Часть 8 : eBPF для сети

Как вы уже видели в главе 1, динамическая природа eBPF позволяет нам настраивать поведение ядра. В мире сетей существует огромный диапазон желаемого поведения, зависящего от приложения. Например, телекоммуникационному оператору может потребоваться взаимодействовать со специфическими телекоммуникационными протоколами, такими как SRv6; среду `Kubernetes` может потребоваться интегрировать с устаревшими приложениями; выделенные аппаратные балансировщики нагрузки можно заменить программами XDP, работающими на общедоступном оборудовании. eBPF позволяет программистам создавать сетевые функции для удовлетворения конкретных потребностей, не навязывая их всем вышестоящим пользователям ядра.

Сетевые инструменты, основанные на eBPF, в настоящее время широко используются и доказали свою эффективность в большом масштабе. Например, проект Cilium CNCF (eBPF-based Networking, Observability, Security — https://cilium.io/) использует eBPF в качестве платформы для сетей `Kubernetes`, автономной балансировки нагрузки и многого другого, и он же используется облачными пользователями во всех мыслимых отраслевых вертикалях[1]. Компания Meta широко использует eBPF — каждый пакет в Facebook с 2017 года проходит через программу XDP. Другим общедоступным и масштабируемым примером является использование Cloudflare eBPF для защиты от DDoS (распределенная атака на отказ в обслуживании).

Это сложные, готовые к производственному применению решения, и их детали выходят далеко за рамки этой книги, но, прочитав примеры в этой главе, вы сможете понять, как строятся подобные сетевые решения eBPF.

> Примеры кода для этой главы находятся в каталоге chapter8 репозитория по адресу https://github.com/lizrice/learning-ebpf.

Отсечение пакетов

Существует несколько функций сетевой безопасности, которые обеспечивают отбрасывание одних входящих пакетов и разрешение других. Такие функции включают межсетевой экран, защиту от DDoS-атак или устранение уязвимостей типа «пакет смерти»:

- Межсетевой экран включает в себя принятие решения о разрешении пакета для каждого отдельного пакета на основе его исходящего и входящего IP-адресов и/или номеров портов.
- Защита от DDoS-атак добавляет сложности, возможно, отслеживая скорость, с которой пакеты поступают из определенного источника, и/или обнаруживая определенные характеристики содержимого пакетов, чтобы определить, что злоумышленник или группа злоумышленников пытается залить интерфейс трафиком.
- Уязвимость «пакет смерти» — это тип уязвимости ядра, при котором ядро не может безопасно обработать пакет, созданный определенным образом. Злоумышленник, отправляющий пакеты именно в этом формате, может воспользоваться уязвимостью, что потенциально может привести к сбою ядра. Традиционно, при обнаружении подобной уязвимости ядра, требуется установка нового ядра с исправлением, что, в свою очередь, требует простоя компьютера. Но программа eBPF, которая обнаруживает и отбрасывает эти вредоносные пакеты, может быть установлена динамически, мгновенно защищая этот хост, не затрагивая какие-либо приложения работающие на машине.

Алгоритмы принятия решений для подобных функций выходят за рамки этой книги, но давайте рассмотрим, как программы eBPF, подключенные к ловушке XDP на сетевом интерфейсе, отбрасывают определенные пакеты, что является основой для реализации этих вариантов использования.

1 На момент написания этой статьи, около 100 организаций были публично объявлены как использующие Cilium, в их файле USERS.md (Who is using Cilium? - https://github.com/cilium/cilium/blob/main/USERS.md), и это число быстро растет. Cilium также был признан AWS, Google и Майкрософт.

Коды возврата XDP программы

Программа XDP запускается при поступлении очередного сетевого пакета. Программа проверяет пакет, и когда это сделано, код возврата выдает вердикт, указывающий, что делать дальше с этим пакетом:

- XDP_PASS указывает на то, что пакет должен быть отправлен в сетевой стек обычным способом (как это было бы сделано, если бы не было программы XDP).
- XDP_DROP вызывает немедленное отбрасывание пакета.
- XDP_TX отправляет пакет обратно из того же интерфейса, на который он прибыл.
- XDP_REDIRECT используется для отправки на другой сетевой интерфейс.
- XDP_ABORTED приводит к отбрасыванию пакета, но его использование подразумевает случай ошибки или что-то неожиданное, а не «нормальное» решение об отбрасывании пакета.

В некоторых случаях использования (например, брандмауэр) программа XDP просто должна решить, передать пакет или отбросить его. Схема программы XDP, которая решает следует ли отбрасывать пакеты, выглядит примерно так:

```
SEC("xdp")
int hello(struct xdp_md *ctx) {
    bool drop;

    drop = <examine packet and decide whether to drop it>;

    if (drop)
        return XDP_DROP;
    else
        return XDP_PASS;
}
```

Программа XDP даже может манипулировать содержимым пакета, но об этом я расскажу чуть позже в этой главе.

Программы XDP запускаются всякий раз, когда входящий сетевой пакет поступает на интерфейс, к которому она подключена. Параметр `ctx` является указателем на структуру `xdp_md`, которая содержит метаданные о входящем пакете. Давайте посмотрим, как вы можете использовать эту структуру для проверки содержимого пакета, чтобы вынести вердикт.

Анализ пакетов XDP

Вот определение `xdp_md` структуры:

```
struct xdp_md {
    __u32 data;
    __u32 data_end;
    __u32 data_meta;
    /* Below access go through struct xdp_rxq_info */
    __u32 ingress_ifindex; /* rxq->dev->ifindex */
    __u32 rx_queue_index;  /* rxq->queue_index */

    __u32 egress_ifindex;  /* txq->dev->ifindex */
};
```

Не обманывайтесь типом `__u32` для первых трех полей, так как на самом деле они являются указателями. Поле `data` указывает место в памяти, где начинается пакет, а поле `data_end`

показывает, где они заканчивается. Как вы видели в главе 6, для прохождения верификатора eBPF вам придется явно проверить, что любые операции чтения или записи содержимого пакета находятся в диапазоне от `data` до `data_end`.

В памяти перед самим пакетом также есть область между `data_meta` и `data` для хранения метаданных об этом пакете. Это можно использовать для координации между несколькими программами eBPF, которые могут обрабатывать один и тот же пакет в разных местах на его пути прохождения через стек.

Чтобы проиллюстрировать основы синтаксического анализа сетевого пакета, в примере кода есть программа XDP, называемая `ping()`, которая просто генерирует строку трассировки всякий раз, когда обнаруживает пакет `ping` (протокола `ICMP`). Вот код этой программы:

```
SEC("xdp")
int ping(struct xdp_md *ctx) {
   long protocol = lookup_protocol(ctx);
   if (protocol == 1) // ICMP
   {
       bpf_printk("Hello ping");
   }
   return XDP_PASS;
}
```

Вы можете увидеть эту программу в действии, выполнив следующие операции:

1. Запустите `make` в каталоге `chapter8`. Это не просто сборка кода; она также подключает программу XDP к петлевому интерфейсу (называемому `lo`).
2. Выполните команду `ping localhost` в одном из окон терминала.
3. В другом окне терминала просмотрите выходные данные, созданные в канале трассировки, запустив команду `cat /sys/kernel/tracing/trace_pipe`.

Вы должны увидеть две строки трассировки, которые генерируются примерно каждую секунду, и они должны выглядеть так:

```
ping-26622   [000] d.s11 276880.862408: bpf_trace_printk: Hello ping
ping-26622   [000] d.s11 276880.862459: bpf_trace_printk: Hello ping
ping-26622   [000] d.s11 276881.889575: bpf_trace_printk: Hello ping
ping-26622   [000] d.s11 276881.889676: bpf_trace_printk: Hello ping
ping-26622   [000] d.s11 276882.910777: bpf_trace_printk: Hello ping
ping-26622   [000] d.s11 276882.910930: bpf_trace_printk: Hello ping
```

Имеют место две строки трассировки в каждую секунду, поскольку петлевой интерфейс получает как запросы как для проверки связи, так и ответы этой проверки связи. Вы можете легко изменить этот код, чтобы отбрасывать пакеты `ping`, добавив строку кода для возврата XDP_DROP при совпадении протоколов, например:

```
if (protocol == 1) // ICMP
{
   bpf_printk("Hello ping");
   return XDP_DROP;
}
return XDP_PASS;
```

Если вы попытаетесь сделать это, то вы увидите, что выходные данные, подобные приведенным ниже, генерируются в выводе трассировки теперь только единожды за секунду:

```
ping-26639    [002] d.s11 277050.589356: bpf_trace_printk: Hello ping
```

```
ping-26639    [002] d.s11 277051.615329: bpf_trace_printk: Hello ping
ping-26639    [002] d.s11 277052.637708: bpf_trace_printk: Hello ping
```

Петлевой интерфейс `lo` получает запрос `ping`, а программа XDP отбрасывает его, поэтому запрос никогда не проходит через сетевой стек достаточно глубоко, чтобы получить ответ. Большая часть работы в этой программе XDP выполняется функцией `lookup_protocol()`, которая определяет тип протокола уровня 4. Это просто пример, а не качественная реализация парсинга сетевого пакета! Но этого достаточно, чтобы дать вам представление о том, как работает синтаксический анализ в eBPF. Полученный сетевой пакет состоит из строки байтов, расположенных, как это показано на рис. 8-1.

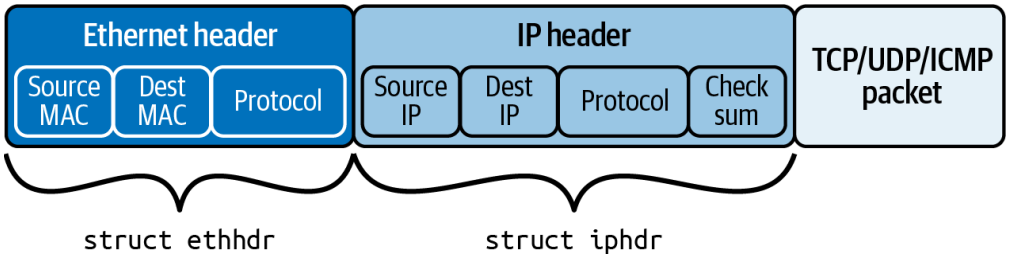

Рисунок 8-1. Схема сетевого пакета IP, начиная с заголовка Ethernet, за которым следует заголовок IP, а затем данные уровня 4 (L4).

Функция `lookup_protocol()` принимает структуру `ctx`, содержащую информацию о том, где в памяти находится этот сетевой пакет, и возвращает тип протокола, найденный в заголовке IP. Код выглядит следующим образом:

```
unsigned char lookup_protocol(struct xdp_md *ctx)
{
   unsigned char protocol = 0;

   void *data = (void *)(long)ctx->data;
   void *data_end = (void *)(long)ctx->data_end;
   struct ethhdr *eth = data;
   if (data + sizeof(struct ethhdr) > data_end)
      return 0;

   // Check that it's an IP packet
   if (bpf_ntohs(eth->h_proto) == ETH_P_IP)
   {
      // Return the protocol of this packet
      // 1 = ICMP
      // 6 = TCP
      // 17 = UDP
      struct iphdr *iph = data + sizeof(struct ethhdr);
      if (data + sizeof(struct ethhdr) + sizeof(struct iphdr) <= data_end)
         protocol = iph→protocol;
   }
   return protocol;
}
```

1. Локальные переменные `data` и `data_end` указывают на начало и конец сетевого пакета.
2. Сетевой пакет должен начинаться с заголовка протокола Ethernet[2].

2 В общем случае — с протокола канального уровня L2, который может быть отличным от Ethernet (TokenRing, Arcnet и др.) (Прим. пер.)

3. Но вы не можете просто неявно предположить, что сам этот сетевой пакет достаточно велик, чтобы вместить этот заголовок Ethernet! Верификатор требует, чтобы вы проверили это явно.
4. Заголовок Ethernet содержит 2-байтовое поле, которое сообщает нам протокол уровня 3 (L3).
5. Если тип протокола указывает, что это пакет IP, заголовок IP следует сразу же за заголовком Ethernet.
6. Вы не можете просто так предположить, что в сетевом пакете достаточно места для этого IP-заголовка. Опять же, верификатор требует, чтобы вы проверили длину явно.
7. Заголовок IP содержит байт протокола, который функция и возвращает вызывающей стороне.

Функция `bpf_ntohs()`, используемая этой программой, гарантирует, что два байта находятся в порядке, ожидаемом на этом хосте. Сетевые протоколы имеют обратный порядок байтов (big-endian), но большинство процессоров имеют обратный порядок байтов (little-endian), то есть они хранят многобайтовые значения в другом порядке. Эта функция преобразует (при необходимости) сетевой порядок байт в хост-упорядочивание. Вы должны использовать эту функцию всякий раз, когда вы извлекаете значение из поля в сетевом пакете, длина которого превышает один байт.

Простой пример показывает, как всего несколько строк кода eBPF могут существенно повлиять на функциональность сети. Нетрудно представить, как более сложные правила о том, какие пакеты следует пропускать, а какие отбрасывать, могут привести к функциям, которые я описала в начале этого раздела: межсетевой экран, защита от DDoS-атак и устранение уязвимостей типа «пакет смерти». Теперь давайте рассмотрим, как можно обеспечить еще большую функциональность, учитывая возможность изменять сетевые пакеты в программах eBPF.

Балансировка нагрузки и форвардинг

Программы XDP не ограничиваются только проверкой содержимого пакета. Они также могут изменять содержимое пакета. Давайте рассмотрим, что требуется, если вы хотите создать простой балансировщик нагрузки, который принимает пакеты, отправленные на заданный IP-адрес, и распределяет эти запросы по нескольким серверным частям, которые могут выполнить запрос.

Пример этого есть в репозитории GitHub[3]. Здесь оборудование представляет собой набор контейнеров, которые все работают на одном хосте. Здесь есть клиент, балансировщик нагрузки и два бэкэнда, каждый из которых работает в своем собственном контейнере. Как показано на рис. 8-2, балансировщик нагрузки получает трафик от клиента и перенаправляет его в один из двух серверных контейнеров.

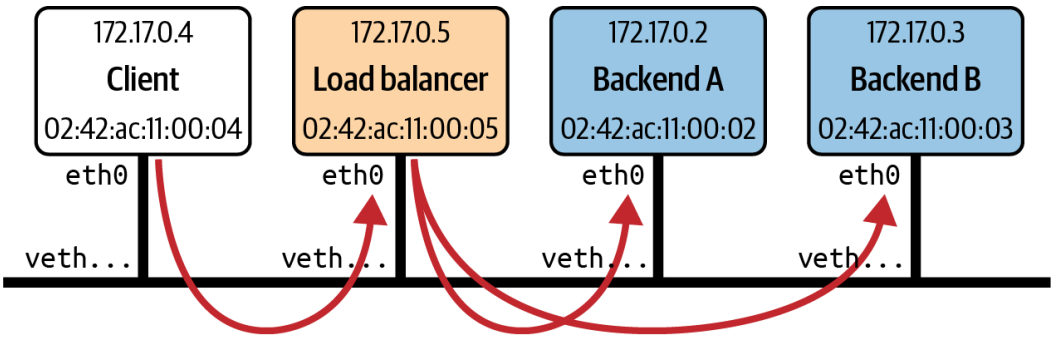

Рисунок 8-2. Пример настройки балансировщика нагрузки

Функция балансировки нагрузки реализована в виде программы XDP, подключенной к сетевому интерфейсу балансировщика нагрузки `eth0`. Код возврата этой программы — `XDP_TX`, указывающий, что пакет должен быть отправлен обратно в тот же интерфейс, на

3 Этот пример основан на моем выступлении на eBPF Summit 2021 под названием «Балансировщик нагрузки с нуля». Создайте балансировщик нагрузки eBPF всего за 15 минут!

который он пришел. Но прежде чем это произойдет, программа должна обновить адресную информацию в заголовках пакетов.

Хотя я думаю, что это полезно в качестве учебного упражнения, этот пример кода очень и очень далек от того, чтобы быть готовым к работе; например, он использует жестко закодированные адреса, которые предполагают точную настройку IP-адресов, показанную на рис. 8-2. Он предполагает, что единственный TCP-трафик, который он когда-либо будет получать, — это запросы от клиента или ответы клиенту. Он также создаёт иллюзию, используя способ, которым `Docker` устанавливает виртуальные MAC-адреса, используя IP-адрес каждого контейнера в качестве последних четырех байтов MAC-адреса для виртуального интерфейса Ethernet для каждого из контейнеров. Этот виртуальный интерфейс Ethernet называется `eth0` с точки зрения самого контейнера.

Вот программа XDP из примера кода балансировщика нагрузки:

```
SEC("xdp_lb")
int xdp_load_balancer(struct xdp_md *ctx)
{
    void *data = (void *)(long)ctx->data;
    void *data_end = (void *)(long)ctx->data_end;

    struct ethhdr *eth = data;
    if (data + sizeof(struct ethhdr) > data_end)
        return XDP_ABORTED;
    if (bpf_ntohs(eth->h_proto) != ETH_P_IP)
        return XDP_PASS;

    struct iphdr *iph = data + sizeof(struct ethhdr);
    if (data + sizeof(struct ethhdr) + sizeof(struct iphdr) > data_end)
        return XDP_ABORTED;

    if (iph->protocol != IPPROTO_TCP)
        return XDP_PASS;

    if (iph->saddr == IP_ADDRESS(CLIENT))
    {
        char be = BACKEND_A;
        if (bpf_get_prandom_u32() % 2)
            be = BACKEND_B;

        iph->daddr = IP_ADDRESS(be);
        eth->h_dest[5] = be;
    }
    else
    {
        iph->daddr = IP_ADDRESS(CLIENT);
        eth->h_dest[5] = CLIENT;
    }
    iph->saddr = IP_ADDRESS(LB);
    eth->h_source[5] = LB;

    iph->check = iph_csum(iph);

    return XDP_TX;
}
```

1. Первая часть этой функции практически такая же, как и в предыдущем примере: она находит заголовок Ethernet, а затем заголовок IP в пакете.
2. Но на этот раз он будет обрабатывать исключительно TCP-пакеты, проталкивая всё остальное, что он получает, вверх по стеку, как будто ничего не произошло.
3. Далее проверяется исходящий IP-адрес. Если этот пакет пришел не от клиента, я буду считать, что это ответ, отправленный клиенту.
4. Далее этот код генерирует псевдослучайный выбор между бэкендами A и B.
5. IP и MAC адреса получателя обновляются в соответствии с выбранным выше бэкэндом...
6. ... или если это ответ от бэкэндом (что является здесь предположением, если он пришел не от клиента), IP и MAC адреса получателя обновляются так, чтобы соответствовать клиенту.
7. Куда бы ни направлялся этот пакет, необходимо обновить адреса источника, чтобы это выглядело так, как будто пакет исходит от балансировщика нагрузки.
8. Заголовок IP включает контрольную сумму, рассчитанную по содержимому пакета, и, поскольку IP-адреса источника и получателя были обновлены, контрольную сумму также необходимо пересчитать и заменить в этом пакете.

    > Поскольку это книга про eBPF, а не о работе с сетью, я не стала вдаваться в подробности, например, того почему необходимо обновлять IP и MAC адреса, или что случится, если этого не сделать. Если вам интересно, я расскажу об этом подробнее в своем видео на YouTube с докладом на саммите eBPF, где я изначально написала этот пример кода (A Load Balancer from scratch – Liz Rice, Isovalent – https://www.youtube.com/watch?v=L3_AOFSNKK8).

Как и в предыдущем примере, `Makefile` содержит инструкции не только для сборки кода, но и для использования `bpftool` для загрузки и подключения программы XDP к интерфейсу, например:

```
xdp: $(BPF_OBJ)
   bpftool net detach xdpgeneric dev eth0
   rm -f /sys/fs/bpf/$(TARGET)
   bpftool prog load $(BPF_OBJ) /sys/fs/bpf/$(TARGET)
   bpftool net attach xdpgeneric pinned /sys/fs/bpf/$(TARGET) dev eth0
```

Здесь команду `make` необходимо выполнять внутри контейнера балансировщика нагрузки, чтобы `eth0` соответствовал его виртуальному интерфейсу Ethernet. Это приводит к интересному моменту: в ядро загружается программа eBPF, которая всего одна; тем не менее, точка подключения может находиться в определенном сетевом пространстве имен и быть видимой только в этом сетевом пространстве имен[4].

Разгрузка XDP

Идея XDP возникла из разговоров о том, насколько полезно было бы, если бы вы могли запускать программы eBPF на сетевой карте для принятия решений об отдельных пакетах еще до того, как они попадут в сетевой стек ядра[5]. Существуют некоторые адаптеры сетевых интерфейсов, которые поддерживают полную *XDP offload* совместимость, когда они могди бы запускать программы eBPF для входящих пакетов на своем собственном процессоре. Это показано на рис. 8-3.

4 Если вы хотите изучить это подробнее, обратитесь к материаламе BPF Summit 2022 (Capture The Flag Challenges for eBPF Summit 2022 — https://github.com/isovalent/eBPF-Summit-2022-CTF). Я не хочу создавать спойлеры здесь, в книге, но вы можете увидеть решение в пошаговом руководстве, данном Даффи Кули и мной (eCHO Episode 64: eBPF Summit CTF #3 — https://www.youtube.com/watch?v=CBUIy0FzxFY).

5 См. презентацию Даниэля Боркманна (https://www.youtube.com/watch?v=99jUcLt3rSk — Daniel Borkmann, Cilium) «Little Helper Minions for Scaling Microservices», которая включает историю eBPF, где он рассказывает этот анекдот.

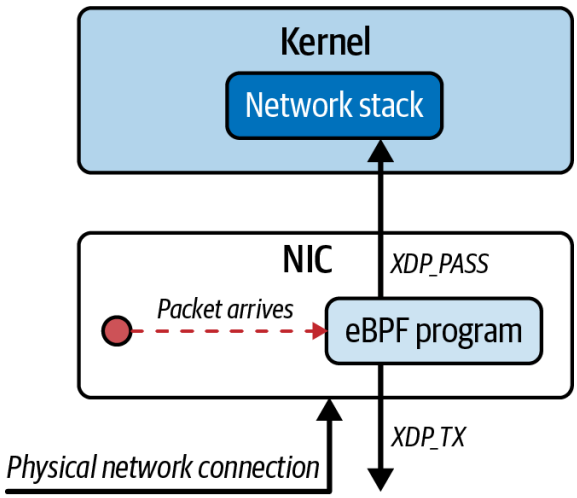

Рисунок 8-3. Карты сетевого интерфейса, которые поддерживают разгрузку XDP, могут обрабатывать, отбрасывать и повторно передавать пакеты без каких-либо действий, требуемых от центрального процессора.

Это означает, что пакет, который отбрасывается или перенаправляется обратно через тот же физический интерфейс (например, в примерах отбрасывания пакетов и балансировки нагрузки ранее в этой главе), никогда не виден ядру хоста, и никакие циклы ЦП на хост-машине никогда не тратятся на обработку. их, так как вся работа выполняется на сетевой карте.

Даже если физическая карта сетевого интерфейса и не поддерживает полную разгрузку XDP, многие драйверы сетевых карт поддерживают перехватчики XDP, что сводит к минимуму копирование памяти, необходимое программе eBPF для обработки пакета[6].

Это может привести к значительному повышению производительности и позволяет очень эффективно выполнять такие функции, как балансировка нагрузки, на обычном оборудовании[7].

Вы видели, как можно использовать XDP для обработки входящих сетевых пакетов, получая к ним доступ как можно быстрее после их поступления на машину. eBPF также можно использовать для обработки трафика и в других точках сетевого стека, в каком бы направлении он ни шел. Давайте продолжим размышления о программах eBPF, подключенных к подсистеме TC.

Контроль трафика (TC)

Я упомянула управление трафиком (TC) в предыдущей главе. К тому времени, когда сетевой пакет достигает этой точки, он уже находится в памяти ядра в виде структуры `sk_buff`[8] (`sk_buff` — https://wiki.linuxfoundation.org/networking/sk_buff). Это структура данных, которая используется во всем сетевом стеке ядра. Программы eBPF, подключенные к подсистеме TC, получают указатель на структуру `sk_buff` в качестве параметра контекста.

> Вам может стать интересно, почему программы XDP не используют эту же самую структуру для своего контекста. Ответ заключается в том, что перехват XDP происходит до того, как сетевые данные достигают сетевого стека и до того ещё, как будет настроена структура `sk_buff`.

Подсистема TC предназначена для управления планированием сетевого трафика. Например, вы можете захотеть ограничить полосу пропускания, доступную для каждого приложения, чтобы все они получили равные шансы. Но когда вы смотрите на планирование отдельных пакетов, *пропускная способность* не является очень значимым термином, поскольку он используется для обозначения среднего объема данных, отправляемых или получаемых.

6 Cilium поддерживает список драйверов (https://docs.cilium.io/en/latest/bpf/progtypes/#xdp), поддерживающих XDP, в своём справочном руководстве по BPF и XDP (https://docs.cilium.io/en/v1.12/bpf/ — BPF and XDP Reference Guide).

7 В этом сообщении Ceznam поделилась данными о повышении производительности, которое ее команда увидела во время экспериментов с балансировщиком нагрузки на основе eBPF (Jul 13, 2022, Cilium Standalone Layer 4 Load Balancer XDP — https://cilium.io/blog/2022/04/12/cilium-standalone-L4LB-XDP/).

8 Сокетный буфер (Прим. пер.)

Конкретное приложение может быть очень нестабильным в интенсивности, а другое приложение может быть критически чувствительным к задержке в сети, поэтому TC дает гораздо более точный контроль над тем, как обрабатываются пакеты и расставляются приоритеты[9].

Программы eBPF, Здесь были представлены чтобы дать пользовательский контроль над алгоритмами, используемыми в TC. Но благодаря возможности манипулировать, отбрасывать или перенаправлять пакеты, программы eBPF, подключенные к TC, также могут использоваться в качестве строительных блоков для сложного сетевого поведения.

Определенная часть сетевых данных в стеке передается в одном из двух направлений: вход (входящий от сетевого интерфейса) или выход (исходящий к сетевому интерфейсу). Программы eBPF могут быть подключены в любом направлении и будут влиять на трафик только в этом направлении. В отличие от XDP, можно подключить несколько программ eBPF, которые будут обрабатываются последовательно.

Традиционное управление трафиком разделено на *классификаторы*, которые классифицируют пакеты на основе некоторого правила, и *отдельные действия*, которые выполняются на основе выходных данных классификатора и определяют, что делать с пакетом. Может быть определён целый ряд классификаторов, каждый из которых определен как часть *qdisc* или очереди *дисциплин*.

Программы eBPF присоединяются в качестве классификатора, но они также могут определять какие действия следует выполнять в рамках этой программы. Действие обозначается кодом возврата программы (значения которого определены в `linux/pkt_cls.h`):

- TC_ACT_SHOT указывает ядру отбросить пакет.
- TC_ACT_UNSPEC ведет себя так, как если бы программа eBPF не выполнялась для этого пакета (поэтому он будет передан следующему классификатору в последовательности, если они такие есть).
- TC_ACT_OK указывает ядру передать пакет на следующий уровень в стеке.
- TC_ACT_REDIRECT отправляет пакет на входной или выходной путь другого сетевого устройства.

Давайте рассмотрим несколько простых примеров программ, которые можно прикрепить к TC. Первый просто генерирует строку трассировки, а затем приказывает ядру отбросить пакет:

```
int tc_drop(struct __sk_buff *skb) {
   bpf_trace_printk("[tc] dropping packet\n");
   return TC_ACT_SHOT;
}
```

Теперь давайте рассмотрим, как отбрасывать только некоторое подмножество пакетов. Этот пример отбрасывает пакеты запросов ICMP (`ping`) и очень похож на пример XDP, который вы видели ранее в этой главе:

```
int tc(struct __sk_buff *skb) {
   void *data = (void *)(long)skb->data;
   void *data_end = (void *)(long)skb->data_end;

   if (is_icmp_ping_request(data, data_end)) {
      struct iphdr *iph = data + sizeof(struct ethhdr);
      struct icmphdr *icmp = data +
                             sizeof(struct ethhdr) +
                             sizeof(struct iphdr);
      bpf_trace_printk("[tc] ICMP request for %x type %x\n", iph->daddr,
```

9 Для более полного обзора TC и его концепций я рекомендую пост Квентина Монне «Понимание режима tc «прямого действия» для BPF» (Understanding tc “direct action” mode for BPF, Apr 11, 2020, Quentin Monnet — https://qmonnet.github.io/whirl-offload/2020/04/11/tc-bpf-direct-action/).

```
                        icmp->type);
      return TC_ACT_SHOT;
   }
   return TC_ACT_OK;
}
```

Структура `sk_buff` имеет в своём составе указатели на начало и конец данных пакета, очень похоже на структуру `xdp_md`, и разбор пакета происходит почти таким же образом. Опять же, чтобы пройти проверку, вы должны явно проверить, что любой доступ к данным находится в диапазоне между `data` и `data_end`.

Вам может быть интересно, зачем реализовывать что-то подобное на уровне TC, если вы уже видели подобную функциональность, реализованную с помощью XDP. Одна веская причина заключается в том, что вы можете использовать программы TC для исходящего трафика, тогда как XDP может обрабатывать только входящий трафик. Во-вторых, поскольку XDP запускается сразу же после прибытия пакета, в этот момент ещё нет структуры данных ядра `sk_buff`, связанной с пакетом. Если программа eBPF заинтересована в этом, или хочет манипулировать с `sk_buff`, который ядро создает для этого пакета, то точка присоединения TC подходит.

> Чтобы лучше понять различия между программами XDP и TC eBPF, перечитайте раздел «Типы программ» в справочном руководстве по BPF и XDP из проекта Cilium (BPF and XDP Reference Guide — https://docs.cilium.io/en/latest/bpf/#program-types).

А теперь давайте рассмотрим пример, который не только просто отбрасывает определенные пакеты. Этот пример идентифицирует полученный запрос `ping` и отвечает ответом `ping`:

```
int tc_pingpong(struct __sk_buff *skb) {
   void *data = (void *)(long)skb->data;
   void *data_end = (void *)(long)skb->data_end;

   if (!is_icmp_ping_request(data, data_end)) {
      return TC_ACT_OK;
   }

   struct iphdr *iph = data + sizeof(struct ethhdr);
   struct icmphdr *icmp = data + sizeof(struct ethhdr) + sizeof(struct iphdr);

   swap_mac_addresses(skb);
   swap_ip_addresses(skb);

   // Change the type of the ICMP packet to 0 (ICMP Echo Reply)
   // (was 8 for ICMP Echo request)
   update_icmp_type(skb, 8, 0);

   // Redirecting a clone of the modified skb back to the interface
   // it arrived on
   bpf_clone_redirect(skb, skb->ifindex, 0);

   return TC_ACT_SHOT;
}
```

1. Функция `is_icmp_ping_request()` анализирует пакет и проверяет не только то, что это ICMP-сообщение, но и то, что это именно эхо-запрос (`ping`).
2. Поскольку эта функция будет отправлять сама ответ отправителю, адреса источника и получателя необходимо поменять местами. (Вы можете прочитать пример кода, если

хотите видеть мельчайшие подробности управления трафиком (TC), что включает и обновление контрольной суммы IP-заголовка.)

3. Преобразуется эхо-ответ, путем изменения типов полей в заголовке ICMP.
4. Эта функция-помощник (`bpf_clone_redirect()`) отправляет клон пакета обратно через интерфейс (`skb->ifindex`), на котором он и был получен.
5. Поскольку функция-помощник уже клонировала пакет перед отправкой ответа, то исходный пакет следует отбросить.

В обычных обстоятельствах запрос `ping` будет обрабатываться сетевым стеком ядра позже, но этот небольшой пример демонстрирует, как в более общем плане сетевые функции могут быть заменены реализацией eBPF.

Многие сетевые возможности на сегодня обрабатываются службами пользовательского адресного пространства, но там, где их можно заменить программами eBPF, это, вероятно, будет отлично с точки зрения производительности. Пакет, который обрабатывается в ядре, не должен завершать свое путешествие через всю остальную часть стека; ему не нужно переходить в пространство пользователя для обработки, а ответ не требует перехода обратно в ядро. Более того, они могут работать параллельно — программа eBPF может возвращать `TC_ACT_OK` для любого пакета, требующего сложной обработки, которую она не может сама обработать, чтобы он как обычно передавался в службу пользовательского пространства.

Для меня это важный аспект реализации сетевого функционала в eBPF. По мере развития платформы eBPF (например, более поздние ядра, позволяющие программы из миллиона инструкций) становится возможной реализация в ядре все более сложных аспектов работы в сети. Части, которые еще не реализованы в eBPF, по-прежнему могут обрабатываться либо традиционным стеком в ядре, либо в пользовательском пространстве. Со временем все больше и больше функций могут быть перемещены из пользовательского пространства в ядро, а гибкость и динамическая природа eBPF означают, что вам не придется ждать, пока они станут частью самого дистрибутива ядра. Вы можете загружать реализации eBPF немедленно, как я уже говорила в главе 1.

Я вернусь к реализации сетевых функций далее, в разделе «eBPF и сеть Kubernetes». Но сначала давайте рассмотрим еще один вариант использования, который позволяет реализовать eBPF: проверка расшифрованного содержимого зашифрованного трафика.

Кодирование и декодирование пакета

Если приложение использует шифрование для защиты данных, которые оно отправляет или получает, то будет момент до того, как оно будет зашифровано, или после того, как оно будет расшифровано, когда данные будут существовать в открытом виде. Вспомните, что eBPF может прикреплять программы практически к любому месту на машине, поэтому, если вы можете подключиться к точке, где передаваемые данные еще не зашифрованы, или сразу после того, как они были расшифрованы, и это позволит вашей программе eBPF наблюдать, эти данные в открытом виде. Нет необходимости предоставлять какие-либо сертификаты для расшифровки трафика, как в традиционном инструменте проверки SSL.

Во многих случаях приложение будет шифровать данные с помощью библиотеки, такой как OpenSSL или BoringSSL, которая находится в пользовательском пространстве. В этом случае трафик уже будет зашифрован к тому времени, когда он достигнет сокета, который является границей пользовательского пространства/ядра для сетевого трафика. Если вы хотите отследить эти данные в незашифрованном виде, то вы должны использовать программу eBPF, прикрепленную к нужному месту кода пользовательского пространства.

SSL библиотеки пространства пользователя

Один из распространенных способов отследить расшифрованное содержимое зашифрованных пакетов — подключиться к вызовам пользовательских библиотек, таких как OpenSSL или BoringSSL. Приложение, использующее OpenSSL, отправляет данные для шифрования, вызывая функцию `SSL_write()`, и извлекает данные открытого текста, которые были получены по сети в зашифрованном виде с помощью `SSL_read()`.

Подключение программ eBPF к этим функциям с помощью `upprobes` позволяет приложению просматривать данные *из любого приложения, использующего эту общую библиотеку*, в открытом виде, до их шифрования или после их расшифровки. И нет необходимости в каких-либо ключах, потому что они уже предоставлены приложением.

Есть довольно простой пример под названием openssl-tracer in the Pixie project[10] (https://github.com/pixie-io/pixie-demos/tree/410447afd6e566050e8bbf4060c66d76660cb30b/openssl-tracer — OpenSSL Trasing Demo), в котором программы eBPF находятся в файле с именем `openssl_tracer_bpf_funcs.c`. Вот часть этого кода, которая отправляет данные в пространство пользователя, используя буфер Perf (аналогично примерам, которые вы видели ранее в этой книге):

```
static int process_SSL_data(struct pt_regs* ctx, uint64_t id,
           enum ssl_data_event_type type, const char* buf) {
   ...
   bpf_probe_read(event->data, event->data_len, buf);
   tls_events.perf_submit(ctx, event, sizeof(struct ssl_data_event_t));

   return 0;
}
```

Вы можете видеть, что данные из `buf` считываются в структуру `event` с помощью функции-помощника `bpf_probe_read()`, а затем эта структура событий отправляется в буфер `Perf`.

Если эти данные отправляются в пространство пользователя, разумно предположить, что это должны быть данные в незашифрованном формате. Так где же берется этот буфер данных? Вы можете понять это, увидев, где вызывается функция `process_SSL_data()`. Она вызывается в двух местах: один для чтения данных и один для записи данных. На рис. 8-4 показано, что происходит в случае чтения данных, поступающих на эту машину в зашифрованном виде.

Когда вы читаете данные, вы передаете указатель на буфер для `SSL_read()`, и когда функция возвращается, этот буфер будет содержать незашифрованные данные. Как и с `kprobes`, входные параметры функции, включая и этот указатель буфера, доступны только для `uprobe`, подключенного к точке входа, поскольку регистры, в которых они хранятся, вполне могут быть перезаписаны во время выполнения функции. Но данные не будут доступны в буфере до выхода из функции, когда вы сможете прочитать их с помощью `uretprobe`.

10 Этот пример также комментируется обсуждениями в блоге https://blog.px.dev/ebpf-openssl-tracing.

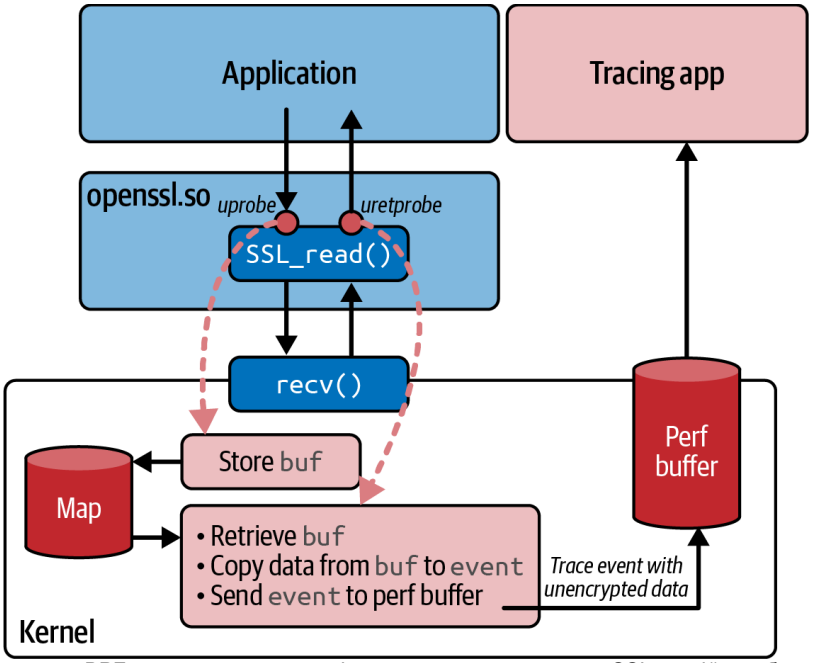

Рисунок 8-4. Программы eBPF подключаются к uprobes при входе и выходе из SSL_read(), чтобы незашифрованные данные можно было прочитать по указателю буфера.

Таким образом, этот пример следует общему шаблону для `kprobes` и `uprobes`, показанному на рис. 8-4, где входной `probe` временно сохраняет входные параметры с помощью карты, из которой выходной `probe` может их извлечь. Давайте посмотрим на код, который это делает, начиная с программы eBPF, прикрепленной к началу `SSL_read()`:

```
// Function signature being probed:
// int SSL_read(SSL *s, void *buf, int num)
int probe_entry_SSL_read(struct pt_regs* ctx) {
   uint64_t current_pid_tgid = bpf_get_current_pid_tgid();
   ...
   const char* buf = (const char*)PT_REGS_PARM2(ctx);

   active_ssl_read_args_map.update(&current_pid_tgid, &buf);
   return 0;
}
```

1. Как описано в комментарии к этой функции, указатель буфера является вторым параметром, передаваемым в функцию `SSL_read()`, к которой будет присоединен этот `probe`. Макрос `PT_REGS_PARM2` получает этот параметр из контекста.
2. Указатель буфера хранится в хеш-таблице, для которой ключом является текущий процесс и идентификатор потока, полученный при старте функции с помощью вызова `bpf_get_current_pid_tgif()`.

А вот это соответствующая программа для выходного `probe`:

```
int probe_ret_SSL_read(struct pt_regs* ctx) {
   uint64_t current_pid_tgid = bpf_get_current_pid_tgid();

   ...
   const char** buf = active_ssl_read_args_map.lookup(&current_pid_tgid);
   if (buf != NULL) {
      process_SSL_data(ctx, current_pid_tgid, kSSLRead, *buf);
   }

   active_ssl_read_args_map.delete(&current_pid_tgid);
```

```
    return 0;
}
```

1. Найдя идентификатор текущего процесса и потока, используйте их в качестве ключа для извлечения указателя буфера из хэш-таблицы.
2. Если это не нулевой указатель, вызовите `process_SSL_data()`, функцию, которую вы видели ранее, которая отправляет данные из этого буфера в пространство пользователя, используя буфер `Perf`.
3. Очистите запись в хэш-таблице, так как каждый входной вызов должен быть соотнесён с выходным.

В этом примере показано, как отследить открытую версию текста из зашифрованных данных, которые отправляются и принимаются приложением пользовательского пространства. Сама трассировка привязана к библиотеке пользовательского пространства, и нет никакой гарантии, что каждое приложение будет использовать именно данную библиотеку SSL. Проект BCC включает утилиту `sslsniff`, которая также поддерживает GnuTLS и NSS. Но если чье-то приложение использует какую-то другую библиотеку шифрования (или даже, не дай Бог, они решили «развернуть свою собственную криптографию»), у `uprobes` просто не будет нужных мест для подключения, и эти инструменты трассировки не будут работать.

Есть еще более общие причины, по которым этот подход, основанный на `probe`, может оказаться неудачным. В отличие от ядра (которое существует только по одному экземпляру на [виртуальную] машину), может быть несколько копий кода библиотеки пользовательского пространства. Если вы используете контейнеры, каждый из них, скорее всего, будет иметь собственный набор всех зависимых библиотек. Вы можете подключиться к `uprobes` в этих библиотеках, но вам нужно будет определить правильную копию именно для специфического контейнера, который вы хотите отслеживать. Другая особенность заключается в том, что вместо использования разделяемой, динамически связываемой, библиотеки приложение может быть статически собрано, так что оно представляет собой один автономный исполняемый файл.

eBPF и сети Kubernetes

Хотя эта книга не о Kubernetes[11], eBPF настолько широко используется для сетей Kubernetes, что является отличной иллюстрацией использования этой платформы для настройки сетевого стека.

В средах Kubernetes приложения развертываются в *модулях*[12]. Каждый модуль — это группа из одного или нескольких контейнеров, которые совместно используют пространства имен и контрольные группы ядра `cgroups`, изолируя модули друг от друга и от общего хост-компьютера, на котором они работают.

В частности (для целей этой главы) модуль обычно имеет свое собственное сетевое пространство имен и собственный IP-адрес[13]. Это означает, что ядро имеет набор структур сетевого стека для этого пространства имен, отделенных от хоста и от других модулей. Как показано на рис. 8-5, модуль подключается к хосту с помощью виртуального соединения Ethernet, и ему назначается собственный IP-адрес.

11 Kubernetes (*K8s*) — открытое программное обеспечение для оркестровки контейнеризированных приложений — автоматизации их развёртывания, масштабирования и координации в условиях кластера (https://kubernetes.io/ru/). (Прим. пер.)

12 Pod (англ. «стручок, кокон», также *модуль, под*) — базовая единица для запуска и управления приложениями: один или несколько контейнеров, которым гарантирован запуск на одном узле, обеспечивается разделение ресурсов и межпроцессорное взаимодействие, и предоставляется уникальный в пределах кластера IP-адрес (Прим. пер.).

13 Модули могут запускаться и в сетевом пространстве имен хоста, чтобы они совместно использовали единый IP-адрес хоста, но обычно так не делается, если только приложение, работающее в модуле, не требует этого.

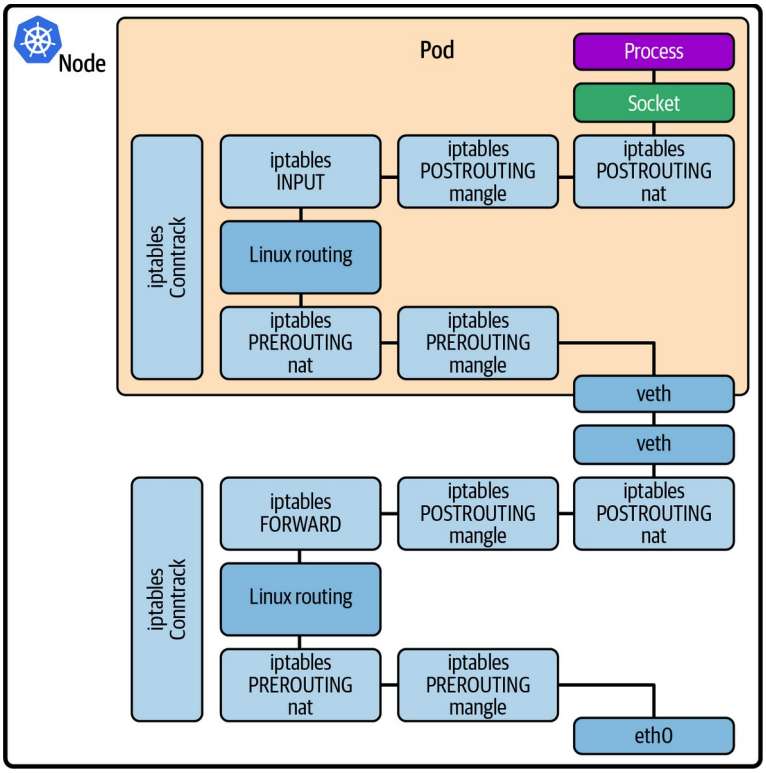

Рисунок 8-5. Сетевой путь в Kubernetes

На рис. 8.5 видно, что пакет, поступающий из-за пределов машины и предназначенный для модуля приложения, должен пройти через сетевой стек на хосте, через виртуальное соединение Ethernet и попасть в сетевое пространство имен модуля, а затем снова пройти его сетевой стек, чтобы добраться до приложения.

Эти два сетевых стека работают в одном и том же ядре, поэтому пакет фактически проходит одну и ту же обработку дважды. Чем больше кода должен пройти сетевой пакет, тем выше задержка, поэтому, если возможно сократить сетевой путь, то это, вероятно, приведет к повышению производительности. Сетевое решение на основе eBPF, такое как у Cilium, может подключиться к сетевому стеку, чтобы переопределить собственное сетевое поведение ядра, как показано на рис. 8-6.

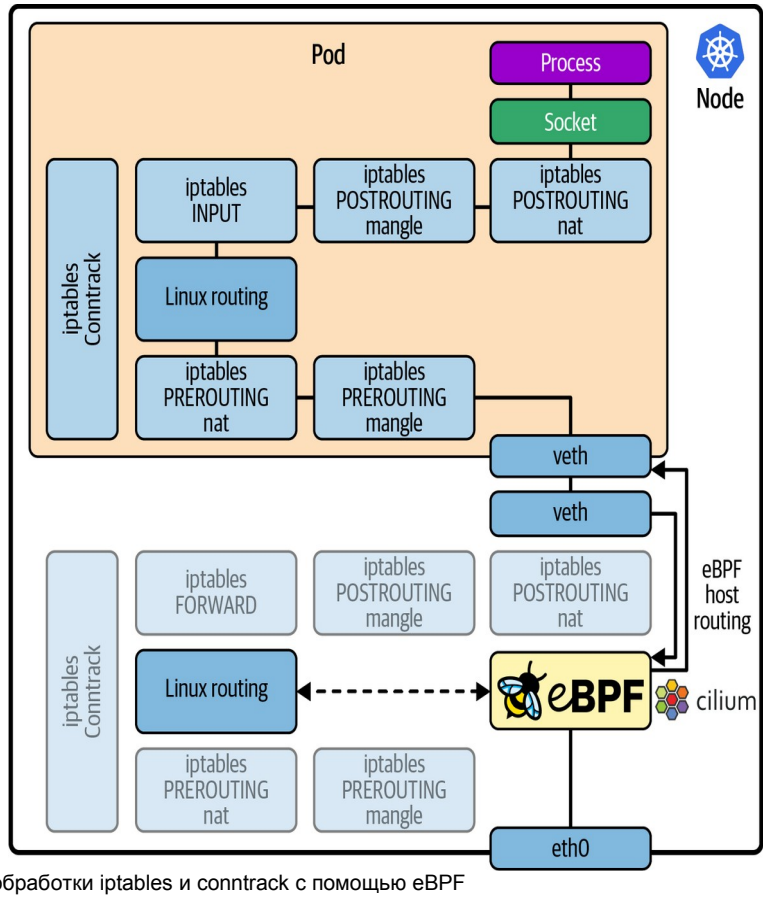

Рисунок 8-6. Обход обработки iptables и conntrack с помощью eBPF

В частности, eBPF позволяет заменить `iptables` и `conntrack` более эффективным решением для управления сетевыми правилами и отслеживания соединений. Давайте обсудим, почему это приводит к значительному повышению производительности в Kubernetes.

Избежание iptables

В Kubernetes есть компонент под названием `kube-proxy`, который реализует балансировку нагрузки, позволяя нескольким модулям выполнять запросы к службе. Это реализовано с помощью правил `iptables`.

Kubernetes предлагает пользователям выбрать, какое сетевое решение использовать, с помощью Container Network Interface (CNI). Некоторые подключаемые модули CNI используют правила `iptables` для реализации сетевой политики L3/L4 в Kubernetes; то есть правила `iptables` указывают, следует ли отбрасывать пакет, потому что он не соответствует сетевой политике.

Хотя `iptables` был эффективен для традиционной (предконтейнерной) сети, у него есть некоторые недостатки при использовании в Kubernetes. В этой среде модули и их IP-адреса возникают и удаляются динамически, и каждый раз, когда модуль добавляется или удаляется, правила `iptables` должны быть полностью переписаны, и это влияет на в масштабы производительности. (В докладе Хайбина Се и Куинтона Хула на KubeCon в 2017 году описывалось, как обновление одного правила для правил `iptables` для 20 000 сервисов может занять пять часов: Scale Kubernetes to Support 50,000 Services [I] - Haibin Xie & Quinton Hoole — https://www.youtube.com/watch?v=4-pawkiazEg&t=863s).

Обновления `iptables` — не единственные проблемы с производительностью: поиск правила требует линейного поиска по таблице ядра, что является операцией O(n), линейно растущей с количеством самих правил.

Cilium использует карты хеш-таблиц eBPF для хранения своих правил сетевой политики, отслеживания соединений и поисковых таблиц балансировщика нагрузки, которые могут заменить `iptables` для `kube-proxy`. Как поиск записи в хеш-таблице, так и вставка новой

записи — это примерно O(1) операций, что означает, что они масштабируются намного лучше.

Вы можете прочитать об улучшении производительности, достигнутом в тестах, в блоге Cilium (May 11, 2021, CNI Benchmark: Understanding Cilium Network Performance — https://cilium.io/blog/2021/05/11/cni-benchmark/). В том же посте вы увидите, что Calico, еще один CNI с опцией eBPF, также достигает лучшей производительности, когда вы выбираете его реализацию eBPF вместо `iptables`. Решение eBPF предлагает наиболее эффективные механизмы для масштабируемых динамических развертываний Kubernetes.

Координированные сетевые программы

Сложная сетевая реализация, такая как Cilium, не может быть написана как одна единая программа eBPF. Как показано на рис. 8-7, она предоставляет несколько разных программ eBPF, которые подключаются к разным частям ядра и его сетевого стека.

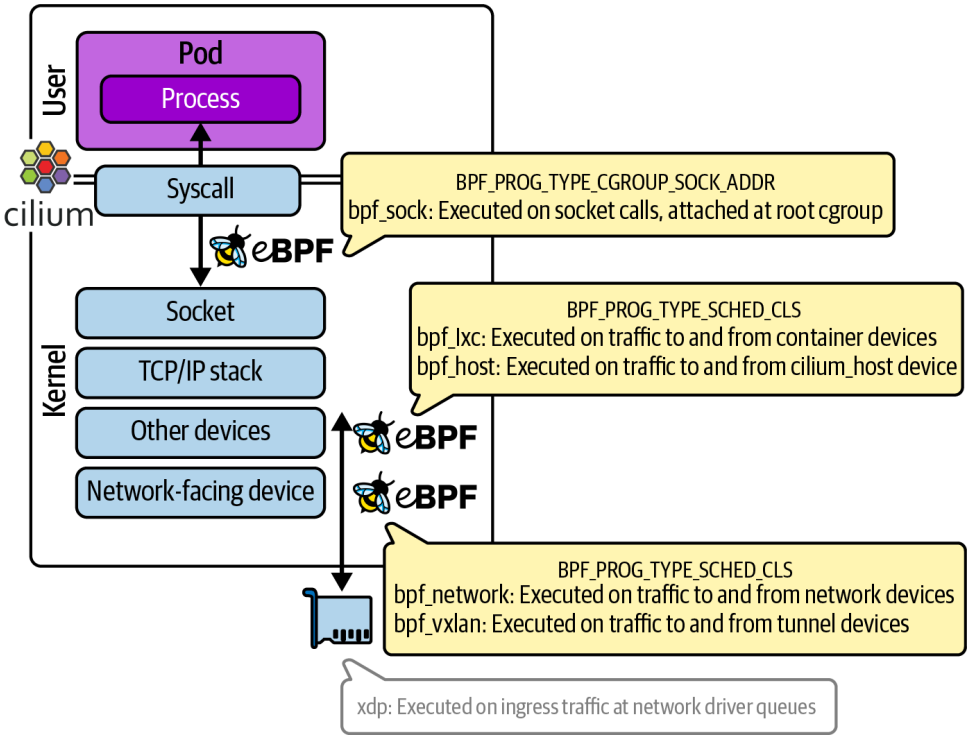

Рисунок 8-7. Cilium состоит из нескольких скоординированных программ eBPF, которые подключаются к разным точкам ядра.

Как правило, Cilium перехватывает трафик, как только может сократить путь обработки для каждого пакета. Сообщения, исходящие из модуля приложения, перехватываются на уровне сокета, как можно ближе к приложению. Входящие пакеты из внешней сети перехватываются с помощью XDP. А как же дополнительные точки прикрепления?

Cilium поддерживает различные сетевые модели, которые подходят для разных сред. Полное описание этого выходит за рамки этой книги (дополнительную информацию вы можете найти на Cilium.io: eBPF-based Networking, Observability, Security — https://cilium.io/), но я дам здесь краткий обзор, чтобы вы поняли, почему существует так много разных программ eBPF!

Существует простой, плоский режим сети, при котором Cilium выделяет IP-адреса для всех модулей в кластере из одного и того же CIDR и напрямую маршрутизируя трафик между ними. Существует также несколько различных режимов туннелирования, в которых трафик, предназначенный для модуля на другом узле, инкапсулируется в сообщение, адресованное IP-адресу этого узла назначения, и декапсулируется на этом узле назначения для окончательного перехода в модуль. Различные программы eBPF вызываются для обработки трафика в зависимости от того, предназначен ли пакет для локального контейнера, локального хоста, другого хоста в этой сети или туннеля.

На рис. 8-7 вы можете увидеть несколько программ TC, которые обрабатывают входящий и исходящий трафик различных устройств. Эти устройства представляют возможные различные реальные и виртуальные сетевые интерфейсы, по которым может проходить пакет:

- Интерфейс к сетевой подсистеме модуля (один конец виртуального соединения Ethernet между модулем и его хостом).
- Интерфейс к сетевому туннелю
- Интерфейс к физическому сетевому устройству на хосте
- Собственный сетевой интерфейс самого хоста

> Если вам интересно узнать больше о том, как пакеты проходят через Cilium, Артур Чиао написал этот подробный и интересный пост в блоге: «Жизнь пакета в Cilium: обнаружение пути трафика модуль-к-сервису и логики обработки BPF» (Life of a Packet in Cilium: Discovering the Pod-to-Service Traffic Path and BPF Processing Logics, Published at 2020-09-12 — https://arthurchiao.art/blog/cilium-life-of-a-packet-pod-to-service/).

Различные программы eBPF, подключенные к этим различным точкам ядра, обмениваются данными, используя карты eBFP и используя метаданные, которые могут быть присоединены к сетевым пакетам по мере их прохождения через стек (о чем я упоминала, когда обсуждала доступ к сетевым пакетам в примере с XDP). Эти программы не просто направляют пакеты к месту назначения; они также используются и для отбрасывания пакетов — как вы видели в предыдущих примерах — на основе сетевых политик.

Применение сетевой политики

В начале этой главы вы видели, как программы eBPF могут отбрасывать пакеты, а это значит, что они просто не дойдут до адресата. Это основа применения сетевой политики, и концептуально это практически одно и то же, независимо от того, думаем ли мы о «традиционном» или «облачном» брандмауэре. Политика определяет, следует ли отбрасывать пакет, на основе информации об его источнике и/или пункте назначения.

В традиционных средах IP-адреса назначаются конкретному серверу на длительный период времени, но в Kubernetes IP-адреса приходят и уходят динамически, и адрес, назначенный сегодня для определенного модуля приложения, вполне может быть повторно использован для совершенно другого приложения завтра. Вот почему традиционные брандмауэры не очень эффективны в облачных средах. Было бы непрактично переопределять правила брандмауэра вручную каждый раз, когда меняются IP-адреса.

Вместо этого Kubernetes поддерживает концепцию сетевой политики ресурса, которая определяет правила брандмауэра на основе меток, применяемых к конкретным модулям, а не на основе их IP-адресов. Хотя этот тип ресурса является базовым для Kubernetes, он не реализован самим Kubernetes. Вместо этого эта функциональность передается любому подключаемому модулю CNI, который вы сами используете. Если вы выбираете CNI, который не поддерживает ресурсы сетевой политики, любые правила, которые вы можете настроить, просто игнорируются. С другой стороны, CNI могут свободно настраивать пользовательские ресурсы, которые позволяют использовать более сложные конфигурации сетевых политик, чем позволяет собственное определение Kubernetes. Например, Cilium поддерживает такие функции, как правила сетевой политики на основе DNS, поэтому вы можете определить, разрешен ли трафик или нет, не на основе IP-адреса, а на основе имени DNS (такие как «example.com»). Вы также можете определить политики для различных протоколов уровня 7, например, разрешая или запрещая трафик для вызовов HTTP GET, но не для вызовов POST для определенного URL-адреса.

> Бесплатное практическое занятие Isovalent «Начало работы с Cilium» (Getting Started with Cilium — https://isovalent.com/labs/getting-started-with-cilium/) поможет вам определить сетевые политики на уровнях 3/4 и 7. Ещё одним очень полезным ресурсом является редактор сетевой политики (Deep Dive into Network Policy — https://networkpolicy.io/), который наглядно представляет эффекты сетевой политики.

Как я говорила ранее в этой главе, можно использовать и правила `iptables` для отбрасывания трафика, и это подход, который некоторые CNI использовали для реализации правил сетевой политики Kubernetes. Cilium использует программы eBPF для отбрасывания трафика, который не соответствует набору действующих правил. Увидев ранее в этой главе примеры отбрасывания пакетов, я надеюсь, что у вас есть модель грубого понимания того, как это должно работать.

Cilium использует идентификаторы Kubernetes, чтобы определить, применяется ли данное правило сетевой политики. Точно так же, как метки определяют, какие модули являются частью службы в Kubernetes, метки также определяют идентификатор безопасности Cilium для модуля. Хэш-таблицы eBPF, проиндексированные этими идентификаторами служб, обеспечивают очень эффективный поиск правил.

Шифрованные соединения

Многие организации предъявляют требования к защите своих инфраструктур и данных пользователей путем шифрования трафика между приложениями. Этого можно достичь путем написания кода непосредственно в каждом приложении, чтобы убедиться, что оно устанавливает безопасные соединения, как правило, с использованием mutual Traffic Layer Security (mTLS), лежащей в основе соединения HTTP или gRPC соединения. Для настройки этих соединений необходимо сначала установить идентификаторы приложений на обоих концах соединения (что обычно достигается путем обмена сертификатами), а только затем зашифровать данные, которые передаются между ними.

В Kubernetes можно разгрузить такое требование к приложениям, либо на уровень сети сервисов, либо на саму базовую сеть. Полное обсуждение сети сервисов выходит за рамки этой книги, но вам может быть интересна статья, которую я написал о новом стеке: «Как eBPF оптимизирует сервис сети» (How eBPF Streamlines the Service Mesh, Oct 25th, 2021, Liz Rice — https://thenewstack.io/how-ebpf-streamlines-the-service-mesh/). А здесь давайте сконцентрируемся на сетевом уровне и на том, как eBPF позволяет внедрить требование шифрования в ядро.

Самый простой способ обеспечить шифрование трафика внутри кластера Kubernetes — использовать прозрачное шифрование. Он называется «прозрачным», потому что полностью выполняется на сетевом уровне и чрезвычайно прост с точки зрения эксплуатации. Сами приложения вообще не должны ничего знать о шифровании, и им не нужно настраивать соединения HTTPS; также этот подход не требует каких-либо дополнительных компонентов инфраструктуры, работающих в Kubernetes.

Существует два широко используемых протокола шифрования в ядре, IPsec и WireGuard®, и оба они поддерживаются в сети Kubernetes CNI и Cilium и Calico. Обсуждение различий между этими двумя протоколами выходит за рамки этой книги, но ключевой момент заключается в том, что они устанавливают безопасный туннель между двумя машинами. CNI может подключить конечную точку eBPF для модуля через этот безопасный туннель.

> В блоге Cilium есть хорошая статья (Transparent Encryption with WireGuard — https://cilium.io/blog/2021/05/20/cilium-110/#wireguard) о том, как Cilium использует WireGuard®, а также и IPsec, для обеспечения зашифрованного трафика между узлами. В посте также дается краткий обзор характеристик производительности обоих.

Безопасный туннель настраивается с использованием идентификаторов узлов на обоих концах. Эти идентификаторы управляются Kubernetes в любом случае, поэтому административная нагрузка на оператора минимальна. Для многих целей этого достаточно, поскольку гарантирует, что весь сетевой трафик в кластере будет зашифрован. Прозрачное шифрование также можно использовать без модификаций с помощью такой сетевой политики, которая использует идентификаторы Kubernetes для управления тем, может ли трафик проходить между различными конечными точками в кластере.

В некоторых организациях используется мульти арендная среда, где необходимы строгие мульти арендные границы, и важно использовать сертификаты для идентификации каждой конечной точки приложения. Обработка этого в каждом приложении является значительным бременем, поэтому в последнее время это было перенесено на уровень сервисов сети, но для

этого требуется развертывание целого дополнительного набора компонентов, что приводит к дополнительному потреблению ресурсов, задержке и операционной сложности.

На сейчас eBPF обеспечивает новый подход (mTLS for Any Network Protocol — https://isovalent.com/blog/post/cilium-service-mesh/#h-mtls-for-any-network-protocol), который основан на прозрачном шифровании, но использующий TLS для первоначального обмена сертификатами и аутентификации конечной точки, чтобы идентификацию могли представлять отдельные приложения, а не узлы, на которых они работают, как это показано на рис. 8-8.

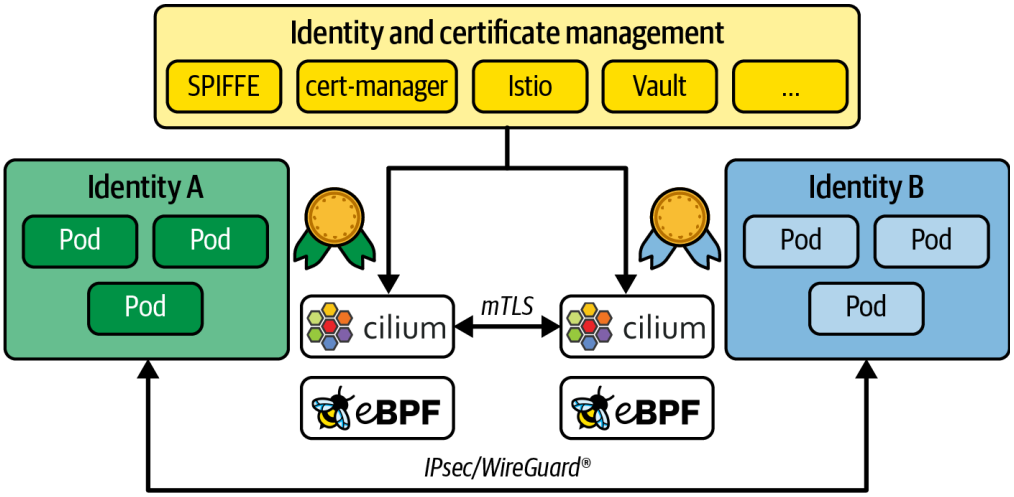

Рисунок 8-8. Прозрачное шифрование между аутентифицированными идентификаторами приложений.

Как только выполнен этап аутентификации, IPsec или WireGuard® в ядре используются для шифрования трафика, проходящего между этими приложениями. Это имеет ряд преимуществ. Это позволяет сторонним инструментам управления сертификатами и идентификацией, таким как cert-manager или SPIFFE/SPIRE, обрабатывать идентификационную часть, а сеть сама заботится о шифровании, поэтому все это полностью прозрачно для приложения. Cilium поддерживает определения сетевой политики, которые указывают конечные точки по их SPIFFE ID, а не только по их меткам Kubernetes. И, возможно, самое главное, этот подход можно использовать с любым протоколом, который перемещается в IP-пакетах. Это большой шаг вперед по сравнению с mTLS, который работает только для соединений на основе TCP. В этой книге недостаточно места, чтобы погрузиться во все внутренности Cilium, но я надеюсь, что этот раздел помог вам понять, насколько eBPF является мощной платформой для создания сложных сетевых функций, таких как полнофункциональный Kubernetes CNI.

Итоги

В этой главе вы видели программы eBPF, подключенные к различным точкам сетевого стека. Я показала примеры базовой обработки пакетов и надеюсь, что они дали вам представление о том, как eBPF может создавать мощные сетевые функции. Вы также видели несколько реальных примеров этих сетевых функций, включая балансировку нагрузки, межсетевой экран, снижение уровня безопасности и работу в сети Kubernetes.

Упражнения и дополнительная литература

Вот несколько способов узнать больше о различных вариантах использования eBPF в сети:

1. Измените пример программы XDP `ping()` так, чтобы она генерировала разные сообщения трассировки для ответов `ping` и запросов `ping`. Заголовок ICMP следует непосредственно за заголовком IP в сетевом пакете (точно так же, как заголовок IP следует за заголовком Ethernet). Скорее всего, вы захотите использовать `struct icmphdr` из файла `linux/icmp.h` и посмотреть, отображается ли в поле типа `ICMP_ECHO` или `ICMP_ECHOREPLY`.

2. Если вы хотите глубже погрузиться в программирование XDP, я рекомендую xdp-руководство (xdp-project/xdp-tutorial — https://github.com/xdp-project/xdp-tutorial).
3. Используйте `sslsniff` из проекта BCC (iovisor/bcc — https://github.com/iovisor/bcc/blob/master/tools/sslsniff.py) для просмотра содержимого зашифрованного трафика.
4. Изучите Cilium, используя учебные пособия и лабораторные работы, ссылки на которые есть на веб-сайте Cilium (https://cilium.io/get-started).
5. Используйте редактор (Policy Editor — https://networkpolicy.io/) чтобы визуализировать влияние сетевых политик в инфраструктуре Kubernetes.