. Можете проверить, что на вашем компьютере после завершения программы появился новый файл по этому пути:

```
ls -la /sys/fs/bpf
total 0
drwxrwxrwt 2 root  root  0 Nov 24 13:56 .
drwxr-xr-x 9 root  root  0 Nov 24 09:29 ..
-rw------- 1 david david 0 Nov 24 13:56 my_map
```

Теперь мы можем написать аналогичную программу, которая загружает эту карту из файловой системы и печатает вставленные элементы. Так мы убедимся, что сохранили карту правильно:

```
static const char * file_path = "/sys/fs/bpf/my_array";

int main(int argc, char **argv) {
  int fd, key, value, result;

  fd = bpf_obj_get(file_path);
  if (fd < 0) {
    printf("Failed to fetch the map: %d (%s)\n", fd, strerror(errno));
    return -1;
  }

  key = 1;
  result = bpf_map_lookup_elem(fd, &key, &value);
  if (result < 0) {
    printf("Failed to read value from the map: %d (%s)\n",
        result, strerror(errno));
    return -1;
  }

  printf("Value read from the map: '%d'\n", value);
  return 0;
```

Возможность сохранять объекты BPF в файловой системе открывает двери для более интересных приложений. Ваши данные и программы больше не привязаны к одному потоку выполнения. Информация может совместно использоваться различными приложениями, и программы BPF способны запускаться даже после того, как приложение, которое их создало, завершит работу. Это обеспечивает им дополнительный уровень доступности, который был бы невозможен без файловой системы BPF.

Резюме

Установление каналов связи между ядром и пользовательским пространством имеет основополагающее значение для широкого использования преимуществ любой программы BPF. В этой главе вы узнали, как создавать карты BPF для установления такой связи и работать с ними. Мы также описали типы карт, которые вы можете применять в своих программах. Далее в книге вы увидите более конкретные примеры карт. Наконец, вы

узнали, как прикрепить определенные карты к системе, чтобы сделать их и информацию, которую они хранят, устойчивыми к сбоям.

Карты BPF являются главной линией связи между ядром и пользовательским пространством. В этой главе мы описали основные понятия, которые вам необходимо знать для формирования такой связи. В следующей главе начнем шире использовать рассмотренные структуры для обмена данными, а также познакомим вас с дополнительными инструментами, которые сделают работу с картами BPF более эффективной.

В следующей главе вы увидите, как программы и карты BPF работают вместе, чтобы обеспечить возможности трассировки в ваших системах с точки зрения ядра. Мы исследуем разные способы, с помощью которых программы могут присоединиться к точкам входа в ядро. Наконец, рассмотрим, как сделать несколько пит-стопов таким образом, чтобы ваши приложения было легче отлаживать и наблюдать.

4 Трассировка с помощью BPF

В программной инженерии трассировка — это метод сбора данных для профилирования и отладки. Цель этих действий — собрать во время выполнения полезную информацию для последующего анализа. Основное преимущество использования BPF для трассировки в том, что вы можете получить доступ практически к любой части информации ядра Linux с помощью своих приложений. При этом BPF не дает заметного снижения производительности и не приводит к большей загрузке системы, по сравнению с другими технологиями трассировки, и от разработчиков не требуется вносить изменения в приложения только с целью сбора данных.

Ядро Linux предоставляет несколько инструментальных возможностей, которые можно использовать вместе с BPF. В этой главе мы поговорим о них. Мы покажем, как все происходит в ядре и как найти информацию, доступную для ваших программ BPF.

Цель трассировки — дать вам глубокое понимание любой системы, получив все доступные данные и представив их вам наиболее полезным способом. Мы поговорим о нескольких разных представлениях данных и о том, как вы можете применить их в разных сценариях.

Начиная с этой главы, мы собираемся задействовать мощный инструментарий для написания BPF-программ — BPF Compiler Collection (BCC). Отметьте, пожалуйста, что GCC — это основной компилятор программ для UNIX. BCC — это набор компонентов, которые делают построение BPF-программ более предсказуемым. Даже если вы владеете Clang и LLVM, то, вероятно, не захотите тратить больше времени, чем необходимо, на создание одних и тех же утилит и обеспечение того, чтобы верификатор BPF не отбрасывал

ваши программы. BCC предоставляет повторно используемые компоненты для общих структур, таких как карты событий Perf, и интеграцию с бэкендом LLVM, чтобы задействовать наилучшие варианты отладки. Кроме того, BCC включает привязки для нескольких языков программирования — в наших примерах использован Python. Эти привязки позволяют вам писать часть программ BPF пользовательского пространства на языке высокого уровня, что значительно упрощает создание программ. В последующих главах мы также применим BCC, чтобы показать реальные примеры.

Первым шагом для трассировки программ в ядре Linux является определение точек расширения, которые оно предоставляет для присоединения программ BPF. Эти точки обычно называют *зондами*.

Зонды

Вот одно из определений в словаре английского языка для слова *«зонд»*: «Беспилотный исследовательский космический корабль, предназначенный для передачи информации об окружающей среде».

Это определение вызывает у нас (вероятно, у вас тоже) воспоминания о научно-фантастических фильмах и эпических миссиях НАСА. Говоря о трассировке с помощью зондов, можно использовать очень похожее определение: «Трассировочные зонды — это исследовательские программы, предназначенные для передачи информации о среде, в которой они работают».

Зонды собирают данные в вашей системе и предоставляют эту информацию для изучения и анализа. Традиционно их применение в Linux подразумевало написание программ, которые были бы скомпилированы в модули ядра, что могло вызвать катастрофические последствия в производственных системах. С годами они стали более безопасными для выполнения, но все еще были громоздкими в написании и тестировании. Такие инструменты, как SystemTap, ввели новые протоколы для написания зондов и упростили получение более конкретной информации из ядра Linux и всех программ, работающих в пространстве пользователя.

BPF использует трассировки для сбора, отладки и анализа информации. Безопасная природа программ BPF делает их более привлекательными, чем те инструменты, которые все еще требуют перекомпиляции ядра. Перекомпиляция ядра с включением внешних модулей может повысить вероятность

сбоев из-за неправильного поведения кода. Верификатор BPF устраняет этот риск, анализируя программу перед загрузкой в ядро. Разработчики BPF (BPF — это сетевой фильтр, разработанный командой FreeBSD, отсюда и название) использовали определения для зондов и изменили ядро так, чтобы, когда встречается одно из этих определений, выполнялись программы BPF, а не модули ядра.

Понимание того, что представляют собой различные типы зондов, которые вы можете задать, чрезвычайно важно для исследования происходящего в вашей системе. В этом разделе мы классифицируем различные определения зондов и расскажем, как их обнаруживать в вашей системе и как связывать с ними BPF-программы.

В этой главе рассмотрим четыре типа зондов.

- *Зонды ядра*. Предоставляют динамический доступ к внутренним компонентам ядра.
- *Точки трассировки*. Обеспечивают статический доступ к внутренним компонентам ядра.
- *Зонды в пользовательском пространстве*. Дают динамический доступ к программам, работающим в пользовательском пространстве.
- *Статически определенные пользователем точки трассировки*. Обеспечивают статический доступ к программам, запущенным в пространстве пользователя.

Начнем с зондов ядра.

Зонды ядра

Зонды ядра позволяют с минимальными издержками устанавливать динамические флаги или точки останова практически в любой инструкции ядра. Дойдя до одного из этих флагов, ядро выполняет код, прикрепленный к тесту, а затем возобновляет обычную работу. Зонды ядра могут дать вам информацию обо всем, что происходит в системе, например об открытых в ней файлах и исполняемых двоичных файлах. Важно иметь в виду, что у зондов ядра нет стабильного двоичного интерфейса приложения (ABI), то есть они могут меняться между версиями ядра. Один и тот же код может перестать работать, если вы попытаетесь подключить один и тот же зонд к двум системам с разными версиями ядра.

Зонды ядра делятся на две категории — *kprobes* и *kretprobes*. Их использование зависит от того, где в цикле выполнения вы можете вставить программу BPF. В этом разделе рассказывается, как с помощью каждой из них присоединять программы BPF к зондам и извлекать информацию из ядра.

Kprobes

Kprobes позволяет вставлять программы BPF перед выполнением любой инструкции ядра. Вам нужно знать сигнатуру функции, которую предстоит исследовать (как говорилось ранее, это нестабильный ABI, поэтому вы должны быть осторожны при настройке зондов, если собираетесь запускать одну и ту же программу на разных версиях ядра). Когда в ходе работы ядра запускается инструкция, в которой вы установили зонд, оно попадает в ваш код, запускает вашу программу BPF и возвращается к выполнению в исходную точку.

Чтобы показать вам, как использовать kprobes, мы напишем программу BPF, которая выводит имя любого двоичного файла, выполняемого в вашей системе. В этом примере применен внешний интерфейс Python для инструментов BCC, но вы можете написать его с помощью любого другого инструмента BPF:

```
from bcc import BPF

bpf_source = """
int do_sys_execve(struct pt_regs *ctx, void filename, void argv, void envp)
{                                                                          ❶
  char comm[16];
  bpf_get_current_comm(&comm, sizeof(comm));
  bpf_trace_printk("executing program: %s", comm);
  return 0;
}
"""

bpf = BPF(text = bpf_source)                                               ❷
execve_function = bpf.get_syscall_fnname("execve")                         ❸
bpf.attach_kprobe(event = execve_function, fn_name = "do_sys_execve")      ❹
bpf.trace_print()
```

❶ Наша BPF-программа начинает работать. Помощник `bpf_get_current_comm` извлечет имя текущей команды, под управлением которой работает ядро, и сохранит его в переменной `comm`. Мы определили это как массив фиксиро-

ванной длины, потому что ядро имеет ограничение 16 символов для имен команд. После получения имени команды мы печатаем его в трассировке отладки, чтобы человек, который запустил сценарий Python, смог увидеть все команды, отображаемые BPF.

❷ Загрузка программы BPF в ядро.

❸ Связывание программы с системным вызовом `execve`. Имя этого системного вызова меняется в разных версиях ядра, и BCC предоставляет функцию для его получения, причем не требуется запоминать, какую версию ядра вы используете.

❹ Код выводит журнал трассировки, поэтому вы можете видеть все команды, которые вы отслеживаете с помощью этой программы.

Kretprobes

Kretprobes запустит вашу программу BPF после выполнения ядром определенной инструкции и вернет значение. Обычно и kprobes, и kretprobes объединяются в одну BPF-программу, чтобы иметь полное представление о поведении инструкции.

Используем пример, аналогичный приведенному в предыдущем разделе, чтобы показать вам, как работает kretprobes:

```
from bcc import BPF

bpf_source = """
int ret_sys_execve(struct pt_regs *ctx) {                                    ❶
  int return_value;
  char comm[16];
  bpf_get_current_comm(&comm, sizeof(comm));
  return_value = PT_REGS_RC(ctx);

  bpf_trace_printk("program: %s, return: %d", comm, return_value);
  return 0;
}
"""

bpf = BPF(text = bpf_source)                                                 ❷
execve_function = bpf.get_syscall_fnname("execve")
bpf.attach_kretprobe(event = execve_function, fn_name = "ret_sys_execve") ❸
bpf.trace_print()
```

❶ Определение функции, которая реализует программу BPF. Ядро выполнит ее сразу после завершения системного вызова `execve`. `PT_REGS_RC` — это макрос, который будет считывать возвращенное значение из реестра BPF для этого конкретного контекста. Мы также используем `bpf_trace_printk`, чтобы отметить команду и возвращенное ей значение в нашем журнале отладки.

❷ Инициализация программы BPF и ее загрузка в ядро.

❸ Замена присоединенной функции на `attach_kretprobe`.

ЧТО ТАКОЕ АРГУМЕНТ КОНТЕКСТА

Возможно, вы заметили, что у обеих BPF-программ первый аргумент в присоединенной функции один и тот же — `ctx`. Этот параметр, называемый контекстом, дает доступ к информации, которую ядро обрабатывает в настоящее время. Следовательно, контекст зависит от типа программы BPF, которую вы используете в данный момент. Процессор будет хранить информацию о текущей задаче, которую выполняет ядро. Эта структура также зависит от архитектуры вашей системы: набор регистров процессора ARM отличается от имеющегося в процессоре x64. Вы можете получить доступ к этим регистрам, не беспокоясь об архитектуре, с помощью макросов, определенных в ядре, например, `PT_REGS_RC`.

Зонды ядра — это мощный способ доступа к ядру. Но как мы упоминали ранее, они могут быть нестабильными, потому что вы привязываетесь к динамическим точкам в исходном коде ядра, которые могут измениться или исчезнуть в следующей версии. Есть другой, более безопасный способ прикрепления программ к ядру.

Точки трассировки

Точки трассировки — это статичные маркеры в коде ядра, которые можно использовать для присоединения кода в работающем ядре. Основное их отличие от kprobes заключается в том, что они встроены в ядро разработчиками. Вот почему мы называем их статичными. ABI для точек трассировки более стабилен: ядро всегда гарантирует, что точки трассировки, имеющиеся в старой версии, будут существовать и в новых версиях. Однако, учитывая, что разработчикам необходимо добавить их в ядро, они не всегда охватывают все подсистемы, из которых состоит ядро.

Как мы упоминали в главе 2, вы можете найти все доступные точки трассировки в своей системе, просмотрев все файлы в `/sys/kernel/debug/tracing/events`. Например, можете найти все точки трассировки для самого BPF, глядя на события, определенные в `/sys/kernel/debug/tracing/events/bpf`:

```
sudo ls -la /sys/kernel/debug/tracing/events/bpf
total 0
drwxr-xr-x  14 root root 0 Feb  4 16:13 .
drwxr-xr-x 106 root root 0 Feb  4 16:14 ..
drwxr-xr-x   2 root root 0 Feb  4 16:13 bpf_map_create
drwxr-xr-x   2 root root 0 Feb  4 16:13 bpf_map_delete_elem
drwxr-xr-x   2 root root 0 Feb  4 16:13 bpf_map_lookup_elem
drwxr-xr-x   2 root root 0 Feb  4 16:13 bpf_map_next_key
drwxr-xr-x   2 root root 0 Feb  4 16:13 bpf_map_update_elem
drwxr-xr-x   2 root root 0 Feb  4 16:13 bpf_obj_get_map
drwxr-xr-x   2 root root 0 Feb  4 16:13 bpf_obj_get_prog
drwxr-xr-x   2 root root 0 Feb  4 16:13 bpf_obj_pin_map
drwxr-xr-x   2 root root 0 Feb  4 16:13 bpf_obj_pin_prog
drwxr-xr-x   2 root root 0 Feb  4 16:13 bpf_prog_get_type
drwxr-xr-x   2 root root 0 Feb  4 16:13 bpf_prog_load
drwxr-xr-x   2 root root 0 Feb  4 16:13 bpf_prog_put_rcu
-rw-r--r--   1 root root 0 Feb  4 16:13 enable
-rw-r--r--   1 root root 0 Feb  4 16:13 filter
```

Каждый подкаталог, указанный в этих выходных данных, соответствует точке трассировки, к которой мы можем присоединить программы BPF. Но там есть еще два дополнительных файла. Первый файл, `enable`, позволяет включать и отключать все точки трассировки для подсистемы BPF. Если содержимое файла равно `0`, точки трассировки отключены, если `1` — включены. Файл `filter` позволяет задать способ фильтрования событий подсистемой Trace в ядре. BPF не использует данный файл. Подробнее об этом можно узнать из документации по трассировке ядра (oreil.ly/miNRd).

Написание программ BPF с применением точек трассировки аналогично трассировке с помощью kprobes. Вот пример, который задействует программу BPF для трассировки всех приложений в вашей системе, загружающих другие программы BPF:

```
from bcc import BPF

bpf_source = """
int trace_bpf_prog_load(void ctx) {                    ❶
  char comm[16];
  bpf_get_current_comm(&comm, sizeof(comm));
```

```
  bpf_trace_printk("%s is loading a BPF program", comm);
  return 0;
}
"""

bpf = BPF(text = bpf_source)
bpf.attach_tracepoint(tp = "bpf:bpf_prog_load",
                      fn_name = "trace_bpf_prog_load")    ❷
bpf.trace_print()
```

❶ Объявление функции, которая определяет программу BPF. Этот код уже должен быть вам знаком. Есть лишь несколько синтаксических изменений в первом примере, который вы видели, когда мы говорили о kprobes.

❷ Основное отличие этой программы: вместо того чтобы прикреплять программу к kprobe, мы прикрепляем ее к точке трассировки. BCC придерживается соглашения об именовании точек трассировки: сначала указывается подсистема для трассировки — в данном случае `bpf`, затем двоеточие, за которым следует точка трассировки в подсистеме `pbf_prog_load`. Это означает, что каждый раз, когда ядро выполняет функцию `bpf_prog_load`, программа получит событие и выведет имя приложения, которое выполняет данную инструкцию `bpf_prog_load`.

Зонды ядра и точки трассировки предоставят вам полный доступ к ядру. Мы рекомендуем использовать точки трассировки всякий раз, когда это возможно, но вы не обязаны применять их только потому, что они безопаснее. Воспользуйтесь преимуществами динамических возможностей зондов ядра. В следующем разделе мы обсудим, как получить аналогичный уровень наблюдаемости в программах, работающих в пользовательском пространстве.

Зонды пользовательского пространства

Зонды пользовательского пространства позволяют устанавливать динамические флаги в программах, работающих в пользовательском пространстве. Они являются эквивалентом зондов ядра для программ, работающих вне ядра. Когда вы определяете uprobe, ядро создает ловушку вокруг прикрепленной инструкции. Когда ваше приложение достигнет этой инструкции, ядро запускает событие, использующее зондовую функцию в качестве обратного вызова. uprobe также дают вам доступ к любой библиотеке, с которой свя-

зана ваша программа, и вы можете отслеживать такие вызовы, если знаете правильное имя инструкции.

Как и зонды ядра, зонды пользовательского пространства подразделяются на две категории: uprobes и uretprobes — в зависимости от того, где в цикле выполнения вы можете вставить свою BPF-программу. Рассмотрим несколько примеров.

Uprobes

Вообще говоря, uprobes — это ловушки, которые ядро вставляет в набор команд программы перед выполнением конкретной инструкции. Вы должны быть осторожны, когда присоединяете uprobes к различным версиям одной и той же программы, потому что сигнатуры функций могут меняться от версии к версии. Единственный способ гарантировать, что программа BPF будет работать в разных версиях, — это убедиться в том, что подпись не изменилась. Можете использовать команду `nm` в Linux, чтобы получить список всех символов, включенных в объектный файл ELF, что является хорошим способом проверить, существует ли еще инструкция, которую вы отслеживаете, в вашей программе, например:

```
package main
import "fmt"

func main() {
        fmt.Println("Hello, BPF")
}
```

Вы можете скомпилировать программу Go, используя `go build -o hello-bpf main.go`. Можно ввести команду `nm`, чтобы получить информацию обо всех точках инструкций, которые содержит двоичный файл. `nm` — это программа, включенная в GNU Development Tools, которая выводит символы из объектных файлов. Если вы отфильтруете то, что содержит `main`, то получите список, подобный следующему:

```
nm hello-bpf | grep main
0000000004850b0 T main.init
00000000567f06 B main.initdone.
00000000485040 T main.main
000000004c84a0 R main.statictmp_0
00000000428660 T runtime.main
```

```
0000000044da30 T runtime.main.func1
00000000044da80 T runtime.main.func2
000000000054b928 B runtime.main_init_done
00000000004c8180 R runtime.mainPC
0000000000567f1a B runtime.mainStarted
```

Теперь, когда у вас есть список символов, можете отслеживать, когда они выполняются, даже между разными процессами, выполняющими один и тот же бинарный код.

Чтобы отследить, когда будет выполнена основная функция в предыдущем примере, напишем программу BPF и прикрепим ее к uprobe, который прервется до того, как какой-либо процесс вызовет эту инструкцию:

```
from bcc import BPF

bpf_source = """
int trace_go_main(struct pt_regs *ctx) {
  u64 pid = bpf_get_current_pid_tgid();                    ❶
  bpf_trace_printk("New hello-bpf process running with PID: %d", pid);
}
"""

bpf = BPF(text = bpf_source)
bpf.attach_uprobe(name = "hello-bpf",
    sym = "main.main", fn_name = "trace_go_main")    ❷
bpf.trace_print()
```

❶ Используем функцию `bpf_get_current_pid_tgid`, чтобы получить идентификатор процесса (PID), который выполняет программу `hello-bpf`.

❷ Подключаем эту программу к uprobe. Вызов должен знать, что объект, который мы хотим отследить, `hello-bpf`, является абсолютным путем к объектному файлу. Ему также нужны символ, который мы отслеживаем внутри объекта, — в данном случае `main.main` — и программа BPF, которую хотим запустить. При этом каждый раз, когда кто-то запускает `hello-bpf` в вашей системе, мы получаем новую запись трассировки.

Uretprobes

Uretprobes — это то же самое, что и зонд kretprobes, но для программ пользовательского пространства. Они присоединяют программы BPF к инструкциям, возвращающим значения, и предоставляют вам возвращенные значения для получения доступа к регистрам из вашего кода BPF.

Комбинирование uprobes и uretprobes позволяет вам писать более сложные программы BPF. Они могут дать более целостное представление о приложениях, работающих в вашей системе. Если вы вставите код трассировки до запуска функции и сразу после ее завершения, то сможете собрать больше данных и оценить поведение приложения. Обычный вариант — измерение времени выполнения функции без отслеживания каждой строки кода в своем приложении.

Мы повторно используем программу Go, описанную в разделе «Uprobes», чтобы измерить, сколько времени потребуется для выполнения основной функции. Этот пример BPF длиннее, чем предыдущие, поэтому мы разделили его на несколько блоков:

```
bpf_source = """
BPF_HASH(cache, u64, u64);                          ❶

int trace_start_time(struct pt_regs *ctx) {
  u64 pid = bpf_get_current_pid_tgid();
  u64 start_time_ns = bpf_ktime_get_ns();           ❷
  cache.update(&pid, &start_time_ns);               ❸
  return 0;
}
"""
```

❶ Создание хеш-карты BPF. Эта таблица позволяет функциям uprobe и uretprobe обмениваться данными. В этом случае мы используем PID приложения в качестве ключа таблицы и сохраняем время запуска функции в качестве значения. Две наиболее интересные операции в функции `uprobe` выполняются так, как описано далее.

❷ Получаем текущее время в системе в наносекундах, как задано ядром.

❸ Создаем запись в нашем кэше с PID программы и текущим временем. Можно предположить, что это время запуска функции приложения.

Теперь объявим функцию uretprobe. Реализуйте функцию, которую нужно вызвать, когда ваша инструкция отработает. Эта функция uretprobe похожа на другие, которые вы видели в разделе «Kretprobes»:

```
bpf_source += """
static int print_duration(struct pt_regs *ctx) {
  u64 pid = bpf_get_current_pid_tgid();                    ❶
  u64 start_time_ns = cache.lookup(&pid);
  if (start_time_ns == 0) {
```

```
    return 0;
  }
  u64 duration_ns = bpf_ktime_get_ns() - start_time_ns;
  bpf_trace_printk("Function call duration: %d", duration_ns);    ❷
  return 0;                                                       ❸
}
"""
```

❶ Получаем PID для нашего приложения — это нужно для того, чтобы определить момент старта. Используем функцию `lookup`, чтобы извлечь это время из карты, где мы его сохранили до запуска функции.

❷ Рассчитываем продолжительность выполнения функции, определив разницу во времени.

❸ Отмечаем задержку в нашем журнале трассировки, чтобы потом отобразить ее в терминале.

Теперь остальная часть программы должна прикрепить две функции BPF к правильным зондам:

```
bpf = BPF(text = bpf_source)
bpf.attach_uprobe(name = "hello-bpf", sym = "main.main",
           fn_name = "trace_start_time")
bpf.attach_uretprobe(name = "hello-bpf", sym = "main.main",
           fn_name = "print_duration")
bpf.trace_print()
```

Мы добавили строку в оригинальный пример uprobe там, где к uretprobe присоединяется функция печати для приложения.

В этом разделе вы увидели, как отслеживать операции, которые протекают в пространстве пользователя, с помощью BPF. Комбинируя функции BPF, выполняющиеся в разные моменты жизненного цикла приложения, можно получить гораздо более богатую информацию. Однако, как упоминалось в начале этого раздела, хотя зонды пользовательского пространства являются мощным средством, они также нестабильны. Функции BPF могут перестать работать только потому, что кто-то изменит имя функции приложения. А сейчас рассмотрим более устойчивый способ трассировки программ пользовательского пространства.

Статические точки трассировки пользовательского пространства

Статически определенные пользователем точки трассировки (USDT) — это статические точки трассировки для приложений в пользовательском пространстве. Их совокупное использование — это удобный способ для инструментов приложений, потому что накладные расходы на точку входа в возможности трассировки, которые предлагает BPF, невелики. Вы можете использовать их и как соглашение для отслеживания приложений в рабочей среде, независимо от языка программирования, на котором они написаны.

USDT были применены в DTrace, инструменте, изначально разработанном в Sun Microsystems для динамического инструментария систем Unix. До недавнего времени DTrace не был доступен в Linux из-за проблем с лицензированием, однако разработчики ядра Linux исходили из опыта создания DTrace при реализации USDT.

Подобно статическим точкам трассировки ядра, USDT требуют, чтобы разработчики дополнили свой код инструкциями, которые ядро будет использовать в качестве ловушек для выполнения программ BPF. Версия USDTs Hello World состоит всего из нескольких строк кода:

```
#include <sys/sdt.h>
int main() {
  DTRACE_PROBE("hello-usdt", "probe-main");
}
```

В этом примере мы применим макрос, который Linux предоставляет для определения нашего первого USDT. Вы уже можете видеть, откуда ядро получает информацию. `DTRACE_PROBE` регистрирует точку трассировки, которую ядро будет использовать для внедрения нашего обратного вызова функции BPF. Первым аргументом в этом макросе является программа, сообщающая о трассировке, вторым — название трассы, о которой мы хотим узнать.

Многие приложения, которые вы, возможно, установили в своей системе, используют этот тип проверки для доступа к данным трассировки во время выполнения каким-либо предсказуемым образом. Например, популярная база данных MySQL предоставляет все виды информации, задействуя статически определенные точки трассировки. Вы можете собирать информацию

из запросов, выполняемых на сервере, а также из многих других пользовательских операций. Node.js — среда выполнения JavaScript, построенная на основе движка Chrome V8, — также предоставляет точки трассировки, которые можно использовать для извлечения информации о времени выполнения.

Прежде чем показать вам, как присоединять программы BPF к определенной пользовательской точке трассировки, нужно поговорить о том, как их обнаружить. Поскольку точки трассировки определены в двоичном формате внутри исполняемых файлов, нам нужен способ найти список зондов, определенных программой, не копаясь в исходном коде. Одним из способов извлечь эту информацию является непосредственное чтение двоичного файла в формате ELF. Мы можем перекомпилировать предыдущий пример Hello World USDT с помощью GCC:

```
gcc -o hello_usdt hello_usdt.c
```

Эта команда сгенерирует двоичный файл с именем `hello_usdt`, с помощью которого можно применить несколько инструментов для нахождения точек трассировки. Linux предоставляет утилиту `readelf` для отображения информации о файлах ELF. Испробуйте ее на только что скомпилированном примере:

```
readelf -n ./hello_usdt
```

Вы можете увидеть USDT, который мы определили на основе вывода команды:

```
Displaying notes found in: .note.stapsdt
  Owner             Data size      Description
  stapsdt           0x00000033     NT_STAPSDT (SystemTap probe descriptors)
  Provider: "hello-usdt"
  Name: "probe-main"
```

`readelf` способна дать намного больше информации о двоичном файле. В нашем небольшом примере она показывает лишь несколько строк информации, но ее вывод слишком подробен для анализа более сложных двоичных файлов.

Лучшим вариантом для обнаружения точек трассировки, определенных в двоичном файле, является использование инструмента BCC `tplist`, который может отображать как точки трассировки ядра, так и USDT. Преимущество этого инструмента в простоте его вывода: он показывает только опреде-

ления точек трассировки без дополнительной информации об исполняемом файле. Используется примерно так же, как `readelf`:

```
tplist -l ./hello_usdt
```

Здесь перечислены все точки трассировки, которые вы задаете. В нашем примере отображается только одна строка с определением `probe-main`:

```
./hello_usdt "hello-usdt":"probe-main"
```

После того как вы узнаете поддерживаемые точки трассировки в своем двоичном файле, можете присоединить к ним BPF-программы аналогично тому, как было показано в предыдущих примерах:

```
from bcc import BPF, USDT

bpf_source = """
#include <uapi/linux/ptrace.h>
int trace_binary_exec(struct pt_regs *ctx) {
  u64 pid = bpf_get_current_pid_tgid();
  bpf_trace_printk("New hello_usdt process running with PID: %d", pid);
}
"""

usdt = USDT(path = "./hello_usdt")                                          ❶
usdt.enable_probe(probe = "probe-main", fn_name = "trace_binary_exec")      ❷
bpf = BPF(text = bpf_source, usdt = usdt)                                   ❸
bpf.trace_print()
```

В этом примере есть серьезное изменение, которое требует объяснения.

❶ Создать объект USDT. Мы не делали этого в предыдущих примерах. USDT не являются частью BPF, то есть вы можете использовать их, не взаимодействуя с виртуальной машиной BPF. Поскольку они не зависят друг от друга, их применение не зависит от кода BPF.

❷ Присоединить функцию BPF для отслеживания выполнения программы к зонду в нашем приложении.

❸ Инициализировать среду BPF с помощью определения точки трассировки, которое мы только что создали. Это сообщит BCC, что ему необходимо сгенерировать код для соединения нашей программы BPF с определением зонда в создаваемом двоичном файле. Когда они оба присоединены, мы можем получить трассировки, сгенерированные BPF-программой, чтобы отследить выполнение основной программы.

USDT и другие языки программирования

Можно использовать USDT также для отслеживания приложений, написанных на других языках программирования, помимо C. Вы найдете на GitHub привязки для Python, Ruby, Go, Node.js и многих других языков. Привязки для Ruby являются одними из наших любимых из-за их простоты и совместимости с такими фреймворками, как Rails. Дейл Хэмел, который в настоящее время работает в Shopify, написал в своем блоге отличный отчет о применении USDT (https://oreil.ly/7pgNO). Он также поддерживает библиотеку *ruby-static-tracing* (https://oreil.ly/ge6cu), которая делает отслеживание приложений Ruby и Rails еще более простым.

Библиотека статической трассировки Хэмела позволяет задействовать возможности трассировки на уровне класса, не требуя добавления логики трассировки для каждого метода в этом классе. В сложных сценариях это обеспечивает удобные методы самостоятельной регистрации выделенных конечных точек трассировки.

Чтобы использовать `ruby-static-tracing` в своих приложениях, сначала укажите, когда будут включены точки трассировки. Можете включить их по умолчанию при запуске приложения, но если хотите избежать непрерывного сбора данных, активируйте их с помощью сигнала системного вызова. Дэйл Хэмел рекомендует взять в качестве этого сигнала `PROF`:

```
require 'ruby-static-tracing'

StaticTracing.configure do |config|
  config.mode = StaticTracing::Configuration::Modes::SIGNAL
  config.signal = StaticTracing::Configuration::Modes::SIGNALS::SIGPROF
end
```

В такой конфигурации вы можете добавить команду `kill` для включения статических точек трассировки вашего приложения по желанию. В следующем примере мы предполагаем, что на машине работает только процесс Ruby, и можем получить его идентификатор процесса с помощью `pgrep`:

```
kill -SIGPROF 'pgrep -nx ruby'
```

Помимо настройки активизации точек трассировки, вы можете использовать некоторые встроенные механизмы трассировки, которые предоставляет `ruby-static-tracing`. На момент написания книги библиотека содержала точки трассировки для измерения задержки и сбора трассировок стека.

Нам очень нравится, что с помощью этого встроенного модуля утомительная задача измерения задержки функции становится почти тривиальной. Вначале нужно добавить в первоначальную конфигурацию трассировщик задержки:

```
require 'ruby-static-tracing'
require 'ruby-static-tracing/tracer/concerns/latency_tracer'

StaticTracing.configure do |config|
  config.add_tracer(StaticTracing::Tracer::Latency)
end
```

После этого каждый класс, включающий модуль задержки, генерирует статические точки трассировки для каждого определенного открытого метода. Когда трассировка включена, вы можете запрашивать эти точки для сбора данных о времени. В следующем примере `ruby-static-tracing` генерирует статическую точку трассировки с именем `usdt:/proc/X/fd/:user_model:find`, выполняя соглашение об использовании имени класса в качестве пространства имен для точки трассировки и имени метода — в качестве имени точки трассировки:

```
class UserModel
  def find(id)
  end

  include StaticTracing::Tracer::Concerns::Latency
end
```

Теперь можно задействовать BCC для извлечения информации о задержке для каждого вызова метода `find`. Для этого мы используем встроенные функции BCC `bpf_usdt_readarg` и `bpf_usdt_readarg_p`. Они читают аргументы, устанавливаемые каждый раз, когда выполняется код нашего приложения. `ruby-static-tracing` всегда задает имя метода в качестве первого аргумента для точки трассировки, тогда как в качестве второго аргумента устанавливается вычисленное значение. Следующий фрагмент реализует программу BPF, которая получает информацию о точке трассировки и выводит ее в журнале трассировки:

```
bpf_source = """
#include <uapi/linux/ptrace.h>
int trace_latency(struct pt_regs *ctx) {
  char method[64];
  u64 latency;
```

```
  bpf_usdt_readarg_p(1, ctx, &method, sizeof(method));
  bpf_usdt_readarg(2, ctx, &latency);

  bpf_trace_printk("method %s took %d ms", method, latency);
}
"""
```

Нам также нужно загрузить предыдущую BPF-программу в ядро. Поскольку мы отслеживаем конкретное приложение, которое уже запущено на компьютере, то можем прикрепить программу к определенному идентификатору процесса:

```
parser = argparse.ArgumentParser()
parser.add_argument("-p", "--pid", type = int, help = "Process ID")    ❶
args = parser.parse_args()

usdt = USDT(pid = int(args.pid))
usdt.enable_probe(probe = "latency", fn_name = "trace_latency")        ❷
bpf = BPF(text = bpf_source, usdt = usdt)
bpf.trace_print()
```

❶ Указываем PID.

❷ Включаем зонд, загружаем программу в ядро и печатаем журнал трассировки. (Этот раздел очень похож на тот, который вы видели ранее.)

Мы показали, как анализировать приложения, которые статически определяют точки трассировки. Многие известные библиотеки и языки программирования включают в себя эти зонды, что помогает отлаживать запущенные приложения, особенно чтобы отслеживать их работу в производственных средах. Однако это лишь верхушка айсберга — получив данные, вы должны разобраться в них. Это именно то, чем мы займемся в дальнейшем.

Визуализация данных трассировки

Пока мы приводили примеры, которые выводят данные в нашем отладочном отчете. Это не очень полезно в производственной среде. Разобраться в этих данных стоит, однако никто не любит копаться в длинных сложных журналах. Если мы хотим отслеживать изменения в том, почему происходят задержки и загрузка ЦП, лучше просмотреть графики за определенный период, чем разбирать числа из файла.

В этом разделе рассматриваются различные способы представления данных трассировки BPF. Мы покажем, как программы BPF могут структурировать информацию. Также вы узнаете, как экспортировать эту информацию в переносимое представление, как использовать готовые инструменты для получения более информативного представления и делиться своими результатами с коллегами.

Флейм-графы

Флейм-графы — это диаграммы, которые помогают визуализировать то, на что ваша система тратит время. Они могут дать вам четкое представление о том, какой код в приложении выполняется чаще. Брендан Грегг, разработчик флейм-графов, поддерживает набор сценариев для простой генерации этих форматов визуализации в GitHub (oreil.ly/3iiZx). Мы используем такие сценарии для создания флейм-графов на основе данных, собранных с помощью BPF, далее в этом разделе. Как выглядят графики, вы можете увидеть на рис. 4.1.

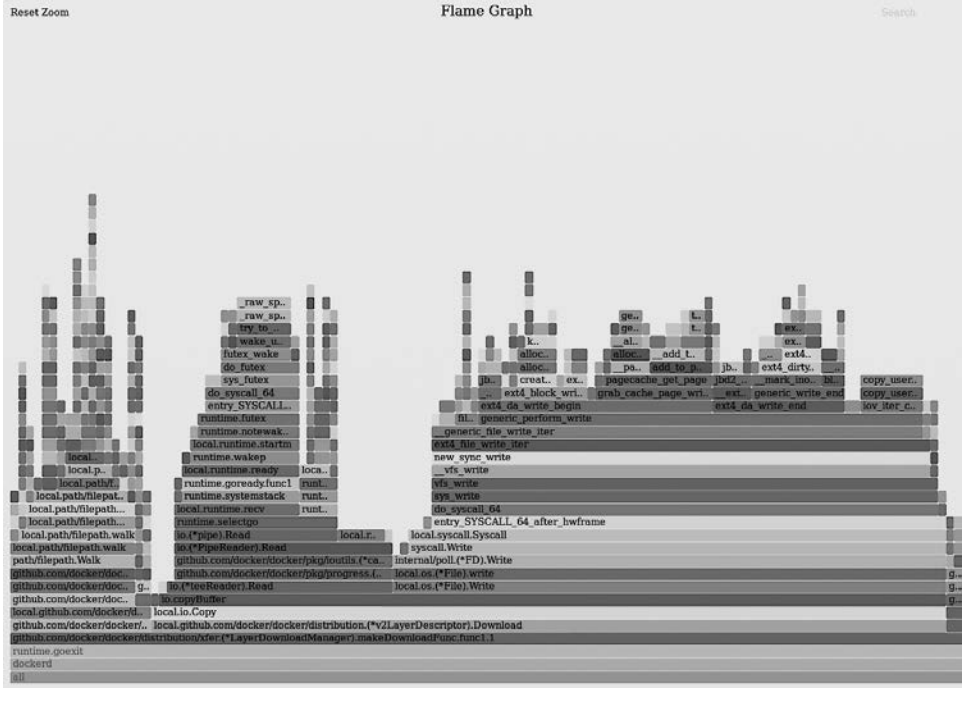

Рис. 4.1. Флейм-граф процессора

Следует помнить две важные вещи.

- На оси *X* данные упорядочены в алфавитном порядке. Ширина каждого стека показывает, как часто он появляется в собираемых данных, что может быть связано с тем, как часто этот путь кода посещался, когда был включен профилировщик.
- По оси *Y* показаны трассы стека, упорядоченные по мере их чтения профилировщиком, с сохранением иерархии трасс.

Самые известные флейм-графы показывают интенсивность кода, наиболее часто используемого процессором. Они называются *графиками по процессору*. Другой интересной визуализацией являются графики вне процессора, они показывают время, которое процессор тратит на задачи, не связанные с вашим приложением. Комбинируя графики на процессоре и вне процессора, вы можете получить полное представление о том, на что система расходует циклы процессора.

Графики и на процессоре, и вне его применяют трассировки стека, чтобы показать, на что система тратит время. Некоторые языки программирования, такие как Go, всегда включают информацию трассировки в свои двоичные файлы, а другие, такие как C++ и Java, требуют дополнительных усилий, чтобы сделать трассировки стека читабельными. После того как ваше приложение сделает доступной информацию о трассировке стека, программы BPF могут использовать ее для объединения наиболее часто задействуемых путей кода, как это видит ядро.

У объединения трассировки стека в ядре есть свои преимущества и недостатки. С одной стороны, это эффективный способ подсчета частоты трассировки стека, потому что все происходит в ядре, никакая информация стека не отправляется в пространство пользователя и тем самым уменьшается обмен данными между ядром и пространством пользователя. С другой — количество событий, которые нужно обработать для графиков вне процессора, может быть довольно большим, потому что вы отслеживаете каждое событие, которое происходит во время переключения контекста вашего приложения. Это может значительно нагрузить вашу систему, если вы попытаетесь профилировать его слишком долго. Имейте это в виду, работая с флейм-графами.

BCC предоставляет несколько утилит, которые помогут вам объединить и визуализировать трассировки стека, но особенно важен макрос `BPF_STACK_TRACE`. Он генерирует карту BPF типа `BPF_MAP_TYPE_STACK_TRACE` для хранения

стеков, которые накапливает ваша программа BPF. Кроме того, эта карта BPF расширена за счет методов, позволяющих извлекать информацию о стеке из контекста программы и просматривать накопленные трассировки стека, если вы хотите использовать их после объединения.

В следующем примере создадим простой профилировщик BPF, который выводит трассировки стека, собранные из приложений пользовательского пространства. Мы генерируем флейм-графы ЦП с трассировками, которые собирает профилировщик. Чтобы протестировать его, напишем маленькую программу на Go, которая создает нагрузку на процессор. Вот код для этого приложения:

```
package main

import "time"

func main() {
        j := 3
        for time.Since(time.Now()) < time.Second {
                for i := 1; i < 1000000; i++ {
                        j *= i
                }
        }
}
```

Если вы сохраните этот код в файле с именем `main.go` и запустите его с помощью `go run main.go`, то увидите, что загрузка ЦП вашей системы значительно возросла. Вы можете остановить выполнение, нажав сочетание клавиш Ctrl+C, и загрузка ЦП вернется к норме.

Первая часть программы BPF инициализирует структуры профилировщика:

```
bpf_source = """
#include <uapi/linux/ptrace.h>
#include <uapi/linux/bpf_perf_event.h>
#include <linux/sched.h>

struct trace_t {                    ❶
  int stack_id;
}

BPF_HASH(cache, struct trace_t);    ❷
BPF_STACK_TRACE(traces, 10000);     ❸
"""
```

❶ Инициализируем структуру, где будет храниться ссылочный идентификатор для каждого из стековых кадров, которые получает профилировщик. Применим эти идентификаторы позже, чтобы узнать, каким путем код выполнялся в то время.

❷ Инициализируем хеш-карту BPF, с помощью которой соберем сведения о том, как часто появляется один и тот же кадр. Сценарии флейм-графа используют это агрегированное значение, чтобы определить, как часто выполняется один и тот же код.

❸ Инициализируем нашу карту трассировки стека BPF. Мы устанавливаем максимальный размер этой карты, но он может варьироваться в зависимости от объема данных, которые требуется обработать. Было бы лучше, если бы этим значением была переменная, но так как приложение на Go не очень большое, то 10 000 элементов будет достаточно.

Далее реализуем функцию, которая объединяет трассировки стека в профилировщике:

```
bpf_source += """
int collect_stack_traces(struct bpf_perf_event_data *ctx) {
  u32 pid = bpf_get_current_pid_tgid() >> 32;           ❶
  if (pid != PROGRAM_PID)
    return 0;

  struct trace_t trace = {                              ❷
    .stack_id = traces.get_stackid(&ctx->regs, BPF_F_USER_STACK)
  };

  cache.increment(trace);                               ❸
  return 0;
}
"""
```

❶ Проверяем то, что идентификатор процесса для программы в текущем контексте BPF является идентификатором для нашего приложения Go, в противном случае событие игнорируется. На данный момент мы не определили значение для `PROGRAM_PID`. Давайте заменим эту строку в части Python профилировщика перед инициализацией программы BPF. Это текущее ограничение для способа, которым BCC инициализирует программу BPF: мы не можем передавать переменные из пространства пользователя, и, как правило, эти строки заменяются в коде перед инициализацией.

❷ Создаем трассировку, чтобы собрать все возможные данные. Извлекаем идентификатор стека из контекста программы с помощью встроенной функции `get_stackid`. Это один из помощников, который BCC добавляет к нашей карте трассировки стека. Мы также используем флаг `BPF_F_USER_STACK`, чтобы указать, что следует получить идентификатор стека для приложения пользовательского пространства, при этом неважно, что происходит в ядре.

❸ Добавляем значение счетчика трассировки, чтобы отследить частоту выполнения одного и того же кода.

Теперь присоединим наш сборщик трассировки стека ко всем событиям Perf в ядре:

```
program_pid = int(sys.argv[0])                                    ❶
bpf_source = bpf_source.replace('PROGRAM_PID', program_pid)       ❷

bpf = BPF(text = bpf_source)
bpf.attach_perf_event(ev_type = PerfType.SOFTWARE,                ❸
                      ev_config = PerfSWConfig.CPU_CLOCK,
                      fn_name = 'collect_stack_traces')
```

❶ Первый аргумент — для нашей программы Python. Это идентификатор процесса для приложения Go, которое мы профилируем.

❷ Используем встроенную функцию Python `replace` для замены строки `PROGRAM_ID` в источнике BPF аргументом, предоставленным профилировщику.

❸ Подсоединяем программу BPF ко всем событиям Software Perf, при этом будут игнорироваться любые другие события, например сообщения от оборудования. Мы также настраиваем нашу программу BPF на использование тактов процессора как показателя времени, чтобы понять, сколько же времени заняло выполнение приложения.

Наконец, нам нужно реализовать код, который будет выгружать трассировки стека в стандартный вывод при прерывании работы профилировщика:

```
try:
  sleep(99999999)
except KeyboardInterrupt:
  signal.signal(signal.SIGINT, signal_ignore)

for trace, acc in sorted(cache.items(), key=lambda cache: cache[1].value):  ❶
  line = []
```

```
  if trace.stack_id < 0 and trace.stack_id == -errno.EFAULT          ❷
    line = ['Unknown stack']
  else
    stack_trace = list(traces.walk(trace.stack_id))
    for stack_address in reversed(stack_trace)                       ❸
      line.extend(bpf.sym(stack_address, program_pid))               ❹

frame = b";".join(line).decode('utf-8', 'replace')                   ❺
print("%s %d" % (frame, acc.value))
```

❶ Просмотрим все собранные трассировки, чтобы вывести их по порядку.

❷ Теперь посмотрим, что у нас есть идентификаторы стека, которые позже можно соотнести с конкретными строками кода. Если мы получим недопустимое значение, то будем использовать заполнитель в флейм-графе.

❸ Перебираем все записи в трассировке стека в обратном порядке. Это нужно для того, чтобы найти первый путь выполнения программы, который использовался самым последним, как в любой трассировке стека.

❹ Вызываем вспомогательный BCC `sym`, чтобы преобразовать адрес памяти для стекового фрейма в имя функции, использованное в нашем исходном коде.

❺ Форматируем строку трассировки стека через точку с запятой. Это формат, который понятен сценариям флейм-графа и применяется, чтобы сгенерировать нужную визуализацию.

Завершив наш профилировщик BPF, можем запустить его, используя `sudo` для сбора трассировок стека для нашей занятой программы Go. Нужно передать идентификатор процесса программы Go профилировщику, чтобы убедиться, что мы собираем только трассировки для этого приложения. Можем найти этот PID с помощью `pgrep`. Так можно запустить профилировщик, если нужно сохранить его вывод в файле с именем `profile.out`:

```
./profiler.py `pgrep -nx go` > /tmp/profile.out
```

`pgrep` найдет PID запущенного в вашей системе процесса по имени. Отправляем вывод профилировщика во временный файл, чтобы потом сгенерировать визуализацию флейм-графа.

Как упоминалось ранее, мы применим сценарии FlameGraph Брендана Грегга для генерирования файла SVG для нашего графика. Эти сценарии есть в его

репозитории GitHub (https://oreil.ly/orqcb). После того как вы загрузили этот репозиторий, можете создать график с помощью `Flamegraph.pl`. Откройте график в своем браузере (у нас Firefox):

```
./flamegraph.pl /tmp/profile.out > /tmp/flamegraph.svg && \
  firefox /tmp/flamefraph.svg
```

Профилировщик такого типа полезен для отслеживания проблем с производительностью в вашей системе. BCC включает в себя более оптимальный профилировщик, чем тот, что задействован в нашем примере, можете использовать его в своих производственных средах. Помимо профилировщика, в BCC есть инструменты, которые помогут создавать флейм-графы вне процессора и многие другие визуализации для анализа систем.

Флейм-графы полезны для анализа производительности. Мы часто применяем их в повседневной работе. Во многих сценариях, помимо визуализации путей кода, требуется определять, как часто происходят события в системах. Рассмотрим этот вопрос подробнее.

Гистограммы

Гистограммы — это диаграммы, показывающие, как часто встречаются несколько диапазонов значений. Числовые данные для такого представления делятся на сегменты, и каждый из них содержит количество вхождений любой точки данных в сегменте. Частота, которую измеряют гистограммы, является комбинацией высоты и ширины каждого сегмента. Если сегменты разделены на равные диапазоны, эта частота соответствует высоте гистограммы, но, если диапазоны неравные, нужно умножить каждую высоту на каждую ширину, чтобы вычислить нужную частоту.

Гистограммы являются основой для анализа производительности систем. Это отличный инструмент для представления распределения измеримых событий, таких как время выполнения инструкций, потому что с их помощью можно увидеть больше, чем посредством других измерений, например, средних значений.

BPF-программы могут создавать гистограммы на основе множества метрик. Вы можете использовать карты BPF, чтобы собирать информацию, классифицировать ее по группам, а затем генерировать представление гистограммы для своих данных. Реализация этой логики несложна, но она утомляет, если

вы хотите получать гистограммы каждый раз, когда нужно проанализировать вывод программы. BCC включает в себя готовую реализацию, которую можно повторно использовать в каждой программе, при этом не требуется каждый раз выполнять группировку и вычислять частоту вручную. Однако в коде ядра имеется прекрасная возможность, которую мы рекомендуем применять в примерах BPF.

В качестве забавного эксперимента покажем, как использовать гистограммы BCC для визуализации задержки, возникающей при загрузке BPF-программ, когда приложение вызывает инструкцию `bpf_prog_load`. С помощью kprobes определим, сколько времени потребуется для выполнения этой инструкции, и будем накапливать результаты в гистограмме, которую представим позже. Мы разделили этот пример на несколько частей, чтобы легче было понять ход действий.

Первая часть содержит исходный код нашей программы BPF:

```
bpf_source = """
#include <uapi/linux/ptrace.h>

BPF_HASH(cache, u64, u64);
BPF_HISTOGRAM(histogram);

int trace_bpf_prog_load_start(void ctx) {    ❶
  u64 pid = bpf_get_current_pid_tgid();      ❷
  u64 start_time_ns = bpf_ktime_get_ns();
  cache.update(&pid, &start_time_ns);        ❸
  return 0;
}
"""
```

❶ Используйте макрос, чтобы создать хеш-карту BPF для сохранения момента срабатывания инструкции `bpf_prog_load`.

❷ Возьмите новый макрос для создания карты гистограммы BPF. Это не присущая BPF карта — BCC содержит данный макрос, чтобы вам было проще создавать такие визуализации. Внутри этой гистограммы BPF используются карты массивов для хранения информации. У макроса есть несколько помощников для разделения и создания окончательного графика.

❸ Применяйте PID-программы для того, чтобы найти приложение, когда оно запускает команду, которую мы хотим отследить. (Эта функция покажется вам знакомой — мы взяли ее из примера, в котором исследовали uprobes.)

Посмотрим, как вычислить дельту для задержки и сохранить ее в гистограмме. Начальные строки нового блока кода также будут выглядеть знакомо, потому что мы все еще используем пример, о котором говорили в подразделе «Uprobes»:

```
bpf_source += """
int trace_bpf_prog_load_return(void ctx) {
  u64 *start_time_ns, delta;
  u64 pid = bpf_get_current_pid_tgid();
  start_time_ns = cache.lookup(&pid);
  if (start_time_ns == 0)
    return 0;

  delta = bpf_ktime_get_ns() - *start_time_ns;    ❶
  histogram.increment(bpf_log2l(delta));          ❷
  return 0;
}
"""
```

❶ Рассчитайте дельту между временем, когда инструкция была вызвана, и временем, которое потребовалось нашей программе, чтобы выполнить код до данного момента. Можно предположить, что это также время выполнения инструкции.

❷ Сохраните дельту в нашей гистограмме. В этой строке мы выполняем две операции. Сначала используем встроенную функцию `bpf_log2l`, чтобы сгенерировать идентификатор сегмента для значения дельты. Эта функция создает стабильное распределение значений во времени. Затем берем функцию `increment`, чтобы добавить в этот сегмент новый элемент. По умолчанию приращение добавляет 1 к значению, если сегмент был в гистограмме, или запускает новый сегмент со значением 1, поэтому вам не нужно заранее беспокоиться о том, существует ли значение.

Последний фрагмент кода, который нам нужно написать, присоединяет эти две функции к работающему kprobes и выводит гистограмму на экран, чтобы можно было увидеть распределение задержек. В этом разделе мы инициализируем нашу программу BPF и станем отслеживать события, чтобы сгенерировать гистограмму:

```
bpf = BPF(text = bpf_source)                    ❶
bpf.attach_kprobe(event = "bpf_prog_load",
    fn_name = "trace_bpf_prog_load_start")
bpf.attach_kretprobe(event = "bpf_prog_load",
    fn_name = "trace_bpf_prog_load_return")
```

```
try:                                          ❷
  sleep(99999999)
except KeyboardInterrupt:
  print()

bpf["histogram"].print_log2_hist("msecs")     ❸
```

❶ Инициализируйте BPF и присоедините ваши функции к kprobes.

❷ Включите в программу ожидание, чтобы собрать столько информации о событиях, сколько нужно получить из нашей системы.

❸ Выведите карту гистограммы в терминале с отслеженным распределением событий — это еще один макрос BCC, который позволяет получить карту гистограммы.

Как мы упоминали в начале этого раздела, гистограммы могут быть полезны для наблюдения аномалий в вашей системе. Инструменты BCC содержат многочисленные сценарии, которые используют гистограммы для представления данных. Настоятельно рекомендуем задействовать их, когда вам нужна визуализация, чтобы лучше понять, как работает ваша система.

События Perf

Мы считаем, что события Perf, вероятно, являются наиболее важным методом коммуникации, который вам необходимо освоить, чтобы успешно использовать трассировку BPF. Мы говорили о картах массива событий BPF Perf в предыдущей главе. Они позволяют вам помещать данные в кольцо буфера, которое в реальном времени синхронизируется с программами пользовательского пространства. Прекрасно, если вы собираете большой объем данных в своей программе BPF и хотите перенести обработку и визуализацию в программу пользовательского пространства. Это позволит лучше контролировать уровень представления, потому что вы не ограничены виртуальной машиной BPF в плане программирования. Большинство программ трассировки BPF, которые вам удастся найти, используют события Perf именно для этой цели.

Теперь рассмотрим, как их применить для извлечения информации о выполнении и ее классификации, чтобы вывести, какие двоичные файлы наиболее часто исполняются в вашей системе. Мы разделили этот пример на два блока

кода, чтобы было проще. В первом блоке определяем программу BPF и присоединяем ее к kprobe, как делали ранее в подразделе «Probes»:

```
bpf_source = """
#include <uapi/linux/ptrace.h>

BPF_PERF_OUTPUT(events);                                    ❶
int do_sys_execve(struct pt_regs *ctx, void filename, void argv, void envp) {
  char comm[16];
  bpf_get_current_comm(&comm, sizeof(comm));

  events.perf_submit(ctx, &comm, sizeof(comm));             ❷
  return 0;
}
"""

bpf = BPF(text = bpf_source)
execve_function = bpf.get_syscall_fnname("execve")        ❸
bpf.attach_kprobe(event = execve_function, fn_name = "do_sys_execve")
```

В первой строке этого примера импортируем библиотеку из стандартной библиотеки Python. Мы собираемся применять счетчик Python для агрегирования событий, которые получаем от нашей программы BPF.

❶ Используем `BPF_PERF_OUTPUT`, чтобы объявить карту событий Perf. Это удобный макрос, который BCC предоставляет для объявления карты такого типа. Назовем эту карту `events`.

❷ Отправляем эту карту в пользовательское пространство для агрегации после того, как у нас будет имя программы, которую выполнило ядро. Мы делаем это с помощью `perf_submit`. Эта функция обновляет карту событий Perf, внося в нее новую информацию.

❸ Инициализируем программу BPF и подключаем ее к kprobe, которая будет включаться при запуске новой программы в нашей системе.

Теперь, когда мы написали код для всех программ, выполняемых в нашей системе, нужно собрать их в пространстве пользователя. В следующем фрагменте кода содержится много информации, рассмотрим самую важную:

```
from collections import Counter
aggregates = Counter()                                      ❶
```

```
def aggregate_programs(cpu, data, size):                ❷
  comm = bpf["events"].event(data)                      ❸
  aggregates[comm] += 1

bpf["events"].open_perf_buffer(aggregate_programs)      ❹
while True:
    try:
      bpf.perf_buffer_poll()
    except KeyboardInterrupt:                           ❺
      break

for (comm, times) in aggregates.most_common():
  print("Program {} executed {} times".format(comm, times))
```

❶ Объявляем счетчик для хранения информации о нашей программе. Поскольку мы используем имя программы в качестве ключа, то счетчиками будут значения. Задействуем функцию `aggregate_programs` для сбора данных из карты событий Perf. В этом примере вы можете увидеть, как макрос BCC применяется для доступа к карте и извлечения последующего события входящих данных из вершины стека.

❷ Увеличиваем количество поступлений событий с тем же именем программы.

❸ Используем функцию `open_perf_buffer`, чтобы сообщить BCC, что нужно запускать функцию `aggregate_programs` каждый раз при получении событий из карты событий Perf.

❹ BCC опрашивает события после открытия кольцевого буфера, пока мы не прервем эту программу Python. Чем дольше вы собираете результаты, тем больше информации придется обрабатывать. Здесь продемонстрировано, как для этой цели применяется `perf_buffer_poll`.

❺ Функция `most_common` дает список элементов в счетчике и цикле, чтобы сначала распечатать наиболее часто исполняемые в системе программы.

События Perf могут помочь при обработке всех данных, которые BPF предоставляет новыми и неожиданными способами. Мы рассмотрели пример сбора произвольных данных из ядра. Много других примеров вы можете найти в инструментах, которые включены в BCC для отслеживания.

Резюме

В этой главе мы лишь поверхностно рассмотрели трассировку с помощью BPF. Ядро Linux дает вам доступ к информации, которую труднее получить с помощью других инструментов. BPF делает этот процесс более предсказуемым, предоставляя общий интерфейс для доступа к данным. В последующих главах вы увидите больше примеров, в которых используются некоторые из описанных методов, например присоединение функций к точкам трассировки. Они помогут вам начать применять на практике то, что вы уже узнали.

В этой главе для большинства примеров мы использовали платформу BCC. Если вам интересно, попробуйте реализовать те же примеры на C, как мы делали в предыдущих главах. Однако BCC предоставляет несколько встроенных функций, которые упрощают написание программ трассировки по сравнению с тем, как это происходит с применением C.

В следующей главе расскажем о некоторых инструментах, которые сообщество разработало для использования поверх BPF для анализа и отслеживания производительности. Написание собственных программ — это хорошо, но уже созданные специальные инструменты дают вам доступ к большей части информации. Так что вам не нужно создавать инструменты — они уже существуют.

5 Утилиты BPF

До сих пор мы говорили о том, как написать BPF-программы для лучшей наблюдаемости системы. Многие разработчики занимались тем же и создали утилиты для BPF с той же целью. В этой главе поговорим о нескольких готовых инструментах, которыми вы можете пользоваться каждый день. Одни из них являются продвинутыми версиями программ BPF, которые вы уже видели. Другие — это инструменты, способные обеспечить вам наблюдаемость в собственных программах BPF.

В этой главе речь пойдет об инструментах для повседневной работы с BPF. Начнем с описания BPFTool — утилиты командной строки, которая поможет вам получить больше информации о своих программах BPF. Рассмотрим BPFTrace и `kubectl-trace`, которые позволят более эффективно писать программы BPF с помощью краткого предметно-ориентированного языка (DSL). Наконец, поговорим о eBPF Exporter — проекте с открытым исходным кодом для интеграции BPF с Prometheus.

BPFTool

BPFTool — это утилита ядра для проверки программ и карт BPF. Этот инструмент не устанавливается по умолчанию ни в одном дистрибутиве Linux, к тому же сейчас он в стадии разработки, поэтому нужно скомпилировать версию, которая лучше всего поддерживает ваше ядро Linux. Мы рассмотрим версию BPFTool, распространяемую с версией 5.1 ядра Linux.

В следующих разделах обсудим, как установить BPFTool в вашу систему и использовать его для наблюдения программ и карт BPF с терминала и изменения их поведения.

Установка

Чтобы установить BPFTool, вам необходимо скачать копию исходного кода ядра. В вашем конкретном онлайн-дистрибутиве Linux могут быть некоторые пакеты, но мы расскажем, как установить его из исходного кода — это не слишком сложно.

1. Используйте Git для клонирования хранилища GitHub с помощью команды `git clone https://github.com/torvalds/linux`.
2. Проверьте конкретный тег версии ядра с помощью `git checkout v5.1`.
3. В исходном коде ядра перейдите в каталог, где хранится исходный код BPFTool, с помощью `cd tools/bpf/bpftool`.
4. Скомпилируйте и установите этот инструмент с помощью `make && sudo make install`.

Можете убедиться в правильности установки BPFTool, проверив его версию:

```
# bpftool --version
bpftool v5.1.0
```

Вывод функциональных возможностей

Одна из основных операций, которую можно выполнять с BPFTool, — это сканирование системы. Так вы сможете узнать, к каким функциям BPF у вас есть доступ. Это очень полезно, если вы не помните версию ядра, версию программы или если у вас включен JIT-компилятор BPF. Чтобы получить ответ на эти и многие другие вопросы, выполните следующую команду:

```
# bpftool feature
```

Вы получите длинный вывод с подробной информацией обо всех поддерживаемых в ваших системах функциях BPF. Для краткости покажем его неполную версию:

```
Scanning system configuration...
bpf() syscall for unprivileged users is enabled
JIT compiler is enabled
```

```
...
Scanning eBPF program types...
eBPF program_type socket_filter is available
eBPF program_type kprobe is NOT available
...
Scanning eBPF map types...
eBPF map_type hash is available
eBPF map_type array is available
```

В этом выводе можно увидеть, что наша система позволяет непривилегированным пользователям выполнять `syscall bpf` и этот вызов ограничен определенными операциями. Также видно, что JIT включен. Более новые версии ядра включают JIT по умолчанию, и это очень помогает при компиляции BPF-программ. Если в вашей системе он не включен, можете выполнить следующую команду, чтобы сделать это:

```
# echo 1 > /proc/sys/net/core/bpf_jit_enable
```

К особенностям вывода относится то, какие типы программ и карт включены в вашей системе. Эта команда предоставляет гораздо больше информации, чем мы здесь показываем, например сведения о помощниках BPF, поддерживаемых типом программы и многими другими директивами конфигурации. Почитайте описания при изучении своей системы.

Знание того, какие функции имеются в вашем распоряжении, может оказаться полезным, особенно если вам необходимо изучить неизвестную систему. После этого мы готовы перейти к другим интересным функциям BPFTool, таким как проверка загруженных программ.

Инспекция программ BPF

BPFTool предоставляет информацию о программах BPF прямо из ядра. Это позволяет отследить, что уже работает в вашей системе. А также дает возможность загружать и прикреплять новые программы BPF, которые были предварительно скомпилированы в командной строке.

Лучшей отправной точкой для изучения способов применения BPFTool является работа с программами проверки того, что функционирует в вашей системе. Для этого вы можете запустить команду `bpftool prog show`. Если используете Systemd в качестве системы инициализации, у вас, вероятно, уже есть несколько программ BPF, загруженных и подключенных к некоторым

контрольным группам (поговорим об этом чуть позже). Результат выполнения этой команды будет выглядеть так:

```
52: cgroup_skb  tag 7be49e3934a125ba
        loaded_at 2019-03-28T16:46:04-0700  uid 0
        xlated 296B  jited 229B  memlock 4096B  map_ids 52,53
53: cgroup_skb  tag 2a142ef67aaad174
        loaded_at 2019-03-28T16:46:04-0700  uid 0
        xlated 296B  jited 229B  memlock 4096B  map_ids 52,53
54: cgroup_skb  tag 7be49e3934a125ba
        loaded_at 2019-03-28T16:46:04-0700  uid 0
        xlated 296B  jited 229B  memlock 4096B  map_ids 54,55
```

Стоящие слева числа с двоеточием — это идентификаторы программ, мы используем их позже, чтобы выяснить, что представляют собой эти программы. В данном выводе вы также можете увидеть, какие программы запущены в вашей системе. Сейчас система выполняет три программы BPF, подключенные к буферам сокетов контрольных групп. Время запуска программ, скорее всего, будет совпадать с моментом загрузки вашей системы, если они были действительно запущены Systemd. Вы также можете увидеть количество памяти, используемой этими программами в настоящее время, и идентификаторы карт, связанных с ними. Вся эта информация полезна для начала, дальнейшие исследования будут обширнее, так как у нас есть идентификаторы программ.

Вы можете добавить идентификатор программы к предыдущей команде в качестве дополнительного аргумента: `bpftool prog show id 52`. При этом BPFTool покажет вам ту же информацию, которую вы видели ранее, но только для программы с идентификатором 52. Таким образом можно отфильтровать ненужную информацию. Эта команда также поддерживает флаг `--json` для генерации вывода JSON. Вывод JSON очень удобен. Например, такие инструменты, как `jq`, позволят получить более структурированный вывод данных:

```
# bpftool prog show --json id 52 | jq
{
  "id": 52,
  "type": "cgroup_skb",
  "tag": "7be49e3934a125ba",
  "gpl_compatible": false,
  "loaded_at": 1553816764,
  "uid": 0,
  "bytes_xlated": 296,
  "jited": true,
```

```
  "bytes_jited": 229,
  "bytes_memlock": 4096,
  "map_ids": [
    52,
    53
  ]
}
```

Вы также можете выполнить более сложные манипуляции и отфильтровать только ту информацию, которая вас интересует. В следующем примере нам нужны только идентификатор программы BPF, ее тип и время загрузки в ядро:

```
# bpftool prog show --json id 52 | jq -c '[.id, .type, .loaded_at]'
[52,"cgroup_skb",1553816764]
```

Зная идентификатор программы, вы можете получить дамп всей программы, применяя BPFTool. Это может быть полезно, когда нужно отладить байт-код BPF, сгенерированный компилятором:

```
# bpftool prog dump xlated id 52
   0: (bf) r6 = r1
   1: (69) r7 = *(u16 *)(r6 +192)
   2: (b4) w8 = 0
   3: (55) if r7 != 0x8 goto pc+14
   4: (bf) r1 = r6
   5: (b4) w2 = 16
   6: (bf) r3 = r10
   7: (07) r3 += -4
   8: (b4) w4 = 4
   9: (85) call bpf_skb_load_bytes#7151872
   ...
```

Эта программа, которую Systemd загружает в ядро, инспектирует пакетные данные с применением помощника `bpf_skb_load_bytes`.

Если вы хотите получить более наглядное представление о программе, включая переходы инструкций, используйте ключевое слово `visual` в этой команде. Выходные данные будут выведены в формате, который вы можете преобразовать в графическое представление с помощью таких инструментов, как `dotty`, или любой другой программы для отображения графиков:

```
# bpftool prog dump xlated id 52 visual &> output.out
# dot -Tpng output.out -o visual-graph.png
```

Визуальное представление для маленькой программы Hello World показано на рис. 5.1.

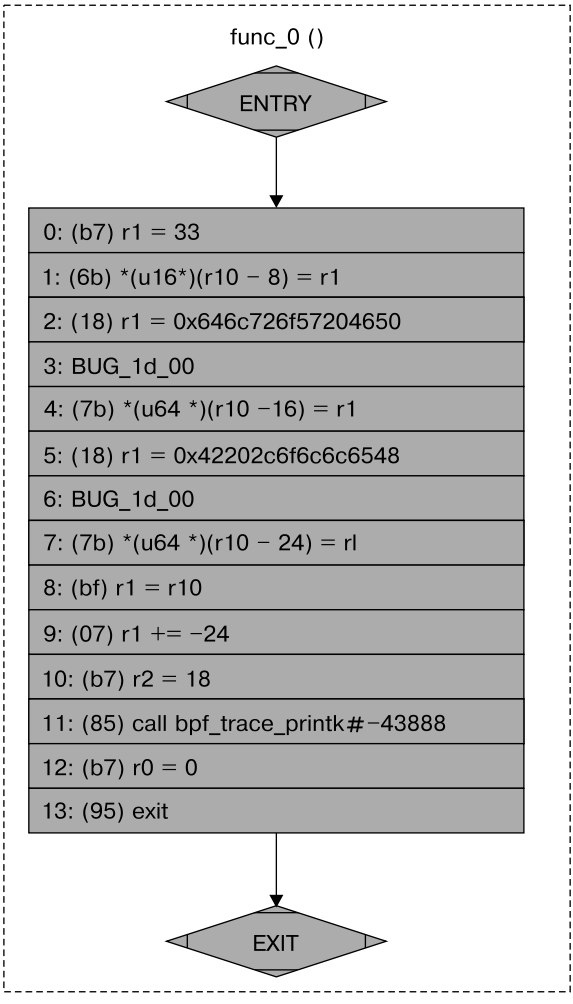

Рис. 5.1. Визуальное представление программы BPF

Если вы используете ядро версии 5.1 или более новое, у вас также будет доступ к статистике времени выполнения. С ее помощью можно увидеть, сколько времени ядро тратит на ваши программы BPF. Эта возможность может быть не включена в вашей системе по умолчанию, поэтому сначала

придется выполнить следующую команду, чтобы ядро узнало, что должно предоставить вам эти данные:

```
# sysctl -w kernel.bpf_stats_enabled=1
```

После включения сбора статистики при запуске BPFTool вы получите данные еще двух счетчиков: общее количество времени, которое ядро потратило на выполнение данной программы (`run_time_ns`), и сколько раз ядро выполняло программу (`run_cnt`):

```
52: cgroup_skb  tag 7be49e3934a125ba  run_time_ns 14397 run_cnt 39
        loaded_at 2019-03-28T16:46:04-0700  uid 0
        xlated 296B  jited 229B  memlock 4096B  map_ids 52,53
```

Но BPFTool не только обеспечивает проверку работы ваших программ, он также позволяет загружать новые программы в ядро и подключать некоторые из них к сокетам и контрольным группам. Например, мы можем загрузить одну из рассмотренных ранее программ и привязать ее к файловой системе BPF с помощью следующей команды:

```
# bpftool prog load bpf_prog.o /sys/fs/bpf/bpf_prog
```

Поскольку программа привязана к файловой системе, она не завершится после запуска и мы сможем увидеть, что она все еще загружена с помощью упомянутой команды `show`:

```
# bpftool prog show
52: cgroup_skb  tag 7be49e3934a125ba
        loaded_at 2019-03-28T16:46:04-0700  uid 0
xlated 296B  jited 229B  memlock 4096B  map_ids 52,53
53: cgroup_skb  tag 2a142ef67aaad174
        loaded_at 2019-03-28T16:46:04-0700  uid 0
        xlated 296B  jited 229B  memlock 4096B  map_ids 52,53
54: cgroup_skb  tag 7be49e3934a125ba
        loaded_at 2019-03-28T16:46:04-0700  uid 0
        xlated 296B  jited 229B  memlock 4096B map_ids 54,55
60: perf_event  name bpf_prog tag c6e8e35bea53af79
        loaded_at 2019-03-28T20:46:32-0700  uid 0
        xlated 112B  jited 115B  memlock 4096B
```

Как видите, BPFTool предоставляет вам богатую информацию о программах, загруженных в ядро, при этом нет необходимости писать и компилировать какой-либо код. Теперь рассмотрим, как работать с картами BPF.

Инспекция карт BPF

Помимо предоставления вам доступа для инспекции программ BPF и управления ими, BPFTool может обеспечить доступ к картам BPF, которые используются этими программами. Команда для отображения всех карт и их фильтрации по идентификаторам аналогична команде `show`, которую вы видели ранее. Вместо того чтобы запрашивать у BPFTool информацию для `prog`, попросим показать нам информацию для карты:

```
# bpftool map show
52: lpm_trie  flags 0x1
        key 8B  value 8B  max_entries 1  memlock 4096B
53: lpm_trie  flags 0x1
        key 20B  value 8B  max_entries 1  memlock 4096B
54: lpm_trie  flags 0x1
        key 8B  value 8B  max_entries 1  memlock 4096B
55: lpm_trie  flags 0x1
        key 20B  value 8B  max_entries 1  memlock 4096B
```

Эти карты соответствуют идентификаторам, которые вы видели прежде в своих программах. Можете фильтровать карты по их идентификатору, так же как фильтровали программы ранее.

Можно использовать BPFTool для создания и обновления карт, а также для вывода списка всех элементов на карте. Для создания новой карты требуется такая же информация, как и при инициализации карты одной из ваших программ. Нужно указать, какой тип карты мы хотим создать, размер ключей и значений и имя карты. Поскольку мы не инициализируем карту одновременно с программой, нужно также привязать ее к файловой системе BPF, чтобы можно было работать с ней позже:

```
# bpftool map create /sys/fs/bpf/counter
    type array key 4 value 4 entries 5 name counter
```

Если вы выведете список карт в системе после выполнения этой команды, то увидите новую карту внизу списка:

```
52: lpm_trie  flags 0x1
        key 8B  value 8B  max_entries 1  memlock 4096B
53: lpm_trie  flags 0x1
        key 20B  value 8B  max_entries 1  memlock 4096B
```

```
54: lpm_trie  flags 0x1
        key 8B  value 8B  max_entries 1  memlock 4096B
55: lpm_trie  flags 0x1
        key 20B  value 8B  max_entries 1  memlock 4096B
56: lpm_trie  flags 0x1
        key 8B  value 8B  max_entries 1  memlock 4096B
57: lpm_trie  flags 0x1
        key 20B  value 8B  max_entries 1  memlock 4096B
58: array  name counter flags 0x0
        key 4B  value 4B  max_entries 5  memlock 4096B
```

После создания карты вы можете обновлять и удалять элементы, как мы это делали в программе BPF.

Помните, что вы не можете удалять элементы из массивов фиксированного размера — разрешается только обновить их. Но можно полностью удалить элементы из других карт, например из хеш-карт.

Если вы хотите добавить в карту новый элемент или обновить существующий, используйте команду обновления карты. Как получить идентификатор карты, показано в предыдущем примере:

```
# bpftool map update id 58 key 1 0 0 0 value 1 0 0 0
```

Если вы попытаетесь обновить элемент с неверным ключом или значением, BPFTool возвратит ошибку:

```
# bpftool map update id 58 key 1 0 0 0 value 1 0 0
Error: value expected 4 bytes got 3
```

BPFTool может дать вам дамп всех элементов на карте, если нужно проверить значения. BPF инициализирует все элементы нулевым значением при создании карт массивов фиксированного размера:

```
# bpftool map dump id 58
key: 00 00 00 00  value: 00 00 00 00
key: 01 00 00 00  value: 01 00 00 00
key: 02 00 00 00  value: 00 00 00 00
key: 03 00 00 00  value: 00 00 00 00
key: 04 00 00 00  value: 00 00 00 00
```

Одна из наиболее мощных возможностей, предлагаемых BPFTool, — то, что вы можете прикреплять предварительно созданные карты к новым программам и заменять карты, которые они будут инициализировать, этими предварительно подобранными картами. Таким образом, вы можете предоставить программам доступ к сохраненным данным с самого начала, даже если не написали программу для чтения карты из файловой системы BPF. Для этого нужно определить карту, которую вы хотите инициализировать при загрузке программы с помощью BPFTool. Можете указать карту по упорядоченному идентификатору, который будет у нее при загрузке программы, например `0` для первой карты, `1` — для второй и т. д. Или по ее имени — обычно это удобнее:

```
# bpftool prog load bpf_prog.o /sys/fs/bpf/bpf_prog_2 \
    map name counter /sys/fs/bpf/counter
```

В этом примере мы присоединяем карту, которую только что создали, к новой программе. Карту заменили ее именем, потому что знаем, что программа инициализирует карту с именем `counter`. Вы также можете использовать позицию индекса карты с ключевым словом `idx`, например, `idx 0`, если это легче запомнить.

Доступ к картам BPF напрямую из командной строки полезен, когда вам нужно отладить передачу сообщений в режиме реального времени. BPFTool предоставляет вам удобный прямой доступ. Можете применять BPFTool не только для самоанализа программ и карт, но и для извлечения гораздо более обширной информации из ядра. Давайте посмотрим, как получить доступ к определенным интерфейсам.

Инспекция программ, подключенных к определенным интерфейсам

Иногда требуется узнать, какие программы подключены к определенным интерфейсам. BPF может загружать программы, работающие на основе контрольных групп, событий Perf и сетевых пакетов. Подкоманды `cgroup`, `perf` и `net` помогут вам отследить подключения по интерфейсам.

Подкоманда `perf` перечисляет все программы, связанные с точками трассировки в системе, такими как kprobes, uprobes и tracepoints. Увидеть этот список можно, выполнив `bpftool perf show`.

Подкоманда `net` перечисляет программы, подключенные к XDP и Traffic Control. Другие подсоединения, такие как фильтры сокетов и программы повторного использования порта, доступны только тогда, когда применен `iproute2`. Вы можете просмотреть подключения к XDP и TC с помощью `bpftool net show`, как это делалось для других объектов BPF.

Наконец, подкоманда `cgroup` перечисляет все программы, прикрепленные к контрольным группам. Она немного отличается от тех, что вы видели раньше. `bpftool cgroup show` требуется путь к контрольной группе, которую вы проверяете. Если вы хотите перечислить все подключения во всех контрольных группах в системе, используйте `bpftool cgroup tree`, как показано в примере:

```
# bpftool cgroup tree
CgroupPath
ID       AttachType      AttachFlags     Name
/sys/fs/cgroup/unified/system.slice/systemd-udevd.service
    5        ingress
    4        egress
/sys/fs/cgroup/unified/system.slice/systemd-journald.service
    3        ingress
    2        egress
/sys/fs/cgroup/unified/system.slice/systemd-logind.service
    7        ingress
    6        egress
```

Благодаря BPFTool вы можете убедиться, что ваши программы правильно подключены к любому интерфейсу в ядре и быстро предоставляют вам визуальный отчет о контрольных группах, Perf и сетевом интерфейсе.

До сих пор мы говорили о том, как использовать различные команды в терминале для отладки программ BPF. Однако помнить все команды непросто, особенно когда они нужны срочно. Далее расскажем, как загрузить несколько команд из текстовых файлов, то есть создать набор сценариев, которые можно держать под рукой, не стараясь запомнить каждую опцию.

Загрузка команд в пакетном режиме

Часто бывает нужно выполнить несколько команд много раз, чтобы проанализировать поведение одной или нескольких систем. Можно составить набор

команд, с которыми вы часто работаете, и применять как часть цепочки инструментов. Пакетный режим BPFTool подойдет, если нет желания вводить эти команды каждый раз повторно.

В пакетном режиме запишите команды, которые хотите выполнить, в файл и запускайте их все сразу. Можно сопроводить этот файл комментариями, начиная строки с #. Однако этот режим выполнения не является атомарным. BPFTool выполняет команды построчно, и в случае сбоя одной из них прервет выполнение, оставив систему в том состоянии, в котором она находилась после выполнения последней успешной команды.

Вот краткий пример файла, который может обрабатываться в пакетном режиме:

```
# Создать новую хеш-карту
map create /sys/fs/bpf/hash_map type hash key 4 value 4 entries 5 name hash_map
# Теперь показать все карты в системе
map show
```

Сохранив эти команды в файле `/tmp/batch_example.txt`, вы сможете загрузить его командой `bpftool batch file /tmp/batch_example.txt`. Когда запустите ее в первый раз, получите вывод, подобный следующему фрагменту:

```
# bpftool batch file /tmp/batch_example.txt
2: lpm_trie  flags 0x1
        key 8B  value 8B  max_entries 1  memlock 4096B
3: lpm_trie  flags 0x1
        key 20B  value 8B  max_entries 1  memlock 4096B
18: hash  name hash_map  flags 0x0
        key 4B  value 4B  max_entries 5  memlock 4096B
processed 2 commands
```

Если попытаетесь запустить команду снова, она отработает без вывода, потому что в системе уже есть карта с именем `hash_map`, так что пакетное выполнение оборвется на первой же строке.

Пакетный режим — одна из наших любимых возможностей BPFTool. Мы рекомендуем хранить пакетные файлы в системе контроля версий, чтобы поделиться ими со своей командой и создать собственный набор утилит. Прежде чем перейти к нашей следующей любимой утилите, давайте посмотрим, как BPFTool может помочь вам лучше понять BPF Type Format.

Отображение информации BTF

BPFTool может отображать информацию о формате типа BPF (BTF) для любого заданного двоичного объекта, если эта информация существует. Как упоминалось в главе 2, BTF аннотирует программные структуры метаданными, чтобы помочь вам отлаживать программы. Например, он может дать вам исходный файл и номера строк для каждой инструкции в программе BPF, когда вы добавляете ключевое слово `linum` в `prog dump`.

Более поздние версии BPFTool включают новую подкоманду `btf`, которая поможет вам разобраться в своих программах. Изначальной целью этой команды являлась визуализация типов структуры. Например, `bpftool btf dump id 54` покажет все типы BFT для программы, загруженной с идентификатором 54.

Например, вы можете использовать BPFTool для точки входа с низкими потерями для любой системы, особенно если вы не работаете с ней ежедневно.

BPFTrace

BPFTrace — язык трассировки высокого уровня для BPF. Он позволяет писать программы BPF с кратким DSL и сохранять их как сценарии, которые для выполнения не требуется компилировать и загружать в ядро вручную. Язык основан на других известных инструментах, таких как awk и DTrace. Если вы знакомы с DTrace и вам всегда не хватало возможности работать с ним в Linux, то BPFTrace станет отличной заменой.

Одним из преимуществ использования BPFTrace над написанием программ непосредственно с помощью BCC или других инструментов BPF является то, что BPFTrace предоставляет множество встроенных функций, которые вам не нужно реализовывать самостоятельно, таких как компоновка информации и создание гистограмм. В то же время возможности языка, который применяет BPFTrace, намного меньше, и он не позволит реализовать сложные программы. В этом разделе мы покажем наиболее важные аспекты языка. Рекомендуем посетить репозиторий BPFTrace в GitHub (https://github.com/iovisor/bpftrace), чтобы подробнее прочитать об этом.

Установка

Установить BPFTrace можно несколькими способами, хотя его разработчики рекомендуют использовать один из готовых пакетов для конкретного дистрибутива Linux. В своем хранилище они поддерживают актуальность документации со всеми вариантами установки и предварительными условиями для вашей системы. Вы найдете инструкции в справочнике по установке (oreil.ly/h9Pha).

Справочник по языку

BPFTrace не содержит сложных команд. Синтаксис краток. Можно разделить команды на три раздела: заголовок, блоки действий и окончание. Заголовок — это специальный блок, который BPFTrace выполняет при загрузке программы. Обычно он используется для печати в верхней части вывода какой-то информации, например преамбулы. Точно так же окончание — это специальный блок, который BPFTrace выполняет один раз перед завершением программы. Заголовок и окончание — необязательные разделы программы BPFTrace. Однако она должна иметь хотя бы один *блок действий*. В них мы указываем зонды, которые хотим отслеживать, и действия, которые выполняем, когда ядро запускает события для этих зондов. Следующий простой пример показывает, как используются эти три раздела:

```
BEGIN
{
  printf("starting BPFTrace program\n")
}

kprobe:do_sys_open
{
  printf("opening file descriptor: %s\n", str(arg1))
}

END
{
  printf("exiting BPFTrace program\n")
}
```

Раздел заголовка всегда помечается ключевым словом `BEGIN`, а раздел окончания — ключевым словом `END`. Эти ключевые слова зарезервированы

в BPFTrace. Идентификаторы блоков действий определяют зонд, к которому вы хотите присоединить действие BPF. В предыдущем примере мы выводили строку в журнал каждый раз, когда ядро открывало файл.

В предыдущих примерах, помимо определения разделов программы, мы видим несколько дополнительных деталей синтаксиса языка. BPFTrace предоставляет помощники, которые преобразуются в код BPF при компиляции программы. Помощник `printf` — это оболочка для функции C `printf`, которая печатает из программы то, что вам нужно. `str` — встроенный помощник, который переводит указатель C в его строковое представление. Многие функции ядра получают указатели на символы в качестве аргументов, этот помощник переводит данные указатели в строки.

BPFTrace можно считать динамическим языком в том смысле, что он не знает количества аргументов, которые может получить зонд при выполнении ядром. Вот почему BPFTrace предоставляет помощники аргументов для доступа к информации, обрабатываемой ядром. BPFTrace генерирует эти помощники динамически, в зависимости от количества аргументов, которые получает блок, и вы можете получить доступ к информации по ее позиции в списке аргументов. В предыдущем примере `arg1` является ссылкой на второй аргумент в системном вызове `open`, который ссылается на путь к файлу.

Чтобы выполнить этот пример, вы можете сохранить его в файл и запустить BPFTrace с путем к файлу в качестве первого аргумента:

```
# bpftrace /tmp/example.bt
```

Язык BPFTrace разработан с учетом сценариев. В предыдущих примерах вы видели краткую версию языка, так что уже знакомы с ней. Но можно разместить многие программы, которые вы можете написать с помощью BPFTrace, в одной строке. Для выполнения этих однострочных программ вам не нужно хранить их в файлах — можете запустить их с опцией `-e` при запуске BPFTrace. Например, приведенный ранее пример счетчика может быть однострочным, если сделать так:

```
# bpftrace -e "kprobe:do_sys_open { @opens[str(arg1)] = count() }"
```

Теперь, когда вы немного больше узнали о языке BPFTrace, посмотрим, как его использовать, на примере нескольких сценариев.

Фильтрация

Запуская код из предыдущего примера, вы, вероятно, получаете список файлов, которые система постоянно открывает, пока не будет нажато сочетание клавиш Ctrl+C для выхода из программы. Так происходит потому, что мы указали BPF печатать каждый дескриптор файла, который открывает ядро. Существуют ситуации, когда нужно выполнить блок действий только в определенных условиях. В BPFTrace это называется *фильтрацией*.

Можете использовать по одному фильтру на каждый блок действий. Они считаются блоками действий, но действие не выполняется, если фильтр возвращает ложное значение. У них также есть доступ к остальной части языка, включая аргументы зондов и помощники. Эти фильтры заключены в две косые черты и стоят после заголовка действия:

```
kprobe:do_sys_open /str(arg1) == "/tmp/example.bt"/
{
  printf("opening file descriptor: %s\n", str(arg1))
}
```

В этом примере мы указываем, что блок действий должен выполняться только тогда, когда файл, открывающий ядро, является файлом, который мы используем для хранения этого примера. Если вы запустите программу с новым фильтром, то увидите, что она печатает заголовок и на этом вывод заканчивается. Так происходит потому, что каждый файл, который запускал это действие раньше, теперь пропускается из-за применения нового фильтра. Если вы несколько раз откроете файл примера в другом терминале, то увидите, как ядро выполняет действие, когда фильтр соответствует пути к нашему файлу:

```
# bpftrace /tmp/example.bt
Attaching 3 probes...
starting BPFTrace program
opening file descriptor: /tmp/example.bt
opening file descriptor: /tmp/example.bt
opening file descriptor: /tmp/example.bt
^Cexiting BPFTrace program
```

Возможности фильтрации BPFTrace очень полезны для сокрытия ненужной информации и позволяют выводить только данные, которые вам действительно необходимы. Далее поговорим о том, как BPFTrace облегчает работу с картами.

Динамическое отображение

Динамические ассоциации карт — удобная особенность BPFTrace. Карты BPF можно генерировать динамически и использовать для многих операций, которые вы видели в книге. Все ассоциации карт начинаются с символа `@`, за ним следует имя карты, которую вы хотите создать. Можно также связать элементы обновления в этих картах, присвоив им значения.

Для примера, с которого начался этот раздел, мы можем собрать сведения о том, как часто система открывает определенные файлы. Для этого нужно подсчитать, сколько раз ядро выполняет системный вызов `open` для определенного файла, а затем сохранить эти данные в карте. Чтобы идентифицировать эти сведения, мы можем использовать путь к файлу в качестве ключа карты. Вот как в таком случае будет выглядеть блок действий:

```
kprobe:do_sys_open
{
  @opens[str(arg1)] = count()
}
```

Снова запустив программу, вы получите примерно такой вывод:

```
# bpftrace /tmp/example.bt
Attaching 3 probes...
starting BPFTrace program
^Cexiting BPFTrace program

@opens[/var/lib/snapd/lib/gl/haswell/libdl.so.2]: 1
@opens[/var/lib/snapd/lib/gl32/x86_64/libdl.so.2]: 1
...
@opens[/usr/lib/locale/en.utf8/LC_TIME]: 10
@opens[/usr/lib/locale/en_US/LC_TIME]: 10
@opens[/usr/share/locale/locale.alias]: 12
@opens[/proc/8483/cmdline]: 12
```

Как видите, BPFTrace печатает содержимое карты, когда останавливает выполнение программы. И, как мы и ожидали, собирает сведения о том, как часто ядро открывает файлы в системе. По умолчанию BPFTrace всегда выводит содержимое каждой карты, которую создает при завершении работы. Вам не нужно указывать, что требуется напечатать карту, — всегда предполагается, что это будет сделано по умолчанию. Вы можете изменить такое поведение, очистив карту внутри блока `END` с помощью встроенной функции `clear`. Это работает, потому что печать карт всегда происходит после выполнения завершающего блока.

Динамическое отображение BPFTrace очень удобно. Оно позволяет избежать множества рутинных задач, необходимых для работы с картами, и помогает легко собирать данные.

BPFTrace — это мощный инструмент для выполнения повседневных задач. Его язык сценариев обеспечивает гибкость, достаточную для доступа ко всем аспектам системы, не требуя вручную компилировать и загружать BPF-программы в ядро, что поможет отследить и устранить проблемы в системе с самого начала работы. Обратитесь к справочному руководству в своем репозитории GitHub, чтобы узнать, как воспользоваться всеми встроенными возможностями, такими как автоматические гистограммы и агрегирование трассировки стека.

В следующем разделе рассмотрим, как применять BPFTrace внутри Kubernetes.

kubectl-trace

`kubectl-trace` — фантастический плагин для командной строки Kubernetes, `kubectl`. Он помогает планировать программы BPFTrace в вашем кластере Kubernetes, не требуя устанавливать какие-либо дополнительные пакеты или модули. Это достигается планированием задания Kubernetes с помощью образа контейнера, в котором есть все необходимое для запуска уже установленной программы. Данное изображение называется `trace-runner`, оно доступно также в общем реестре Docker.

Установка

Необходимо установить `kubectl-trace` из исходного репозитория, используя набор инструментов Go, потому что его разработчики не предоставляют никаких бинарных пакетов:

```
go get -u github.com/iovisor/kubectl-trace/cmd/kubectl-trace
```

Система плагинов `kubectl` автоматически обнаружит это дополнение после того, как набор инструментов Go скомпилирует программу и пропишет путь к ней. `kubectl-trace` автоматически загружает образы Docker, необходимые при первом запуске в кластере.

Инспекция узлов Kubernetes

Вы можете использовать `kubectl-trace` для узлов и модулей, в которых выполняются контейнеры, а также для процессов, реализуемых в этих контейнерах. В первом случае можно запустить практически любую BPF-программу, а во втором — только те программы, которые связывают с этими процессами зонды пользовательского пространства.

Если вы хотите запустить программу BPF на определенном узле, вам нужен правильный идентификатор, чтобы Kubernetes планировал задание в соответствующем месте. После получения этого идентификатора запуск программы аналогичен запуску программ, которые вы видели ранее. Запустим нашу однострочную систему для подсчета открытий файлов:

```
# kubectl trace run node/node_identifier -e \
  "kprobe:do_sys_open { @opens[str(arg1)] = count() }"
```

Как видите, программа точно такая же, как и прежде, но мы вводим команду `kubectl trace run`, чтобы запланировать ее на конкретном узле кластера. Воспользуемся синтаксисом `node/...`, чтобы сообщить `kubectl-trace`, что мы нацеливаемся на узел в кластере. Если требуется нацелиться на конкретный модуль, меняем `node/` на `pod/`.

Запуск программы в определенном контейнере требует более длинного синтаксиса. Сначала посмотрим на пример и пройдемся по нему:

```
# kubectl trace run pod/pod_identifier -n application_name -e <<PROGRAM
uretprobe:/proc/$container_pid/exe:"main.main" {
  printf("exit: %d\n", retval)
}
PROGRAM
```

В этой команде нужно выделить две интересные вещи. Во-первых, нам требуется имя приложения, запущенного в контейнере, чтобы найти его процесс. В нашем примере это соответствует `application_name`. Далее захочется использовать имя двоичного файла, который выполняется в контейнере, например `nginx` или `memcached`. Обычно контейнеры запускают только один процесс, но это дает дополнительные гарантии того, что мы присоединяем нашу программу к правильному процессу. Вторым аспектом, который стоит выделить, является включение `$container_pid` в программу BPF. Это не по-

мощник BPFTrace, а заполнитель, который `kubectl-trace` применяет в качестве замены для идентификатора процесса. Перед запуском программы BPF трассировщик заменяет заполнитель соответствующим идентификатором и присоединяет программу к правильному процессу.

Если вы запускаете Kubernetes в производственной среде, `kubectl-trace` намного упростит вашу жизнь, когда вам нужно будет проанализировать поведение ваших контейнеров.

В этом и предыдущих разделах мы сосредоточились на инструментах, которые помогут вам более эффективно запускать программы BPF даже в контейнерных средах. В следующем разделе поговорим о хорошем инструменте для интеграции сбора данных из программ BPF с Prometheus — известной системой мониторинга с открытым исходным кодом.

eBPF Exporter

eBPF Exporter — это инструмент, который позволяет вам экспортировать пользовательские метки трассировки BPF в Prometheus. Prometheus — это масштабируемая система мониторинга и оповещения. Одним из ключевых факторов, который отличает Prometheus от других систем мониторинга, является то, что он задействует стратегию извлечения для получения метрик, вместо того чтобы ожидать, что клиент предоставит их ему. Это позволяет пользователям создавать собственные экспортеры, которые могут собирать метрики из любой системы, а Prometheus будет извлекать их по четко определенной схеме API. eBPF Exporter реализует этот API для получения метрик трассировки из программ BPF и их импорта в Prometheus.

Установка

Хотя eBPF Exporter предлагает бинарные пакеты, мы рекомендуем устанавливать его из исходных кодов, потому что часто релизы запаздывают. К тому же сборка из исходных кодов дает вам доступ к новым функциям, созданным на основе современных версий BCC — коллекции компилятора BPF.

Чтобы установить eBPF Exporter из исходных кодов, на вашем компьютере уже должен быть установлен набор инструментов BCC и Go. Выполнив эти

требования, вы можете использовать Go для загрузки и сборки двоичного файла:

```
go get -u github.com/cloudflare/ebpf_exporter/...
```

Экспорт метрик из BPF

eBPF Exporter настраивается с помощью файлов YAML, в которых вы можете указать метрики, которые хотите собирать из системы, программу BPF, генерирующую их, и то, как их переводить в Prometheus. Когда Prometheus отправляет запрос в eBPF Exporter для получения метрик, этот инструмент преобразует информацию, которую собирают программы BPF, в значения метрик. К счастью, eBPF Exporter объединяет множество программ, собирающих очень полезную информацию из системы, например инструкции, выполняемые в каждом цикле, и частоту обращений в кэш ЦП.

Простой файл конфигурации для eBPF Exporter состоит из трех основных разделов. В первом из них вы определяете метрики, которые Prometheus должен извлечь из системы. Здесь вы переводите данные, собранные в карты BPF, в метрики, понятные Prometheus. Далее приведен пример таких переводов:

```
programs:
  - name: timers
    metrics:
      counters:
        - name: timer_start_total
          help: Timers fired in the kernel
          table: counts
          labels:
            - name: function
              size: 8
              decoders:
                - name: ksym
```

Мы определяем метрику `timer_start_total`, которая собирает частоту запуска таймера ядром. Кроме того, указываем, что хотим получить эту информацию из карты BPF, которая называется `counts`. Наконец, определяем функцию перевода для ключей карты. Это необходимо, потому что ключи карты обычно являются указателями на информацию, а мы хотим отправить Prometheus реальные имена функций.

Во втором разделе примера описаны зонды, к которым мы хотим присоединить программу BPF. В данном случае требуется отследить запуск таймера, для этого мы используем точку трассировки `timer:timer_start`:

```
tracepoints:
  timer:timer_start: tracepoint__timer__timer_start
```

Здесь мы указываем eBPF Exporter, что хотим присоединить функцию BPF `tracepoint__timer__timer_start` к конкретной точке трассировки. Посмотрим, как объявить эту функцию:

```
code: |
  BPF_HASH(counts, u64);
  // Генерирует функцию tracepoint__timer__timer_start
  TRACEPOINT_PROBE(timer, timer_start) {
      counts.increment((u64) args->function);
      return 0;
}
```

Программа BPF встроена в файл YAML. Это одна из самых нелюбимых нами частей данного инструмента, потому что для YAML особенно важны пробелы, но он работает с небольшими программами, подобными рассматриваемой. eBPF Exporter использует BCC для компиляции программ, поэтому у нас есть доступ ко всем его макросам и помощникам. В предыдущем фрагменте задействован макрос `TRACEPOINT_PROBE` для генерации последней функции, которую мы прикрепим к точке трассировки `tracepoint__timer__timer_start`.

Cloudflare задействует eBPF Exporter для мониторинга показателей во всех своих центрах обработки данных. Компания позаботилась о том, чтобы объединить наиболее распространенные метрики, которые вы хотите экспортировать из своих систем. Но, как видите, список довольно легко расширить с помощью новых метрик.

Резюме

В этой главе мы рассмотрели некоторые из наших любимых инструментов для системного анализа. Они достаточно универсальны, так что нужно иметь их под рукой, чтобы отследить любую аномалию в системе. Как видите, все эти инструменты инкапсулируют концепции, которые мы рассматривали в предыдущих главах, чтобы помочь вам использовать BPF, даже если среда

не готова к такому. Это одно из множества преимуществ BPF перед другими инструментами анализа. Поскольку любое современное ядро Linux включает виртуальную машину BPF, вы можете надстраивать новые инструменты, которые используют эти действительно хорошие возможности.

Есть много других инструментов, которые применяют BPF для аналогичных целей, таких как Cilium и Sysdig, рекомендуем попробовать поработать с ними.

Эта глава и глава 4 касались в основном системного анализа и трассировки, но с BPF вы можете сделать гораздо больше. В дальнейшем углубимся в его сетевые возможности. Мы покажем, как анализировать трафик в любой сети и использовать BPF для управления сообщениями в вашей сети.

6 Сетевое взаимодействие в Linux и BPF

С точки зрения сетевого взаимодействия мы используем программы BPF для двух основных целей — захвата пакетов и фильтрации. Это означает, что программа пользовательского пространства может прикрепить фильтр к любому сокету, извлечь информацию о пакетах, проходящих через него, и разрешить, или запретить, или перенаправить определенные типы пакетов в зависимости от того, как они определены на этом уровне.

Цель этой главы — объяснить, как программы BPF могут взаимодействовать со структурой буфера сокетов на разных этапах пути к сетевым данным в сетевом стеке ядра Linux. Пока определим два наиболее общих случая:

- программы, связанные с сокетами;
- программы, написанные для основанного на BPF классификатора для управления трафиком.

Socket Buffer, также называемая SKB или sk_buff, — это структура ядра, которая создается и используется для каждого отправленного или полученного пакета. Читая SKB, вы можете передавать или отбрасывать пакеты и заполнять карты BPF для создания статистики и метрик потока, иллюстрирующих трафик.

Кроме того, некоторые программы BPF позволяют манипулировать SKB и, соответственно, преобразовывать конечные пакеты, чтобы перенаправить или изменить их. Например, в системе только с IPv6 вы можете написать программу, которая преобразует все полученные пакеты из IPv4 в IPv6 с помощью SKB.

Понимание различий между разными видами программ, которые мы можем написать, и того, как разные программы позволяют достичь одной и той же

цели, является ключом к пониманию BPF и eBPF в сети. В следующем разделе рассмотрим первые два способа фильтрации на уровне сокетов: с помощью классических фильтров BPF и программ eBPF, подключенных к сокетам.

BPF и фильтрация пакетов

Как уже говорилось, фильтры BPF и программы eBPF являются основными вариантами применения программ BPF в контексте сетевого взаимодействия, однако изначально программы BPF были синонимом фильтрации пакетов.

Фильтрация пакетов по-прежнему является одним из наиболее важных участков. В свое время она была расширена от классического BPF (cBPF) до современного eBPF в Linux 3.19 добавлением функций, связанных с картами, к типу программы фильтра `BPF_PROG_TYPE_SOCKET_FILTER`.

Фильтры используются в основном в следующих трех сценариях высокого уровня.

- Отбрасывание трафика в реальном времени (например, разрешение только трафика протокола пользовательских дейтаграмм (UDP) и отбрасывание чего-либо еще).
- Наблюдение в реальном времени за отфильтрованным набором пакетов, поступающих в работающую систему.
- Ретроспективный анализ сетевого трафика, захваченного в работающей системе, например, в *формате pcap*.

Термин pcap происходит от соединения двух слов: packet («пакет») и capture («захват»). Формат pcap реализован как специфичный для домена API для захвата пакетов в библиотеке Packet Capture Library (libpcap). Этот формат полезен в сценариях отладки, когда нужно сохранить набор пакетов, захваченных в действующей системе, непосредственно в файл для последующего анализа с помощью инструмента, который может читать поток пакетов, экспортируемых в формате pcap.

В следующих разделах мы рассмотрим два способа применения концепции фильтрации пакетов в программах BPF. Сначала покажем, как обычный широко распространенный инструмент, такой как `tcpdump`, выступает в ка-

честве высокоуровневого интерфейса для программ BPF, работающих как фильтры. Затем напишем и загрузим собственную программу, используя тип BPF-программы `BPF_PROG_TYPE_SOCKET_FILTER`.

Выражения tcpdump и BPF

Одним из инструментов командной строки для анализа и наблюдения трафика в реальном времени, о котором знают почти все, является `tcpdump`. По сути, это интерфейс для `libpcap`, который позволяет пользователю определять высокоуровневые выражения фильтрации. `tcpdump` выполняет чтение пакетов из выбранного вами сетевого (или любого другого) интерфейса, а затем записывает содержимое полученных пакетов в стандартный вывод или файл. Затем поток пакетов может быть отфильтрован с применением синтаксиса фильтра pcap. Синтаксис фильтра pcap — это предметно-ориентированный язык (DSL) для фильтрации пакетов с использованием высокоуровневого набора выражений, созданных с помощью набора примитивов, которые, как правило, легче для запоминания, чем ассемблер BPF. В этой главе не описываются все возможные примитивы и выражения в синтаксисе фильтра pcap, поскольку весь набор можно найти в `man 7 pcap-filter`, но мы рассмотрим некоторые примеры, чтобы вы могли почувствовать его мощь.

Представим, что мы работаем на машине Linux, на которой имеется веб-сервер на порте 8080. Этот веб-сервер не регистрирует запросы, и мы хотим узнать, получает ли он их на самом деле и как, потому что клиент обслуживаемого приложения жалуется на невозможность получить ответ при просмотре страницы продуктов. На данный момент мы знаем только, что он подключается к одной из наших страниц, где представлены продукты, используя веб-приложение, обслуживаемое этим веб-сервером. Но, как это обычно бывает, понятия не имеем, что может быть причиной проблемы, потому что конечные пользователи обычно не собираются отлаживать сервисы за нас. Вдобавок, к сожалению, мы не позаботились о какой-либо стратегии ведения журналов или отчетов об ошибках, поэтому совершенно не понимаем, в чем кроется проблема. К счастью, есть инструмент, способный прийти нам на помощь! Это `tcpdump`, который можно настроить для фильтрации только пакетов IPv4, проходящих в нашей системе и использующих протокол управления передачей (TCP) через порт 8080. Так мы сможем проанализировать трафик веб-сервера и выявить ошибочные запросы.

Отфильтровать пакеты с помощью `tcpdump` мы можем командой:

```
# tcpdump -n 'ip and tcp port 8080'
```

Рассмотрим, что здесь происходит:

- `-n` указывает `tcpdump` не преобразовывать адреса в соответствующие имена, так как мы хотим видеть именно адреса источника и получателя;
- `ip and tcp port 8080` — это выражение фильтра pcap, которое `tcpdump` будет использовать для фильтрации пакетов. `ip` означает IPv4, `and` является соединением для задания более сложного фильтра, что позволяет добавлять больше выражений для сопоставления. Затем мы указываем, что нас интересуют только пакеты TCP, поступающие в порт 8080 или из него, с помощью `tcp port 8080`. В данном конкретном случае лучше было бы задействовать выражение `tcp dst port 8080`, потому что нас интересуют только пакеты, чей порт назначения — 8080, а не те, что выходят из него.

Результат работы этой команды будет примерно таким (без лишних частей, например полных TCP-рукопожатий):

```
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on wlp4s0, link-type EN10MB (Ethernet), capture size 262144 bytes
12:04:29.593703 IP 192.168.1.249.44206 > 192.168.1.63.8080: Flags [P.],
   seq 1:325, ack 1, win 343,
   options [nop,nop,TS val 25580829 ecr 595195678],
   length 324: HTTP: GET / HTTP/1.1
12:04:29.596073 IP 192.168.1.63.8080 > 192.168.1.249.44206: Flags [.],
   seq 1:1449, ack 325, win 507,
   options [nop,nop,TS val 595195731 ecr 25580829],
   length 1448: HTTP: HTTP/1.1 200 OK
12:04:29.596139 IP 192.168.1.63.8080 > 192.168.1.249.44206: Flags [P.],
   seq 1449:2390, ack 325, win 507,
   options [nop,nop,TS val 595195731 ecr 25580829],
   length 941: HTTP
12:04:46.242924 IP 192.168.1.249.44206 > 192.168.1.63.8080: Flags [P.],
   seq 660:996, ack 4779, win 388,
   options [nop,nop,TS val 25584934 ecr 595204802],
   length 336: HTTP: GET /api/products HTTP/1.1
12:04:46.243594 IP 192.168.1.63.8080 > 192.168.1.249.44206: Flags [P.],
   seq 4779:4873, ack 996, win 503,
   options [nop,nop,TS val 595212378 ecr 25584934],
   length 94: HTTP: HTTP/1.1 500 Internal Server Error
12:04:46.329245 IP 192.168.1.249.44234 > 192.168.1.63.8080: Flags [P.],
   seq 471:706, ack 4779, win 388,
```

```
   options [nop,nop,TS val 25585013 ecr 595205622],
   length 235: HTTP: GET /favicon.ico HTTP/1.1
12:04:46.331659 IP 192.168.1.63.8080 > 192.168.1.249.44234: Flags [.],
   seq 4779:6227, ack 706, win 506,
   options [nop,nop,TS val 595212466 ecr 25585013],
   length 1448: HTTP: HTTP/1.1 200 OK
12:04:46.331739 IP 192.168.1.63.8080 > 192.168.1.249.44234: Flags [P.],
   seq 6227:7168, ack 706, win 506,
   options [nop,nop,TS val 595212466 ecr 25585013],
   length 941: HTTP
```

Теперь ситуация во многом прояснилась! У нас есть множество запросов, которые возвращают код состояния `200 OK`, и еще один — с кодом внутренней ошибки `500 Internal Server Error` в конечной точке `/api/products`. Клиент прав: у нас проблема с перечислением продуктов!

Вы можете спросить: какое отношение средства фильтрации pcap и `tcpdump` имеют к программам BPF, если у них собственный синтаксис? Фильтры pcap в Linux скомпилированы в программы BPF! И, поскольку `tcpdump` использует фильтры pcap для фильтрации, это означает, что каждый раз, выполняя `tcpdump` с применением фильтра, вы фактически компилируете и загружаете программу BPF для фильтрации пакетов. К счастью, передав `tcpdump` флаг `-d`, вы можете вывести инструкции BPF, которые будут загружены с помощью указанного фильтра:

```
tcpdump  -d  'ip and tcp port 8080'
```

Фильтр такой же, как и в предыдущем примере, но благодаря флагу `-d` вывод представляет собой набор инструкций ассемблера BPF.

Вывод команды будет следующим:

```
(000) ldh     [12]
(001) jeq     #0x800        jt 2    jf 12
(002) ldb     [23]
(003) jeq     #0x6          jt 4    jf 12
(004) ldh     [20]
(005) jset    #0x1fff       jt 12   jf 6
(006) ldxb    4*([14]&0xf)
(007) ldh     [x + 14]
(008) jeq     #0x1f90       jt 11   jf 9
(009) ldh     [x + 16]
(010) jeq     #0x1f90       jt 11   jf 12
(011) ret     #262144
(012) ret     #0
```

Проанализируем полученное.

- `ldh [12]`. Загрузить (`ld`) половинное слово (`h`) (16 бит) из аккумулятора со смещением 12, что является полем Ethertype (рис. 6.1).

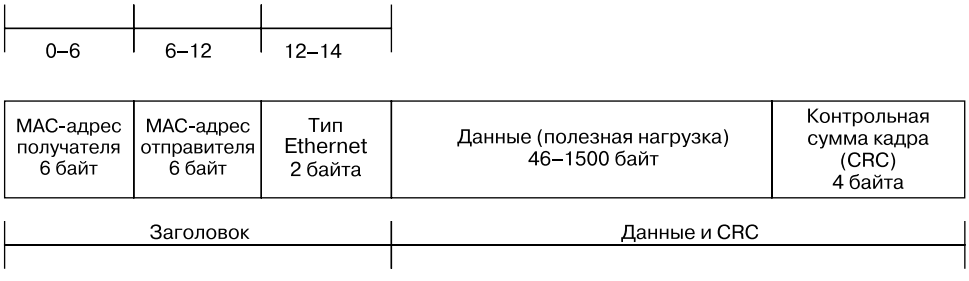

Рис. 6.1. Структура кадра Ethernet канального уровня

- `jeq #0x800 jt 2 jf 12`. Выполнить условный переход: проверить, равно ли значение Ethertype из предыдущей инструкции `0x800`, что является идентификатором для IPv4, и затем использовать пункты назначения перехода — `2`, если истина (`jt`), и `12` — если ложь (`jf`). Так что обработка будет продолжена до следующей инструкции, если интернет-протокол — IPv4, в ином случае инструкция перейдет в конец и вернет ноль.
- `ldb [23]`. Эта инструкция загрузит поле протокола более высокого уровня из кадра IP, который может быть найден по смещению `23`. Смещение `23` определяется суммированием 14 байт заголовков в кадре канального уровня Ethernet (см. рис. 6.1) и позиции протокола в заголовке IPv4, которая равна 9: 14 + 9 = 23.
- `jeq #0x6 jt 4 jf 12`. Снова условный переход. В данном случае убеждаемся, что предыдущий извлеченный протокол равен `0 x 6`, что соответствует TCP. Если это так, переходим к следующей инструкции (`4`), если не так — переходим к концу (`12`) и отбрасываем пакет.
- `ldh [20]`. Еще одна инструкция загрузки половинного слова — в данном случае загрузка суммы значений смещения пакета и смещения фрагмента из заголовка IPv4.
- `jset #0x1fff jt 12 6`. Инструкция `jset` перейдет на `12`, если какие-либо данные, найденные в смещении фрагмента, верны, в ином случае перейдет на `6`, что является следующей инструкцией. Смещение после инструкции

`jset 0x1fff` приказывает инструкции рассматривать только последние 13 байт данных (в расширенном виде `0001 1111 1111 1111`).

- `ldxb 4*([14]&0xf)`. В `x` загружается (`ld`) то, чем является `b`. Эта инструкция загрузит в `x` значение длины заголовка IP.
- `ldh [x + 14]`. Еще одна инструкция загрузки полуслова, которая получит значение со смещением (`x + 14`), то есть длину IP-заголовка + 14, что в пакете соответствует расположению порта источника.
- `jeq #0x1f90 jt 11 jf 9`. Если значение в (`x + 14`) равно `0x1f90` (`8080` в десятичном виде), то есть исходный порт `8080`, то инструкция перейдет к `11`, а если это не так, проверит, находится ли пункт назначения на порте `8080`, перейдя на `9`.
- `ldh [x + 16]`. Еще одна инструкция загрузки половинного слова, которая получает значение по смещению (`x + 16`), что соответствует местоположению порта назначения в пакете.
- `jeq #0x1f90 jt 11 jf 12`. Еще один условный переход. На этот раз проверяется, равен ли пункт назначения `8080`: если это так — перейти к `11`, если нет — перейти к `12` и отбросить пакет.
- `ret #262144`. Когда эта инструкция достигнута, совпадение найдено и нужно вернуть соответствующую длину (по умолчанию 262 144 байта). Этот параметр можно настроить с помощью ключа `-s` в `tcpdump`.

Приведем «правильный» пример. Как мы говорили, для нашего веб-сервера нужно учитывать только пакеты, у которых 8080 — порт назначения, а не источник, поэтому в фильтре `tcpdump` следует указать это значение в поле `dst`:

```
tcpdump -d 'ip and tcp dst port 8080'
```

В этом случае дамп набора инструкций такой же, как в предыдущем примере, но, как видите, в нем отсутствует часть, которая отвечала за поиск пакетов, для которых порт 8080 — источник. Действительно, здесь нет `ldh [x + 14]` и соответствующей инструкции `jeq #0x1f90 jt 11 jf 9`:

```
(000) ldh      [12]
(001) jeq      #0x800           jt 2    jf 10
(002) ldb      [23]
(003) jeq      #0x6             jt 4    jf 10
(004) ldh      [20]
```

```
(005) jset    #0x1fff       jt 10   jf 6
(006) ldxb    4*([14]&0xf)
(007) ldh     [x + 16]
(008) jeq     #0x1f90       jt 9    jf 10
(009) ret     #262144
(010) ret     #0
```

Помимо простого анализа сгенерированного ассемблерного кода из `tcpdump`, как мы это делали, вы можете написать собственный код для фильтрации сетевых пакетов. Оказывается, самая большая проблема в этом случае — отладить выполнение кода, чтобы убедиться, что он соответствует вашим ожиданиям. Для этого в дереве исходников ядра в `tools/bpf` есть инструмент `bpf_dbg.c`, который, по сути, является отладчиком, позволяющим загрузить программу и файл `.pcap` для пошагового выполнения.

tcpdump может также читать напрямую из файла .pcap и применять к нему фильтры BPF.

Фильтрация пакетов для сырых сокетов

Тип программы `BPF_PROG_TYPE_SOCKET_FILTER` позволяет присоединить программу BPF к сокету. Все полученные сокетом пакеты будут переданы программе в виде структуры `sk_buff`, и затем программа сможет решить, следует их отбрасывать или разрешать. Программы такого вида также имеет возможность доступа к картам и работы в них.

Рассмотрим на примере, как можно использовать такую программу BPF.

Целью программы-примера является подсчет количества пакетов TCP, UDP и ICMP, передаваемых через интересующий нас интерфейс. Нам нужны:

- BPF-программа, которая может видеть потоки пакетов;
- код для загрузки программы и ее подключения к сетевому интерфейсу;
- сценарий для компиляции программы и запуска загрузчика.

На этом этапе можно написать программу BPF двумя способами: в виде кода C, который затем компилируется в файл *ELF*, или непосредственно

в виде сборки BPF. Мы решили использовать код на C, чтобы показать абстракцию более высокого уровня и способ применения Clang для компиляции программы. Важно отметить, что для создания программы мы берем заголовки и помощники, доступные только в дереве исходных кодов ядра Linux, поэтому первое, что нужно сделать, — получить их копию с помощью Git. Чтобы избежать коллизий, воспользуйтесь тем же коммитом SHA, с помощью которого создан этот пример:

```
export KERNEL_SRCTREE=/tmp/linux-stable
git clone git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git
  $KERNEL_SRCTREE
cd $KERNEL_SRCTREE
git checkout 4b3c31c8d4dda4d70f3f24a165f3be99499e0328
```

Для поддержки BPF вам потребуется clang> = 3.4.0 с llvm> = 3.7.1. Чтобы проверить, поддерживается ли BPF в вашей установке, используйте команду llc -version и посмотрите, есть ли у нее цель BPF.

Теперь, когда вы разбираетесь в фильтрации сокетов, можно приступить к BPF-программе типа `socket`.

Программа BPF

Основное, что должна сделать программа BPF, — получить доступ к пакету, который она получает. Проверьте тип протокола: TCP, UDP или ICMP, а затем увеличьте значение счетчика в массиве карты на конкретном ключе для найденного протокола.

Для этой программы мы собираемся воспользоваться механизмом загрузки, который анализирует ELF-файлы, задействуя помощники, находящиеся в `samples/bpf/bpf_load.c` в дереве исходного кода ядра. Функция загрузки `load_bpf_file` способна распознавать некоторые конкретные заголовки разделов ELF и связывать их с соответствующими типами программ. Вот как выглядит этот код:

```
bool is_socket = strncmp(event, "socket", 6) == 0;
bool is_kprobe = strncmp(event, "kprobe/", 7) == 0;
bool is_kretprobe = strncmp(event, "kretprobe/", 10) == 0;
```

```
bool is_tracepoint = strncmp(event, "tracepoint/", 11) == 0;
bool is_raw_tracepoint = strncmp(event, "raw_tracepoint/", 15) == 0;
bool is_xdp = strncmp(event, "xdp", 3) == 0;
bool is_perf_event = strncmp(event, "perf_event", 10) == 0;
bool is_cgroup_skb = strncmp(event, "cgroup/skb", 10) == 0;
bool is_cgroup_sk = strncmp(event, "cgroup/sock", 11) == 0;
bool is_sockops = strncmp(event, "sockops", 7) == 0;
bool is_sk_skb = strncmp(event, "sk_skb", 6) == 0;
bool is_sk_msg = strncmp(event, "sk_msg", 6) == 0;
```

Первое, что делает код, — это создает ассоциацию между заголовком раздела и внутренней переменной, например, как для `SEC("socket")`, в итоге получим `bool is_socket = true`.

Дальше в этом же файле мы увидим набор инструкций `if`, которые создают связь между заголовком и фактическим типом `prog_type`, поэтому для `is_socket` мы получаем `BPF_PROG_TYPE_SOCKET_FILTER`:

```
if (is_socket) {
        prog_type = BPF_PROG_TYPE_SOCKET_FILTER;
} else if (is_kprobe || is_kretprobe) {
        prog_type = BPF_PROG_TYPE_KPROBE;
} else if (is_tracepoint) {
        prog_type = BPF_PROG_TYPE_TRACEPOINT;
} else if (is_raw_tracepoint) {
        prog_type = BPF_PROG_TYPE_RAW_TRACEPOINT;
} else if (is_xdp) {
        prog_type = BPF_PROG_TYPE_XDP;
} else if (is_perf_event) {
        prog_type = BPF_PROG_TYPE_PERF_EVENT;
} else if (is_cgroup_skb) {
        prog_type = BPF_PROG_TYPE_CGROUP_SKB;
} else if (is_cgroup_sk) {
        prog_type = BPF_PROG_TYPE_CGROUP_SOCK;
} else if (is_sockops) {
        prog_type = BPF_PROG_TYPE_SOCK_OPS;
} else if (is_sk_skb) {
        prog_type = BPF_PROG_TYPE_SK_SKB;
} else if (is_sk_msg) {
        prog_type = BPF_PROG_TYPE_SK_MSG;
} else {
        printf("Unknown event '%s'\n", event);
        return -1;
}
```

Теперь, поскольку мы хотим написать программу типа BPF_PROG_TYPE_SOCKET_FILTER, нужно указать SEC("socket") в качестве заголовка ELF для нашей функции, которая будет действовать как точка входа для программы BPF.

Как видно из этого списка, существует множество типов программ, связанных с сокетами и общими сетевыми операциями. В этой главе мы показываем примеры с BPF_PROG_TYPE_SOCKET_FILTER, а определение всех других типов программ приведено в главе 2. Более того, в главе 7 мы обсудим программы XDP с типом BPF_PROG_TYPE_XDP.

Поскольку мы хотим сохранять количество пакетов для каждого протокола, с которым сталкиваемся, нам необходимо создать карту «ключ/значение», где протокол является ключом, а пакеты считаются значениями. Для этой цели можем использовать BPF_MAP_TYPE_ARRAY:

```
struct bpf_map_def SEC("maps") countmap = {
    .type = BPF_MAP_TYPE_ARRAY,
    .key_size = sizeof(int),
    .value_size = sizeof(int),
    .max_entries = 256,
};
```

Карта определяется с помощью структуры bpf_map_def, для ссылок в программе она будет называться countmap.

На этом этапе можно написать какой-то код для подсчета пакетов. Мы знаем, что программы типа BPF_PROG_TYPE_SOCKET_FILTER для этого подходят, так как помогают увидеть все пакеты, проходящие через интерфейс. Поэтому мы прикрепляем программу к правильному заголовку с помощью SEC("socket"):

```
SEC("socket")
int socket_prog(struct __sk_buff *skb) {
  int proto = load_byte(skb, ETH_HLEN + offsetof(struct iphdr, protocol));
  int one = 1;
  int *el = bpf_map_lookup_elem(&countmap, &proto);
  if (el) {
    (*el)++;
  } else {
    el = &one;
  }
  bpf_map_update_elem(&countmap, &proto, el, BPF_ANY);
  return 0;
}
```

После присоединения заголовка ELF мы можем использовать функцию `load_byte` для извлечения раздела протокола из структуры `sk_buff`. Затем применяем идентификатор протокола в качестве ключа для выполнения операции `bpf_map_lookup_elem`, чтобы извлечь текущее значение счетчика из таблицы подсчетов и иметь возможность увеличить его или установить в `1`, если это лишь первый пакет. Теперь можно обновить карту с увеличенным значением с помощью `bpf_map_update_elem`.

Чтобы скомпилировать программу в файл ELF, мы просто берем Clang с `-target bpf`. Эта команда создает файл `bpf_program.o`, который будем загружать с помощью загрузчика:

```
clang -O2 -target bpf -c bpf_program.c -o bpf_program.o
```

Загрузка и подключение к сетевому интерфейсу

Загрузчик — это программа, которая фактически открывает наш скомпилированный двоичный файл BPF ELF `bpf_program.o` и связывает определенную программу BPF и ее карты с сокетом, который создается на основе наблюдаемого интерфейса (в нашем случае `lo` — локального петлевого интерфейса).

Наиболее важной частью работы загрузчика является сама загрузка файла ELF:

```
if (load_bpf_file(filename)) {
  printf("%s", bpf_log_buf);
  return 1;
}

sock = open_raw_sock("lo");

if (setsockopt(sock, SOL_SOCKET, SO_ATTACH_BPF, prog_fd,
               sizeof(prog_fd[0]))) {
  printf("setsockopt %s\n", strerror(errno));
  return 0;
}
```

Это дополнит массив `prog_fd` еще одним элементом, то есть файловым дескриптором нашей загруженной программы, который мы теперь можем связать с дескриптором сокета петлевого интерфейса, открытого с помощью `open_raw_sock`.

Связывание выполняется установкой опции SO_ATTACH_BPF для необработанного сокета, открытого в интерфейсе. На этом этапе загрузчик пользовательского пространства может найти элементы карты по мере того, как ядро отправляет их:

```
for (i = 0; i < 10; i++) {
  key = IPPROTO_TCP;
  assert(bpf_map_lookup_elem(map_fd[0], &key, &tcp_cnt) == 0);

  key = IPPROTO_UDP;
  assert(bpf_map_lookup_elem(map_fd[0], &key, &udp_cnt) == 0);

  key = IPPROTO_ICMP;
  assert(bpf_map_lookup_elem(map_fd[0], &key, &icmp_cnt) == 0);

  printf("TCP %d UDP %d ICMP %d packets\n", tcp_cnt, udp_cnt, icmp_cnt);
  sleep(1);
}
```

Для выполнения поиска мы подсоединяемся к карте массива, используя цикл for и bpf_map_lookup_elem, чтобы прочитать и вывести значения для счетчиков пакетов TCP, UDP и ICMP.

Теперь осталось лишь скомпилировать программу!

Поскольку она использует библиотеку libbpf, нужно скомпилировать ее из дерева исходного кода ядра, которое мы только что клонировали:

```
$ cd $KERNEL_SRCTREE/tools/lib/bpf
$ make
```

Сейчас, когда у нас есть libbpf, можем скомпилировать загрузчик с помощью следующего сценария:

```
KERNEL_SRCTREE=$1
LIBBPF=${KERNEL_SRCTREE}/tools/lib/bpf/libbpf.a
clang -o loader-bin -I${KERNEL_SRCTREE}/tools/lib/bpf/ \
  -I${KERNEL_SRCTREE}/tools/lib -I${KERNEL_SRCTREE}/tools/include \
  -I${KERNEL_SRCTREE}/tools/perf -I${KERNEL_SRCTREE}/samples \
  ${KERNEL_SRCTREE}/samples/bpf/bpf_load.c \
  loader.c "${LIBBPF}" -lelf
```

Как видите, сценарий включает в себя несколько заголовков и библиотеку libbpf из самого ядра, поэтому он должен знать, где найти исходный код ядра.

Для этого вы можете заменить в нем `$KERNEL_SRCTREE` или просто записать этот сценарий в файл и использовать его:

```
$ ./build-loader.sh /tmp/linux-stable
```

На данном этапе загрузчик создаст файл `loader-bin`, который может быть наконец запущен вместе с файлом ELF программы BPF (требуются права root):

```
# ./loader-bin bpf_program.o
```

После загрузки и запуска программа сделает десять дампов, по одному за секунду, показывая количество пакетов для каждого из трех рассмотренных протоколов. Поскольку программа подключена к петлевому устройству `lo`, наряду с загрузчиком вы можете запустить `ping` и увидеть увеличение счетчика ICMP.

Выполните `ping` для генерации ICMP-трафика на localhost:

```
$ ping -c 100 127.0.0.1
```

При этом localhost пингуется 100 раз и получается что-то вроде следующего:

```
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.100 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.107 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.093 ms
64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.102 ms
64 bytes from 127.0.0.1: icmp_seq=5 ttl=64 time=0.105 ms
64 bytes from 127.0.0.1: icmp_seq=6 ttl=64 time=0.093 ms
64 bytes from 127.0.0.1: icmp_seq=7 ttl=64 time=0.104 ms
64 bytes from 127.0.0.1: icmp_seq=8 ttl=64 time=0.142 ms
```

Затем в другом терминале мы, наконец, можем запустить программу BPF:

```
# ./loader-bin bpf_program.o
```

Начало вывода будет выглядеть так:

```
TCP 0 UDP 0 ICMP 0 packets
TCP 0 UDP 0 ICMP 4 packets
TCP 0 UDP 0 ICMP 8 packets
TCP 0 UDP 0 ICMP 12 packets
TCP 0 UDP 0 ICMP 16 packets
TCP 0 UDP 0 ICMP 20 packets
TCP 0 UDP 0 ICMP 24 packets
TCP 0 UDP 0 ICMP 28 packets
TCP 0 UDP 0 ICMP 32 packets
TCP 0 UDP 0 ICMP 36 packets
```

Теперь вы знаете достаточно о том, что нужно для фильтрации пакетов в Linux с помощью программы eBPF фильтра сокетов. Но это не единственный способ! Возможно, вы захотите подключить подсистему планирования пакетов с помощью ядра, а не непосредственно на сокетах. В следующем разделе рассмотрим эту возможность.

Классификатор управления трафиком на основе BPF

Управление трафиком относится к архитектуре подсистемы ядра для планирования пакетов. Она состоит из механизмов и систем очередей, которые могут определять, как пакеты передаются и принимаются.

Некоторые варианты управления трафиком включают следующие действия:

- определение приоритетов для пакетов конкретных типов;
- отбрасывание пакетов определенных типов;
- распределение пропускной способности,

но не ограничиваются ими.

Учитывая, что в общем случае управление трафиком — это способ перераспределить сетевые ресурсы в системе для того, чтобы извлечь максимальную выгоду, следует использовать конкретные специфические конфигурации управления трафиком в зависимости от типа приложений, которые вы собираетесь запустить. Управление трафиком предоставляет программируемый классификатор, называемый `cls_bpf`, позволяющий подключаться к различным уровням операций планирования, на которых они могут считывать и обновлять буфер сокета и метаданные пакета для таких процедур, как формирование трафика, трассировка, предварительная обработка и многое другое.

Поддержка eBPF в `cls_bpf` была реализована в ядре 4.1, что означает: программа такого типа имеет доступ к картам eBPF, способна поддерживать завершающие вызовы, может обращаться к метаданным туннеля IPv4/IPv6 и в общем случае использовать помощники и утилиты, поставляемые с eBPF.

Инструменты для взаимодействия с сетевой конфигурацией, связанной с управлением трафиком, являются частью пакета iproute2 (https://oreil.ly/SYGwI), в который включены `ip` и `tc` — инструменты для конфигурирования сетевых интерфейсов и параметров управления трафиком соответственно.

На этом этапе для изучения управления трафиком может потребоваться специальная терминология. Этому посвящен следующий раздел.

Терминология

Как уже упоминалось, между управлением трафиком и программами BPF существуют точки взаимодействия, поэтому вам необходимо понимать некоторые концепции управления трафиком. Если вы уже освоили управление трафиком, можете пропустить раздел терминологии и перейти непосредственно к примерам.

Организация очереди

При организации очереди (qdisc) определяются объекты планирования, используемые для постановки в очередь пакетов, отправленных на интерфейс, путем изменения способа их отправки. Эти объекты могут быть бесклассовыми или классифицированными.

По умолчанию qdisc — это `pfifo_fast`, который является бесклассовым и ставит в очередь пакеты в трех очередях FIFO («первым пришел — первым вышел»), взятые из очереди в зависимости от их приоритета. Этот qdisc не используется для виртуальных устройств, таких как loopback (`lo`) или устройства Virtual Ethernet (`veth`), которые вместо этого применяют `noqueue`. Помимо того что `pfifo_fast` является хорошим выбором по умолчанию для своего алгоритма планирования, он не требует никакого конфигурирования для работы.

Виртуальные интерфейсы можно отличить от физических интерфейсов (устройств) с помощью псевдофайловой системы `/sys`:

```
ls -la /sys/class/net
total 0
drwxr-xr-x  2 root root 0 Feb 13 21:52 .
drwxr-xr-x 64 root root 0 Feb 13 18:38 ..
```

```
lrwxrwxrwx  1 root root 0 Feb 13 23:26 docker0 ->
../../devices/virtual/net/docker0
Lrwxrwxrwx  1 root root 0 Feb 13 23:26 enp0s31f6 ->
../../devices/pci0000:00/0000:00:1f.6/net/enp0s31f6
Lrwxrwxrwx  1 root root 0 Feb 13 23:26 lo -> ../../devices/virtual/net/lo
```

Если пока не все ясно, не расстраивайтесь. Если вы никогда не слышали о qdisc, можете использовать команду `ip a` для отображения списка сетевых интерфейсов, настроенных в текущей системе:

```
ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue
state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
    valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
    valid_lft forever preferred_lft forever
2: enp0s31f6: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc
fq_codel stateDOWN group default
qlen 1000
link/ether 8c:16:45:00:a7:7e brd ff:ff:ff:ff:ff:ff
6: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc
noqueue state DOWN group default
link/ether 02:42:38:54:3c:98 brd ff:ff:ff:ff:ff:ff
inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
    valid_lft forever preferred_lft forever
inet6 fe80::42:38ff:fe54:3c98/64 scope link
    valid_lft forever preferred_lft forever
```

Этот список уже о чем-то говорит. Можете ли вы найти в нем слово `qdisc`? Проанализируем ситуацию.

- В этой системе три сетевых интерфейса — `lo`, `enp0s31f6` и `docker0`.
- Интерфейс `lo` — виртуальный, поэтому у него есть `qdisc noqueue`.
- `enp0s31f6` — это физический интерфейс. А почему здесь имеется qdisc с `fq_codel` (честная задержка в очереди)? Разве `pfifo_fast` не должен быть установлен по умолчанию? Оказывается, система, на которой мы тестируем команды, работает под управлением Systemd, которая по-разному устанавливает qdisc по умолчанию, используя параметр ядра `net.core.default_qdisc`.
- Интерфейс `docker0` является интерфейсом моста, поэтому он задействует виртуальное устройство, а для него применяется `noqueue` qdisc.

noqueue qdisc не имеет классов, планировщика или классификатора. Все, что он делает, — пытается отправить пакеты немедленно. Как уже говорилось, noqueue по умолчанию используется виртуальными устройствами, но это также qdisc, который вступает в силу для любого интерфейса, если вы удаляете qdisc, связанный с этим интерфейсом в настоящий момент.

fq_codel — это бесклассовый qdisc, который классифицирует входящие пакеты с помощью стохастической модели, чтобы иметь возможность честно ставить в очередь потоки трафика.

Теперь ситуация должна быть более понятной: мы задействовали команду ip, чтобы найти информацию о qdisc, но оказалось, что среди инструментов iproute2 есть инструмент под названием tc, имеющий специальную подкоманду, которую вы можете использовать для вывода списка всех qdisc:

```
tc qdisc ls
qdisc noqueue 0: dev lo root refcnt 2
qdisc fq_codel 0: dev enp0s31f6 root refcnt 2 limit 10240p flows 1024
quantum 1514
target 5.0ms interval 100.0ms memory_limit 32Mb ecn
qdisc noqueue 0: dev docker0 root refcnt 2
```

Здесь гораздо больше информации! Для docker0 и lo мы в основном видим ту же информацию, что и для ip a, но для enp0s31f6, например, получаем следующее:

- количество входящих пакетов, которые он может обработать, ограничено — 10 240;
- стохастическая модель, используемая fq_codel, хочет направить трафик в разные потоки, и этот вывод содержит информацию о том, сколько их существует, — 1024.

Теперь, когда ключевые понятия qdisc описаны, в следующем разделе можем подробнее рассмотреть классовые и бесклассовые qdisc, чтобы понять, в чем их различия и какие из них подходят для программ BPF.

Классовые qdisc, фильтры и классы

Классовые qdisc позволяют определять классы для разных типов трафика, поэтому к ним можно применять разные правила. Наличие класса для qdisc означает, что он способен содержать дополнительные qdisc. При таком типе

иерархии мы можем использовать фильтр (классификатор) для классификации трафика путем определения следующего класса, в который должен быть помещен пакет.

С помощью *фильтров* пакеты назначают определенному классу в зависимости от их типа. Их задействуют внутри классовых qdisc, чтобы определить, в каком классе следует поместить пакет в очередь (два или более фильтра могут отражать один и тот же класс) (рис. 6.2). Каждый фильтр применяет классификатор для сортировки пакетов на основе содержащейся в них информации.

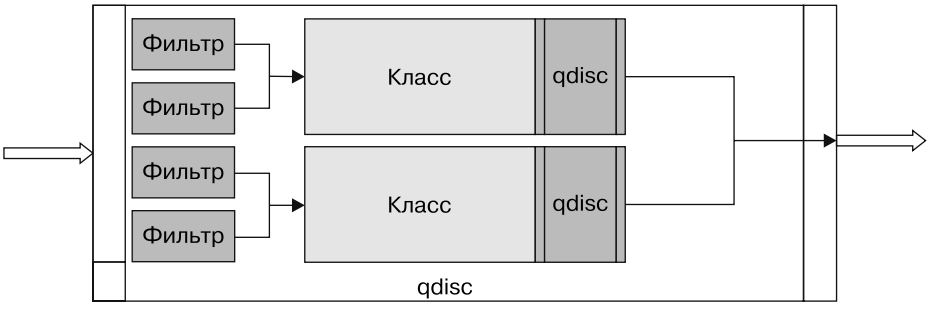

Рис. 6.2. Классовый qdisc с фильтрами

Как уже упоминалось, `cls_bpf` — это классификатор, который мы хотим использовать при написании BPF-программ для управления трафиком. В следующих разделах будет приведен конкретный пример того, как это делается.

Классы — это объекты, которые могут существовать только в классовом qdisc, с их помощью управляют трафиком для создания иерархий. Сложные иерархии возможны благодаря тому, что к классу могут быть прикреплены фильтры, которые в дальнейшем можно применять в качестве точки входа для другого класса или qdisc.

Бесклассовые qdisc

Бесклассовый qdisc — это qdisc, у которого не может быть дочерних элементов, потому что ему не разрешено иметь связанные классы. Это означает, что к бесклассовому qdisc невозможно прикрепить фильтры. Поскольку бесклассовые qdisc не могут иметь дочерних элементов, мы не можем добавлять к ним фильтры и классификаторы, так что они неинтересны с точки

зрения BPF, но все же полезны для решения простых задач управления трафиком.

Получив определенные знания о qdisc, фильтрах и классах, рассмотрим, как писать программы BPF для классификатора `cls_bpf`.

Программа классификатора управления трафиком с использованием cls_bpf

Как уже говорилось, управление трафиком — это мощный механизм, который стал еще более мощным благодаря классификаторам. Среди классификаторов есть такой, который позволяет запрограммировать путь сетевых данных, — классификатор `cls_bpf`. Это особенный классификатор, потому что он может запускать программы BPF. Это означает, что `cls_bpf` позволит вам внедряться в свои программы BPF непосредственно на входном и выходном уровнях, а запуск программ BPF, подключенных к этим уровням, означает, что они смогут получить доступ к структуре `sk_buff` для соответствующих пакетов.

Чтобы лучше понять взаимосвязь между управлением трафиком и программами BPF, посмотрите на рис. 6.3, где показано, как программы BPF загружаются в классификатор `cls_bpf`. Отметьте, что такие программы подключены к входным и выходным qdisc. Все другие взаимодействия в контексте также описаны. Считая сетевой интерфейс точкой входа для сетевого трафика, вы увидите следующее.

- Сначала трафик направляется на вход управления трафиком.
- Затем ядро выполняет программу BFP, загруженную из пространства пользователя для каждого входящего запроса.
- После выполнения входной программы управление передается сетевому стеку, который информирует приложение пользователя о сетевом событии.
- После того как приложение дает ответ, управление передается на выход управления трафиком с использованием другой программы BPF, которая после завершения возвращает управление ядру.
- Клиенту дается ответ.

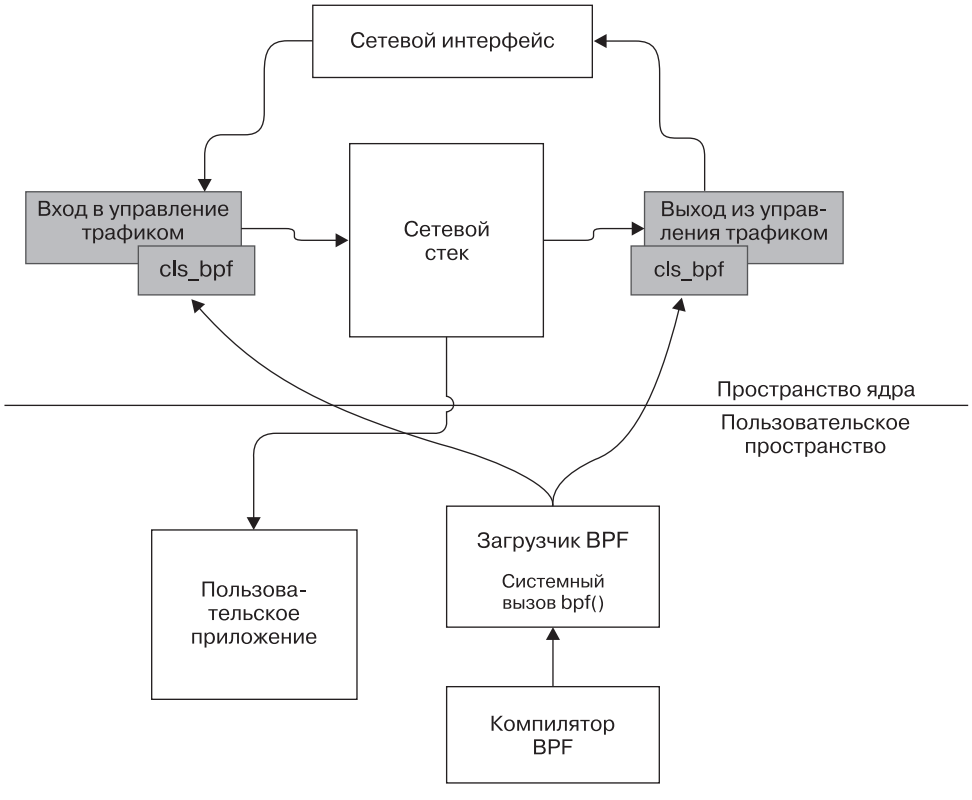

Рис. 6.3. Загрузка BPF-программ и управление трафиком

Вы можете писать программы BPF для управления трафиком на C и компилировать их, используя LLVM/Clang с бэкендом BPF.

Входящие и исходящие qdisc позволяют подключать управление трафиком к входящему (входному) и исходящему (выходному) трафику соответственно.

Чтобы этот пример работал, нужно запустить его в ядре, которое было скомпилировано с `cls_bpf` напрямую или в виде модуля. Чтобы убедиться, что у вас есть все необходимое, сделайте следующее:

```
cat /proc/config.gz| zcat  | grep -i BPF
```

Убедитесь, что видите примерно такой вывод с `y` или `m`:

```
CONFIG_BPF=y
CONFIG_BPF_SYSCALL=y
CONFIG_NET_CLS_BPF=m
CONFIG_BPF_JIT=y
CONFIG_HAVE_EBPF_JIT=y
CONFIG_BPF_EVENTS=y
```

Теперь посмотрим, как написать классификатор:

```
SEC("classifier")
static inline int classification(struct __sk_buff *skb) {
  void *data_end = (void *)(long)skb->data_end;
  void *data = (void *)(long)skb->data;
  struct ethhdr *eth = data;

  __u16 h_proto;
  __u64 nh_off = 0;
  nh_off = sizeof(*eth);

  if (data + nh_off > data_end) {
    return TC_ACT_OK;
  }
```

Роль функции `main` нашего классификатора выполняет функция `classification`. Она аннотирована заголовком раздела под названием `classifier`, так что `tc` знает, что использовать нужно этот классификатор.

В данный момент нам нужно извлечь некоторую информацию из `skb`. Член `data` содержит все данные для текущего пакета и все его детали протокола. Чтобы программа знала, что внутри, нужно перевести данные в кадр Ethernet (в данном случае с переменной `*eth`). Чтобы удовлетворить статический верификатор, следует убедиться, что данные, суммируемые с размером указателя `eth`, не превышают объем пространства, в котором находится `data_end`. После этого мы можем перейти на один уровень глубже и получить тип протокола из члена `h_proto` в `*eth`:

```
if (h_proto == bpf_htons(ETH_P_IP)) {
  if (is_http(skb, nh_off) == 1) {
      trace_printk("Yes! It is HTTP!\n");
    }
  }

  return TC_ACT_OK;
}
```

Получив протокол, нам нужно преобразовать его с хоста, чтобы проверить, действительно ли это IPv4, который нас интересует. Если это так, мы проверяем, является ли внутренний пакет пакетом HTTP, используя нашу собственную функцию `is_http`. Если все правильно, печатаем отладочное сообщение о том, что нашли HTTP-пакет:

```
void *data_end = (void *)(long)skb->data_end;
void *data = (void *)(long)skb->data;
struct iphdr *iph = data + nh_off;

if (iph + 1 > data_end) {
  return 0;
}

if (iph->protocol != IPPROTO_TCP) {
  return 0;
}
__u32 tcp_hlen = 0;
```

Функция `is_http` аналогична нашей функции классификатора, но мы запускаем ее из `skb`, уже зная начальное смещение для данных протокола IPv4. Как и ранее, мы должны выполнить проверку перед получением доступа к данным протокола IP с помощью переменной `*iph`, чтобы статический верификатор узнал о наших намерениях.

После того как это сделано, для продолжения работы мы просто проверяем, содержит ли заголовок IPv4 пакет TCP. Если протокол пакета относится к типу `IPPROTO_TCP`, нужно сделать еще несколько проверок, чтобы получить фактический заголовок TCP в переменной `*tcph`:

```
  plength = ip_total_length - ip_hlen - tcp_hlen;
  if (plength >= 7) {
    unsigned long p[7];
    int i = 0;
    for (i = 0; i < 7; i++) {

      p[i] = load_byte(skb, poffset + i);
    }
    int *value;
    if ((p[0] == 'H') && (p[1] == 'T') && (p[2] == 'T') && (p[3] == 'P')) {
      return 1;
    }
  }

  return 0;
}
```

Теперь, когда заголовок TCP известен, мы можем загрузить первые 7 байт из структуры `skb` со смещением `poffset` полезной нагрузки TCP. На этом этапе можно проверить, является ли массив байтов последовательностью, означающей HTTP. Зная, что это протокол уровня 7 и это HTTP, можем вернуть `1`, в противном случае возвращаем `0`.

Как видите, наша программа проста. Разрешено почти все, а получив HTTP-пакет, программа скажет нам об этом в отладочном сообщении.

Вы можете скомпилировать программу с помощью Clang, используя цель `bpf`, как мы делали ранее с примером фильтра сокетов. Мы не можем скомпилировать эту программу для управления трафиком подобным образом — это сгенерирует файл ELF `classifier.o`, который будет на этот раз загружен `tc`, а не нашим пользовательским загрузчиком:

```
clang -O2 -target bpf -c classifier.c -o classifier.o
```

КОДЫ ВОЗВРАТА УПРАВЛЕНИЯ ТРАФИКОМ

Из `man 8 tc-bpf`:

- `TC_ACT_OK (0)` — завершает конвейер обработки пакетов и позволяет пакету продвигаться дальше;
- `TC_ACT_SHOT (2)` — завершает конвейер обработки пакетов и отбрасывает пакет;
- `TC_ACT_UNSPEC (-1)` — использует действие по умолчанию, настроенное из `tc` (аналогично возвращению `-1` из классификатора);
- `TC_ACT_PIPE (3)` — переходит к следующему действию, если оно доступно;
- `TC_ACT_RECLASSIFY (1)` — завершает конвейер обработки пакетов и начинает классификацию с начального `else`.

Все остальное — неопределенный код возврата.

Теперь мы можем установить программу на интересующий нас интерфейс. В нашем случае это был `eth0`.

Первая команда заменит qdisc по умолчанию для устройства `eth0`, а вторая фактически загрузит наш классификатор `cls_bpf` в этот классовый планиров-

щик входящего трафика (ingress). Это означает, что наша программа будет обрабатывать весь трафик, поступающий в данный интерфейс. Для обработки исходящего трафика нужно использовать вместо этого egress.

```
# tc qdisc add dev eth0 handle 0: ingress
# tc filter add dev eth0 ingress bpf obj classifier.o flowid 0:
```

Программа загружена, теперь все, что нам нужно, — отправить HTTP-трафик в интерфейс. Для этого можно задействовать любой HTTP-сервер в этом интерфейсе. Тогда вы сможете обратиться к IP-адресу интерфейса с помощью curl.

Если у вас нет HTTP-сервера, можете использовать тестовый вариант с Python 3 и модулем http.server. Откроется порт 8000 со списком каталогов текущего рабочего каталога:

```
python3 -m http.server
```

Сейчас можно вызвать сервер с помощью curl:

```
$ curl http://192.168.1.63:8080
```

После этого вы должны увидеть ответ от HTTP-сервера. Теперь можете получить сообщения отладки, созданные с помощью trace_printk, подтвердив это специальной командой tc:

```
# tc exec bpf dbg
```

Вывод будет примерно таким:

```
Running! Hang up with ^C!

        python3-18456 [000] ..s1 283544.114997: 0: Yes! It is HTTP!
        python3-18754 [002] ..s1 283566.008163: 0: Yes! It is HTTP!
```

Поздравляем! Вы только что создали свой первый классификатор управления трафиком BPF.

Вместо сообщения отладки, как в этом примере, вы можете применить карту, чтобы сообщить пространству пользователя, что интерфейс только что получил пакет HTTP. Поупражняйтесь в этом самостоятельно. Посмотрите на classifier.c в предыдущем примере, чтобы получить представление о том, как это сделать, — например, так, как мы использовали карту countmap.

Возможно, теперь вам понадобится выгрузить классификатор. Можете сделать это, удалив входящий qdisc, который только что подключили к интерфейсу:

```
# tc qdisc del dev eth0 ingress
```

Примечания к act_bpf и отличия cls_bpf

Вы могли заметить, что существует другой объект для BPF-программ, называемый `act_bpf`. На самом деле `act_bpf` — это действие, а не классификатор. Это означает другую функциональность, потому что действия являются объектами, прикрепленными к фильтрам. Из-за этого они не могут выполнять фильтрацию напрямую, требуя, чтобы управление трафиком сначала рассмотрело все пакеты. Поэтому обычно предпочтительно использовать классификатор `cls_bpf` вместо прямого действия `act_bpf`.

Но, поскольку `act_bpf` может быть присоединен к любому классификатору, могут возникнуть случаи, когда вам покажется полезным повторно применить уже имеющийся классификатор и присоединить к нему программу BPF.

Различия между управлением трафиком и XDP

Хотя программы управления трафиком `cls_bpf` и XDP выглядят очень похожими, они во многом разнятся. Программы XDP выполняются раньше по ходу движения входных данных, прежде чем те поступят в основной сетевой стек ядра, поэтому данные программы не имеют доступа к структуре буфера сокетов `sk_buff`, как в случае с `tc`. Вместо этого они принимают другую структуру, называемую `xdp_buff`, которая является урезанным представлением пакета без метаданных. Все это имеет свои преимущества и недостатки. Например, будучи выполненными еще до кода ядра, программы XDP могут эффективно отбрасывать пакеты. По сравнению с программами управления трафиком программы XDP могут быть подключены только к трафику, входящему в систему.

Вы можете задаться вопросом: когда выгодно использовать одни и другие? Ответ таков: из-за того что XDP-программы не содержат обогащенные ядром структуры данных и метаданные, они лучше подходят для случаев, охватывающих уровни OSI вплоть до 4-го. Но оставим это до следующей главы!

Резюме

Теперь вам должно быть совершенно ясно, что программы BPF полезны для обеспечения видимости и контроля на разных уровнях пути передачи сетевых данных. Вы видели, как использовать их для фильтрации пакетов с помощью высокоуровневых инструментов, которые генерируют сборку BPF. Мы загрузили программу в сетевой сокет и в конце подключили наши программы к входному qdisc управления трафиком, чтобы классифицировать его с помощью программ BPF. В этой главе мы также кратко обсудили XDP, а в главе 7 полностью раскроем данную тему. Например, расскажем, как создаются программы XDP, какие программы XDP существуют и как их писать и тестировать.

7 Express Data Path

Express Data Path (XDP) — это безопасный, программируемый, высокопроизводительный процессор пакетов, интегрированный в ядро, который используется при прохождении сетевых данных в Linux. Выполняет программы BPF по получении пакета драйвером NIC. Это позволяет программам XDP в кратчайшие сроки принять решение по поводу данного пакета — отбросить, изменить или просто разрешить его.

Точка выполнения — это не единственное, за счет чего программы XDP работают быстро, определенную роль играют в этом и другие решения, реализованные в ходе разработки.

- При обработке пакетов с помощью XDP не нужно выделять память.
- Программы XDP работают только с линейными нефрагментированными пакетами и задействуют начальный и конечный указатели пакета.
- Нет доступа к полным метаданным пакета, поэтому входной контекст, который получает этот тип программы, будет иметь тип `xdp_buff` вместо структуры `sk_buff`, с которой вы познакомились в главе 6.
- Являясь программами eBPF, программы XDP ограничены по времени выполнения, и, как следствие, их использование имеет фиксированную стоимость в сетевом конвейере.

Говоря о XDP, важно помнить, что это не механизм обхода ядра — он предназначен для интеграции с другими компонентами ядра и внутренней моделью безопасности Linux.

Структура xdp_buff применяется для представления контекста пакета программе BPF, которая использует механизм прямого доступа к пакету, обеспечиваемый платформой XDP. Считайте это облегченной версией sk_buff.

Разница между ними заключается в том, что sk_buff поддерживает также метаданные пакетов (proto, mark, type), которые доступны только на более высоком уровне в сетевом конвейере, и позволяет смешиваться с ними. То, что xdp_buff создается рано и не зависит от других уровней ядра, является одной из причин, по которым он быстрее получает и обрабатывает пакеты с помощью XDP. Другая причина заключается в том, что xdp_buff не содержит ссылок на маршруты, ловушек управления трафиком или других метаданных пакета, как происходит в программах, использующих sk_buff.

В этой главе мы рассмотрим характеристики XDP-программ, различные их виды, а также способы компиляции и загрузки. После этого мы приведем реальные варианты их применения.

Обзор программ XDP

По сути, что делают программы XDP? Они решают, как поступить с принятым пакетом, а затем могут редактировать его содержимое или просто возвращать код результата. Этот код используется для определения того, что происходит с пакетом. Вы можете отбросить пакет, передать его через тот же интерфейс или переслать остальному сетевому стеку. Кроме того, для взаимодействия с сетевым стеком программы XDP могут вставлять и извлекать заголовки пакета. Например, если текущее ядро не поддерживает формат инкапсуляции или протокол, программа XDP может деинкапсулировать его или преобразовать протокол и отправить результат в ядро для обработки.

Но подождите, какова связь между XDP и eBPF?

Оказывается, программы XDP управляются через системный вызов `bpf` и загружаются с использованием типа программы `BPF_PROG_TYPE_XDP`. Кроме того, ловушка драйвера выполнения выполняет байт-код BPF.

При написании XDP-программ важно понимать, что контексты, в которых они будут выполняться, называются также *режимами работы*.

Режимы работы

У XDP есть три режима работы: для удобного тестирования функций, на заказном оборудовании от поставщиков и на обычным образом скомпилированных ядрах без специального оборудования. Рассмотрим их.

Нативный XDP

Такой режим задан по умолчанию. В этом случае программа XDP BPF запускается непосредственно в ходе раннего приема сетевого драйвера. При использовании этого режима важно проверить, поддерживает ли его драйвер. Вы можете сделать это, выполнив следующую команду для исходного дерева данной версии ядра:

```
# Клонировать стабильный репозиторий linux
git clone git://git.kernel.org/pub/scm/linux/kernel/git/stable/
linux-stable.git\
linux-stable

# Проверить тег для текущей версии вашего ядра
cd linux-stable
git checkout tags/v4.18

# Проверить доступные драйверы
git grep -l XDP_SETUP_PROG drivers/
```

Вывод будет подобен следующему:

```
drivers/net/ethernet/broadcom/bnxt/bnxt_xdp.c
drivers/net/ethernet/cavium/thunder/nicvf_main.c
drivers/net/ethernet/intel/i40e/i40e_main.c
drivers/net/ethernet/intel/ixgbe/ixgbe_main.c
drivers/net/ethernet/intel/ixgbevf/ixgbevf_main.c
drivers/net/ethernet/mellanox/mlx4/en_netdev.c
drivers/net/ethernet/mellanox/mlx5/core/en_main.c
drivers/net/ethernet/netronome/nfp/nfp_net_common.c
drivers/net/ethernet/qlogic/qede/qede_filter.c
drivers/net/netdevsim/bpf.c
drivers/net/tun.c
drivers/net/virtio_net.c
```

Итак, ядро 4.18 поддерживает:

- сетевой драйвер Broadcom NetXtreme-C/E bnxt;
- драйвер Cavium thunderx;
- драйвер Intel i40;
- драйверы Intel ixgbe и ixgvevf;
- драйверы Mellanox mlx4 и mlx5;
- Netronome Network Flow Processor;
- QLogic qede NIC Driver;
- TUN/TAP;
- Virtio.

Получив четкое представление о режиме работы по умолчанию, приступим к рассмотрению того, как инструкции программы XDP могут напрямую обрабатываться сетевыми картами с использованием выгруженного XDP.

Выгруженный XDP

В этом режиме программа XDP BPF напрямую выгружается в NIC, а не выполняется на центральном процессоре хоста. Исключив выполнение из ЦП, этот режим получает выигрыш в производительности по сравнению с родным XDP.

Мы можем повторно применить дерево исходного кода ядра, которое только что клонировали, чтобы проверить, какие драйверы NIC в 4.18 поддерживают аппаратную разгрузку. Для этого поищем `XDP_SETUP_PROG_HW`:

```
git grep -l XDP_SETUP_PROG_HW drivers/
```

В выводе должно получиться что-то вроде этого:

```
include/linux/netdevice.h
866:    XDP_SETUP_PROG_HW,

net/core/dev.c
8001:           xdp.command = XDP_SETUP_PROG_HW;
```

```
drivers/net/netdevsim/bpf.c
200:    if (bpf->command == XDP_SETUP_PROG_HW && !ns->bpf_xdpoffload_accept)
{
205:    if (bpf->command == XDP_SETUP_PROG_HW) {
560:    case XDP_SETUP_PROG_HW:

drivers/net/ethernet/netronome/nfp/nfp_net_common.c
3476:   case XDP_SETUP_PROG_HW:
```

Здесь видно, что только Netronome Network Flow Processor (nfp) поддерживает данный режим. Это значит, что он может работать в обоих режимах, поддерживая аппаратную разгрузку наряду с родным XDP.

Теперь такой вопрос: что мне делать, если у меня нет сетевых карт и драйверов, чтобы попробовать мои программы XDP? Ответ прост: общий XDP!

Общий XDP

Он предусмотрен для тестового режима разработки, когда нужно писать и запускать программы XDP, не имея возможностей родного или разгрузочного XDP. Общий XDP поддерживается с версии ядра 4.12. Вы можете использовать этот режим, например, на устройствах `veth`, а мы применим его в последующих примерах для демонстрации возможностей XDP, не требуя, чтобы вы покупали специальное оборудование.

Но кто координирует общее взаимодействие между всеми компонентами и режимами работы? В следующем разделе рассмотрим пакетный процессор.

Пакетный процессор

Процессор пакетов XDP позволяет выполнять программы BPF для пакетов XDP и координирует взаимодействие между ними и сетевым стеком. Он является компонентом ядра для программ XDP, который обрабатывает пакеты в очереди приема (RX) напрямую — так, как они представлены NIC. Это гарантирует, что пакеты доступны для чтения и записи, и позволяет выполнять постобработку в пакетном процессоре. Обновление атомарных программ и загрузка новых программ в процессор пакетов могут выполняться во время работы, не требуя прерывать обслуживание с точки зрения сети

и связанного трафика. Во время работы XDP можно использовать в режиме занятого опроса, что позволяет зарезервировать процессоры, которые будут иметь дело со всеми очередями RX. Это позволяет избежать переключения контекста и обеспечивает обработку пакетов немедленно по прибытии, независимо от их соответствия IRQ. Другой режим, в котором можно задействовать XDP, — это режим, управляемый прерыванием. Он не резервирует ЦП, а выдает прерывание, действующее в среде событий, чтобы проинформировать ЦПУ о том, что ему, выполняя обычную обработку, нужно иметь дело с новым событием.

На рис. 7.1 вы видите точки взаимодействия между RX/TX, приложениями, процессором пакетов и программами BPF, применяемыми к пакетам.

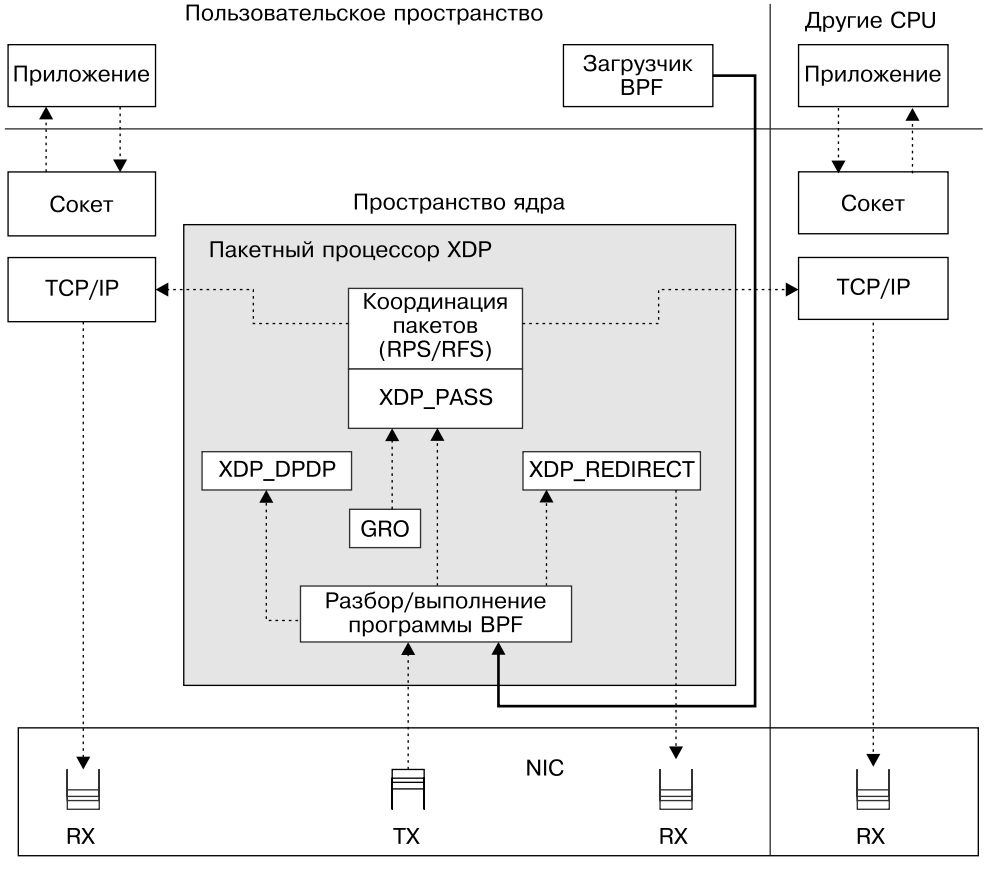

Рис. 7.1. Пакетный процессор

Обратите внимание, что здесь есть несколько блоков со строкой, добавленной к `XDP_`. Это коды результата XDP, которые мы рассмотрим далее.

Коды результата XDP (действия процессора пакетов)

Принятое в процессоре пакетов решение о том, что делать с пакетом, можно задать с использованием одного из пяти кодов возврата. Они затем позволят инструктировать сетевой драйвер о том, как обрабатывать пакет. Рассмотрим действия, которые выполняет процессор пакетов.

- *Отбросить (XDP_DROP)*. Отбрасывает пакет. Это происходит на самой ранней стадии приема в драйвере и означает просто его возврат в очередь RX, в которую он только что прибыл. Отбрасывание пакета как можно раньше помогает предотвратить отказ в обслуживании (DoS). Таким образом отброшенные пакеты используют как можно меньше процессорного времени и мощности процессора.
- *Передать (XDP_TX)*. Пересылает пакет. Это может произойти до или после его изменения. Пересылка пакета подразумевает возврат страницы принятого пакета на тот сетевой адаптер, на котором она была получена.
- *Перенаправить (XDP_REDIRECT)*. Подобно `XDP_TX` в том плане, что он может передавать пакет XDP, но делает это через другой сетевой адаптер или в `cpumap` BPF. В случае `cpumap` BPF ЦП, обслуживающие XDP в очередях приема NIC, могут продолжать работу и передавать пакет для обработки верхнего стека ядра на удаленный ЦП. Это похоже на `XDP_PASS`, но с возможностью того, что программа XDP BPF продолжит обслуживать входящую высокую нагрузку, а не станет тратить время на передачу текущего пакета на верхние уровни.
- *Пропустить (XDP_PASS)*. Передает пакет в обычный сетевой стек для обработки. Эквивалентно поведению обработки пакетов по умолчанию без XDP. Это можно сделать одним из двух способов:
  - *обычный прием* распределяет метаданные (`sk_buff`), принимает пакет в стек и направляет его другому процессору для обработки. Здесь допустимы сырые интерфейсы для пользовательского пространства. Может происходить до или после изменения пакета;

- *общая разгрузка приема* (GRO) может предусматривать прием больших пакетов и объединение пакетов одного и того же соединения. GRO в конечном итоге пропускает пакет через поток обычного приема после обработки.

- *Код ошибки (XDP_ABORTED)*. Обозначает ошибку программы eBPF и приводит к удалению пакета. Это не то, что функциональная программа должна использовать в качестве кода возврата. Например, `XDP_ABORTED` будет возвращено, если программа делит что-либо на ноль. Значение `XDP_ABORTED` всегда равно нулю. При этом оказывается пройденной точка трассировки `trace_xdp_exception`, которую можно дополнительно исследовать для выявления неправильного поведения.

Эти коды действий записаны в заголовочном файле `linux/bpf.h` следующим образом:

```
enum xdp_action {
    XDP_ABORTED = 0,
    XDP_DROP,
    XDP_PASS,
    XDP_TX,
    XDP_REDIRECT,
};
```

Действия XDP определяют различное поведение и являются внутренним механизмом процессора пакетов. На рис. 7.2 приведена упрощенная, ориентированная только на возвратные действия версия того, что изображено на рис. 7.1.

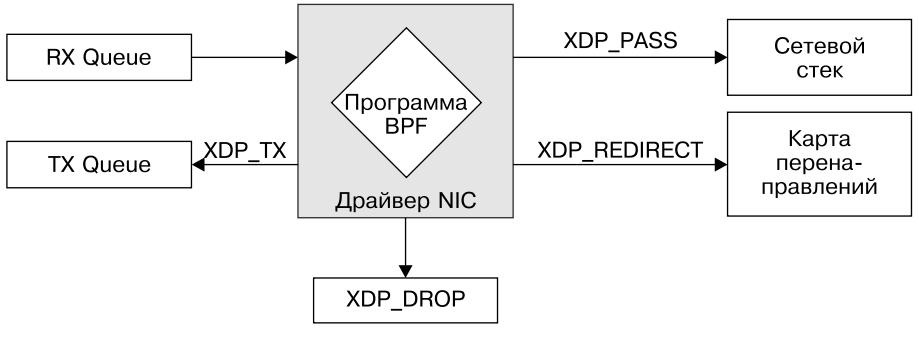

Рис. 7.2. Коды действий XDP

Интересной особенностью XDP-программ является то, что для них обычно не нужно писать загрузчик. На большинстве машин Linux есть хороший загрузчик, реализованный командой `ip`. В следующем разделе расскажем, как его использовать.

XDP и iproute2 в качестве загрузчика

Команда `ip`, доступная в iproute2 (https://oreil.ly/65zuT), может выступать в качестве интерфейса для загрузки программ XDP, скомпилированных в формате ELF, и полностью поддерживает карты и их перемещение, вызов конечного оператора и фиксацию объекта.

Поскольку загрузка программы XDP может быть выражена как конфигурация существующего сетевого интерфейса, загрузчик реализован как часть команды `ip link`, которая выполняет настройку сетевого устройства.

Синтаксис для загрузки программы XDP прост:

```
# ip link set dev eth0 xdp obj program.o sec mysection
```

Проанализируем параметры этой команды по порядку:

- `ip` — вызывает команду `ip`;
- `link` — настраивает сетевые интерфейсы;
- `set` — изменяет атрибуты устройства;
- `dev eth0` — определяет сетевое устройство, на котором мы хотим работать и куда загружать программу XDP;
- `xdp obj program.o` — загружает программу XDP из ELF-файла (объектный файл) с именем `program.o`. Часть `xdp` этой команды указывает системе использовать собственный драйвер, когда он доступен, в противном случае — общий. Вы можете принудительно задать один или другой режим, применив более специфический селектор:
  - `xdpgeneric` для общего XDP;
  - `xdpdrv` для родного XDP;
  - `xdpoffload` для выгруженного XDP;

- `sec mysection` — указывает имя секции `mysection`, содержащее программу BPF для использования из файла ELF. Если оно не задано, будет применяться раздел с именем `prog`. Если в программе не указан раздел, вы должны задать `sec .text` в вызове `ip`.

Рассмотрим практический пример.

У нас есть система с веб-сервером на порте 8000, для которой мы хотим заблокировать любой доступ к его страницам на общедоступной сетевой карте, запретив все TCP-подключения к ней.

Первое, что нам понадобится, — это рассматриваемый веб-сервер. Если у вас его еще нет, запустите его с помощью команды `python3`:

```
$ python3 -m http.server
```

После запуска веб-сервера порт, на котором он работает, будет показан в открытых сокетах с помощью `ss`. Веб-сервер привязан к любому интерфейсу, `*: 8000`, поэтому на данный момент любой внешний абонент, имеющий доступ к нашим общедоступным интерфейсам, сможет видеть его содержимое!

```
$  ss -tulpn
Netid  State      Recv-Q Send-Q Local Address:Port   Peer Address:Port
tcp    LISTEN     0      5      *:8000               *:*
```

Статистика сокетов (ss в терминале) — это утилита командной строки, используемая для исследования сетевых сокетов в Linux. По сути, это современная версия netstat, и ее применение похоже на применение Netstat, что означает: вы можете передавать те же аргументы и получать сопоставимые результаты.

На этом этапе можно проверить сетевые интерфейсы на компьютере, где работает HTTP-сервер:

```
$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
defau
lt qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
```

```
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP g
roup default qlen 1000
    link/ether 02:1e:30:9c:a3:c0 brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.15/24 brd 10.0.2.255 scope global dynamic enp0s3
       valid_lft 84964sec preferred_lft 84964sec
    inet6 fe80::1e:30ff:fe9c:a3c0/64 scope link
       valid_lft forever preferred_lft forever
3: enp0s8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP g
roup default qlen 1000
    link/ether 08:00:27:0d:15:7d brd ff:ff:ff:ff:ff:ff
    inet 192.168.33.11/24 brd 192.168.33.255 scope global enp0s8
       valid_lft forever preferred_lft forever
    inet6 fe80::a00:27ff:fe0d:157d/64 scope link
       valid_lft forever preferred_lft forever
```

Обратите внимание, что машина имеет три интерфейса, но топология сети проста:

- `lo` — это петлевой интерфейс для внутренней коммуникации;
- `enp0s3` — это уровень сети управления, администраторы будут использовать данный интерфейс, чтобы подключиться к веб-серверу для выполнения своих задач;
- `enp0s8` — это открытый интерфейс, наш веб-сервер должен быть скрыт от этого интерфейса.

Теперь перед загрузкой любой программы XDP мы можем проверить открытые на сервере порты с другого компьютера, который может получить доступ к его сетевому интерфейсу, в нашем случае по адресу IPv4 192.168.33.11.

Вы можете проверить открытые порты на удаленном хосте, используя `nmap` следующим образом:

```
# nmap -sS 192.168.33.11
Starting Nmap 7.70 ( https://nmap.org ) at 2019-04-06 23:57 CEST
Nmap scan report for 192.168.33.11
Host is up (0.0034s latency).
Not shown: 998 closed ports
PORT     STATE SERVICE
22/tcp   open  ssh
8000/tcp open  http-alt
```

Прекрасно! Порт 8000 виден, а сейчас его нужно заблокировать!

Network Mapper (nmap) — это сетевой сканер, который наряду с операционной системой может обнаружить хост, службу, сеть и порт. Он используется в основном для аудита безопасности и сканирования сети. При сканировании хоста на наличие открытых портов nmap будет пробовать войти на каждый порт в указанном (или полном) диапазоне.

Наша программа состоит из одного исходного файла `program.c`, поэтому посмотрим, что в него следует включить.

Он должен задействовать структуры заголовков `ethhdr` IPv4 `iphdr` и Ethernet Frame, а также константы протокола и другие структуры. Давайте включим в него необходимые заголовки, как показано далее:

```
#include <linux/bpf.h>
#include <linux/if_ether.h>
#include <linux/in.h>
#include <linux/ip.h>
```

После включения заголовков мы можем объявить макрос `SEC` (его мы уже встречали в предыдущих главах), используемый для объявления атрибутов ELF:

```
#define SEC(NAME) __attribute__((section(NAME), used))
```

Теперь можем объявить основную точку входа для нашей программы `myprogram` и имя раздела ELF `mysection`. Программа применяет в качестве входного контекста структурный указатель `xdp_md`, эквивалентный BPF для встроенного драйвера `xdp_buff`. Затем определяем нужные переменные, такие как указатели данных, структуры уровня Ethernet и IP:

```
SEC("mysection")
int myprogram(struct xdp_md *ctx) {
  int ipsize = 0;
  void *data = (void *)(long)ctx->data;
  void *data_end = (void *)(long)ctx->data_end;
  struct ethhdr *eth = data;
  struct iphdr *ip;
```

Поскольку данные содержат кадр Ethernet, можем извлечь из него пакет уровня IPv4. Проверяем также, что смещение, в котором мы ищем уровень IPv4, не превышает все пространство указателя, так что статический верификатор

позволяет нам это сделать. Если мы вышли за пределы адресного пространства, то просто отбрасываем пакет:

```
ipsize = sizeof(*eth);
ip = data + ipsize;
ipsize += sizeof(struct iphdr);
if (data + ipsize > data_end) {
  return XDP_DROP;
}
```

Теперь, после всех проверок и настроек, можно реализовать настоящую логику для программы, которая в основном отбрасывает каждый TCP-пакет, пропуская все остальное:

```
  if (ip->protocol == IPPROTO_TCP) {
    return XDP_DROP;
  }

  return XDP_PASS;
}
```

Программа готова, сохраним ее под именем `program.c`.

Следующим шагом является компиляция файла ELF `program.o` из нашей программы с использованием Clang. Его можно выполнить на любом компьютере, поскольку двоичные файлы BPF ELF не зависят от платформы:

```
$ clang -O2 -target bpf -c program.c -o program.o
```

Вернувшись на компьютер, на котором размещен наш веб-сервер, мы наконец-то можем загрузить `program.o` с общедоступным сетевым интерфейсом `enp0s8`, используя утилиту `ip` с командой `set`, как описывалось ранее:

```
# ip link set dev enp0s8 xdp obj program.o sec mysection
```

Точкой входа в программу выбран раздел `mysection`.

На данном этапе, если эта команда вернула ноль в качестве кода завершения без ошибок, мы можем проверить сетевой интерфейс, чтобы увидеть, была ли программа загружена правильно:

```
# ip a show enp0s8
3: enp0s8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 xdpgeneric/id:32
    qdisc fq_codel state UP group default qlen 1000
    link/ether 08:00:27:0d:15:7d brd ff:ff:ff:ff:ff:ff
```

```
    inet 192.168.33.11/24 brd 192.168.33.255 scope global enp0s8
       valid_lft forever preferred_lft forever
    inet6 fe80::a00:27ff:fe0d:157d/64 scope link
       valid_lft forever preferred_lft forever
```

Как видите, вывод для `ip a` дает еще кое-что: после MTU он показывает `xdpgeneric/id: 32`, который содержит два интересных момента:

- использованный драйвер — `xdpgeneric`;
- идентификатор программы XDP — `32`.

И последнее — следует убедиться, что загруженная программа действительно выполняет то, что должна делать. Мы можем проверить это, снова запустив `nmap` на внешней машине, чтобы увидеть, что порт 8000 недоступен:

```
# nmap -sS 192.168.33.11
Starting Nmap 7.70 ( https://nmap.org ) at 2019-04-07 01:07 CEST
Nmap scan report for 192.168.33.11
Host is up (0.00039s latency).
Not shown: 998 closed ports
PORT    STATE SERVICE
22/tcp  open  ssh
```

Еще одним тестом для проверки работоспособности может быть попытка получить доступ к программе через браузер или выполнение любого HTTP-запроса. Любой вид теста должен быть неудачным при попытке доступа к 192.168.33.11 в качестве пункта назначения для порта 8000. Вы хорошо поработали, поздравляем с загрузкой вашей первой программы XDP!

Если вы проделали все это на своем компьютере, который необходимо привести в исходное состояние, всегда можете отсоединить программу и отключить XDP на устройстве:

```
# ip link set dev enp0s8 xdp off
```

Интересно, правда? Загрузка программ XDP кажется легкой, не так ли?

По крайней мере, используя `iproute2` в качестве загрузчика, вы можете не писать его самостоятельно. В этом примере мы сосредоточились на `iproute2`, который уже реализует загрузчик для программ XDP. Но на самом деле программы являются BPF-программами, поэтому, даже если `iproute2` иногда может быть полезен, вы всегда должны помнить, что можете загружать свои программы с помощью BCC, как показано в следующем разделе, или

задействовать системный вызов bpf напрямую. Наличие собственного загрузчика позволяет вам управлять жизненным циклом программы и взаимодействием с пространством пользователя.

XDP и BCC

Как и любые другие программы BPF, программы XDP можно компилировать, загружать и запускать с помощью BCC. В следующем примере показана программа XDP, похожая на ту, которую мы использовали для iproute2, но со специальным загрузчиком пространства пользователя, созданным с помощью BCC. Загрузчик в этом случае необходим, потому что мы также хотим подсчитать количество пакетов, которые отбрасываем при фильтрации пакетов TCP.

Как и прежде, сначала создаем программу для пространства ядра с именем program.c.

В примере с iproute2 нашей программе необходимо было импортировать требуемые заголовки для определений структуры и функций, связанных с BPF и протоколами. Здесь мы делаем то же самое, а также объявляем карту типа BPF_MAP_TYPE_PERCPU_ARRAY, используя макрос BPF_TABLE. Карта будет содержать счетчик пакетов для каждого индекса протокола IP, из-за чего размер окажется 256 (спецификация IP содержит только 256 значений). Мы хотим задействовать тип BPF_MAP_TYPE_PERCPU_ARRAY, потому что он гарантирует атомарность счетчиков на уровне ЦП без блокировки:

```
#define KBUILD_MODNAME "program"
#include <linux/bpf.h>
#include <linux/in.h>
#include <linux/ip.h>

BPF_TABLE("percpu_array", uint32_t, long, packetcnt, 256);
```

После этого объявляем нашу основную функцию myprogram, которая принимает в качестве параметра структуру xdp_md. В первую очередь она должна содержать объявления переменных для кадров Ethernet IPv4:

```
int myprogram(struct xdp_md *ctx) {
  int ipsize = 0;
```

```
  void *data = (void *)(long)ctx->data;
  void *data_end = (void *)(long)ctx->data_end;
  struct ethhdr *eth = data;
  struct iphdr *ip;
  long *cnt;
  __u32 idx;

  ipsize = sizeof(*eth);
  ip = data + ipsize;
  ipsize += sizeof(struct iphdr);
```

После того как мы выполнили все объявления переменных и получили доступ к указателю данных, который теперь содержит кадр Ethernet и указатель `ip` с пакетом IPv4, можем проверить, не вышли ли мы за пределы доступной памяти. Если это так, отбрасываем пакет. Если пространство памяти в порядке, извлекаем протокол и ищем массив `packetcnt`, чтобы получить предыдущее значение счетчика пакетов для текущего протокола в переменной `idx`. Затем увеличиваем счетчик на единицу. Когда приращение обработано, мы можем продолжить и проверить, является ли это протоколом TCP. Если это так, просто отбрасываем пакет без вопросов, в противном случае пропускаем его:

```
if (data + ipsize > data_end) {
  return XDP_DROP;
}

idx = ip->protocol;
cnt = packetcnt.lookup(&idx);
if (cnt) {
  *cnt += 1;
}

if (ip->protocol == IPPROTO_TCP) {
  return XDP_DROP;
}
  return XDP_PASS;
}
```

Теперь напишем загрузчик `loader.py`. Он состоит из двух частей: фактической логики загрузки и цикла, который выводит количество пакетов.

Для логики загрузки открываем нашу программу, читая файл `program.c`. С помощью `load_func` указываем системному вызову `bpf` использовать функцию

myprogram в качестве main, применяя тип программы BPF.XDP. Это то же самое, что и BPF_PROG_TYPE_XDP.

После загрузки мы получаем доступ к карте BPF с именем packetcnt, используя get_table.

Обязательно измените переменную устройства с enp0s8 на интерфейс, с которым хотите работать.

```
#!/usr/bin/python

from bcc import BPF
import time
import sys

device = "enp0s8"
b = BPF(src_file="program.c")
fn = b.load_func("myprogram", BPF.XDP)
b.attach_xdp(device, fn, 0)
packetcnt = b.get_table("packetcnt")
```

Оставшаяся часть, которую нам нужно написать, — это фактически цикл для вывода количества пакетов. Без этого наша программа уже сможет отбрасывать пакеты, но мы хотим увидеть, что там происходит. У нас есть два цикла. Внешний цикл получает события клавиатуры и завершается при появлении сигнала на прерывание программы. Когда внешний цикл прерывается, вызывается функция remove_xdp и интерфейс освобождается от программы XDP.

Во внешнем цикле внутренний цикл должен возвращать значения из карты packetcnt и печатать их в формате protocol: counter pkt/s:

```
prev = [0] * 256
print("Printing packet counts per IP protocol-number, hit CTRL+C to stop")
while 1:
    try:
        for k in packetcnt.keys():
            val = packetcnt.sum(k).value
            i = k.value
            if val:
        delta = val - prev[i]
        prev[i] = val
        print("{}: {} pkt/s".format(i, delta))
```

```
    time.sleep(1)
except KeyboardInterrupt:
    print("Removing filter from device")
    break

b.remove_xdp(device, 0)
```

Хорошо! Теперь можем протестировать эту программу, просто запустив загрузчик с правами администратора:

```
# python program.py
```

Она будет каждую секунду выводить строку со счетчиками пакетов:

```
Printing packet counts per IP protocol-number, hit CTRL+C to stop
6: 10 pkt/s
17: 3 pkt/s
^CRemoving filter from device
```

Мы обнаружили только два типа пакетов: `6` обозначает TCP, а `17` — UDP.

Сейчас вы, вероятно, подумаете о проектах использования XDP, и это очень хорошо! Но, как всегда в программной инженерии, если вы хотите создать хорошую программу, важно сначала написать тесты или вообще написать тесты! Это важно! В следующем разделе говорится о том, как тестировать программы XDP.

Тестирование программ XDP

Самая сложная часть работы с программами XDP заключается в том, что для проверки фактического потока пакетов необходимо воспроизвести среду, в которой все компоненты усреднены. Хотя в настоящее время с применением технологий виртуализации создание рабочей среды может оказаться легкой задачей, верно также и то, что сложная настройка способна ограничивать воспроизводимость и программируемость тестовой среды. Кроме того, при анализе аспектов производительности высокопроизводительных программ XDP в виртуализированной среде стоимость виртуализации делает тест неэффективным, поскольку он стоит гораздо больше, чем фактическая обработка пакетов.

К счастью, у разработчиков ядра есть решение. Они реализовали команду, которая может использоваться для тестирования программ XDP и называется `BPF_PROG_TEST_RUN`.

По сути, BPF_PROG_TEST_RUN получает программу XDP для выполнения вместе с входным и выходным пакетами. Когда программа выполняется, переменная выходного пакета заполняется и возвращается код выхода XDP. Это означает, что вы можете использовать выходной пакет и код возврата в тестовых проверках! Такую технику можно применять и для программ skb.

Чтобы обеспечить полноту и простоту этого примера, возьмем язык Python и его фреймворк для модульного тестирования.

XDP-тестирование с использованием фреймворка Python для тестирования модулей

Написание XDP-тестов с BPF_PROG_TEST_RUN и их интеграция с тестом инфраструктуры модульного тестирования Python — хорошая идея по нескольким причинам.

- Вы можете загружать и выполнять программы BPF с помощью библиотеки Python BCC.
- В Python одна из лучших доступных библиотек для создания и анализа пакетов — scapy.
- Python интегрируется со структурами C с использованием контрольных типов.

Как уже говорилось, нам нужно импортировать все необходимые библиотеки — это первое, что мы сделаем в файле с именем test_xdp.py:

```
from bcc import BPF, libbcc
from scapy.all import Ether, IP, raw, TCP, UDP

import ctypes
import unittest

class XDPExampleTestCase(unittest.TestCase):
    SKB_OUT_SIZE = 1514  # mtu 1500 + 14 ethernet size
    bpf_function = None
```

После того как все нужные библиотеки импортированы, можем продолжить и создать класс тестового примера с именем XDPExampleTestCase. Этот тестовый класс будет содержать все наши тесты и метод-член (_xdp_test_run),

который мы будем использовать для выполнения утверждений и вызова `bpf_prog_test_run`.

В следующем коде показано, как выглядит `_xdp_test_run`:

```
def _xdp_test_run(self, given_packet, expected_packet, expected_return):
    size = len(given_packet)

    given_packet = ctypes.create_string_buffer(raw(given_packet), size)
    packet_output = ctypes.create_string_buffer(self.SKB_OUT_SIZE)

    packet_output_size = ctypes.c_uint32()
    test_retval = ctypes.c_uint32()
    duration = ctypes.c_uint32()
    repeat = 1
    ret = libbcc.lib.bpf_prog_test_run(self.bpf_function.fd,
                                       repeat,
                                       ctypes.byref(given_packet),
                                       size,
                                       ctypes.byref(packet_output),
                                       ctypes.byref(packet_output_size),
                                       ctypes.byref(test_retval),
                                       ctypes.byref(duration))
    self.assertEqual(ret, 0)
    self.assertEqual(test_retval.value, expected_return)

    if expected_packet:
        self.assertEqual(
            packet_output[:packet_output_size.value], raw(expected_packet))
```

Требуются три аргумента:

- `given_packet` — это необработанный полученный интерфейсом пакет, с которым мы тестируем нашу программу XDP;
- `expected_packet` — это пакет, который мы ожидаем получить обратно после обработки программой XDP. Когда программа XDP возвращает `XDP_DROP` или `XDP_ABORT`, ожидается, что это будет `None`, во всех остальных случаях пакет остается таким же, как прежде, или может быть изменен;
- `expected_return` — это ожидаемое возвращение программы XDP после обработки *данного* пакета.

Если не учитывать аргументы, сам метод довольно прост. Он выполняет преобразование в типы C с помощью библиотеки `ctypes`, а затем вызывает `libbcc`,

эквивалентный BPF_PROG_TEST_RUN, — libbcc.lib.bpf_prog_test_run, используя в качестве аргументов теста пакеты и их метаданные. Затем выполняет все утверждения на основе результатов тестового вызова вместе с заданными значениями.

Теперь, когда функция создана, можно написать контрольные примеры, создав различные пакеты для проверки их поведения при прохождении через программу XDP. Но перед этим требуется написать метод setUp для теста.

Эта часть работы очень важна, потому что программа установки выполняет фактическую загрузку нашей программы BPF с именем myprogram, открывая и компилируя исходный файл program.c (в нем будет содержаться код XDP):

```
def setUp(self):
    bpf_prog = BPF(src_file=b"program.c")
    self.bpf_function = bpf_prog.load_func(b"myprogram", BPF.XDP)
```

Следующим после завершения настройки шагом является написание кода для того, что мы, собственно, хотим в данном случае. Не будучи слишком изобретательными, проверим, что отбрасываем все TCP-пакеты. Поэтому мы создаем пакет в given_packet, который является просто TCP-пакетом IPv4. Затем, используя метод подтверждения _xdp_test_run, проверяем, что с учетом нашего пакета мы получим обратно XDP_DROP без обратного пакета:

```
def test_drop_tcp(self):
    given_packet = Ether() / IP() / TCP()
    self._xdp_test_run(given_packet, None, BPF.XDP_DROP)
```

Поскольку этого недостаточно, проверяем, разрешены ли все пакеты UDP. Затем создаем два UDP-пакета — для given_packet и expected_packet, которые, по сути, одинаковы. Таким образом, мы также проверяем, что UDP-пакеты не изменяются, хотя и разрешены с помощью XDP_PASS:

```
def test_pass_udp(self):
    given_packet = Ether() / IP() / UDP()
    expected_packet = Ether() / IP() / UDP()
    self._xdp_test_run(given_packet, expected_packet, BPF.XDP_PASS)
```

Чтобы немного усложнить задачу, мы решили, что эта система затем передаст разрешенные пакеты TCP при условии, что они перейдут на порт 9090. Когда это произойдет, они также будут перезаписаны, чтобы изменился их MAC-

адрес назначения для перенаправления на определенный сетевой интерфейс с адресом `08:00:27:dd:38:2a`.

Например, сделаем следующим образом. Для `given_packet` в качестве порта назначения используется 9090, и нам требуется `expected_packet` с новым назначением и портом 9090:

```
def test_transform_dst(self):
    given_packet = Ether() / IP() / TCP(dport=9090)
    expected_packet = Ether(dst='08:00:27:dd:38:2a') / \
        IP() / TCP(dport=9090)
    self._xdp_test_run(given_packet, expected_packet, BPF.XDP_TX)
```

Имея множество тестов, напишем точку входа для тестовой программы, которая просто вызовет `unittest.main()`, а затем загрузит и выполнит тесты:

```
if __name__ == '__main__':
    unittest.main()
```

Итак, мы написали тесты для нашей программы XDP! Теперь, когда у нас есть тест — конкретный пример того, что мы хотим получить, — можем написать программу XDP, которая реализует его, создав файл с именем `program.c`.

Наша программа очень простая, она содержит только функцию XDP `myprogram` с проверенной логикой. Как всегда, первое, что нужно сделать, — включить необходимые заголовки, которые говорят сами за себя. У нас есть программа BPF, которая обрабатывает TCP/IP, передаваемый через Ethernet:

```
#define KBUILD_MODNAME "kmyprogram"

#include <linux/bpf.h>
#include <linux/if_ether.h>
#include <linux/tcp.h>
#include <linux/in.h>
#include <linux/ip.h>
```

Вновь, как и в ходе работы с другими программами в этой главе, нужно проверить смещения и заполнить переменные для трех уровней пакета: `ethhdr`, `iphdr` и `tcphdr` для Ethernet, IPv4 и TCP соответственно:

```
int myprogram(struct xdp_md *ctx) {
  int ipsize = 0;
  void *data = (void *)(long)ctx->data;
```

```
  void *data_end = (void *)(long)ctx->data_end;
  struct ethhdr *eth = data;
  struct iphdr *ip;
  struct tcphdr *th;

  ipsize = sizeof(*eth);
  ip = data + ipsize;
  ipsize += sizeof(struct iphdr);
  if (data + ipsize > data_end) {
    return XDP_DROP;
  }
```

Теперь, получив значения, можем реализовать нашу логику.

В первую очередь проверяем, является ли протокол TCP `ip-> protocol == IPPROTO_TCP`. Если это так, мы всегда применяем `XDP_DROP`, в противном случае используем `XDP_PASS` для всего остального.

При проверке протокола TCP выполняем еще одну проверку, чтобы выяснить, является ли порт назначения 9090 — `th-> dest == htons (9090)`. Если это так, изменяем MAC-адрес назначения на уровне Ethernet и возвращаем `XDP_TX`, чтобы отослать пакет через тот же NIC:

```
if (ip->protocol == IPPROTO_TCP) {
  th = (struct tcphdr *)(ip + 1);
  if ((void *)(th + 1) > data_end) {
    return XDP_DROP;
  }

  if (th->dest == htons(9090)) {
    eth->h_dest[0] = 0x08;
    eth->h_dest[1] = 0x00;
    eth->h_dest[2] = 0x27;
    eth->h_dest[3] = 0xdd;
    eth->h_dest[4] = 0x38;
    eth->h_dest[5] = 0x2a;
    return XDP_TX;
  }

  return XDP_DROP;
}

  return XDP_PASS;
}
```

Ну что ж, теперь можем запустить наши тесты:

```
sudo python test_xdp.py
```

Результат сообщит, что эти три теста пройдены:

```
...
--------------------------------
Ran 3 tests in 4.676s

OK
```

На данном этапе разобраться в происходящем несложно. Мы можем изменить последний `XDP_PASS` на `XDP_DROP` в `program.c` и посмотреть, что произойдет:

```
.F.
======================================================================
FAIL: test_pass_udp (__main__.XDPExampleTestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_xdp.py", line 48, in test_pass_udp
    self._xdp_test_run(given_packet, expected_packet, BPF.XDP_PASS)
  File "test_xdp.py", line 31, in _xdp_test_run
    self.assertEqual(test_retval.value, expected_return)
AssertionError: 1 != 2

----------------------------------------------------------------------
Ran 3 tests in 4.667s

FAILED (failures=1)
```

Тест не пройден — код состояния не совпадает и тестовая структура сообщила об ошибке. Это именно то, что нам требовалось! Получена эффективная среда тестирования, позволяющая уверенно писать программы XDP. Теперь у нас есть возможность делать что-то на конкретных этапах и изменять данное поведение в соответствии с тем, что нужно получить. Затем мы пишем код, чтобы показать это поведение в форме программы XDP.

MAC-адрес слишком короток для адреса Media Access Controll. Это уникальный идентификатор, состоящий из двух групп шестнадцатеричных цифр, которые есть у каждого сетевого интерфейса и которые применяются на канальном уровне (уровень 2 в модели OSI) для соединения устройств по таким технологиям, как Ethernet, Bluetooth и Wi-Fi.

Варианты использования XDP

Знакомясь с XDP, безусловно, полезно знать варианты его реализации различными организациями по всему миру. Это может помочь вам понять, почему в некоторых случаях XDP лучше, чем другие методы, такие как фильтрация сокетов или управление трафиком.

Начнем с наиболее часто употребляемого — мониторинга.

Мониторинг

Сейчас большинство систем мониторинга сети реализуются либо написанием модулей ядра, либо путем доступа к файлам процедур из пользовательского пространства. Написание, распространение и компиляция модулей ядра — задача не для всех, это опасная операция. Их нелегко поддерживать и отлаживать. Однако альтернатива может быть еще хуже. Чтобы получить такую информацию, как, например, количество пакетов, принятых картой в секунду, вам нужно открыть и уметь прочитать файл, в данном случае `/sys/class/net/eth0/statistics/rx_packets`. Это может показаться хорошей идеей, но для получения простой информации требуется много вычислений, поскольку открытый системный вызов в некоторых случаях обходится недешево в системном смысле.

Так что нам нужно решение, которое позволит реализовать функции, аналогичные функциям модуля ядра, без потери производительности. XDP идеально подходит для этого, потому что мы можем с помощью программы XDP отправлять данные, которые хотим извлечь в карту. Затем ее использует загрузчик, способный сохранять показатели в бэкенд-хранилище и применять к нему алгоритмы или отображать результаты в виде графика.

Миграция DDoS

Возможность видеть пакеты на уровне NIC гарантирует, что любой из них будет перехвачен на первом этапе, когда система еще не потратила значительную вычислительную мощность, чтобы понять, будут ли пакеты полезны для нее. В типичном сценарии карта `bpf` может инструктировать программу XDP для пакетов `XDP_DROP` из определенного источника. Список пакетов

может быть сгенерирован в пространстве пользователя после анализа пакетов, полученных через другую карту. Как только пакет, поступающий в программу XDP, совпадает с элементом списка, происходит его обработка. Пакет отбрасывается, так что ядру не придется тратить цикл процессора на его обработку. Это приводит к тому, что цель злоумышленника становится труднодостижимой, поскольку в таком случае он не может тратить впустую дорогостоящие вычислительные ресурсы.

Балансировка нагрузки

Интересный пример использования программ XDP — балансировка нагрузки, однако XDP может повторно передавать пакеты только на тот же сетевой адаптер, куда они поступили. Это означает, что XDP — не лучший вариант для реализации классического балансировщика нагрузки, который расположен перед всеми вашими серверами и перенаправляет трафик на них. Однако это не означает, что XDP не подходит как вариант. Если мы перенесем балансировку нагрузки с внешнего сервера на те же машины, которые обслуживают приложение, вы сразу увидите, как их сетевые карты можно использовать для балансировки.

Таким образом, мы можем создать распределитель нагрузки, где каждая машина, на которой размещается приложение, помогает распределять трафик на соответствующие серверы.

Брандмауэры

Когда люди представляют брандмауэр в Linux, они обычно думают о `iptables` или `net filter`. С XDP вы можете получить ту же функциональность полностью программируемым способом непосредственно в сетевой карте или ее драйвере. Обычно брандмауэры — это дорогие машины, расположенные поверх сетевого стека или между узлами для контроля связи. Однако при использовании XDP сразу становится ясно: поскольку программы XDP очень дешевы и быстры, мы могли бы внедрить логику брандмауэра непосредственно в сетевые адаптеры узлов вместо набора выделенных машин. Обычный вариант — иметь загрузчик XDP, который управляет картой с набором правил, измененных с помощью API удаленного вызова процедур. Затем набор правил, содержащихся в карте, динамически передается программам XDP,

загруженным на каждый конкретный компьютер, для управления тем, что он может получать, от кого и в какой ситуации.

Эта альтернатива не просто делает брандмауэр менее дорогим — каждый узел может развертывать собственный уровень межсетевого экрана, не полагаясь на программное обеспечение пользовательского пространства или ядро. Когда развертывание выполняется с помощью выгруженного XDP в рабочем режиме, мы получаем максимальное преимущество, потому что центральный процессор даже не обрабатывает пакеты.

Резюме

Вы получили действительно нужные знания! Мы обещаем, что XDP поможет вам думать о сетевых потоках совершенно по-другому. Необходимость полагаться на такие инструменты, как `iptables` или другие инструменты пользовательского пространства, работая с сетевыми пакетами, часто разочаровывает и замедляет процесс. XDP интересен тем, что он быстрее, так как способен обрабатывать пакеты напрямую, а вы можете написать собственную логику для работы с сетевыми пакетами. Поскольку весь созданный вами код может работать с картами и взаимодействовать с другими программами BPF, у вас есть множество вариантов его применения, для того чтобы изобретать и улучшать собственные архитектуры!

Несмотря на то что в следующей главе речь идет не о сети, мы рассмотрим многие концепции, описанные в главах 6 и 7. К тому же BPF используется для фильтрации некоторых условий на основе заданного ввода, а также того, что может сделать программа. Не забывайте, что F в BPF означает фильтр!

8 Безопасность ядра Linux, его возможности и Seccomp

BPF предоставляет мощный способ расширения ядра без ущерба для стабильности, безопасности и скорости. По этой причине разработчики ядра подумали, что было бы неплохо использовать его универсальность для улучшения изоляции процессов в Seccomp путем реализации фильтров Seccomp, поддерживаемых программами BPF, известными также как Seccomp BPF. В этой главе мы расскажем, что такое Seccomp и как он применяется. Затем вы узнаете, как писать фильтры Seccomp с помощью программ BPF. После этого рассмотрим встроенные ловушки BPF, которые есть в ядре для модулей безопасности Linux.

Модули безопасности Linux (LSM) — это платформа, предоставляющая набор функций, которые можно применять для стандартизированной реализации различных моделей безопасности. LSM может использоваться непосредственно в дереве исходного кода ядра, например Apparmor, SELinux и Tomoyo.

Начнем с обсуждения возможностей Linux.

Возможности

Суть возможностей Linux заключается в том, что вам нужно предоставить непривилегированному процессу разрешение на выполнение определенной задачи, но без использования `suid` для этой цели, или иным образом сделать процесс привилегированным, уменьшая возможность атак и обеспечивая процессу возможность выполнения определенных задач. Например, если вашему приложению нужно открыть привилегированный порт, допустим, 80,

вместо запуска процесса от имени root можете просто предоставить ему возможность CAP_NET_BIND_SERVICE.

Рассмотрим программу Go с именем main.go:

```
package main

import (
         "net/http"
         "log"
)

func main() {
    log.Fatalf("%v", http.ListenAndServe(":80", nil))
}
```

Эта программа обслуживает HTTP-сервер на порте 80 (это привилегированный порт). Обычно мы запускаем ее сразу после компиляции:

```
$ go build -o capabilities main.go
$ ./capabilities
```

Однако, поскольку мы не предоставляем привилегии root, этот код выдаст ошибку при привязке порта:

```
2019/04/25 23:17:06 listen tcp :80: bind: permission denied
exit status 1
```

capsh (средство управления оболочкой) — это инструмент, который запускает оболочку с определенным набором возможностей.

В этом случае, как уже говорилось, вместо предоставления полных прав root можно разрешить привязку привилегированных портов, предоставив возможность cap_net_bind_service наряду со всеми остальными, которые уже есть в программе. Для этого можем заключить нашу программу в capsh:

```
# capsh --caps='cap_net_bind_service+eip cap_setpcap,cap_setuid,cap_
setgid+ep' \
    --keep=1 --user="nobody" \
    --addamb=cap_net_bind_service -- -c "./capabilities"
```

Немного разберемся в этой команде.

- `capsh` — используем `capsh` в качестве оболочки.
- `--caps='cap_net_bind_service+eip cap_setpcap,cap_setuid,cap_setgid+ep'` — поскольку нам нужно сменить пользователя (мы не хотим запускаться с правами root), укажем `cap_net_bind_service` и возможность фактически изменить идентификатор пользователя с `root` на `nobody`, а именно `cap_setuid` и `cap_setgid`.
- `--keep=1` — хотим сохранить установленные возможности, когда выполнено переключение с аккаунта root.
- `--user="nobody"` — конечным пользователем, запускающим программу, будет `nobody`.
- `--addamb=cap_net_bind_service` — задаем очистку связанных возможностей после переключения из режима root.
- `-- -c "./capabilities"` — просто запускаем программу.

Связанные возможности — это особый вид возможностей, которые наследуются дочерними программами, когда текущая программа выполняет их с помощью execve(). Наследоваться могут только возможности, разрешенные как связанные, или, другими словами, как возможности окружения.

Вероятно, вам интересно, что значит `+eip` после указания возможности в опции `--caps`. Эти флаги используются для определения того, что возможность:

- должна быть активирована (`p`);
- доступна для применения (`e`);
- может быть унаследована дочерними процессами (`i`).

Поскольку мы хотим использовать `cap_net_bind_service`, нужно сделать это с флагом `e`. Затем запустим оболочку в команде. В результате запустится двоичный файл `capabilities`, и нам нужно пометить его флагом `i`. Наконец, мы хотим, чтобы возможность была активирована (мы это сделали, не меняя UID) с помощью `p`. Это выглядит как `cap_net_bind_service+eip`.

Можете проверить результат с помощью `ss`. Немного сократим вывод, чтобы он поместился на странице, но он покажет связанный порт и идентификатор пользователя, отличные от 0, в данном случае 65 534:

```
# ss -tulpn -e -H | cut -d' ' -f17-
128    *:80    *:*
users:(("capabilities",pid=30040,fd=3)) uid:65534 ino:11311579 sk:2c v6only:0
```

В этом примере мы использовали `capsh`, но вы можете написать оболочку с помощью `libcap`. За дополнительной информацией обращайтесь к `man 3 libcap`.

При написании программ разработчик довольно часто не знает заранее всех возможностей, необходимых программе во время выполнения; более того, в новых версиях эти возможности могут измениться.

Чтобы лучше понять возможности нашей программы, можем взять инструмент BCC `capable`, который устанавливает kprobe для функции ядра `cap_capable`:

```
 /usr/share/bcc/tools/capable
TIME      UID    PID    TID    COMM            CAP  NAME                 AUDIT
10:12:53  0      424    424    systemd-udevd   12   CAP_NET_ADMIN        1
10:12:57  0      1103   1101   timesync        25   CAP_SYS_TIME         1
10:12:57  0      19545  19545  capabilities    10   CAP_NET_BIND_SERVICE 1
```

Добиться того же самого мы можем, используя `bpftrace` с однострочным kprobe в функции ядра `cap_capable`:

```
bpftrace -e \
    'kprobe:cap_capable {
        time("%H:%M:%S ");
        printf("%-6d %-6d %-16s %-4d %d\n", uid, pid, comm, arg2, arg3);
    }' \
    | grep -i capabilities
```

Это выведет что-то вроде следующего, если возможности нашей программы будут активированы после kprobe:

```
12:01:56  1000  13524  capabilities  21  0
12:01:56  1000  13524  capabilities  21  0
12:01:56  1000  13524  capabilities  21  0
12:01:56  1000  13524  capabilities  12  0
12:01:56  1000  13524  capabilities  12  0
12:01:56  1000  13524  capabilities  12  0
12:01:56  1000  13524  capabilities  12  0
12:01:56  1000  13524  capabilities  10  1
```

Пятый столбец — это возможности, в которых нуждается процесс, и, поскольку эти выходные данные включают в том числе неаудитные события, мы видим все неаудитные проверки и, наконец, требуемую возможность с флагом аудита (последний в выводе), установленным в `1`. Возможность, которая нас интересует, — `CAP_NET_BIND_SERVICE`, она определяется как константа в исходном коде ядра в файле `include/uapi/linux/ability.h` с идентификатором `10`:

```
/* Allows binding to TCP/UDP sockets below 1024 */
/* Allows binding to ATM VCIs below 32 */

#define CAP_NET_BIND_SERVICE 10
```

Возможности часто задействуются во время выполнения контейнеров, таких как runC или Docker, чтобы они работали в непривилегированном режиме, но им были разрешены только те возможности, которые необходимы для запуска большинства приложений. Когда приложению требуются определенные возможности, в Docker можно обеспечить их с помощью `--cap-add`:

```
docker run -it --rm --cap-add=NET_ADMIN ubuntu ip link add dummy0 type dummy
```

Эта команда предоставит контейнеру возможность `CAP_NET_ADMIN`, что позволит ему настроить сетевую ссылку для добавления интерфейса `dummy0`.

В следующем разделе показано использование таких возможностей, как фильтрация, но с помощью другого метода, который позволит нам программно реализовать собственные фильтры.

Seccomp

Seccomp означает Secure Computing, это уровень безопасности, реализованный в ядре Linux, который позволяет разработчикам фильтровать определенные системные вызовы. Хотя Seccomp сопоставим с возможностями Linux, его способность управлять определенными системными вызовами делает его намного более гибким по сравнению с ними.

Seccomp и возможности Linux не исключают друг друга, их часто используют вместе, чтобы получить пользу от обоих подходов. Например, вы можете захотеть предоставить процессу возможность `CAP_NET_ADMIN`, но

не разрешить ему принимать соединения через сокет, блокируя системные вызовы `accept` и `accept4`.

Способ фильтрации Seccomp основан на фильтрах BPF, работающих в режиме `SECCOMP_MODE_FILTER`, и фильтрация системных вызовов выполняется так же, как и для пакетов.

Фильтры Seccomp загружаются с использованием `prctl` через операцию `PR_SET_SECCOMP`. Эти фильтры имеют форму программы BPF, которая выполняется для каждого пакета Seccomp, представленного с помощью структуры `seccomp_data`. Эта структура содержит эталонную архитектуру, указатель инструкций процессора во время системного вызова и максимум шесть аргументов системного вызова, выраженных как `uint64`.

Вот как выглядит структура `seccomp_data` из исходного кода ядра в файле `linux/seccomp.h`:

```
struct seccomp_data {
        int nr;
        __u32 arch;
        __u64 instruction_pointer;
        __u64 args[6];
};
```

Как видно из этой структуры, мы можем фильтровать по системному вызову, его аргументам или их комбинации.

После получения каждого пакета Seccomp фильтр обязан выполнить обработку, чтобы принять окончательное решение, и сообщить ядру, что делать дальше. Окончательное решение выражается одним из возвращаемых значений (кодов состояния).

- `SECCOMP_RET_KILL_PROCESS` — завершение всего процесса сразу же после фильтрации системного вызова, который из-за этого не выполняется.
- `SECCOMP_RET_KILL_THREAD` — завершение текущего потока сразу же после фильтрации системного вызова, который из-за этого не выполняется.
- `SECCOMP_RET_KILL` — алиас для `SECCOMP_RET_KILL_THREAD`, оставлен для обратной совместимости.
- `SECCOMP_RET_TRAP` — системный вызов запрещен, и сигнал `SIGSYS` (Bad System Call) отправляется вызывающей его задаче.

- SECCOMP_RET_ERRNO — системный вызов не выполняется, и часть возвращаемого значения фильтра SECCOMP_RET_DATA передается в пространство пользователя как значение errno. В зависимости от причины ошибки возвращаются разные значения errno. Список номеров ошибок приведен в следующем разделе.
- SECCOMP_RET_TRACE — используется для уведомления трассировщика ptrace с помощью PTRACE_O_TRACESECCOMP для перехвата, когда выполняется системный вызов, чтобы видеть и контролировать этот процесс. Если трассировщик не подключен, возвращается ошибка, errno устанавливается в -ENOSYS, а системный вызов не выполняется.
- SECCOMP_RET_LOG — системный вызов разрешен и зарегистрирован в журнале.
- SECCOMP_RET_ALLOW — системный вызов просто разрешен.

ptrace — это системный вызов для реализации механизмов трассировки в процессе, называемом tracee, с возможностью наблюдения и контроля за выполнением процесса. Программа трассировки может эффективно влиять на выполнение и изменять регистры памяти tracee. В контексте Seccomp ptrace используется, когда запускается кодом состояния SECCOMP_RET_TRACE, следовательно, трассировщик может предотвратить выполнение системного вызова и реализовать собственную логику.

Ошибки Seccomp

Время от времени, работая с Seccomp, вы будете сталкиваться с различными ошибками, которые идентифицируются возвращаемым значением типа SECCOMP_RET_ERRNO. Чтобы сообщить об ошибке, системный вызов seccomp вернет -1 вместо 0.

Возможны следующие ошибки:

- EACCESS — вызывающей стороне не разрешено делать системный вызов. Обычно это происходит потому, что у нее нет привилегий CAP_SYS_ADMIN или же не установлен no_new_privs с помощью prctl (об этом поговорим позже);
- EFAULT — переданные аргументы (args в структуре seccomp_data) не имеют действительного адреса;

- `EINVAL` — здесь может быть четыре причины:
  - запрошенная операция неизвестна или не поддерживается ядром в текущей конфигурации;
  - указанные флаги недействительны для запрошенной операции;
  - операция включает `BPF_ABS`, но есть проблемы с указанным смещением, которое может превышать размер структуры `seccomp_data`;
  - количество инструкций, переданных в фильтр, превышает максимальное;
- `ENOMEM` — недостаточно памяти для выполнения программы;
- `EOPNOTSUPP` — операция указала, что с `SECCOMP_GET_ACTION_AVAIL` действие было доступно, однако ядро не поддерживает возврат в аргументах;
- `ESRCH` — возникла проблема при синхронизации другого потока;
- `ENOSYS` — нет трассировщика, прикрепленного к действию `SECCOMP_RET_TRACE`.

prctl — это системный вызов, который позволяет программе пользовательского пространства управлять (устанавливать и получать) конкретными аспектами процесса, такими как порядковый номер байтов, имена потоков, режим защищенных вычислений (Seccomp), привилегии, события Perf и т. д.

Seccomp может показаться вам технологией песочницы, но это не так. Seccomp — утилита, которая позволяет пользователям разрабатывать механизм песочницы. Теперь рассмотрим, как создаются программы пользовательских взаимодействий с использованием фильтра, вызываемого непосредственно системным вызовом Seccomp.

Пример фильтра BPF Seccomp

Здесь мы покажем, как соединить два рассмотренных ранее действия, а именно:

- напишем программу Seccomp BPF, которая будет применяться в качестве фильтра с различными кодами возврата в зависимости от принимаемых решений;
- загрузим фильтр, используя `prctl`.

Для начала нужны заголовки из стандартной библиотеки и ядра Linux:

```
#include <errno.h>
#include <linux/audit.h>
#include <linux/bpf.h>
#include <linux/filter.h>
#include <linux/seccomp.h>
#include <linux/unistd.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/prctl.h>
#include <unistd.h>
```

Прежде чем пытаться выполнить этот пример, мы должны убедиться, что ядро скомпилировано с `CONFIG_SECCOMP` и `CONFIG_SECCOMP_FILTER`, установленными в `y`. На рабочей машине это можно проверить так:

```
cat /proc/config.gz| zcat   | grep -i CONFIG_SECCOMP
```

Остальная часть кода представляет собой функцию `install_filter`, состоящую из двух частей. Первая часть содержит наш список инструкций по фильтрации BPF:

```
static int install_filter(int nr, int arch, int error) {
  struct sock_filter filter[] = {
    BPF_STMT(BPF_LD + BPF_W + BPF_ABS, (offsetof(struct seccomp_data, arch))),
    BPF_JUMP(BPF_JMP + BPF_JEQ + BPF_K, arch, 0, 3),
    BPF_STMT(BPF_LD + BPF_W + BPF_ABS, (offsetof(struct seccomp_data, nr))),
    BPF_JUMP(BPF_JMP + BPF_JEQ + BPF_K, nr, 0, 1),
    BPF_STMT(BPF_RET + BPF_K, SECCOMP_RET_ERRNO | (error & SECCOMP_RET_DATA)),
    BPF_STMT(BPF_RET + BPF_K, SECCOMP_RET_ALLOW),
  };
```

Инструкции устанавливаются с помощью макросов `BPF_STMT` и `BPF_JUMP`, определенных в файле `linux/filter.h`.

Пройдемся по инструкциям.

- `BPF_STMT(BPF_LD + BPF_W + BPF_ABS (offsetof(struct seccomp_data, arch)))` — система загружает и накапливает с `BPF_LD` в форме слова `BPF_W`, пакетные данные располагаются с фиксированным смещением `BPF_ABS`.
- `BPF_JUMP(BPF_JMP + BPF_JEQ + BPF_K, arch, 0, 3)` — проверяет с применением `BPF_JEQ`, является ли значение архитектуры в константе аккумулятора

BPF_K равным arch. Если это так, переходит со смещением 0 к следующей инструкции, в противном случае, чтобы выдать ошибку, перепрыгнет со смещением 3 (в данном случае), потому что arch не совпадает.

- BPF_STMT(BPF_LD + BPF_W + BPF_ABS (offsetof(struct seccomp_data, nr))) — загружает и накапливает с BPF_LD в форме слова BPF_W, которое является номером системного вызова, содержащимся в фиксированном смещении BPF_ABS.
- BPF_JUMP(BPF_JMP + BPF_JEQ + BPF_K, nr, 0, 1) — сравнивает номер системного вызова со значением переменной nr. Если они равны, переходит к следующей инструкции и запрещает системный вызов, в противном случае разрешает системный вызов с SECCOMP_RET_ALLOW.
- BPF_STMT(BPF_RET + BPF_K, SECCOMP_RET_ERRNO | (error & SECCOMP_RET_DATA)) — завершает программу с BPF_RET и в результате выдает ошибку SECCOMP_RET_ERRNO с номером из переменной err.
- BPF_STMT(BPF_RET + BPF_K, SECCOMP_RET_ALLOW) — завершает программу с BPF_RET и разрешает выполнение системного вызова с помощью SECCOMP_RET_ALLOW.

SECCOMP — ЭТО CBPF

Возможно, вам интересно, почему вместо скомпилированного объекта ELF или программы на C, скомпилированной с JIT, используется список инструкций.

На это есть две причины.

- Во-первых, Seccomp применяет cBPF (классический BPF), а не eBPF, что означает: у него нет реестров, а есть только аккумулятор для хранения последнего результата вычислений, как можно увидеть в примере.
- Во-вторых, Seccomp принимает указатель на массив инструкций BPF напрямую и ничего больше. Макросы, которые мы использовали, всего лишь помогают указать эти инструкции в удобной для программистов форме.

Если вам нужна дополнительная помощь, чтобы понять эту сборку, рассмотрите псевдокод, который делает то же самое:

```
if (arch != AUDIT_ARCH_X86_64) {
    return SECCOMP_RET_ALLOW;
}
```

```
if (nr == __NR_write) {
    return SECCOMP_RET_ERRNO;
}
return SECCOMP_RET_ALLOW;
```

После определения кода фильтра в структуре `socket_filter` нужно определить `sock_fprog`, содержащий код и вычисленную длину фильтра. Эта структура данных необходима в качестве аргумента для объявления работы процесса в дальнейшем:

```
struct sock_fprog prog = {
  .len = (unsigned short)(sizeof(filter) / sizeof(filter[0])),
  .filter = filter,
};
```

Осталось только одно, что нужно сделать в функции `install_filter`, — загрузить саму программу! Для этого используем `prctl`, взяв `PR_SET_SECCOMP` в качестве опции, чтобы войти в режим защищенных вычислений. Затем укажем режиму загружать фильтр с помощью `SECCOMP_MODE_FILTER`, который содержится в переменной `prog` типа `sock_fprog`:

```
  if (prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, &prog)) {
    perror("prctl(PR_SET_SECCOMP)");
    return 1;
  }
  return 0;
}
```

Наконец, можем воспользоваться нашей функцией `install_filter`, но перед этим нужно задействовать `prctl`, чтобы установить `PR_SET_NO_NEW_PRIVS` для текущего выполнения и тем самым избежать ситуации, когда дочерние процессы получают более широкие привилегии, чем родительские. При этом мы можем делать следующие вызовы `prctl` в функции `install_filter`, не имея прав root.

Теперь мы можем вызвать функцию `install_filter`. Заблокируем все системные вызовы `write`, относящиеся к архитектуре `X86-64`, и просто дадим разрешение, блокирующее все попытки. После установки фильтра продолжаем выполнение, используя первый аргумент:

```
int main(int argc, char const *argv[]) {
  if (prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0)) {
    perror("prctl(NO_NEW_PRIVS)");
    return 1;
  }
```

```
  install_filter(__NR_write, AUDIT_ARCH_X86_64, EPERM);
  return system(argv[1]);
}
```

Приступим. Для компиляции нашей программы мы можем использовать либо `clang`, либо `gcc`, в любом случае это просто компиляция файла `main.c` без специальных опций:

```
clang main.c -o filter-write
```

Как было отмечено, мы заблокировали все записи в программе. Чтобы проверить это, нужна программа, которая что-то выводит, — `ls` кажется хорошим кандидатом. Вот как она обычно себя ведет:

```
ls -la
total 36
drwxr-xr-x 2 fntlnz users  4096 Apr 28 21:09 .
drwxr-xr-x 4 fntlnz users  4096 Apr 26 13:01 ..
-rwxr-xr-x 1 fntlnz users 16800 Apr 28 21:09 filter-write
-rw-r--r-- 1 fntlnz users    19 Apr 28 21:09 .gitignore
-rw-r--r-- 1 fntlnz users  1282 Apr 28 21:08 main.c
```

Прекрасно! Вот как выглядит использование нашей программы-оболочки: мы просто передаем программу, которую хотим протестировать, в качестве первого аргумента:

```
./filter-write "ls -la"
```

После выполнения эта программа выдает совершенно пустой вывод. Тем не менее мы можем применить `strace`, чтобы увидеть, что происходит:

```
strace -f ./filter-write "ls -la"
```

Результат работы сильно укорочен, но соответствующая его часть показывает, что записи блокируются с ошибкой `EPERM` — той самой, которую мы настроили. Это означает, что программа ничего не выводит, потому что не может получить доступ к системному вызову `write`:

```
[pid 25099] write(2, "ls: ", 4)         = -1 EPERM (Operation not permitted)
[pid 25099] write(2, "write error", 11) = -1 EPERM (Operation not permitted)
[pid 25099] write(2, "\n", 1)           = -1 EPERM (Operation not permitted)
```

Теперь вы понимаете, как работает Seccomp BPF, и хорошо представляете, что можно сделать с его помощью. Но разве не хотелось бы добиться того же с помощью eBPF вместо cBPF, чтобы использовать всю его мощь?

Размышляя о программах eBPF, большинство людей думает, что они просто пишут их и загружают с привилегиями администратора. Хотя это утверждение в целом верно, ядро реализует набор механизмов для защиты объектов eBPF на различных уровнях. Эти механизмы называются ловушками BPF LSM.

Ловушки BPF LSM

Чтобы обеспечить независимый от архитектуры контроль системных событий, LSM реализует концепцию ловушек. Технически вызов ловушки похож на системный вызов, однако независим от системы и интегрирован с инфраструктурой. LSM предоставляет новую концепцию, в которой уровень абстракции способен помочь избежать проблем, возникающих в ходе работы с системными вызовами на разных архитектурах.

На момент написания книги ядро имело семь ловушек, связанных с BPF-программами, и SELinux — единственный встроенный LSM, реализующий их.

Исходный код ловушек размещен в дереве ядра в файле `include/linux/security.h`:

```
extern int security_bpf(int cmd, union bpf_attr *attr, unsigned int size);
extern int security_bpf_map(struct bpf_map *map, fmode_t fmode);
extern int security_bpf_prog(struct bpf_prog *prog);
extern int security_bpf_map_alloc(struct bpf_map *map);
extern void security_bpf_map_free(struct bpf_map *map);
extern int security_bpf_prog_alloc(struct bpf_prog_aux *aux);
extern void security_bpf_prog_free(struct bpf_prog_aux *aux);
```

Каждая из них будет вызываться на разных этапах выполнения:

- `security_bpf` — проводит начальную проверку выполненных системных вызовов BPF;
- `security_bpf_map` — проверяет, когда ядро возвращает файловый дескриптор для карты;
- `security_bpf_prog` — проверяет, когда ядро возвращает дескриптор файла для программы eBPF;
- `security_bpf_map_alloc` — проверяет, инициализировано ли поле безопасности внутри карт BPF;

- `security_bpf_map_free` — проверяет, выполняется ли очистка поля безопасности внутри карт BPF;
- `security_bpf_prog_alloc` — проверяет, инициализируется ли поле безопасности внутри BPF-программ;
- `security_bpf_prog_free` — проверяет, очищается ли поле безопасности внутри BPF-программ.

Теперь, видя все это, мы понимаем: идея перехватчиков LSM BPF заключается в том, что они могут обеспечить защиту каждого объекта eBPF, гарантируя, что только те из них, которые имеют соответствующие привилегии, могут выполнять операции над картами и программами.

Резюме

Безопасность — это не то, что вы можете внедрить универсальным способом для всего, что хотите уберечь. Важно иметь возможность защищать системы на разных уровнях и разными способами. Хотите верьте, хотите нет, лучший способ обезопасить систему — организовывать разные уровни защиты с разных позиций, чтобы снижение безопасности одного уровня не позволило получить доступ ко всей системе. Разработчики ядра проделали большую работу, предоставив нам набор различных слоев и точек взаимодействия. Мы надеемся, что дали вам хорошее представление о том, что такое слои и как использовать программы BPF для работы с ними.

9 Реальные способы применения

При внедрении новой технологии нужно задать себе очень важный вопрос: «Каковы варианты использования всего этого?» Вот почему мы решили взять интервью у создателей некоторых самых интересных проектов BPF, чтобы они могли поделиться своими идеями.

Режим God Mode от Sysdig eBPF

Sysdig — компания, которая создает одноименный инструмент для устранения неполадок Linux с открытым исходным кодом, начала работать с eBPF в 2017 году под ядром 4.11.

Поначалу модуль ядра использовался для извлечения и выполнения всей работы на стороне ядра, но по мере увеличения базы пользователей и когда все больше и больше компаний начали экспериментировать, Sysdig признала, что это ограничивает большинство внешних участников процесса во многих аспектах.

- Растет число пользователей, которые не могут загружать модули ядра на свои машины. Облачные платформы становятся все более и более строгими в отношении того, что могут делать программы во время выполнения.
- И новые, и даже старые участники не понимают архитектуру модуля ядра. Это уменьшает их общее количество и является ограничивающим фактором для роста проекта.

- Обслуживание модулей ядра затруднено не только из-за написания кода, но и из-за усилий, необходимых для обеспечения его безопасности и правильной организации.

В связи с этим Sysdig решила попробовать подход, предусматривающий написание того же набора функций, что и в модуле, но с помощью программы eBPF. Еще одно преимущество, которое автоматически обеспечивает внедрение eBPF, — это возможность для Sysdig в дальнейшем использовать другие полезные функции трассировки eBPF. Например, довольно легко прикрепить программы eBPF к определенным точкам выполнения в приложении пользовательского пространства с помощью пользовательских зондов, как описано в разделе «Зонды пользовательского пространства».

Кроме того, теперь проект может задействовать собственные вспомогательные возможности в программах eBPF для отслеживания треков запущенных процессов, чтобы понять, что именно происходит в системе. Это дает пользователям еще больше информации для устранения неполадок.

Несмотря на то что сейчас все иначе, вначале Sysdig столкнулась с некоторыми проблемами из-за ограничений виртуальной машины eBPF, поэтому главный архитектор проекта Джанлука Борелло решил улучшить ее, добавив патчи для восходящих потоков к самому ядру, в том числе:

- возможность с самого начала работать со строками в программах eBPF (https://oreil.ly/ZJ09y);
- несколько патчей для улучшения семантики аргументов в программах eBPF 1 (https://oreil.ly/lPcGT), eBPF 2 (https://oreil.ly/5S_tR) и eBPF 3 (https://oreil.ly/HLrEu).

Последнее особенно важно для работы с аргументами системного вызова — вероятно, наиболее важного источника данных, доступного в инструменте.

На рис. 9.1 показана архитектура режима eBPF в Sysdig.

Ядром реализации является набор пользовательских программ eBPF, отвечающих за инструментарий. Эти программы написаны в подмножестве языка программирования C. Они скомпилированы с использованием последних версий Clang и LLVM, которые переводят высокоуровневый код C в байткод eBPF.

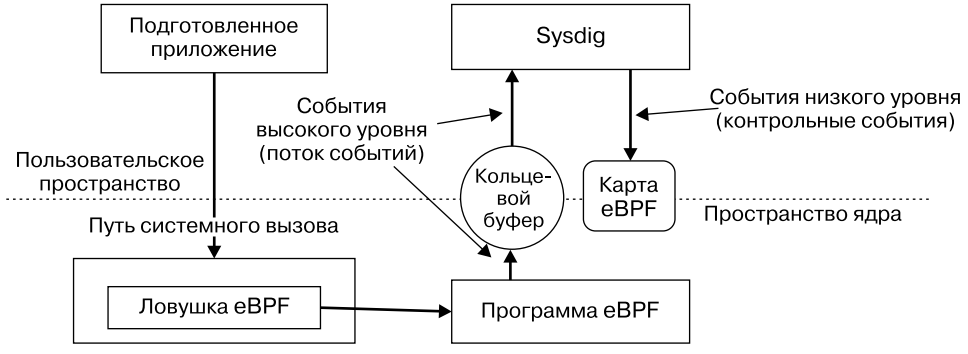

Рис. 9.1. Архитектура Sysdig eBPF

Существует одна программа eBPF для каждой отдельной точки выполнения, где Sysdig использует ядро. В настоящее время программы eBPF привязаны к следующим статическим точкам трассировки:

- входной путь системного вызова;
- путь выхода системного вызова;
- переключение контекста процесса;
- завершение процесса;
- незначительные и основные ошибки страницы;
- процесс доставки сигнала.

Каждая программа принимает данные точки выполнения (например, для системных вызовов — аргументы, переданные вызывающим процессом) и начинает обрабатывать их. Обработка зависит от типа системного вызова. Для простых системных вызовов аргументы просто копируются дословно в карту eBPF для временного хранения, пока не будет сформирован весь кадр события. Для других, более сложных вызовов программы eBPF включают логику для перевода или дополнения аргументов. Это позволяет приложению Sysdig в пользовательском пространстве задействовать данные целиком.

Некоторые из дополнительных данных:

- данные, связанные с сетевым подключением (TCP/UDP, IPv4/IPv6, имена сокетов UNIX и т. д.);
- высокодетализированные метрики о процессе (показатели счетчиков памяти, количество сбоев страниц, длина очереди сокетов и т. д.);

- данные, специфичные для контейнера, такие как контрольные группы, к которым принадлежит процесс, выдавший системный вызов, а также пространства имен, в которых существует процесс.

Как показано на рис. 9.1, захватив все необходимые данные для определенного системного вызова, программа eBPF использует специальную встроенную функцию BPF для передачи данных в набор кольцевых буферов для каждого CPU, которые может задействовать приложение пользовательского пространства, читая с очень высокой пропускной способностью. Именно здесь применение eBPF в Sysdig отличается от типичной парадигмы применения карт eBPF для обмена с пространством пользователя небольшими данными, созданными в пространстве ядра. Чтобы узнать больше о картах и о том, как пространствам пользователя и ядра общаться между собой, обратитесь к главе 3.

С точки зрения производительности результаты хорошие! На рис. 9.2 показано, что затраты на инструментарий eBPF в Sysdig незначительно превышают расходы на классические инструменты модуля ядра.

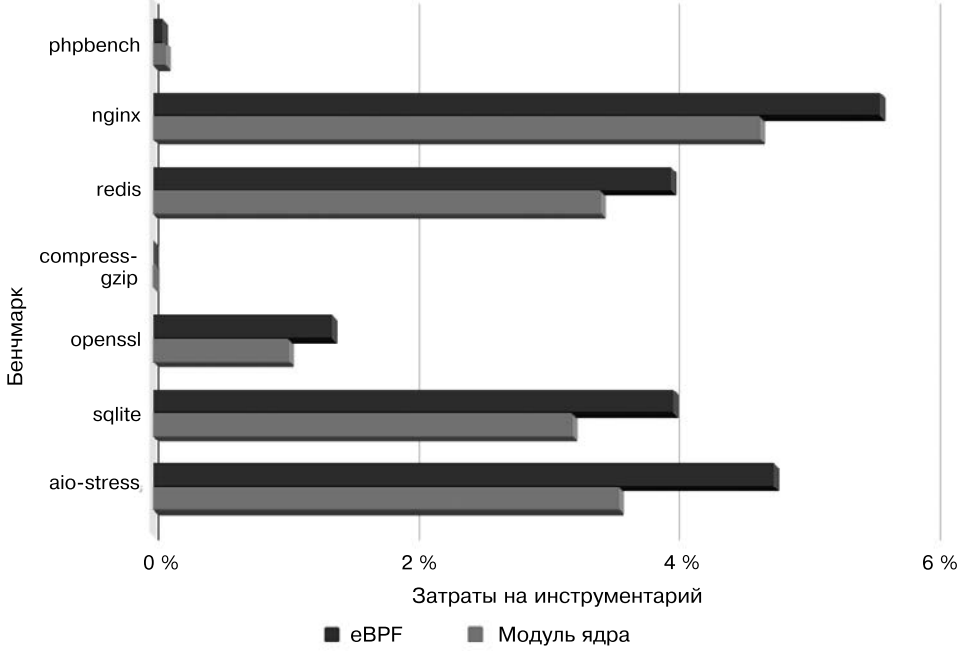

Рис. 9.2. Сравнение производительности Sysdig eBPF

Вы можете попробовать применить Sysdig и воспользоваться его поддержкой eBPF, следуя инструкциям по использованию (https://oreil.ly/luHKp), и обязательно посмотрите код драйвера BPF (https://oreil.ly/AJddM).

Flowmill

Стартап Flowmill появился на основе научного исследовательского проекта Flowtune (https://oreil.ly/e9heR), основанного Джонатаном Перри. В рамках Flowtune изучали, как эффективно планировать отдельные пакеты в перегруженных сетях центров обработки данных. Одним из основных элементов технологии, необходимых для этой работы, был способ сбора сетевой телеметрии с чрезвычайно низкими издержками. Flowmill адаптировал эту технологию для наблюдения, агрегирования и анализа связей между каждым компонентом в распределенном приложении для решения следующих задач:

- обеспечения точного представления о том, как сервисы взаимодействуют в распределенной системе;
- определения областей, в которых произошли статистически значимые изменения в скорости трафика, ошибках или задержке.

Flowmill использует зонды ядра eBPF для отслеживания каждого открытого сокета и периодического сбора метрик операционных систем. Это сложно по ряду причин.

- Необходимо подключать как новые, так и существующие соединения, уже открытые на момент установления зондов eBPF. Кроме того, зонды должны учитывать TCP, UDP, а также пути кода IPv4 и IPv6 через ядро.
- Для систем на основе контейнеров каждый сокет должен быть отнесен к соответствующей контрольной группе и присоединен к метаданным соединителя с платформой, такой как Kubernetes или Docker.
- Трансляция сетевых адресов, выполняемая через conntrack, должна быть оснащена инструментами для установления соответствия между сокетами и их видимыми IP-адресами. Например, в Docker и Kubernetes общая сетевая модель задействует исходный NAT для маскировки контейнеров

за IP-адресом хоста, а виртуальный IP-адрес службы используется для представления набора контейнеров.

- Данные, собранные программами eBPF, следует обработать, чтобы сопоставить сведения, полученные на двух сторонах соединения.

Добавление зондов ядра eBPF обеспечивает гораздо более эффективный и надежный способ сбора этих данных. При этом полностью исключается риск пропущенных соединений и все может быть сделано с минимальными издержками на каждом сокете в течение секунды. Подход Flowmill основан на агенте, который сочетает в себе набор зондов ядра eBPF и сбор метрик пространства пользователя, а также агрегирование и постобработку. Здесь активно используются кольца Perf для передачи метрик, собранных в каждом сокете, в пространство пользователя для дальнейшей обработки. Кроме того, он применяет хеш-карту для отслеживания открытых сокетов TCP и UDP.

Компания Flowmill обнаружила, что, как правило, существует две стратегии проектирования измерительных методик eBPF. Простой подход находит одну-две функции ядра, которые вызываются для каждого инструментованного события, но требует, чтобы код BPF поддерживал больше информации и выполнял больше работы за каждый вызов, причем очень часто. Чтобы уменьшить опасения по поводу того, что контрольно-измерительные средства влияют на производственные нагрузки, Flowmill применил другую стратегию: инструментам назначались более специфические функции, которые вызываются реже и обозначают важное событие. Это значительно снижает накладные расходы, но требует бóльших усилий для охвата всех важных путей кода, особенно в разных версиях ядра по мере его развития.

Например, `tcp_v4_do_rcv` перехватывает весь трафик TCP RX по установленным соединениям и имеет доступ к носителю `struct`, но у него чрезвычайно большой объем вызовов. Вместо этого пользователи могут задействовать функции, связанные с ACK, обработкой пакетов не по порядку, оценкой RTT, и многие другие, которые позволяют обрабатывать определенные события, влияющие на известные метрики.

При таком подходе к TCP, UDP, процессам, контейнерам, conntrack и другим подсистемам система достигает чрезвычайно хорошей производительности с довольно низкими издержками, которые зачастую трудно измерить. Загрузка процессора обычно составляет 0,1... 0,25 % на ядро, включая eBPF

и компоненты пользовательского пространства, и зависит главным образом от скорости создания новых сокетов.

Подробнее о Flowmill и Flowtune можно узнать на их сайте (https://www.flowmill.com).

Sysdig и Flowmill являются пионерами в использовании BPF для создания инструментов мониторинга и наблюдения, но они не единственные. В книге мы упоминали о других компаниях, например Cillium и Facebook, которые выбрали BPF в качестве основы для разработки высоконадежной и эффективной сетевой инфраструктуры. Мы будем очень рады узнать о том, что вам удалось создать с помощью BPF.

Об авторах

Дэвид Калавера является техническим директором в Netlify. Он работал в службе поддержки Docker и участвовал в разработке инструментов Runc, Go и BCC, а также других проектов с открытым исходным кодом. Известен своей работой над проектами Docker и развитием экосистемы плагинов Docker. Дэвид очень любит флейм-графы и всегда стремится к оптимизации производительности.

Лоренцо Фонтана трудится в команде разработчиков ПО с открытым исходным кодом в Sysdig, где в основном занимается Falco — проектом Cloud Native Computing Foundation, который обеспечивает безопасность среды выполнения контейнеров и обнаружение аномалий через модуль ядра и eBPF. Он увлечен распределенными системами, программно определяемыми сетями, ядром Linux и анализом производительности.

Об обложке

Птица на обложке — это совка Санда *(Otus lempiji)*. Совки — это маленькие совы с «ушками» из перьев на голове. Совка Санда родом из Юго-Восточной Азии, известна также как сингапурская совка. Изначально эти птицы обитали в лесах, но приспособились к урбанизации и сегодня живут в садах.

Совки Санда светло-коричневые, с черными крапинками. Они достигают примерно 20 сантиметров в высоту, весят около 170 граммов. Ранней весной самки откладывают два-три яйца в дуплах деревьев. Совки питаются насекомыми, чаще жуками, а также охотятся на грызунов, ящериц и мелких птиц.

Многие из животных на обложках книг, выпущенных издательством O'Reilly, находятся под угрозой исчезновения, но их существование важно для нашего мира.

Иллюстрация выполнена Сюзи Вивиотт на основе черно-белой гравюры из книги *British Birds* («Птицы Британии»).

Дэвид Калавера, Лоренцо Фонтана

BPF для мониторинга Linux

Перевел с английского *С. Черников*

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*С. Давид*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Н. Рощина*
Художественный редактор	*В. Мостипан*
Корректоры	*Е. Павлович, Е. Рафалюк-Бузовская*
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: 07.2020. 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 08.07.20. Формат 70x100/16. Бумага офсетная. Усл. п. л. 16,770. Тираж 1000. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87