Часть 3 : Анатомия eBPF программы

В предыдущей главе вы видели простую программу eBPF «Hello World», написанную с использованием инфраструктуры BCC. В этой главе приведен пример версии программы «Hello World», полностью написанной на C, чтобы вы могли увидеть некоторые детали, о которых BCC позаботилась за кулисами.

В этой главе также показаны этапы, которые проходит программа eBPF на пути от исходного кода к выполнению, как показано на рис. 3-1.

Рисунок 3-1. Исходный код C (или Rust) компилируется в байт-код eBPF, который затем либо JIT-компилируется, либо интерпретируется в собственные инструкции машинного кода.

Программа eBPF представляет собой набор инструкций байт-кода eBPF. Можно написать код eBPF непосредственно в этом байт-коде, так же как можно и программировать на языке ассемблера. Но людям обычно проще иметь дело с языком программирования более высокого уровня, и, по крайней мере, на момент написания этой статьи я бы сказала что подавляющее большинство кода eBPF написано на C[1], а затем уже скомпилировано в байт-код eBPF.

Концептуально этот байт-код выполняется на виртуальной машине eBPF внутри ядра.

Виртуальная машина eBPF

Виртуальная машина eBPF, как и любая виртуальная машина, представляет собой программно реализованный компьютер. Он принимает программу в виде инструкций байт-кода eBPF, и их необходимо преобразовать в собственные машинные инструкции, которые и выполняются на ЦП.

В ранних реализациях eBPF инструкции байт-кода интерпретировались внутри ядра, то есть каждый раз, когда запускается программа eBPF, ядро проверяет его инструкции и преобразует их в машинный код, который затем выполняется. С тех пор интерпретация была в значительной степени заменена компиляцией JIT («на лету») из соображений производительности и во избежание проявлений возможности некоторых, связанных со Spectre[2], уязвимостей в интерпретаторе eBPF. Компиляция JIT означает что преобразование в собственные машинные инструкции происходит только один, первый, раз, когда программа загружается в ядро.

Байт-код eBPF состоит из последовательности инструкций, и эти инструкции воздействуют на (виртуальные) регистры eBPF. Набор инструкций eBPF и модель регистров были

1 Все чаще программы eBPF также пишутся на языке Rust, поскольку компилятор Rust непосредственно поддерживает байт-код eBPF в качестве цели.

2 Четвертого января 2018 года стало известно о масштабной концептуальной уязвимости, которая присутствует в большинстве процессоров Intel и позволяет любой непривилегированный программе получить доступ к данным, которые хранятся в оперативной памяти компьютера. Эта уязвимость получила название Meltdown. Чуть позже был обнаружен способ эксплуатации этой уязвимости почти на всех процессорах, который получил название уязвимость Spectre.

разработаны для точного сопоставления с распространенными архитектурами ЦП, поэтому этап компиляции в машинный код или интерпретации байт-кода является достаточно простым.

Регистры eBPF

Виртуальная машина eBPF использует 10 регистров общего назначения, пронумерованных от 0 до 9. Кроме того, регистр 10 используется в качестве указателя кадра стека (и может только читаться, но не записываться). По мере выполнения программы BPF значения сохраняются в этих регистрах для отслеживания состояния.

Важно понимать, что эти регистры eBPF в виртуальной машине eBPF реализованы программно. Вы можете увидеть их перечисление от `BPF_REG_0` до `BPF_REG_10` в заголовочном файле `include/uapi/linux/bpf.h` исходного кода ядра Linux (https://elixir.bootlin.com/linux/v5.19.17/source/include/uapi/linux/bpf.h).

Аргумент контекста программы eBPF загружается в регистр 1 перед началом выполнения программы. Возвращаемое функцией значение сохраняется в регистре 0.

Перед вызовом функции из кода eBPF аргументы этой функции помещаются в регистры с 1 по 5 (не все регистры используются, если аргументов меньше пяти).

Инструкции eBPF

Тот же заголовочный файл `linux/bpf.h` определяет структуру с именем `bpf_insn`, которая описывает инструкцию BPF:

```
struct bpf_insn {
__u8 code;        /* opcode */
__u8 dst_reg:4;   /* dest register */
__u8 src_reg:4;   /* source register */
__s16 off;        /* signed offset */
__s32 imm;        /* signed immediate constant */
};
```

1). У каждой инструкции есть код операции (`code`), который определяет, какую операцию должна выполнять инструкция: например, добавление значения к содержимому регистра или переход к другой инструкции в программе[3]. Документ «Неофициальная спецификация eBPF» проекта Iovisor (https://github.com/iovisor/bpf-docs/blob/master/eBPF.md) содержит список действующих инструкций.

2). Различные операции могут включать до двух регистров (`dst_reg` и `src_reg`).

3). `off, imm` — в зависимости от операции может быть значение смещения и/или «непосредственное» целочисленное значение.

Эта структура `bpf_insn` имеет длину 64 бита (или 8 байтов). Однако иногда для инструкции может потребоваться занимать более 8 байт. Если вы хотите установить для регистра 64-битное значение, вы не можете каким-то образом втиснуть все 64 бита этого значения в структуру вместе с кодом операции и информацией о регистре. В таких случаях в

---

3 Есть несколько инструкций, в которых операция «модифицируется» значением других полей в инструкции. Например, в ядре 5.12 появился набор атомарных инструкций, которые включают арифметическую операцию (ADD, AND, OR, XOR), указанную в поле `imm`.

инструкции используется *расширокая кодировка инструкций*, общая длина которой составляет 16 байт. Вы увидите пример этого в этой главе.

При загрузке в ядро байт-код программы eBPF представляется серией этих структур `bpf_insn`. Верификатор выполняет несколько проверок этой информации, чтобы убедиться что код безопасен для запуска. Вы узнаете больше о процессе верификации в главе 6.

Большинство разнообразных кодов операций распределяются по следующим категориям:

- Загрузка значения в регистр (либо непосредственное значение, либо значение, прочитанное из памяти или из другого регистра)
- Сохранение значения из регистра в память
- Выполнение арифметических операций, таких как добавление значения к содержимому регистра.
- Переход к другой инструкции, если выполняется определенное условие.

> Для обзора архитектуры eBPF я рекомендую «Справочное руководство по BPF и XDP» (https://docs.cilium.io/en/stable/bpf/), которое включено в документацию проекта Cilium. Если вам нужны подробности, то в документации ядра достаточно четко описаны инструкции и кодировка eBPF (https://docs.kernel.org/bpf/instruction-set.html).

Давайте возьмем еще один простой пример программы eBPF и проследим ее путь от исходного кода C через байт-код eBPF к инструкциям машинного кода.

> Если вы хотите создать и запустить этот код самостоятельно, вы найдете код вместе с инструкциями по настройке среды для этого на https://github.com/lizrice/learning-ebpf. Код для этой главы находится в каталоге Chapter3.
>
> Примеры в этой главе написаны на C с использованием библиотеки `libbpf`. Вы узнаете больше об этой библиотеке в главе 5.

eBPF «Hello World» для сетевого интерфейса

Примеры из предыдущей главы выдавали трассировку «Hello World», запускаемую системным вызовом `kprobe`; а на этот раз я собираюсь показать программу eBPF, которая записывает строку трассировки, которая активируется прибытием сетевого пакета.

Обработка пакетов — очень распространенное применение eBPF. Я расскажу об этом более подробно в главе 8, а сейчас может быть полезно ознакомиться с основной идеей программы eBPF, которая активируется для каждого пакета данных, поступающего на сетевой интерфейс. Программа может проверить и даже изменить содержимое этого пакета, и принять решение (или вердикт) о том, что ядро должно далее делать с этим пакетом. Этот вердикт может указать ядру продолжить обработку в обычном режиме, отбросить его, или перенаправить в другое место.

В простом примере, который я здесь показываю, программа ничего не делает с сетевым пакетом; она просто выводит слова *Hello World* и счетчик в канал трассировки каждый раз при получении очередного сетевого пакета. Пример программы находится в `Chapter3/hello.bpf.c`. Это довольно распространенное соглашение помещать

программы eBPF в имена файлов оканчивающиеся на `bpf.c` чтобы отличить их от кода C пользовательского пространства, который может находиться в том же каталоге исходного кода. Вот вся программа:

```
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>
int counter = 0;
SEC("xdp")
int hello(void *ctx) {
   bpf_printk("Hello World %d", counter);
   counter++;
   return XDP_PASS;
}
char LICENSE[] SEC("license") = "Dual BSD/GPL";
```

1). Этот пример начинается с включения некоторых заголовочных файлов *.h. На всякий случай, если вы не знакомы с кодированием C, каждая программа должна включать файлы заголовков, которые определяют любые структуры или функции, которые программа собирается использовать. По именам можно догадаться что эти заголовочные файлы связаны с BPF.

2). В этом примере показано, как программы eBPF могут использовать глобальные переменные. Этот счетчик `counter` будет увеличиваться при каждом выполнении программы.

3). Макрос `SEC()` определяет раздел с именем `xdp`, который вы сможете увидеть в скомпилированном объектном файле. Я вернусь к тому, как имя раздела используется в главе 5, но сейчас вы можете просто думать о нем как об определении того что это вид eXpress Data Path (XDP) программ eBPF.

4). Далее вы можете увидеть реальную программу eBPF. В eBPF имя программы — это имя функции, поэтому эта программа называется `hello`. Она использует функцию-помощник `bpf_printk()` для записи строки текста, увеличивает значение счетчика глобальной переменной, а затем возвращает значение `XDP_PASS`. Этот результат указывает ядру что оно должно обработать этот сетевой пакет как обычно.

5). Наконец, есть еще один макрос `SEC()`, который определяет строку лицензии, и это важное требование для программ eBPF. Некоторые функции-помощники BPF в ядре определены как «только GPL»[4]. Если вы хотите использовать любую из этих функций, ваш код BPF должен быть объявлен как имеющий лицензию, совместимую с GPL. Верификатор (которого мы обсудим в главе 6) будет возражать, если объявленная лицензия несовместима с функциями, которые использует программа. Некоторые типы программ eBPF, включая те, которые используют BPF LSM (о которых вы узнаете в главе 9), также должны быть совместимы с GPL (https://docs.kernel.org/bpf/bpf_licensing.html#using-bpf-programs-in-the-linux-kernel).

> Вам может стать любопытно интересно, почему в предыдущей главе использовалась функция `bpf_trace_printk()`, а в этой — `bpf_printk()`.

4 В ядре Linux все экспортируемые (доступные для использования) имена (функций и данных) могут определяться как «экспортируемые для всех» и как «экспортируемые только для кодов под лицензией GNU». (Прим. пер.)

> Короткий ответ: версия из BCC называется `bpf_trace_printk()`, а версия из `libbpf` — `bpf_printk()`, но обе они являются оболочками функции ядра `bpf_trace_printk()`. Хороший пост об этом написал в своем блоге Андрей Накрыйко (https://nakryiko.com/posts/bpf-tips-printk/).

Это пример программы eBPF, которая подключается к точке подключения XDP на сетевом интерфейсе. Вы можете думать о событии XDP как о таком, которое запускается в момент поступления нового сетевого пакета на (физический или виртуальный) сетевой интерфейс.

> Некоторые сетевые карты поддерживают облегчение нагрузки программ XDP тем, что их можно выполнять на самом сетевом адаптере. Это означает, что каждый поступающий сетевой пакет может быть предварительно обработан самим сетевым адаптером ещё до того, как он поступит к центральному процессору машины. Программы XDP могут проверять и даже изменять каждый сетевой пакет, поэтому это очень полезно для выполнения таких задач, как защита от DDoS-атак, межсетевой экран или балансировка нагрузки с высокой производительностью. Вы узнаете больше об этом в главе 8.

Вы увидели исходный код C, и ттеперь следующим шагом будет его компиляция в объект, понятный ядру.

Компиляция объектного файла eBPF

Наш исходный код eBPF необходимо скомпилировать в машинные инструкции, понятные виртуальной машине eBPF: в байт-код eBPF. Компилятор Clang из проекта LLVM сделает это, если вы укажете `-target bpf`. Ниже приведен фрагмент `Makefile`, который будет выполнять компиляцию:

```
hello.bpf.o: %.o: %.c
    clang \
        -target bpf \
        -I/usr/include/$(shell uname -m)-linux-gnu \
        -g \
        -O2 -c $< -o $@
```

Такая компиляция создает объектный файл с именем `hello.bpf.o` из исходного кода в `hello.bpf.c`. Флаг `-g` здесь необязателен[5], но он генерирует отладочную информацию чтобы вы могли видеть исходный код вместе с байт-кодом при проверке объектного файла. Давайте рассмотрим этот объектный файл, чтобы лучше понять содержащийся в нем код eBPF.

Анализ объектного файла eBPF

Утилита `file` обычно используется для определения принадлежности файла по его содержимому:

```
$ file hello.bpf.o
```

5 Флаг -g необходим для генерации информации BTF, которая понадобится вам для CO-RE программ eBPF, о которых я расскажу в главе 5.

```
hello.bpf.o: ELF 64-bit LSB relocatable, eBPF, version 1 (SYSV), with debug_info,
not stripped
```

Вывод показывает, что это файл ELF (Executable and Linkable Format), содержащий код eBPF, для 64-битной платформы с архитектурой LSB (наименее значащий бит). Он включает отладочную информацию, если вы использовали флаг `-g` на этапе компиляции. Вы можете дополнительно проверить этот объект с помощью `llvm-objdump` чтобы увидеть инструкции eBPF:

```
$ llvm-objdump -S hello.bpf.o
```

Даже если вы не дружите с дизассемблированием, вывод этой команды не так уж сложен для понимания:

```
hello.bpf.o:                  file format elf64-bpf
Disassembly of section xdp:
0000000000000000 <hello>:
; bpf_printk("Hello World %d", counter");
   0:   18 06 00 00 00 00 00 00 00 00 00 00 00 00 00 00 r6 = 0 ll
   2:   61 63 00 00 00 00 00 00 r3 = *(u32 *)(r6 + 0)
   3:   18 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 r1 = 0 ll
   5:   b7 02 00 00 0f 00 00 00 r2 = 15
   6:   85 00 00 00 06 00 00 00 call 6
; counter++;
   7:   61 61 00 00 00 00 00 00 r1 = *(u32 *)(r6 + 0)
   8:   07 01 00 00 01 00 00 00 r1 += 1
   9:   63 16 00 00 00 00 00 00 *(u32 *)(r6 + 0) = r1
; return XDP_PASS;
  10:   b7 00 00 00 02 00 00 00 r0 = 2
  11:   95 00 00 00 00 00 00 00 exit
```

1). Первая строка дает дополнительное подтверждение того что `hello.bpf.o` — это 64-битный файл ELF с кодом eBPF (нет особой рифмы или причины, по которой одни инструменты используют термин BPF, а другие — eBPF; как я уже говорила ранее, эти термины теперь практически используются взаимозаменяемо).

2). Далее следует дизассемблирование секции с меткой `xdp`, которая соответствует определению `SEC()` в исходном коде C.

3). Этот раздел представляет собой функцию под названием `hello`.

4). Далее пять строк инструкций байт-кода eBPF, которые соответствуют исходной строке кода: `bpf_printk("Hello World %d", counter");`.

5). Три строки инструкций байт-кода eBPF, которые соответствуют инкременту переменной `counter`.

6). И еще две строки байт-кода генерируются из исходного кода возврата: `return XDP_PASS;`.

Если вы не сильно заинтересованы во всём этом, вам не нужно точно понимать, как каждая строка байт-кода связана с исходным кодом. Компилятор сам заботится о создании байт-кода, так что вам не нужно об этом думать! Но давайте рассмотрим вывод более подробно, чтобы

вы могли понять, как этот вывод соотносится с инструкциями и регистрами eBPF, о которых вы узнали ранее в этой главе.

Слева от каждой строки байт-кода вы можете увидеть смещение этой инструкции от места расположения `hello` в памяти. Как описано ранее в этой главе, инструкции eBPF обычно имеют длину 8 байтов, а поскольку на 64-разрядной платформе каждая ячейка памяти может содержать 8 байтов, смещение обычно увеличивается на единицу для каждой инструкции. Однако первая инструкция в этой программе представляет собой кодировку расширенной инструкции, которая требует 16 байтов чтобы установить в регистре 6 64-битное значение 0. Это помещает инструкцию во вторую строку вывода по смещению 2. После этого есть (через одну) еще одна 16-байтовая инструкция, устанавливающая для регистра 1 64-битное значение 0. И после этого каждая оставшаяся инструкция умещается в 8 байтов, поэтому смещение увеличивается на единицу в каждой строке.

Первый байт каждой строки — это код операции, сообщающий ядру, какую операцию выполнять, а в правой части каждой строки инструкции — удобочитаемая интерпретация инструкции. На момент написания этой статьи проект Iovisor имеет наиболее полную документацию (https://github.com/iovisor/bpf-docs/blob/master/eBPF.md) кодов операций eBPF, но официальная документация (https://docs.kernel.org/bpf/instruction-set.html) ядра Linux её нагоняет, а сообщество eBPF Foundation работает над стандартной документацией (https://github.com/ietf-wg-bpf/ebpf-docs), не привязанной к какой-то конкретной операционной системе.

Например, возьмем инструкцию по смещению 5, которая выглядит так:

```
5:   b7 02 00 00 0f 00 00 00 r2 = 15
```

Код операции — 0xb7, и документация говорит нам что соответствующий ему псевдокод — `dst = imm`, что можно прочитать как «Установить место назначения на непосредственное значение». Место назначения определяется вторым байтом, 0x02, что означает «Регистр 2». «Непосредственное» (или буквальное) значение здесь — 0x0f, что равно 15 в десятичном виде. Итак, мы можем понять что эта инструкция говорит ядру «установить для регистра 2 значение 15». Это и соответствует выводу, который мы видим в правой части инструкции: r2 = 15.

Инструкция по смещению 10 аналогична:

```
10:   b7 00 00 00 02 00 00 00 r0 = 2
```

Эта строка также имеет код операции 0xb7, и на этот раз она устанавливает значение регистра 0 в 2. Когда программа eBPF завершает работу, регистр 0 содержит код возврата, а `XDP_PASS` имеет значение 2. Это соответствует исходному коду, который всегда возвращает `XDP_PASS`.

Теперь вы знаете, что `hello.bpf.o` содержит программу eBPF в байт-коде. Следующим шагом будет загрузка его в ядро.

Загрузка программы в ядро

В этом примере мы будем использовать утилиту под названием `bpftool`. Вы также можете загружать программы программно, и вы увидите примеры этого позже в книге.

> Некоторые дистрибутивы Linux предоставляют пакет, включающий утилиту `bpftool`, или вы можете скомпилировать её из исходного кода (https://github.com/libbpf/bpftool). Вы можете найти более подробную информацию

об установке или сборке этого инструмента в блоге Квентина Монне (https://qmonnet.github.io/whirl-offload/2021/09/23/bpftool-features-thread/), а также дополнительную документацию и информацию об использовании на сайте Cilium (https://docs.cilium.io/en/latest/bpf/#bpftool).

Ниже приведен пример использования `bpftool` для загрузки программы в ядро. Обратите внимание, что вам, вероятно, потребуется `root` (или использовать `sudo`), чтобы получить привилегии BPF, которые требуются `bpftool`.

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

Это загружает программу eBPF из нашего скомпилированного объектного файла и «прикрепляет» ее к местоположению `/sys/fs/bpf/hello`[6]. Отсутствие выходного ответа на выполнение команды указывает на успех, но вы можете подтвердить что программа на месте, используя `ls`:

```
$ ls /sys/fs/bpf
hello
```

Программа eBPF успешно загружена. Давайте воспользуемся утилитой (всё той же) `bpftool` чтобы узнать больше о программе и ее статусе в ядре.

Проверка загруженной программы

Утилита `bpftool` может перечислить все программы, загруженные в ядро. Если вы сами попробуете это сделать, то вы, вероятно, увидите в этом выводе несколько ранее загруженных программ eBPF, но для ясности я просто покажу строки, относящиеся только к нашему примеру «Hello World»:

```
# bpftool prog list
...
540: xdp name hello tag d35b94b4c0c10efb gpl
loaded_at 2022-08-02T17:39:47+0000 uid 0
xlated 96B jited 148B memlock 4096B map_ids 165,166
btf_id 254
```

Программе был присвоен идентификатор 540. Этот идентификатор представляет собой номер, присваиваемый каждой программе по мере ее загрузки. Зная ID, вы можете попросить `bpftool` показать больше информации об этой программе. На этот раз давайте получим вывод в предварительно подготовленном формате JSON, чтобы имена полей были видны, а также их значения:

```
# bpftool prog show id 540 --pretty
{
    "id": 540,
    "type": "xdp",
    "name": "hello",
    "tag": "d35b94b4c0c10efb",
    "gpl_compatible": true,
```

6 В общем случае, это необязательно — программы eBPF можно загружать в ядро без привязки к местоположению файла — но это не есть опционально для утилиты `bpftool`, который всегда должен закреплять загружаемые программы. Причина этого более подробно будет описана позже в разделе «Справочные материалы по программам и картам BPF».

```
    "loaded_at": 1659461987,
    "uid": 0,
    "bytes_xlated": 96,
    "jited": true,
    "bytes_jited": 148,
    "bytes_memlock": 4096,
    "map_ids": [165,166
    ],
    "btf_id": 254
}
```

Учитывая названия полей, многое из этого понять несложно:

- `id` программы равный 540.
- Поле типа сообщает нам что эта программа может быть подключена к сетевому интерфейсу с помощью события XDP. Несколько других типов программ BPF могут быть привязаны к различным типам событий, и мы обсудим это подробнее в главе 7.
- Имя программы — `hello`, это имя функции из исходного кода.
- `tag` — еще один идентификатор этой программы, о котором я вскоре расскажу подробнее.
- Программа определяется лицензией, совместимой с GPL.
- Имеется временная метка, показывающая, когда программа была загружена.
- Пользователь с идентификатором `uid` 0 (`root`) загрузил программу.
- В этой программе есть 96 байт транслированного байт-кода eBPF, о чём я вам вскоре расскажу.
- Эта программа была скомпилирована JIT, и в результате компиляции было получено 148 байт машинного кода. Я расскажу об этом тоже в ближайшее время.
- Поле `bytes_memlock` сообщает нам что эта программа резервирует 4096 байт памяти, которые не будут выгружаться.
- Эта программа обращается к картам BPF с идентификаторами 165 и 166. Это может показаться удивительным, поскольку в исходном коде нет явной ссылки на карты. Позже в этой главе вы увидите, как семантика карты используется для обработки глобальных данных в программах eBPF.
- Вы узнаете о BTF в главе 5, а сейчас просто знайте что `btf_id` указывает на наличие блока информации BTF для этой программы. Эта информация включается в объектный файл, только если вы компилируете с флагом `-g`.

Тег программы BPF

Тег представляет собой сумму инструкций программы SHA (Secure Hashing Algorithm), которую можно использовать в качестве другого идентификатора программы. Идентификатор может меняться каждый раз, когда вы загружаете или выгружаете программу, но тег остается

прежним. Утилита bpftool принимает ссылки на программу BPF по идентификатору, имени, тегу или закрепленному пути, поэтому в приведенном здесь примере все следующие действия дадут один и тот же результат:

- `bpftool prog show id 540`
- `bpftool prog show name hello`
- `bpftool prog show tag d35b94b4c0c10efb`
- `bpftool prog show pinned /sys/fs/bpf/hello`

У вас может быть несколько программ с одним и тем же именем и даже несколько экземпляров программ с одним и тем же тегом, но идентификатор и закрепленный путь всегда будут уникальными.

Транслированный байт-код

Поле `bytes_xlated` сообщает нам, сколько байтов содержит «транслированный» код eBPF. Это байт-код eBPF после того, как он прошел проверку (и, возможно, был изменен ядром по причинам, которые я буду обсуждать позже в этой книге).

Давайте используем `bpftool` чтобы показать эту транслированную версию нашего кода «Hello World»:

```
# bpftool prog dump xlated name hello
int hello(struct xdp_md * ctx):
; bpf_printk("Hello World %d", counter);
   0: (18) r6 = map[id:165][0]+0
   2: (61) r3 = *(u32 *)(r6 +0)
   3: (18) r1 = map[id:166][0]+0
   5: (b7) r2 = 15
   6: (85) call bpf_trace_printk#-78032
; counter++;
   7: (61) r1 = *(u32 *)(r6 +0)
   8: (07) r1 += 1
   9: (63) *(u32 *)(r6 +0) = r1
; return XDP_PASS;
  10: (b7) r0 = 2
  11: (95) exit
```

Это очень похоже на дизассемблированный код, который вы видели ранее в выходных данных `llvm-objdump`. Адреса смещения одинаковы, и инструкции выглядят одинаково — например, мы можем видеть что инструкция по смещению 5 это r2 = 15.

JIT-компилируемый машинный код

Транслированный байт-код довольно низкоуровневый, но это еще не совсем машинный код. eBPF использует JIT-компилятор для преобразования байт-кода eBPF в машинный код, который нативно запускается на целевом ЦП. Поле `bytes_jited` показывает что после этого диалога длина программы составляет 108 байт.

> Для повышения производительности программы eBPF обычно компилируются JIT. Альтернативой является интерпретация байт-кода eBPF во время выполнения. Набор инструкций и регистры eBPF были разработаны таким образом чтобы

достаточно точно отображать собственные машинные инструкции, чтобы сделать эту интерпретацию простой и, следовательно, относительно быстрой, но скомпилированные программы будут работать быстрее, и большинство архитектур на сегодня поддерживают JIT[7].

Утилита `bpftool` может создать дамп этого JIT-кода на языке ассемблера. Не волнуйтесь, если вы не знакомы с языком ассемблера, и это выглядит совершенно непонятно! Я включила его только для того чтобы проиллюстрировать все те преобразования, через которые проходит код eBPF из исходного кода до исполняемых машинных инструкций. Вот команда и ее вывод:

```
# bpftool prog dump jited name hello
int hello(struct xdp_md * ctx):
bpf_prog_d35b94b4c0c10efb_hello:
; bpf_printk("Hello World %d", counter);
   0:   hint    #34
   4:   stp     x29, x30, [sp, #-16]!
   8:   mov     x29, sp
   c:   stp     x19, x20, [sp, #-16]!
  10:   stp     x21, x22, [sp, #-16]!
  14:   stp     x25, x26, [sp, #-16]!
  18:   mov     x25, sp
  1c:   mov     x26, #0
  20:   hint    #36
  24:   sub     sp, sp, #0
  28:   mov     x19, #-140733193388033
  2c:   movk    x19, #2190, lsl #16
  30:   movk    x19, #49152
  34:   mov     x10, #0
  38:   ldr     w2, [x19, x10]
  3c:   mov     x0, #-205419695833089
  40:   movk    x0, #709, lsl #16
  44:   movk    x0, #5904
  48:   mov     x1, #15
  4c:   mov     x10, #-6992
  50:   movk    x10, #29844, lsl #16
  54:   movk    x10, #56832, lsl #32
  58:   blr     x10
  5c:   add     x7, x0, #0
; counter++;
  60:   mov     x10, #0
  64:   ldr     w0, [x19, x10]
  68:   add     x0, x0, #1
  6c:   mov     x10, #0
```

7 Параметр ядра `CONFIG_BPF_JIT` должен быть включен чтобы воспользоваться преимуществами JIT-компиляции, и его можно включить или отключить во время выполнения с помощью параметра `net.core.bpf_jit_enable` команды `sysctl`. Дополнительные сведения о поддержке JIT для различных архитектур микросхем см. в документации.

```
  70:   str     w0, [x19, x10]
; return XDP_PASS;
  74:   mov     x7, #2
  78:   mov     sp, sp
  7c:   ldp     x25, x26, [sp], #16
  80:   ldp     x21, x22, [sp], #16
  84:   ldp     x19, x20, [sp], #16
  88:   ldp     x29, x30, [sp], #16
  8c:   add     x0, x7, #0
  90:   ret
```

> В некоторых пакетных дистрибутивах версии `bpftool` еще не включают поддержку дампа вывода JIT, и если это так, то вы увидите сообщение «Ошибка: нет поддержки libbfd». Вы можете собрать `bpftool` для себя из исходного кода, следуя инструкциям на странице https://github.com/libbpf/bpftool.

Вы увидели, что программа «Hello World» была загружена в ядро, но на данный момент она еще не связана с событием, поэтому ничто не произведёт ее запуск. Код нужно привязать к событию.

Присоединение к событию

Тип программы должен соответствовать типу события, к которому она привязывается; вы узнаете об этом больше в главе 7. В данном случае это программа XDP, и вы можете использовать утилиту `bpftool` для присоединения примера программы eBPF к событию XDP на сетевом интерфейсе, например:

```
# bpftool net attach xdp id 540 dev eth0
```

> На момент написания этого текста утилита `bpftool` ещё не поддерживала возможность подключения для всех типов программ, но недавно она была расширена для автоматического подключения k(ret)probes, u(ret)probes и точек трассировки.

Здесь я использовала идентификатор программы 540, но вы также можете использовать имя (при условии что оно уникально) или тег для идентификации присоединяемой программы. В этом примере я подключил программу к сетевому интерфейсу `eth0`.

Вы можете просмотреть все программы eBPF, подключенные к сети, с помощью `bpftool`:

```
# bpftool net list
xdp:
eth0(2) driver id 540

tc:

flow_dissector:
```

Программа с ID 540 привязана к событию XDP на интерфейсе `eth0`. Этот вывод также дает некоторые подсказки о некоторых других типах потенциальных событиях в сетевом стеке, к

которым вы можете прикрепить программы eBPF: `tc` и `flow_dissector`. Подробнее об этом в главе 7.

Вы также можете проверить сетевые интерфейсы, используя команду `ip link`, и вы увидите вывод, который выглядит примерно так (некоторые детали были удалены для ясности):

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT
group default qlen 1000
    ...
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 xdp qdisc fq_codel state UP
mode DEFAULT group default qlen 1000
    ...
    prog/xdp id 540 tag 9d0e949f89f1a82c jited
    ...
```

В этом примере есть два интерфейса: петлевой интерфейс `lo`, который используется для отправки трафика процессам внутри этой машине; и интерфейс `eth0`, который соединяет эту машину с внешним миром. Эти выходные данные также показывают что `eth0` имеет JIT-компилированную программу eBPF с идентификатором 540 и тегом 9d0e949f89f1a82c, прикрепленную к его ловушке XDP.

> Вы также можете использовать комаду `ip link …` для подключения и отключения программ XDP к сетевому интерфейсу. Я включила это в качестве упражнения в конец этой главы, а дополнительные примеры будут приведены в главе 7.

В таком состоянии программа `hello` eBPF должна производить вывод трассировки каждый раз при получении нового сетевого пакета. Вы можете проверить это, запустив команду `cat /sys/kernel/debug/tracing/trace_pipe`. Это должно показать массированный вывод, похожий на вот такое:

```
<idle>-0       [003] d.s.. 655370.944105: bpf_trace_printk: Hello World 4531
<idle>-0       [003] d.s.. 655370.944587: bpf_trace_printk: Hello World 4532
<idle>-0       [003] d.s.. 655370.944896: bpf_trace_printk: Hello World 4533
```

Если вы изо всех сил пытаетесь вспомнить расположение (путь) канала трассировки, вы можете получить тот же результат, просто используя команду: `bpftool prog tracelog`.

По сравнению с выводом, который вы видели в главе 2, на этот раз с каждым из этих событий не связано ни команды, ни идентификатора процесса; вместо этого вы видите <idle>-0 в начале каждой строки трассировки. В главе 2 каждое событие системного вызова происходило из-за того что процесс, выполняющий приложение в пользовательском пространстве, вызывал API системного вызова. Этот идентификатор процесса и команда являются частью контекста, в котором выполнялась программа eBPF. Но в приведенном здесь примере событие XDP происходит из-за прибытия сетевого пакета. С этим пакетом не связано никакого пользовательского процесса — в момент запуска программы `hello` eBPF система ничего не сделала с пакетом, кроме как приняла его в память, и она понятия не имеет что это за пакет и куда он направляется.

Вы можете видеть, что отслеживаемое значение счетчика каждый раз увеличивается на единицу, как и ожидалось. В исходном коде `counter` является глобальной переменной. Давайте посмотрим, как это реализовано в eBPF с использованием карты.

Глобальные переменные

Как вы узнали из предыдущей главы, карта eBPF — это структура данных, к которой можно получить доступ или из программы eBPF или из пользовательского пространства. Поскольку к одной и той же карте можно обращаться неоднократно при разных активациях одной и той же программы, то ее можно использовать для хранения состояния от одного выполнения к другому. Несколько программ также могут получить доступ к одной и той же карте. Благодаря этим особенностям семантика карты может быть переопределена для использования в качестве глобальных переменных.

> До того, как в 2019 году была добавлена поддержка глобальных переменных (https://lore.kernel.org/bpf/20190228231829.11993-7-daniel@iogearbox.net/t/#u), программистам eBPF приходилось явно писать карты для выполнения той же задачи.

Ранее вы видели, что `bpftool` показывает пример этой программы, используя две карты с идентификаторами 165 и 166. (Вы, вероятно, увидите разные идентификаторы, если попробуете это повторить сами, поскольку идентификаторы назначаются при создании карт в ядре.) Давайте рассмотрим что в этих картах.

Утилита `bpftool` умеет показывать карты, загруженные в ядро. Для ясности я покажу в выводе только записи 165 и 166, которые относятся к примеру программы «Hello World»:

```
# bpftool map list
165: array name hello.bss  flags 0x400
        key 4B value 4B max_entries 1 memlock 4096B
        btf_id 254
166: array name hello.rodata flags 0x80
        key 4B value 15B max_entries 1 memlock 4096B
        btf_id 254 frozen
```

Секция `bss`[8] в объектном файле, скомпилированном из программы на C, обычно содержит глобальные переменные, и вы можете использовать `bpftool` для проверки его содержимого, например:

```
# bpftool map dump name hello.bss
[{
        "value": {
            ".bss": [{
                    "counter": 11127
                }
            ]
        }я
    }
]
```

Я могла бы также использовать идентификатор дампа карты `bpftool` 165 для получения той же информации. Если я снова запущу любую из этих команд, я увижу, что счетчик увеличился, так как программа запускалась каждый раз, когда получен сетевой пакет.

---

8 Здесь bss означает «блок, начинающийся с символа».

Как вы узнаете из главы 5, `bpftool` может красиво печатать имена полей из карты (здесь — это имя переменной `counter`) только в том случае, если доступна информация BTF, а эта информация включается только в том случае, если вы компилируете с опцией `-g`. Если вы опустите эту опцию на этапе компиляции, вы увидите что-то похожее на это:

```
# bpftool map dump name hello.bss
key: 00 00 00 00 value: 19 01 00 00
Found 1 element
```

Без информации BTF утилита `bpftool` не может узнать, какое имя переменной использовалось в исходном коде. Вы можете сделать вывод, что, поскольку на этой карте есть только один элемент, шестнадцатеричное значение 19 01 00 00 должно быть текущим значением `counter` (281 в десятичном формате, поскольку байты упорядочены, начиная с младшего значащего байта).

Вы видели здесь, что программа eBPF использует семантику карты для чтения и записи в глобальную переменную. Карты также используются для хранения статических данных, в чем вы можете убедиться, изучив другую карту.

Тот факт, что другая карта называется `hello.rodata`, намекает на то, что это могут быть данные только для чтения, относящиеся к нашей программе `hello`. На дампе содержимого этой карты вы можете увидеть что она содержит строку, используемую программой eBPF для трассировки:

```
# bpftool map dump name hello.rodata
[{
        "value": {
            ".rodata": [{
                    "hello.____fmt": "Hello World %d"
                        }
            ]
        }
    }
]
```

Если вы не компилировали объект с флагом `-g`, то вы увидите вывод подобный следующему:

```
# bpftool map dump id 166
key: 00 00 00 00 value: 48 65 6c 6c 6f 20 57 6f 72 6c 64 20 25 64 00
Found 1 element
```

В этой карте присутствует одна пара ключ-значение, а значение содержит 12 байтов данных, оканчивающихся на 0. Вероятно, вас не удивит что эти байты представляют собой ASCII-представление строки «Hello World %d».

Теперь, когда мы закончили проверку этой программы и ее карт, пришло время почистить её. Мы начнем с открепления её от события, которое её запускает.

Отключение программы

Вы можете отключить программу от сетевого интерфейса подобным образом:

```
# bpftool net detach xdp dev eth0
```

Если эта команда выполняется успешно, то вывода не будет никакого, но вы можете убедиться, что программа больше не подключена по отсутствию записей XDP в выводе команды `bpftool net list`:

```
$ bpftool net list
xdp:

tc:

flow_dissector:
```

Однако программа всё ещё загружена в ядро:

```
$ bpftool prog show name hello
395: xdp name hello tag 9d0e949f89f1a82c gpl
        loaded_at 2022-12-19T18:20:32+0000 uid 0
        xlated 48B jited 108B memlock 4096B map_ids 4
```

Выгрузка программ

У команды `bpftool` нет способа обратной выгрузки программы (по крайней мере, на момент написания этого текста), но вы можете удалить программу из ядра, просто удалив закрепленный псевдофайл:

```
# rm /sys/fs/bpf/hello
# bpftool prog show name hello
#
```

Теперь вывод команды `bpftool` отсутствует, так как программа больше не загружена в ядро.

Вызовы к BPF из BPF

В предыдущей главе вы видели хвостовые вызовы в действии, и я упомянула что теперь также есть возможность непосредственно вызывать функции из программы eBPF. Давайте рассмотрим простой пример, который, как и пример хвостового вызова, можно присоединить к точке трассировки `sys_enter`, за исключением того что на этот раз он будет отслеживать код операции для выполняемого системного вызова. Вы найдете код в `Chapter3/hello-func.bpf.c`.

Для иллюстративных целей я написала очень простую функцию, которая извлекает код операции системного вызова из аргументов в точке трассировки:

```
static __attribute((noinline)) int get_opcode(struct bpf_raw_tracepoint_args *ctx) {
   return ctx->args[1];
}
```

Имея выбор, компилятор, вероятно, встроил бы (inline) эту очень простую функцию, которую я буду вызывать всего только из одного места. Так как это противоречит замыслу данного примера, я добавила `__attribute((noinline))` для принудительного указания компилятору. В нормальных обстоятельствах вы, вероятно, должны были бы опустить это и позволить компилятору оптимизировать код по своему усмотрению.

Функция eBPF, вызывающая эту функцию, выглядит подобно следующему:

```
SEC("raw_tp")
int hello(struct bpf_raw_tracepoint_args *ctx) {
   int opcode = get_opcode(ctx);
   bpf_printk("Syscall: %d", opcode);
   return 0;
}
```

После компиляции этого кода в объектный файл eBPF вы можете загрузить его в ядро и проверить что он загружен с помощью `bpftool`:

```
# bpftool prog load hello-func.bpf.o /sys/fs/bpf/hello
# bpftool prog list name hello
893: raw_tracepoint name hello tag 3d9eb0c23d4ab186 gpl
        loaded_at 2023-01-05T18:57:31+0000 uid 0
        xlated 80B jited 208B
        btf_id 302
```

Интересная часть этого эксперимента — проверка байт-кода eBPF чтобы увидеть функцию `get_opcode()`:

```
# bpftool prog dump xlated name hello
int hello(struct bpf_raw_tracepoint_args * ctx):
; int opcode = get_opcode(ctx);
   0: (85) call pc+7#bpf_prog_cbacc90865b1b9a5_get_opcode
; bpf_printk("Syscall: %d", opcode);
   1: (18) r1 = map[id:193][0]+0
   3: (b7) r2 = 12
   4: (bf) r3 = r0
   5: (85) call bpf_trace_printk#-73584
; return 0;
   6: (b7) r0 = 0
   7: (95) exit
int get_opcode(struct bpf_raw_tracepoint_args * ctx):
; return ctx->args[1];
   8: (79) r0 = *(u64 *)(r1 +8)
; return ctx->args[1];
   9: (95) exit
```

1). Здесь вы можете увидеть, как программа `hello()` eBPF делает вызов `get_opcode()`. Инструкция eBPF по смещению 0 имеет значение 0x85, что в документации по набору инструкций соответствует «вызову функции». Вместо выполнения следующей инструкции, которая была бы по смещению 1, выполнение перескочит на семь инструкций вперед (pc+7), что означает инструкцию по смещению 8.

2). Ниже байт-код для `get_opcode()`: и, как вы и могли надеяться, первая инструкция его и находится по смещению 8.

Инструкция вызова функции требует помещения текущего состояния в стек виртуальной машины eBPF, чтобы при завершении вызываемой функции выполнение могло продолжаться в вызывающей функции. Поскольку размер стека ограничен 512 байтами, вызовы к BPF из BPF не могут быть очень глубоко вложены.

Для более подробной информации о хвостовых вызовах и вызовах BPF к BPF есть отличный пост Якуба Ситницки в блоге Cloudflare: «Assembly within! BPF tail calls on x86 and ARM» (10.10.2022 https://blog.cloudflare.com/assembly-within-bpf-tail-calls-on-x86-and-arm/).

Резюме

В этой главе вы видели пример исходного кода C, преобразованного в байт-код eBPF, а затем скомпилированного в машинный код, чтобы он был готов к выполнению в ядре. Вы также узнали, как использовать утилиту `bpftool` для проверки программ и карт загруженных в ядро, и для подключения к событиям XDP.

Помимо этого, вы видели примеры различных типов программ eBPF, запускаемых разными типами событий. Событие вида `XDP` запускается при поступлении пакета данных на сетевой интерфейс, тогда как события видов `kprobe` и `tracepoint` запускаются при попадании в определенную точку внутри кода ядра. Я расскажу о некоторых других типах программ eBPF в главе 7.

Вы также узнали как карты используются для реализации глобальных переменных для программ eBPF, и увидели вызовы функций BPF из программ BPF.

Следующая глава посвящена другому уровню детализации, поскольку я покажу вам, что происходит на уровне системных вызовов, когда `bpftool` — или любой другой код пользовательского пространства — загружает программы и прикрепляет их к событиям.

Упражнения

Вот несколько вещей, которые стоит попробовать, если вы хотите глубже изучить программы BPF:

1. Попробуйте использовать команды `ip link`, подобные приведенным ниже, для подключения и отключения программы XDP:

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

2. Запустите любой из примеров BCC из главы 2. Во время работы программы используйте второе окно терминала для проверки загруженной программы с помощью `bpftool`. Вот пример того, что я увидела, запустив пример `hello-map.py`:

   ```
   # bpftool prog show name hello
   197: kprobe name hello tag ba73a317e9480a37 gpl
           loaded_at 2022-08-22T08:46:22+0000 uid 0
           xlated 296B jited 328B memlock 4096B map_ids 65
           btf_id 179
           pids hello-map.py(2785)
   ```

3. Запустите `hello-tail.py` из каталога Chapter2 и, пока он работает, посмотрите на загруженные им программы. Вы увидите, что каждая программа хвостового вызова указана отдельно, например:

   ```
   # bpftool prog list
   ...
   ```

```
120: raw_tracepoint name hello tag b6bfd0e76e7f9aac gpl
        loaded_at 2023-01-05T14:35:32+0000 uid 0
        xlated 160B jited 272B memlock 4096B map_ids 29
        btf_id 124
        pids hello-tail.py(3590)
121: raw_tracepoint name ignore_opcode tag a04f5eef06a7f555 gpl
        loaded_at 2023-01-05T14:35:32+0000 uid 0
        xlated 16B jited 72B memlock 4096B
        btf_id 124
        pids hello-tail.py(3590)
122: raw_tracepoint name hello_exec tag 931f578bd09da154 gpl
        loaded_at 2023-01-05T14:35:32+0000 uid 0
        xlated 112B jited 168B memlock 4096B
        btf_id 124
        pids hello-tail.py(3590)
123: raw_tracepoint name hello_timer tag 6c3378ebb7d3a617 gpl
        loaded_at 2023-01-05T14:35:32+0000 uid 0
        xlated 336B jited 356B memlock 4096B
        btf_id 124
        pids hello-tail.py(3590)
```

Вы также можете использовать `bpftool prog dump xlated`, чтобы просмотреть инструкции байт-кода и сравнить их с тем, что вы видели в разделе «Вызовы BPF из BPF».

4. *Будьте осторожны с этим, так как может быть лучше просто подумать о том, почему это происходит, а не пытаться!* Если вы возвращаете значение 0 из программы XDP, это соответствует `XDP_ABORTED`, который сообщает ядру о прекращении дальнейшей обработки этого пакета. Это может показаться немного нелогичным, учитывая, что значение 0 обычно указывает на успех в C, но так оно и есть. Итак, если вы попытаетесь изменить программу так, чтобы она возвращала 0, и подключить ее к интерфейсу `eth0` виртуальной машины, все сетевые пакеты будут отброшены. Это будет несколько неприятно, если вы используете SSH для подключения к этой машине, и вам, вероятно, придется перезагрузить машину, чтобы восстановить доступ! Вы можете запустить программу в контейнере, чтобы программа XDP была подключена к виртуальному интерфейсу Ethernet, который влияет только на этот контейнер, а не на всю виртуальную машину. Пример этого есть на https://github.com/lizrice/lb-from-scratch.