

Часть 7 : Типы программы eBPF и присоединения

В предыдущих главах вы видели множество примеров программ eBPF и, вероятно, заметили тот факт, что они привязаны к разным типам событий. Некоторые из примеров, которые я показала, подключаются к `kprobes`, но в других примерах я продемонстрировала программы XDP, которые обрабатывают вновь прибывшие сетевые пакеты. Это только две из многих точек присоединения в ядре. В этой главе мы более подробно рассмотрим различные типы программ и способы их привязки к различным событиям.

Вы можете собрать и запустить примеры из этой главы, используя код и инструкции на <https://github.com/lizrice/learning-ebpf>. Код для этой главы находится в каталоге `chapter7`.

На момент написания этой статьи некоторые из примеров не поддерживаются процессорами ARM. Обратитесь к файлу `README` в каталоге `chapter7` для получения более подробной информации и советов.

В настоящее время в `uapi/linux/bpf.h` перечислено около 30 типов программ и более 40 типов присоединения (<https://elixir.bootlin.com/linux/v5.19.14/source/include/uapi/linux/bpf.h>). Тип присоединения более конкретно определяет, куда прикрепляется программа; для многих типов программ тип присоединения может быть выведен из типа программы, но некоторые типы программ могут быть присоединены к нескольким различным точкам в ядре, поэтому также необходимо указать явно тип присоединения. Как вы знаете, эта книга не предназначена для использования в качестве справочного руководства, поэтому я не буду рассматривать каждый отдельный тип программы eBPF. Скорее всего, к тому времени, когда вы будете читать эту книгу, будут добавлены новые типы!

Аргументы контекста программы

Все программы eBPF принимают аргумент контекста, который является указателем, но структура, на которую указатель указывает, зависит от конкретного типа события, вызвавшего программы. Программистам eBPF нужно писать программы, которые принимают соответствующий тип контекста; нет смысла изображать, что аргумент контекста указывает на сетевой пакет, если событие является, скажем, точкой трассировки. Определение различных по типу программ позволяет верификатору гарантировать, что контекстная информация обрабатывается надлежащим образом, и применять правила относительно того, какие функции-помощники здесь допустимы.

Чтобы углубиться в детали контекстных данных, передаваемых в различные типы программ BPF, ознакомьтесь с этой публикацией Алана Магуайра (Alan Maguire) в блоге Oracle (*здесь в оригинале ошибочная ссылка*).

Функции-помощники и коды возврата

Как вы видели в предыдущей главе, верификатор проверяет чтобы все функции-помощники, используемые программой, были совместимы с её типом программы. В примере из предыдущей главы показано, что функция-помощник `bpf_get_current_pid_tgid()` не разрешена в программе XDP. В точке, где получен пакет и срабатывает ловушка XDP, нет никакого процесса или потока пользовательского пространства, поэтому вызов для обнаружения текущего процесса и идентификатора потока не имеет смысла в этом контексте.

Тип программы также определяет значение кода возврата из программы. Опять же, используя XDP в качестве примера, значение кода возврата сообщает ядру, что делать с пакетом после того, как программа eBPF завершит его обработку, что может включать передачу его в сетевой стек, удаление или перенаправление на другой интерфейс. Эти коды возврата не имеют никакого смысла, когда программа eBPF запускается, скажем, при попадании в определенную точку трассировки, в которой нет сетевого пакета.

Существует man-страница¹ для функций-помощников (bpf-helpers(7) — Linux manual page : <https://man7.org/linux/man-pages/man7/bpf-helpers.7.html>) (с вполне разумными оговорками о том, что она может быть неполной из-за продолжающейся разработки подсистемы BPF).

Вы можете получить список функций-помощников, доступных для каждого типа программы в вашей версии ядра, с помощью команды `bpftool feature`. Это показывает конфигурацию системы, и список всех доступных типов программ, и типов карт, и даже список всех функций-помощников, которые поддерживаются для каждого типа программы².

Функции-помощники считаются частью UAPI внешнего стабильного интерфейса ядра Linux. Таким образом, как только некоторая функция-помощник будет определена в ядре, она не должна изменяться в будущем, даже если внутренние функции ядра и структуры данных могут измениться.

Несмотря на риск изменений между версиями ядра, от программистов eBPF исходило требование иметь доступ к некоторым внутренним функциям ядра из программ eBPF. Этого можно добиться с помощью механизма, называемого *ядерными функциями BPF* или `kfuncs` (BPF Kernel Functions (kfuncs) : <https://docs.kernel.org/bpf/kfuncs.html#bpf-kernel-functions-kfuncs>).

Kfuncs

`Kfuncs` позволяет регистрировать внутренние функции ядра в подсистеме BPF, чтобы верификатор позволял вызывать их из программ eBPF. Существует регистрация для каждого типа программы eBPF, которому разрешено вызывать данный `kfunc`.

В отличие от функций-помощников, `kfuncs` не дает гарантий совместимости, поэтому программист eBPF должен учитывать возможность изменений между версиями ядра.

Существует набор (Core kfuncs : <https://docs.kernel.org/bpf/kfuncs.html#core-kfuncs>) «основных» `kfuncs` BPF, который на момент написания этой статьи состоит из функций, позволяющих программам eBPF получать и освобождать ссылки ядра на задачи и `cgroups`. Напомним, что тип программы eBPF определяет к каким событиям она может быть присоединена, что, в свою очередь, и определяет тип получаемой контекстной информации. Тип программы также определяет набор функций-помощников и `kfuncs`, которые она может вызывать. Обычно считается, что типы программ делятся на две категории: типы

1 С тем же успехом (или даже с большим успехом — за счёт большей новизны), все справочные страницы `man`, упоминаемые в книге, могут просматриваться не по ссылкам в Интернет, а командой `man` на выполняющейся Linux системе. В данном конкретном случае команда будет выглядеть так: `man 7 bpf-helpers` (Прим. пер.)

2 Нужно быть готовым к тому, что вывод команды огромный и лучше обеспечить перенаправление потока вывода в файл для изучения. Вот, к примеру, в одном из современных дистрибутивов Linux — 1716 строк:

```
$ sudo bpftool feature | wc -l
```

`1716`
(Прим. пер.)

программ трассировки (или Perf) и типы программ, связанных с сетью. Давайте посмотрим на некоторые примеры.

Трассировка

Программы, которые подключаются к `kprobes`, `tracepoints`, `raw tracepoints`, `fentry/fexit probes` и событиям Perf, были все разработаны, чтобы обеспечить эффективный способ для программ eBPF в ядре сообщать информацию об отслеживании событий в пространство пользователя. Не ожидалось, что эти, связанные с трассировкой, типы повлияют на поведение ядра в ответ на события, к которым они привязаны (хотя, как вы увидите в главе 9, в этой области были некоторые новшества!).

На них иногда ссылаются как на «Perf-программы». Например, подкоманда `bpftool perf` позволяет вам просматривать программы, присоединённые к событиям Perf, следующим образом:

```
$ sudo bpftool perf show
pid 232272 fd 16: prog_id 392
pid 232272 fd 17: prog_id 394
pid 232272 fd 19: prog_id 396
pid 232272 fd 20: prog_id 397
pid 232272 fd 21: prog_id 398
kprobe func __x64_sys_execve offset 0
kprobe func do_execve offset 0
tracepoint sys_enter_execve
raw_tracepoint sched_process_exec
raw_tracepoint sched_process_exec
```

Предыдущий вывод — это то, что я вижу при запуске примера кода из файла `hello.bpf.c` в каталоге `chapter7`, который связывает различные программы с различными событиями, связанными с `execve()`. Я рассмотрю все эти типы в этом разделе, но в качестве обзора вот эти программы:

- `kprobe`, прикрепленный к точке входа в системный вызов `execve()`.
- `kprobe`, присоединенный к функции ядра `do_execve()`.
- `tracepoint`, размещается на входе в системный вызов `execve()`.
- Две версии `raw tracepoint`, вызываемые во время обработки `execve()`. Одна из них, как вы увидите в этом разделе, — это версия с поддержкой BTF.

Вам потребуются привилегии `CAP_PERFMON` и `CAP_BPF`, или `CAP_SYS_ADMIN`, чтобы использовать любой из типов программ eBPF, связанных с трассировкой.

kprobes и kretprobes

Я обсуждала концепцию kprobes в главе 1. Вы можете прикреплять программы kprobe практически к любому месту в ядре³. Обычно они прикрепляются с помощью kprobes ко входу в функцию, а kretprobes — к выходу из функции, но вы можете использовать kprobes для присоединения к инструкции, которая находится на некотором заданном смещении после точки входа в функцию. Если вы решите сделать это⁴, вы должны быть точно уверены, что версия ядра, на которой вы работаете, имеет такую инструкцию, на которую вы хотите прикрепиться именно туда, куда вы планируете! Присоединение к точкам входа и выхода функций ядра может быть относительно безопасным, но произвольные строки кода могут легко изменяться от одного релиза к другому.

В примере вывода результата команды `bpftool perf list`, вы можете видеть, что имеет место смещение 0 для обоих kprobes.

Когда ядро компилируется, также существует вероятность того, что компилятор решит «встроить» любую заданную функцию ядра как `inline`; то есть вместо того, чтобы осуществлять переход с того места где вызывается функция, компилятор может встроить машинный код для реализации тех действий, что эта функция делает внутри вызываемой функции. Если функция окажется встроенной, то у вашей программы eBPF не окажется точки входа kprobe, к которой можно будет прикрепиться.

Прикрепление kprobes к точкам входа системных вызовов

Первый пример программы eBPF для этой главы называется `kprobe_sys_execve`, и это kprobe, прикрепленный к системному вызову `execve()`. Функция и её секция определения выглядят следующим образом:

```
SEC("ksyscall/execve")
int BPF_KPROBE_SYSCALL(kprobe_sys_execve, char *pathname)
```

Это есть то же самое, что вы уже видели в главе 5.

Одной из причин привязки именно к системным вызовам является то, что они представляют собой устойчивые интерфейсы, которые не меняются между версиями ядра (то же самое относится и к точкам трассировки, к которым мы вскоре вернемся). Однако не следует полагаться на kprobes для системных вызовов как обеспечение безопасности по причинам, которые я подробно расскажу в главе 9.

Прикрепление kprobes к другим функциям ядра

Вы можете найти множество примеров, когда инструменты на основе eBPF используют kprobes для присоединения именно к системным вызовам, но, как упоминалось ранее, kprobes также можно прикрепить и к любой не встроенной (не `inline`) функции в ядре. Я представляла пример в `hello.bpf.c`, в котором прикрепление kprobe происходит к функции `do_execve()`, и оно определен следующим образом:

```
SEC("kprobe/do_execve")
int BPF_KPROBE(kprobe_do_execve, struct filename *filename)
```

3 За исключением только нескольких мест в ядре, где kprobes запрещены из соображений безопасности. Они перечислены в `/sys/kernel/debug/kprobes/blacklist`.

4 Единственный пример этого, который я видел до сих пор, — это набор тестов `cilium/ebpf` (https://github.com/cilium/ebpf/blob/365d07f530e5641a7fbeb3d487a692320891f776/link/kprobe_test.go#L56).

Поскольку `do_execve()` не является системным вызовом, между этим и предыдущим примером есть несколько отличий:

- Формат имени секции `SEC` идентичен предыдущей версии, прикрепления к точке входа системного вызова, но здесь нет необходимости определять варианты для конкретной платформы, поскольку `do_execve()`, как и большинство функций ядра, является общей для всех платформ.
- Я использовала макрос `BPF_KPROBE`, а не `BPF_KPROBE_SYSCALL`. Цель точно та же, только последний из них обрабатывает параметры для системного вызова.
- Есть еще одно важное отличие: параметр пути в системном вызове представляет собой указатель на строку (`char*`), но теперь для функции параметр называется именем файла и является указателем на структуру имени файла, и которая и является структурой данных, используемой в ядре.

Вам может стать интересно, как я узнала, что именно этот тип должен использоваться для параметра. Я покажу вам. Функция `do_execve()` в ядре имеет следующую сигнатуру:

```
int do_execve(struct filename *filename,
              const char __user *const __argv,
              const char __user *const __envp)
```

Я решила проигнорировать⁵ вызова `do_execve()` параметры `__argv` и `__envp` и объявила только аргумент `filename`, используя тип `struct filename*`, чтобы соответствовать определению функции ядра. Учитывая способ последовательного размещения аргументов вызова в памяти, можно игнорировать последние `N` параметров, но вы не можете игнорировать предшествующий более ранний аргумент в списке, если хотите использовать более поздний. Эта структура имени файла определена внутри ядра, и это иллюстрация того, как программирование eBPF является именно программированием ядра: мне пришлось искать определение `do_execve()`, чтобы найти его аргументы, и определение `struct filename`. На имя исполняемого файла, который должен быть запущен, указывает поле `filename->name`. Я получаю это имя в примере кода в следующих строках:

```
const char *name = BPF_CORE_READ(filename, name);
bpf_probe_read_kernel(&data.command, sizeof(data.command), name);
```

Итак, резюмируем: параметр контекста для системного вызова `kprobe` представляет собой структуру, представляющую значения, переданные пользовательским пространством в системный вызов. Параметр контекста для «обычного» (не системного) `kprobe` представляет собой структуру, представляющую параметры, переданные для вызываемой функции любым кодом ядра, который ее вызывает, поэтому структура зависит от определения этой функции.

Прикрепления `kretprobes` очень похожи на `kprobes`, за исключением того, что они запускаются, когда функция возвращает значение, и могут обращаться к возвращаемому значению вместо аргументов. Оба, `kprobes` и `kretprobes` — это разумный способ подключиться к функциям ядра, но есть более новый вариант, который следует учитывать, если вы работаете на последних версиях ядра.

5 В своём коде примера. (Прим. пер.)

fentry / fexit

Более эффективный механизм отслеживания входа в функции ядра и выхода из них был представлен вместе с идеей *батута BPF* в версии ядра 5.5 (на процессорах x86; поддержка *батута BPF* не появилась для процессоров ARM до Linux 6.0 — KernelNewbies: Linux 6.0 : https://kernelnewbies.org/Linux_6.0#ARM). Если вы используете достаточно свежее ядро, *fentry/fexit* теперь является предпочтительным методом для отслеживания входа в функцию ядра или выхода из нее. Вы можете написать на выбор один и тот же код программы, использующий внутри или тип *kprobe* или *fentry*.

Пример программы *fentry* с именем *fentry_execve()* находится в файле *chapter7/hello.bpf.c*. Я объявила программу eBPF для этого *kprobe* с помощью макроса *BPF_PROG*, который является еще одной удобной оболочкой, предоставляющей доступ к типизированным параметрам, а не к универсальному указателю контекста, но эта версия используется для программ типов *fentry/fexit* и *tracepoint*. Определение выглядит так:

```
SEC("fentry/do_execve")
int BPF_PROG(fentry_execve, struct filename *filename)
```

Имя секции сообщает *libbpf*, что нужно подключиться к ловушке *fentry* в начале функции ядра *do_execve()*. Как и в примере с *kprobe*, параметры контекста отражают параметры, переданные функции ядра, к которой вы хотите подключить эту программу eBPF. Точки присоединения *fentry/fexit* были разработаны, чтобы быть более эффективными, чем *kprobes*, но есть и еще одно преимущество, когда вы хотите сгенерировать событие в конце функции: ловушка *fexit* имеет доступ ещё и к входным параметрам функции, чего нет у *kretprobe*. Вы можете увидеть пример этого в примерах *libbpf-bootstrap* (*libbpf/libbpf-bootstrap* : <https://github.com/libbpf/libbpf-bootstrap>). Оба, и *kprobe.bpf.c* и *fentry.bpf.c* являются эквивалентными примерами, которые прикрепляются к функции ядра *do_unlinkat()*. Программа eBPF, прикреплённая к *kretprobe*, имеет следующую сигнатуру:

```
SEC("kretprobe/do_unlinkat")
int BPF_KRETPROBE(do_unlinkat_exit, long ret)
```

Макрос *BPF_KRETPROBE* расширяется так, чтобы сделать это средствами *kretprobe* при выходе из *do_unlinkat()*. Единственный параметр, который получает здесь программа eBPF, — это *ret*, который содержит значение, возвращаемое функцией *do_unlinkat()*. Сравните это с версией *fexit*:

```
SEC("fexit/do_unlinkat")
int BPF_PROG(do_unlinkat_exit, int dfd, struct filename *name, long ret)
```

В этой версии программа получает доступ не только к возвращаемому значению *ret*, но и к входным параметрам *do_unlinkat()*, а именно *dfd* и *name*.

Точки трассировки

Точки трассировки (Event Tracing — <https://www.kernel.org/doc/html/v6.0/trace/events.html>) — это отмеченные места в коде ядра (позже в этой главе мы вернемся к точкам трассировки пользовательского пространства). Они ни в коем случае не являются исключительными для eBPF и уже давно используются для генерации вывода трассировки ядра и такими инструментами, как SystemTap (Comparing SystemTap and bpftrace — <https://lwn.net/Articles/852112/>). В отличие от присоединения к произвольным инструкциям с

помощью `kprobes`, точки трассировки стабильны между выпусками ядра (хотя старое ядро может и не иметь полного набора точек трассировки, которые были добавлены в более новое).

Вы можете увидеть доступный набор подсистем трассировки в вашем ядре, заглянув в `/sys/kernel/tracing/available_events` следующим образом:

```
# cat /sys/kernel/tracing/available_events
tls:tls_device_offload_set
tls:tls_device_decrypted
...
syscalls:sys_exit_execveat
syscalls:sys_enter_execveat
syscalls:sys_exit_execve
syscalls:sys_enter_execve
...
```

В моей версии ядра 5.15 в этом списке определено более 1400 точек трассировки. Определение раздела для программы eBPF с точкой трассировки должно соответствовать одному из этих элементов, чтобы `libbpf` мог автоматически присоединить его к точке трассировки. Определение имеет вид: `SEC("tp/tracing subsystem/tracepoint name")`.

В файле `chapter7/hello.bpf.cfiles` вы найдете пример, который соответствует точке трассировки `sys call:sys_enter_execve`, которая срабатывает, когда ядро начинает обрабатывать системный вызов `execve()`. Определение раздела сообщает `libbpf`, что это программа с точкой трассировки, и указывает, куда ее следует присоединить, например:

```
SEC("tp/syscalls/sys_enter_execve")
```

А как насчет параметра контекста для точки трассировки? Как я скоро расскажу, здесь нам может помочь BTF, но сначала давайте рассмотрим, что нужно делать, когда BTF недоступен. Каждая точка трассировки имеет формат, описывающий поля, которые из нее трассируются. В качестве примера, вот формат точки трассировки на входе в системный вызов `execve()`:

```
# cat /sys/kernel/tracing/events/syscalls/sys_enter_execve/format
name: sys_enter_execve
ID: 622
format:
  field:unsigned short common_type;  offset:0; size:2; signed:0;
  field:unsigned char  common_flags;  offset:2; size:1; signed:0;
  field:unsigned char  common_preempt_count; offset:3; size:1; signed:0;
  field:int common_pid; offset:4; size:4; signed:1;

  field:int __syscall_nr;            offset:8; size:4; signed:1;
  field:const char * filename;       offset:16; size:8; signed:0;
  field:const char *const * argv;    offset:24; size:8; signed:0;
  field:const char *const * envp;    offset:32; size:8; signed:0;

print fmt: "filename: 0x%08lx, argv: 0x%08lx, envp: 0x%08lx",
((unsigned long)(REC->filename)), ((unsigned long)(REC->argv)),
```

```
((unsigned long)(REC->envp))
```

Я использовала эту информацию для определения соответствующей структуры под названием `my_syscalls_enter_execve` в файле `chapter7/hello.bpf.c`:

```
struct my_syscalls_enter_execve {
    unsigned short common_type;
    unsigned char common_flags;
    unsigned char common_preempt_count;
    int common_pid;
    long syscall_nr;
    long filename_ptr;
    long argv_ptr;
    long envp_ptr;
};
```

Программам eBPF не разрешен доступ к первым четырем из этих полей. Если вы попытаетесь получить к ним доступ, программа не пройдет проверку с ошибкой недопустимого доступа к `bpf_context`.

В моем примере программа eBPF, которая подключается к этой точке трассировки, может использовать указатель на этот тип в качестве параметра контекста, например:

```
int tp_sys_enter_execve(struct my_syscalls_enter_execve *ctx) {
```

Затем вы можете получать доступ к содержимому этой структуры. К примеру, вы можете получить указатель имени файла следующим образом:

```
    bpf_probe_read_user_str(&data.command, sizeof(data.command), ctx->filename_ptr);
```

Когда вы используете тип программы с точкой трассировки, структура, передаваемая программе eBPF, уже сопоставлена с набором необработанных аргументов. Для повышения производительности вы можете получить непосредственный доступ к этим необработанным аргументам с помощью типа программы eBPF: `raw_tracepoint`. Определение раздела должно начинаться с `raw_tp` (или `raw_tracepoint`) вместо `tp`. Вам потребуется преобразовать аргументы из `__u64` в любой тип, который использует структура точки трассировки (когда точка трассировки является входом в системный вызов, эти аргументы зависят от архитектуры чипа процессора).

Точки трассировки разрешаемые BTF

В предыдущем примере я описала структуру с именем `my_syscalls_enter_execve` для определения параметра контекста для моей программы eBPF. Но когда вы определяете структуру в своем коде eBPF или анализируете необработанные аргументы, то существует риск, что ваш код может не соответствовать тому ядру, на котором он работает. Хорошая новость заключается в том, что BTF, с которым вы познакомились в главе 5, также решает эту проблему.

При поддержке BTF, в `vmlinux.h` будет определена структура, которая соответствует структуре контекста, переданной программе eBPF для точки трассировки. Ваша программа eBPF должна использовать определение секции `SEC("tp_btf/tracepoint name")`, где имя точки трассировки является одним из доступных событий, перечисленных в `/sys/kernel/tracing/available_events`. Пример программы в `chapter7/hello.bpf.c` выглядит следующим образом:

```
SEC("tp_btf/sched_process_exec")
int handle_exec(struct trace_event_raw_sched_process_exec *ctx)
```

Как видите, имя структуры совпадает с именем точки трассировки с предшествующим префиксом `trace_event_raw_`.

Прикрепления в пользовательском пространстве

До сих пор я показывала примеры программ eBPF, прикрепляющихся к событиям, определенным в исходном коде ядра. В коде пространства пользователя есть аналогичные точки присоединения: `uprobes` и `uretprobes` для присоединения к входу и выходу функций пространства пользователя, а также статически определенные пользователем точки трассировки (USDT) — для присоединения к указанным точкам трассировки в коде приложения или библиотеках пространства пользователя. Все они используют тип программы `BPF_PROG_TYPE_KPROBE`.

Существует достаточно много общедоступных примеров программ прикрепления в пользовательском адресном пространстве. Вот некоторые из состава проекта BCC:

- <https://github.com/iovisor/bcc/blob/master/libbpf-tools/bashreadline.bpf.c> и <https://github.com/iovisor/bcc/blob/master/libbpf-tools/funclatency.c> прикрепляются к `u(ret)probe`.
- https://github.com/iovisor/bcc/blob/master/examples/usdt_sample/usdt_sample.md — пример USDT в BCC.

Если вы используете `libbpf`, макрос `SEC()` позволяет определить точку автоматического прикрепления для этих `probes` в пользовательском пространстве. Вы сможете найти конкретный формат, необходимый для секции `SEC()`, в документации `libbpf` (Program Types and ELF Sections — https://docs.kernel.org/bpf/libbpf/program_types.html). Например, чтобы прикрепить `uprobe` к началу функции `SSL_write()` в OpenSSL, вы должны определить раздел для программы eBPF следующим образом:

```
SEC("uprobe/usr/lib/aarch64-linux-gnu/libssl.so.3/SSL_write")
```

Есть несколько ошибок, которых следует избегать при инструментировании кода пользовательского пространства:

- Обратите внимание, что путь к этой разделяемой библиотеке в этом примере зависит от архитектуры, и поэтому вам могут понадобиться соответствующие определения для этой своей архитектуры.
- Если вы не контролируете машину, на которой выполняется код, вы не можете узнать, какие библиотеки пользовательского пространства и приложения там будут установлены.
- Приложение может быть создано как статически скомпонованный двоичный файл, поэтому оно не позволит какие-либо `probes`, которые вы могли бы подключить к разделяемым библиотекам.
- Контейнеры обычно работают со своей собственной копией файловой системы, в которой установлен собственный набор связей. Путь к разделяемой библиотеке,

используемой в контейнере, не будет совпадать с путём к разделяемым библиотеке на хост-компьютере.

- Вашей программе eBPF может потребоваться информация о языке программирования, на котором написано приложение. Например, в C аргументы функции обычно передаются с помощью регистров, а в Go они передаются с использованием стека⁶, поэтому структура `pt_args`, содержащая информацию о регистрах, может быть менее полезной.

Тем не менее, существует множество полезных инструментов, которые интегрируют приложения пользовательского пространства с eBPF. Например, вы можете подключиться к библиотеке SSL, чтобы отслеживать расшифрованные версии зашифрованной информации — мы рассмотрим это более подробно в следующей главе. Другой пример — непрерывное профилирование ваших приложений с помощью таких инструментов, как `Parca` (<https://www.parca.dev/>).

LSM

Программы типа `BPF_PROG_TYPE_LSM` присоединяются к API модуля безопасности Linux (LSM), который представляет собой устойчивый интерфейс внутри ядра, изначально предназначенный для использования кодом модулей ядра для обеспечения соблюдения политик безопасности. Как вы увидите в главе 9, где я расскажу об этом более подробно, инструменты безопасности eBPF теперь также могут использовать и этот интерфейс.

Программы типа `BPF_PROG_TYPE_LSM` присоединяются с помощью вызова `bpf(BPF_RAW_TRACEPOINT_OPEN)` и во многом обрабатываются как и программы трассировки. Одна интересная особенность программ `BPF_PROG_TYPE_LSM` заключается в том, что возвращаемое ими значение влияет на поведение ядра. Ненулевой код возврата указывает на то, что проверка безопасности не пройдена, поэтому ядро не будет выполнять какую-либо операцию, которую ему было предложено выполнить. Это существенное отличие от типов программ, связанных с `Perf`, где код возврата игнорируется.

Документация ядра Linux охватывает описание программ LSM BPF (LSM BPF Programs — https://www.kernel.org/doc/html/latest/bpf/prog_lsm.html).

Тип программы LSM не единственный, который играет роль в обеспечении безопасности. Многие типы программ, связанных с сетью, которые вы увидите в следующем разделе, могут использоваться для сетевой безопасности, чтобы разрешать или запрещать сетевой трафик или операции, связанные с сетью. Вы также узнаете больше об использовании eBPF в целях безопасности в главе 9.

До сих пор в этой главе вы видели, как набор типов программ трассировки ядра иди пользовательского пространства обеспечивает наблюдаемость всей системы. Следующий набор типов программ eBPF, которые следует рассмотреть, — это те, которые позволяют нам подключаться к сетевому стеку с возможностью не только наблюдать, но и влиять на то, как стек обрабатывает отправляемые и получаемые данные.

⁶ Это справедливо до версии Go 1.17, когда было введено новое соглашение о вызовах на основе регистров. Тем не менее, я думаю, что исполняемые файлы Go, созданные с более старыми версиями, будут ещё циркулировать в течение некоторого времени.

Сеть

Существует множество различных типов программ eBPF, предназначенных для обработки сетевых сообщений при их прохождении через различные места сетевого стека. На рис. 7-1 показано, где подключаются некоторые из часто используемых типов программ. Все эти типы программ требуют разрешенных привилегий CAP_NET_ADMIN и CAP_BPF или CAP_SYS_ADMIN.

Контекст, передаваемый этим типам программ, представляет собой рассматриваемое сетевое сообщение, хотя конкретный тип структуры зависит от данных, которые ядро имеет в соответствующей точке сетевого стека. В нижней части стека данные хранятся в виде сетевых пакетов уровня 2, которые, по сути, представляют собой последовательность байт, которые или были или готовы к передаче «через среду». В верхней части стека пользовательского приложения используются сокеты, а само ядро создает сокетные буфера для обработки данных, отправляемых и получаемых из этих сокетов.

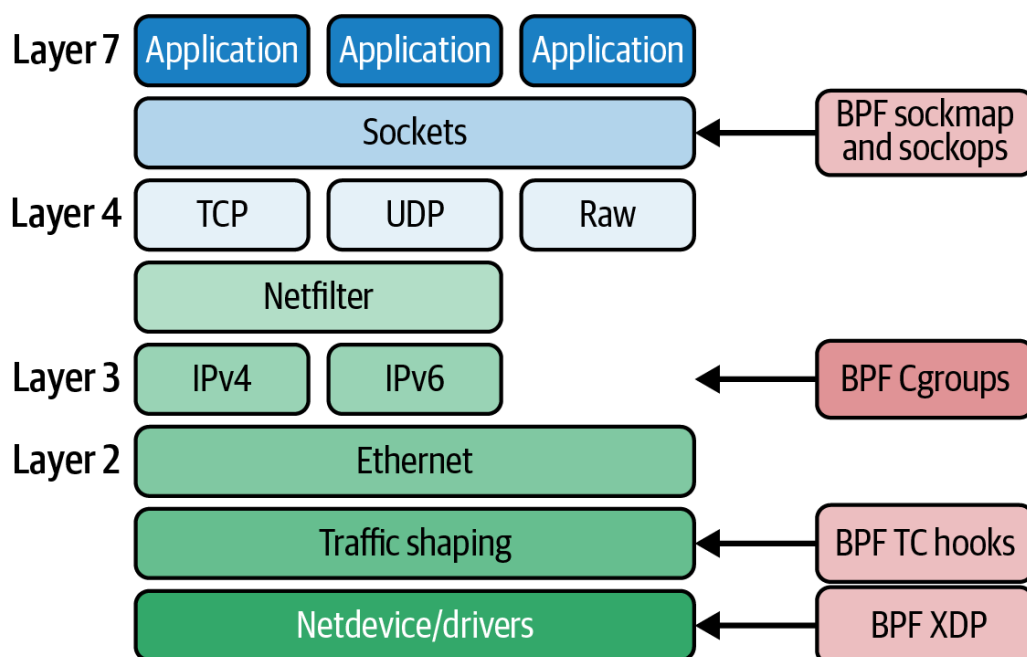


Рисунок 7-1. Типы программ BPF подключаются к различным точкам сетевого стека.

Модель уровней сетевого стека выходит за рамки этой книги, но она рассматривается во многих других книгах, статьях и учебных курсах. Я обсуждала это в главе 10 книги «Container Security» (by Liz Rice, Released April 2020, O'Reilly Media, Inc. — <https://www.oreilly.com/library/view/container-security/9781492056690/> ISBN: 9781492056706). Для целей же текущей книги достаточно знать, что уровень 7 охватывает форматы, предназначенные для использования пользовательскими приложениями, такие как HTTP, DNS или gRPC; протокол TCP находится на уровне 4; протокол IP находится на уровне 3; а протокол Ethernet и WiFi находятся на уровне 2. Одной из ролей сетевого стека является преобразование сообщений между этими различными форматами.

Одно большое различие между типами сетевых программ и типами, связанными с трассировкой, которые вы видели ранее в этой главе, заключается в том, что они обычно предназначены для ручной настройки сетевого поведения. Это включает в себя две основные характеристики:

1. Использование кода возврата из программы eBPF, чтобы сообщить ядру, что делать дальше с сетевым пакетом, что может включать его обычную обработку, удаление или перенаправление в другое место назначения.
2. Разрешение программе eBPF изменять сетевые пакеты, параметры конфигурации сокетов и т. д.

В следующей главе вы увидите несколько примеров того, как эти характеристики используются для создания мощных сетевых возможностей, а пока давайте рассмотрим типы программ eBPF.

Сокеты

В верхней части стека подмножество этих сетевых типов программ относится к сокетам и операциям с сокетами:

- `BPF_PROG_TYPE_SOCKET_FILTER` был первым типом программы, добавленным в ядро. Вы, вероятно, догадались из названия, что это используется для фильтрации сокетов, но что менее очевидно это то, что это не означает фильтрацию данных, отправляемых в приложение или из него. Он используется для фильтрации копии данных сокета, которые могут быть отправлены инструменту наблюдения, такому как `tcpdump`.
- Сокет специфичен для соединения уровня 4 (TCP). `BPF_PROG_TYPE_SOCK_OPS` позволяет программам eBPF перехватывать различные операции и действия, происходящие в сокете, и устанавливать для этого сокета такие параметры, как значения времени ожидания TCP. Сокеты существуют только в конечных точках соединения, но не на промежуточных уровнях сквозь которые обмен может пройти.
- Программы `BPF_PROG_TYPE_SK_SKB` используются в сочетании со специальным типом карты, который содержит набор ссылок на сокеты, чтобы обеспечить так называемые операции `sockmap`: (BPF: sockmap and sk redirect support — <https://lwn.net/Articles/731133/>) перенаправление трафика в разные пункты назначения на уровне сокетов.

Контроль трафика

Далее по сетевому стеку вниз идет «ТС» или управление трафиком (traffic control). В ядре Linux есть целая подсистема, связанная с ТС, и даже самый беглый взгляд на справочную страницу команды `tc` (`tc(8)` — Linux manual page — <https://man7.org/linux/man-pages/man8/tc.8.html>) даст вам представление о том, насколько она сложна, и насколько важно для вычислений в целом иметь глубокие уровни гибкости и конфигурирования в отношении того, как обрабатываются сетевые пакеты.

Можно подключить программы eBPF для предоставления настраиваемых фильтров и классификаторов сетевых пакетов как для входящего, так и для исходящего трафика. Это один из строительных блоков проекта компании Cilium, и я рассмотрю несколько примеров в следующей главе. Если вы не можете дождаться этого момента, в блоге Квентина Монне (Understanding tc “direct action” mode for BPF, April 11, 2020, Quentin Monnet — <https://qmonnet.github.io/whirl-offload/2020/04/11/tc-bpf-direct-action/>) есть несколько хороших примеров. Это можно сделать программно, но у вас также есть возможность использовать команду `tc` для управления такого типа программ eBPF.

XDP

Вы кратко познакомились с программами eBPF XDP (eXpress Data Path) ранее в главе 3. В том примере я загрузила программу eBPF и подключил ее к интерфейсу `eth0`, используя следующие команды:

```
# bpftool prog load hello.bpf.o /sys/fs/bpf/hello
# bpftool net attach xdp id 540 dev eth0
```

Стоит отметить, что программы XDP подключаются к конкретному интерфейсу (или виртуальному интерфейсу), и у вас вполне могут быть разные программы XDP, подключенные к разным интерфейсам. В главе 8 вы узнаете больше о том, как программы XDP могут быть загружены на сетевые карты или выполняться сетевыми драйверами.

Программы XDP — еще один пример программ, которыми можно управлять с помощью сетевых утилит Linux — в данном случае это подкоманда `link` команды `ip` пакета `iproute2` (`ip(8)` — Linux manual page — <https://man7.org/linux/man-pages/man8/ip.8.html>). Примерно эквивалентной командой для загрузки и подключения программы к `eth0` будет следующая:

```
# ip link set dev eth0 xdp obj hello.bpf.o sec xdp
```

Эта команда считывает программу eBPF, помеченную как секция `xdp`, из объекта `hello.bpf.o` и прикрепляет ее к сетевому интерфейсу `eth0`. Команда `ip link show` для этого интерфейса теперь содержит некоторую новую информацию о прикреплении к нему программе XDP:

```
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 xdpgeneric qdisc fq_codel
state UP mode DEFAULT group default qlen 1000
    link/ether 52:55:55:3a:1b:a2 brd ff:ff:ff:ff:ff:ff
    prog/xdp id 1255 tag 9d0e949f89f1a82c jited
```

Открепление этой программы XDP командой `ip link` делается подобно следующего:

```
# ip link set dev eth0 xdp off
```

Вы увидите гораздо больше относительно программ XDP и их применений в следующей главе.

Рассекатель потока

Рассекатель (диссектор) потока используется в различных точках сетевого стека для извлечения деталей из заголовков пакета. Программы eBPF типа `BPF_PROG_TYPE_FLOW_DISSECTOR` могут реализовывать пользовательское разделение пакетов. Здесь в статье LWN есть хорошая статья о написании диссекторов сетевого потока в BPF (Writing network flow dissectors in BPF — <https://lwn.net/Articles/764200/>).

Легковесные туннели

Семейство типов программ `BPF_PROG_TYPE_LWT_*` можно использовать для реализации сетевой инкапсуляции в программах eBPF. Этим типом программ также можно управлять с помощью команды `ip`, но на этот раз задействована будет подкоманда `route`. На практике они используются не часто.

Cgroup-ы

Программы eBPF могут быть присоединены к управляющим группам (сокращение от «контрольные группы»). `cgroup` — это концепция ядра Linux, которая ограничивает набор ресурсов, к которым может иметь доступ данный процесс или группа процессов. `cgroup` — это один из механизмов, изолирующих один контейнер (или один модуль Kubernetes) от другого. Присоединение программ eBPF к `cgroup` позволяет настраивать поведение, которое применяется только к процессам из этой контрольной группы. Все процессы связанные с `cgroup` включают и процессы, которые не выполняются внутри контейнера.

Существует несколько типов программ, связанных с `cgroup`, и еще больше ловушек, к которым их можно прикрепить. По крайней мере, на момент написания этой статьи почти все они связаны с сетью, хотя существует также тип программы `BPF_CGROUP_SYSCTL`, который можно присоединить к командам `sysctl`, влияющим на конкретную `cgroup`.

Например, существуют типы программ, связанных с сокетами, специфичные для `cgroup`, такие как `BPF_PROG_TYPE_CGROUP_SOCK` и `BPF_PROG_TYPE_CGROUP_SKB`. Программы eBPF могут определить разрешено ли данной `cgroup` выполнять запрошенную операцию сокета или передачу данных. Это полезно для применения политики сетевой безопасности (о которой я расскажу в следующей главе). Программы сокетов также могут обмануть вызывающий процесс, заставив его думать, что он подключается к определенному адресу назначения.

Инфракрасные контроллеры

Программы типа `BPF_PROG_TYPE_LIRC_MODE2` (IR decoding using BPF — <https://lwn.net/Articles/755752/>) могут быть прикреплены к файловому дескриптору для инфракрасного контроллера, чтобы обеспечить декодирование инфракрасных протоколов. На момент написания этой статьи для этого типа программы требуется привилегия `CAP_NET_ADMIN`, но я думаю, что это показывает, что разделение типов программ на связанные с трассировкой и связанные с сетью не полностью отражает весь диапазон различных приложений, которые может решать eBPF.

BPF типы прикреплений

Тип вложения предлагает более детальный контроль над тем, где программа может быть прикреплена в системе. Для некоторых типов программ существует взаимно-однозначная корреляция с типом хука, к которому он может быть присоединен, поэтому тип присоединения неявно определяется самим типом программы. Например, программы XDP присоединяются к ловушкам XDP в сетевом стеке. Для некоторых типов программ также необходимо указать явно тип прикрепления.

Тип прикрепления участвует в принятии решения о допустимости функций-помощников, а также, в некоторых случаях, ограничивает доступ к частям контекстной информации. Ранее в этой главе был пример, когда верификатор выдает ошибку: `an invalid bpf_context access error`.

Вы также можете увидеть, какие типы программ требуют указания типа вложения, и какие типы вложений допустимы, в функции ядра `bpf_prog_load_check_attach` (https://elixir.bootlin.com/linux/v5.19.17/C/ident/bpf_prog_load_check_attach, определена в `bpf/syscall.c`: <https://elixir.bootlin.com/linux/v5.19.17/source/kernel/bpf/syscall.c#L2284>).

Например, вот код, проверяющий тип прикрепления для программы типа `CGROUP_SOCK`:

```

case BPF_PROG_TYPE_CGROUP_SOCK:
    switch (expected_attach_type) {
    case BPF_CGROUP_INET_SOCK_CREATE:
    case BPF_CGROUP_INET_SOCK_RELEASE:
    case BPF_CGROUP_INET4_POST_BIND:
    case BPF_CGROUP_INET6_POST_BIND:
        return 0;
    default:
        return -EINVAL;
    }
}

```

Этот тип программы может быть присоединён к нескольким местам: при создании сокета, при освобождении сокета, или после завершения привязки к IPv4 или IPv6. Еще одним местом, где можно найти список допустимых типов вложений для программ, является документация `libbpf` (https://docs.kernel.org/bpf/libbpf/program_types.html — Program Types and ELF Sections), где вы также найдете имена разделов, которые `libbpf` понимает для каждой программы и типа присоединения.

Итоги

В этой главе вы видели, что различные типы программ eBPF используются для присоединения к различным точкам подключения в ядре. Если вы хотите написать код, реагирующий на определенное событие, вам нужно определить тип(ы) программы, которые подходят для присоединения к этому событию. Контекст, передаваемый в программу, зависит от типа программы, и ядро также может по-разному реагировать на код возврата вашей программы в зависимости от ее типа.

Пример кода для этой главы в основном сосредоточен на событиях, связанных с отслеживанием Perf (трассировочных) событий. В следующих двух главах вы увидите более подробную информацию о различных типах программ eBPF, используемых для сетевых приложений и приложений безопасности.

Упражнения

Пример кода для этой главы включает `kprobe`, `fentry`, `tracepoint`, `raw tracepoint` и программы `tracepoint` с поддержкой BTF, которые все присоединены к записи одного и того же системного вызова. Как вы знаете, программы трассировки eBPF могут быть присоединены ко многим различным местам помимо системных вызовов.

1. Запустите пример кода, используя `strace` для захвата системных вызовов `bpf()`, например так:

```
$ strace -e bpf -o outfile ./hello
```

Это запишет информацию о каждом системном вызове `bpf()` в файл с именем `outfile`. Найдите в этом файле инструкции `BPF_PROG_LOAD` и посмотрите, как различается файл `prog_type` для разных программ. Вы можете определить, какая программа есть какая, по полю `prog_name` в трассировке и сопоставить их с исходным кодом в `chapter7/hello.bpf.c`.

2. Пример кода пользовательского пространства в `hello.c` загружает все программные объекты, определенные в `hello.bpf.o`. В качестве упражнения по написанию кода

пользовательского пространства `libbpf` измените загрузку кода примера и подключите только одну из программ eBPF (выберите любую, которая вам нравится), не удаляя сами эти программы из `hello.bpf.c`.

3. Напишите программу `kprobe` и/или `fentry`, которая запускается при вызове ещё какой-либо другой функции ядра. Вы можете найти доступные функции в вашей версии ядра, просмотрев `/proc/kallsyms`.
4. Напишите обычную, `raw` или поддерживаемую BTF программу для `tracepoint`, которая подключается к какой-либо другой точке трассировки ядра. Вы можете найти доступные точки трассировки для этого в `/sys/kernel/tracing/available_events`.
5. Попробуйте подключить к заданному сетевому интерфейсу более одной программы XDP, и убедитесь, что это невозможно! Вы должны увидеть ошибку, которая выглядит примерно так:

```
libbpf: Kernel error message: XDP program already attached
Error: interface xdpgeneric attach failed: Device or resource busy
```