Часть 4 : Системный вызов bpf()

Как вы видели в главе 1, когда приложения пользовательского пространства хотят, чтобы ядро сделало что-то от их имени, они делают запросы, используя API системных вызовов. Поэтому имеет смысл предположить что если приложение пользовательского пространства хочет загрузить программу eBPF в ядро, то должны быть задействованы некоторые системные вызовы. На самом деле существует системный вызов `bpf()`, и в этой главе я покажу вам, как он используется для загрузки и взаимодействия с eBPF, программами и картами.

Стоит отметить, что сам код eBPF, работающий в ядре, не использует системные вызовы для доступа к картам. Интерфейс системного вызова используется только приложениями пользовательского пространства. Вместо этого программы eBPF используют некоторые функции-помощники для чтения и записи карт; вы уже видели примеры этого в предыдущих двух главах.

Если вы продолжите писать программы eBPF самостоятельно, то есть большая вероятность, что вы не будете сами напрямую вызывать эти системные вызовы `bpf()`. Есть библиотеки, о которых я расскажу позже в книге, и которые предлагают абстракции более высокого уровня для упрощения работы. Тем не менее, эти абстракции обычно напрямую сопоставляются с базовыми командами системных вызовов, которые вы увидите в этой главе. Какую бы библиотеку вы ни использовали, вам потребуется понимание основных операций — загрузки программы, создания карт, доступа к ним и так далее — которые вы увидите в этой главе.

Прежде чем я покажу вам примеры системных вызовов `bpf()`, давайте посмотрим, что говорится на странице руководства (https://man7.org/linux/man-pages/man2/bpf.2.html) по `bpf()`, а именно, что `bpf()` используется для «выполнения команд расширенного BPF над картами или программами». Там также говорится, что сигнатура `bpf()` выглядит следующим образом:

```
int bpf(int cmd, union bpf_attr *attr, unsigned int size);
```

Первый аргумент `bpf()`, `cmd` указывает какую команду выполнять. Системный вызов `bpf()` не просто делает какую-то одну вещь — существует множество различных команд, которые можно использовать для управления программами и картами eBPF. На рис. 4-1 показан обзор некоторых распространенных команд, которые код пользовательского пространства может использовать для загрузки программ eBPF, создания карт, присоединения программ к событиям и доступа к парам ключ-значение в карте.

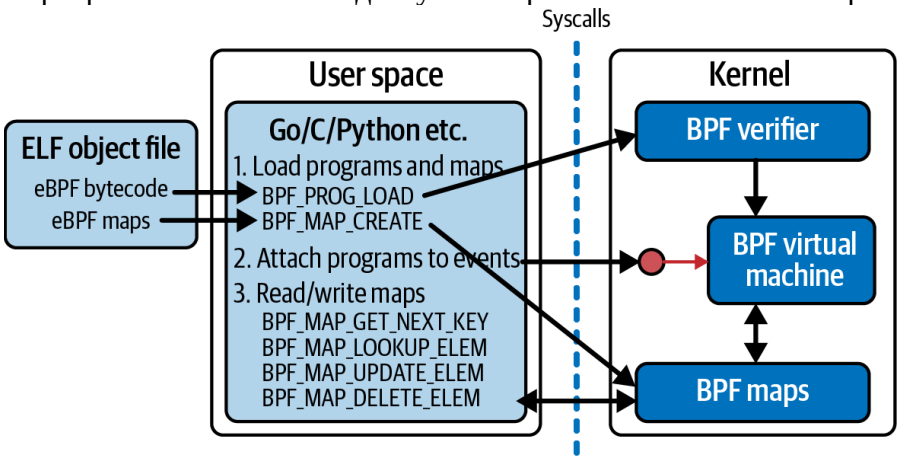

Рисунок 4-1. Программа пользовательского пространства взаимодействует с программами eBPF и картами в ядре с помощью системных вызовов.

Аргумент `attr` системного вызова `bpf()` содержит указатель на все данные, необходимые для параметров команды, а size указывает, сколько байтов данных находится по `attr*`.

Вы уже встречались с командой `strace` в главе 1, когда я использовала ее чтобы показать как код пользовательского пространства выполняет множество запросов через API системных вызовов. В этой главе я буду использовать её, чтобы продемонстрировать, как используется системный вызов `bpf()`. Вывод `strace` включает в себя аргументы каждого системного вызова, но, чтобы не загромождать пример выводом в этой главе, я опускаю многие подробности из аргументов `attr`, если только они не особо интересные.

> Вы найдете код, вместе с инструкциями по настройке среды для его запуска, на http://github.com/lizrice/learning-ebpf. Код для этой главы находится в каталоге `Chapter4`.

В этом примере я собираюсь использовать BCC-программу `hello-buffer-config.py`, основанную на примерах, которые вы уже видели в главе 2. Как и пример `hello-buffer.py`, эта программа отправляет сообщение в буфер Perf всякий раз когда она запускается, передавая информацию из ядра в пространство пользователя о событии системного вызова `execve()`. Что нового в этой версии, так это то, что она позволяет настраивать разные сообщения для каждого идентификатора пользователя.

Вот её eBPF исходный код:

```
struct user_msg_t {
   char message[12];
};

BPF_HASH(config, u32, struct user_msg_t);

BPF_PERF_OUTPUT(output);

struct data_t {
   int pid;
   int uid;
   char command[16];
   char message[12];
};

int hello(void *ctx) {
   struct data_t data = {};
   struct user_msg_t *p;
   char message[12] = "Hello World";

   data.pid = bpf_get_current_pid_tgid() >> 32;
   data.uid = bpf_get_current_uid_gid() & 0xFFFFFFFF;

   bpf_get_current_comm(&data.command, sizeof(data.command));

   p = config.lookup(&data.uid);
```

```
    if (p != 0) {
        bpf_probe_read_kernel(&data.message, sizeof(data.message), p->message);
    } else {
        bpf_probe_read_kernel(&data.message, sizeof(data.message), message);
    }

    output.perf_submit(ctx, &data, sizeof(data));
    return 0;
}
```

1). В начале вводится определение структуры `user_msg_t` для хранения 12-символьного сообщения.

2). Макрос BCC `BPF_HASH` используется для определения хеш-таблицы, называемой `config`. Она будет содержать значения типа `user_msg_t`, проиндексированное ключами типа `u32`, что является соответствующим размером для идентификатора пользователя. (Если вы не укажете типы для ключей и значений, то BCC по умолчанию будет использовать значения типа `u64` для обоих.)

3). Вывод буфера Perf (`BPF_PERF_OUTPUT`) определяется точно так же, как в главе 2. Вы можете отправлять произвольные данные в буфер, поэтому здесь нет необходимости указывать какие-либо типы данных…

4). ... хотя на практике в этом примере программа всегда отправляет структуру `data_t`. Здесь также ничего не изменилось по сравнению с примером из главы 2.

5). Большая часть остальной части программы eBPF не отличается от версии `hello()`, которую вы видели ранее.

6). Единственное отличие состоит в том, что, используя функцию-помощник для получения идентификатора пользователя, код ищет запись в хэш-таблице конфигурации с этим идентификатором пользователя в качестве ключа. Если есть совпадающая запись, значение содержит сообщение, которое используется вместо умалчиваемого «Hello World».

Код Python будет иметь две дополнительных строки:

```
b["config"][ct.c_int(0)] = ct.create_string_buffer(b"Hey root!")
b["config"][ct.c_int(501)] = ct.create_string_buffer(b"Hi user 501!")
```

Они определяют записи (и сообщения) в хэш-таблице конфигурации для идентификаторов пользователей 0 и 501, которые соответствуют пользователю `root` и моему собственному идентификатору пользователя на этой виртуальной машине. В этом коде используется пакет Python `ctypes`[1], чтобы гарантировать, что ключ и значение имеют в точности те же типы, что и используемые в определении C для `user_msg_t`.

Вот некоторый иллюстративный вывод из этого примера вместе с командами, которые я выполняла во втором терминале, чтобы получить этот вывод:

```
Terminal 1                          Terminal 2
# ./hello-buffer-config.py
37926 501 bash Hi user 501!         ls
37927 501 bash Hi user 501!         sudo ls
```

1 В коде: `import ctypes as ct` (Прим. пер.)

```
37929 0 sudo Hey root!
37931 501 bash Hi user 501!            sudo -u daemon ls
37933 1 sudo Hello World
```

Теперь, когда вы получили представление о том, что делает эта программа, я хотела бы показать вам системные вызовы `bpf()`, которые используются при её выполнении. Я запущу его снова, но используяутилиту `strace`, указав опцию `-e bpf`, чтобы обозначить что меня интересует только один системный вызов `bpf()`:

```
# strace -e bpf ./hello-buffer-config.py
```

Вывод, который вы увидите, если попробуете сделать это самостоятельно, показывает несколько вызовов этого системного вызова[2]. Для каждого из них вы увидите команду, указывающую что должен делать системный вызов `bpf()`. Общая схема выглядит так:

```
bpf(BPF_BTF_LOAD, ...) = 3
bpf(BPF_MAP_CREATE, {map_type=BPF_MAP_TYPE_PERF_EVENT_ARRAY...) = 4
bpf(BPF_MAP_CREATE, {map_type=BPF_MAP_TYPE_HASH...) = 5
bpf(BPF_PROG_LOAD, {prog_type=BPF_PROG_TYPE_KPROBE,...prog_name="hello",...) = 6
bpf(BPF_MAP_UPDATE_ELEM, ...}
...
```

Давайте рассмотрим их последовательно один за другим. Ни у вас, читатель, ни у меня нет бесконечного терпения, поэтому я не буду обсуждать каждый вывод до каждой детали! Я сосредоточусь на тех частях, которые, как мне кажется, действительно помогают рассказать историю о том что происходит когда программа пользовательского пространства взаимодействует с программой eBPF.

Загрузка BTF данных

Первый системный вызов `bpf()` который я вижу выглядит так:

```
bpf(BPF_BTF_LOAD, {btf="\237\353\1\0...}, 128) = 3
```

В этом вызове команда, которую вы видите в выводе, — `BPF_BTF_LOAD`. Это всего лишь одна из множества допустимых команд, из числа (по крайней мере, на момент написания этой статьи) наиболее полно задокументированых в исходном коде ядра[3].

Возможно, вы и не увидите вызов этой команды если вы используете относительно старое ядро Linux, поскольку оно связано с BTF или форматом типа BPF[4]. BTF позволяет программам eBPF быть переносимыми между различными версиями ядра, чтобы вы могли скомпилировать программу на одной машине а использовать ее на другой, которая может использовать совсем другую версию ядра и, следовательно, иметь различные структуры данных ядра. Я расскажу об этом более подробно в главе 5.

---

2 … и результатов их выполнения (кодов возвратов). (Прим. пер.)

3 Если вы хотите увидеть полный набор команд BPF, то они задокументированы в заголовочном файле кодов ядра `linux/bpf.h`.

4 BTF был представлен выше в ядре 5.1.0, но, как вы можете видеть из этого обсуждения, он был внедрён в некоторые дистрибутивы Linux ешё раньше этого: https://lists.iovisor.org/g/iovisor-dev/topic/which_is_oldest_linux_kernel/80980657?p=,,,20,0,0,0::recentpostdate%2Fsticky,,,20,0,0,80980657 .

Этот вызов `bpf()` загружает блок данных BTF в ядро, а код возврата из системного вызова `bpf()` (3[5] в моем примере) представляет собой номер файлового дескриптора, который ссылается на эти данные.

> Дескриптор файла — это числовой идентификатор открытого файла (или файлового объекта). Если вы открываете файл (системным вызовом `open()` или `openat()`), код возврата представляет собой дескриптор файла (число), который затем передается в качестве аргумента другим системным вызовам, таким как `read()` или `write()`, для выполнения операций над этим файлом. Здесь блок данных вовсе не совсем файл, но ему сопоставляется файловый дескриптор в качестве идентификатора, который можно будет использовать для будущих операций, которые к нему (блоку данных) относятся.

Создание карт

Следующий вызов `bpf()` создаёт карту `output` в буфере Perf:

```
bpf(BPF_MAP_CREATE, {map_type=BPF_MAP_TYPE_PERF_EVENT_ARRAY, , key_size=4,
value_size=4, max_entries=4, ... map_name="output", ...}, 128) = 4
```

Наверное, вы можете догадаться, по имени кода команды `BPF_MAP_CREATE`, что этот вызов создает карту eBPF. Вы можете видеть, что тип этой карты — `PERF_EVENT_ARRAY`, и она называется выходной. Ключи и значения в этой карте имеют длину 4 байта. Также устанавливается ограничение в четыре пары ключ-значение, которые могут храниться на этой карте, определяемое полем `max_entries`. Позже в этой главе я объясню, почему в этой карте будет четыре записи. Возвращаемое значение 4 — это файловый дескриптор кода пользовательского пространства для доступа к `output` карте.

Следующий системный вызов `bpf()` в выходных данных создает карту `config`:

```
bpf(BPF_MAP_CREATE, {map_type=BPF_MAP_TYPE_HASH, key_size=4, value_size=12,
max_entries=10240... map_name="config", ...btf_fd=3,...}, 128) = 5
```

Эта карта определяется как хеш-таблица с ключами длиной 4 байта (что соответствует 32-разрядному целому числу которое может использоваться для хранения идентификатора пользователя UID) и значениями длиной 12 байтов (что соответствует размеру структуру `msg_t`). Я не указываю здесь размер карты, поскольку для нее был задан размер BCC по умолчанию, равный 10240 элементов.

Этот системный вызов `bpf()` также возвращает дескриптор файла 5, который будет использоваться для ссылки на эту карту конфигурации в последующих системных вызовах.

Вы также можете увидеть здесь поле `btf_fd=3`, которое указывает ядру использовать дескриптор файла BTF 3, который был получен ранее. Как вы увидите в главе 5, информация BTF описывает компоновку структур данных, и включение ее в определение карты означает наличие информации о структуре ключей и типов значений, используемых в этой карте. Это используется такими инструментами, как `bpftool`, для красивой печати дампов карт, что делает их удобочитаемыми — вы видели пример этого в главе 3.

---

5 Здесь вспоминаем, что в Linux первые 3 файловых дескиптора: 0, 1, 2 — зарезервированы по умолчанию (и заняты) в каждом процессе пользовательского пространства за STDIN, STDOUT и STDERR, соответственно. (Прим. пер.)

Загрузка программы

Выше вы видели пример программного кода, использующего системные вызовы для загрузки данных BTF в ядро и создания некоторых карт eBPF. Следующее, что он делает, это загружает eBPF загружаемую программу в ядро с помощью следующего системного вызова `bpf()`:

```
bpf(BPF_PROG_LOAD, {prog_type=BPF_PROG_TYPE_KPROBE, insn_cnt=44,
insns=0xffffa836abe8, license="GPL", ... prog_name="hello", ...
expected_attach_type=BPF_CGROUP_INET_INGRESS, prog_btf_fd=3,...}, 128) = 6
```

Здесь много довольно интересных полей:

- Поле `prog_type` описывает тип программы, которое здесь указывает, что она предназначена для подключения к `kprobe`. Вы узнаете больше о типах программ в главе 7.
- Поле `insn_cnt` означает «количество инструкций». Это число инструкций байт-кода в программе.
- Инструкции байт-кода, составляющие эту программу eBPF, хранятся в памяти по адресу, указанному в поле `insns`.
- Эта программа была указана как лицензированная под лицензией GPL, поэтому она может использовать функции-помощники BPF требующие лицензии GPL.
- Имя программы — `hello`.
- `expected_attach_type` — `BPF_CGROUP_INET_INGRESS` может показаться неожиданным, поскольку это похоже на входящий сетевой трафик, но вы же знаете, что эта программа eBPF будет присоединена к `kprobe`. На самом деле поле `expect_attach_type` используется только для некоторых типов программ, и `BPF_PROG_TYPE_KPROBE` не входит в их число. BPF_CGROUP_INET_INGRESS оказывается первым в списке типов вложений BPF[6], почему он имеет значение 0.
- Поле `prog_btf_fd` сообщает ядру какой блок ранее загруженных данных BTF использовать с этой программой. Значение 3 здесь соответствует дескриптору файла, который вы видели возвращенным из системного вызова с командой `BPF_BTF_LOAD` (и это тот же блок данных BTF, который используется и для карты `config`).

Если бы эта программа не прошла верификацию (о чем я расскажу в главе 6), этот системный вызов вернул бы отрицательное значение, но здесь вы можете видеть что он вернул дескриптор файла 6. Напомним, что на этот момент все дескрипторы файлов имеют значения как показано в таблице 4-1.

---

6 Они определены в `bpf_attach_type` перечислении в `linux/bpf.h`.

Файловый дескриптор	Что представляет
3	BTF данные
4	`output` карта в Perf буфере
5	`config` карта хеш-таблицы
6	EBPF программа `hello`

Изменение карты из пространства пользователя

Вы уже видели строки в исходном коде Python пользовательского пространства, которые настраивают условно специальные сообщения, которые будут отображаться для пользователя `root` с идентификатором пользователя 0 и для пользователя с идентификатором 501:

```
b["config"][ct.c_int(0)] = ct.create_string_buffer(b"Hey root!")
b["config"][ct.c_int(501)] = ct.create_string_buffer(b"Hi user 501!")
```

Вы можете увидеть (листинг выполнения выше), как эти записи были помещены в карту с помощью системных вызовов такого вида:

```
bpf(BPF_MAP_UPDATE_ELEM, {map_fd=5, key=0xffffa7842490, value=0xffffa7a2b410,
flags=BPF_ANY}, 128) = 0
```

Команда `BPF_MAP_UPDATE_ELEM` обновляет пару ключ-значение в карте. Флаг `BPF_ANY` указывает, что если ключ еще не существует в этой карте, то его следует создать. В данном случае есть два таких вызова, соответствующих двум записям, настроенным для двух разных идентификаторов пользователей.

Поле `map_fd` определяет, с какой картой выполняется операция. Вы можете видеть, что в данном случае это 5, значение файлового дескриптора, возвращенное ранее при создании `config` карты.

Дескрипторы файлов назначаются ядром для конкретного процесса, поэтому значение 5 допустимо только для этого конкретного процесса пространства пользователя, в котором выполняется программа Python. Однако несколько программ пользовательского пространства (и несколько программ eBPF в ядре) могут обращаться к одной и той же карте. Двум программам пользовательского пространства, которые обращаются к одной и той же структуре отображения в ядре, вполне могут быть назначены разные значения файловых дескрипторов; точно так же две программы пользовательского пространства могут иметь одно и то же значение файлового дескриптора для совершенно разных карт.

Оба, и ключ, и значение являются указателями, поэтому вы не можете определить числовое значение ни ключа, ни значения из этого вывода утилиты `strace`. Однако вы можете использовать утилиту `bpftool` для просмотра содержимого карты и увидеть что-то вроде этого:

```
# bpftool map dump name config
[{
        "key": 0,
        "value": {
            "message": "Hey root!"
        }
```

```
    },{
        "key": 501,
        "value": {
            "message": "Hi user 501!"
        }
    }
]
```

Откуда `bpftool` знает, как форматировать этот вывод? Например, как он узнает, что значение является структурой с полем, называемым сообщением, и которое содержит строку? Ответ заключается в том, что он использует определения из информации BTF, включенной в системный вызов `BPF_MAP_CREATE`, который и определил эту карту. Подробнее о том, как BTF передает эту информацию, вы узнаете в следующей главе.

Теперь вы увидели, как пространство пользователя взаимодействует с ядром для загрузки программ и карт и для обновления информации в карте. В последовательности системных вызовов, которую вы видели до сих пор, программа еще не была привязана к событию. Этот шаг должен произойти; в противном случае программа никогда не будет активирована.

Честное предупреждение: разные типы программ eBPF привязываются к разным событиям разными способами! Позже в этой главе я покажу вам системные вызовы, используемые в этом примере для подключения к событию `kprobe`, и в этом случае вызовы `bpf()` не используется. Напротив, в упражнениях в конце этой главы я покажу вам другой пример, в котором системный вызов `bpf()` используется для присоединения программы к необработанному событию точки трассировки.

Прежде чем мы перейдем к этим деталям, я хотела бы обсудить что происходит когда вы прекращаете работу с программой. Вы обнаружите, что программа и карты автоматически выгружаются, и это происходит потому, что ядро отслеживает их, используя счетчики ссылок.

Ссылки BPF программ и карт

Вы уже знаете, что загрузка программы BPF в ядро с помощью системного вызова `bpf()` возвращает файловый дескриптор. Внутри ядра этот файловый дескриптор является *ссылкой* на программу. Процесс пользовательского пространства, выполнивший системный вызов, владеет этим файловым дескриптором; когда этот процесс завершается, дескриптор файла освобождается, а счетчик ссылок на программу уменьшается. Когда не остаётся ссылок на программу BPF, ядро удаляет эту программу.

Дополнительная ссылка создается, когда вы прикрепляете программу к файловой системе.

Закрепление

Вы уже видели закрепление в действии в главе 3 производимое с помощью следующей команды:

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

> Такие закрепленные объекты не являются реальными файлами сохраненными на диске. Они создаются в псевдофайловой системе `sysfs`, которая ведет себя подобно обычной дисковой файловой системе с её каталогами и файлами. Но здесь они хранятся в памяти, а, значит, не останутся на месте при перезагрузке системы.

Если бы утилита `bpftool` позволяла вам загружать программу не закрепляя ее, это было бы бессмысленно, потому что дескриптор файла освобождается при выходе из `bpftool`, и если нет более ссылок на программу, то программа будет удалена, так что ничего полезного не будет достигнуто. Но закрепление её (именем) в файловой системе означает наличие дополнительной ссылки на программу, поэтому программа остается загруженной и после завершения выполнения команды.

Счетчик ссылок также увеличивается когда программа BPF присоединяется к ловушке которая её запускает. Само поведение этих счетчиков ссылок зависит от типа программы BPF. Вы узнаете больше об этих типах программ в главе 7, но есть и такие, которые относятся к трассировке (например, `kprobes` и `tracepoints`) и всегда связаны с процессом пользовательского пространства; для этих типов программ eBPF счетчик ссылок ядра уменьшается при завершении этого процесса. Программы, подключенные к сетевому стеку или `cgroups` (сокращение от «контрольные группы»), не связаны ни с одним процессом пользовательского пространства, поэтому они остаются на месте даже после завершения работы программы пользовательского пространства, которая их загружает. Вы уже видели пример этого при загрузке XDP программ с помощью команды `ip link`:

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

Команда `ip` выполнена, а определения закреплённой локации нет, но тем не менее, `bpftool` покажет вам, что в ядре загружена программа XDP:

```
# bpftool prog list
...
1255: xdp name hello tag 9d0e949f89f1a82c gpl
        loaded_at 2022-11-01T19:21:14+0000 uid 0
        xlated 48B jited 108B memlock 4096B map_ids 612
```

Счетчик ссылок для этой программы отличен от нуля из-за привязки к перехватчику XDP, который сохраняется после выполнения команды `ip link`.

Карты eBPF также имеют счетчики ссылок, и они уничтожаются когда их счетчик ссылок падает до нуля. Каждая программа eBPF, использующая карту, увеличивает счетчик, как и каждый файловый дескриптор, который программы пользовательского пространства могут хранить в карте.

Вполне возможно, что исходный код программы eBPF может определять карту, на которую программа фактически не ссылается. Предположим, вы хотите сохранить некоторые метаданные о программе; вы можете определить ее как глобальную переменную, и, как вы видели в предыдущей главе, эта информация сохраняется в карте. Если программа eBPF ничего не делает с этой картой, то автоматически не будет увеличен счетчик ссылок из программы на карту. Существует системный вызов `bpf(BPF_PROG_BIND_MAP)`, который связывает карту с программой, так чтобы карта не очищалась, как только программа загрузчика пользовательского пространства завершает работу и больше не содержит ссылку дескриптора файла на карту.

Также и карты можно закреплять в файловой системе, и программы пользовательского пространства могут получить доступ к карте зная путь к закреплённой карте.

> Алексей Старовойтов написал хорошее описание счетчиков ссылок и файловых дескрипторов BPF в своем блоге «Время жизни объектов BPF»: https://facebookmicrosites.github.io/bpf/blog/2018/08/31/object-lifetime.html

Другой способ создать ссылку на программу BPF — это использовать BPF связь[7].

BPF связи

Связи BPF обеспечивают уровень абстракции между программой eBPF и событием к которому она привязана. Сама связь BPF может быть закреплена в файловой системе, что создает дополнительную ссылку на программу. Это означает, что процесс пользовательского пространства, который загрузил программу в ядро, при этом может завершиться, оставив программу BPF загруженной. Дескриптор файла в программе загрузки пользовательского пространства освобождается, уменьшая количество ссылок на программу, но счетчик ссылок будет оставаться не нулевым из-за связи BPF.

Вы получите возможность увидеть связей BPF в действии, если выполните упражнения в конце этой главы. А пока давайте вернемся к последовательности системных вызовов `bpf()` используемых в `hello-buffer-config.py`.

Дополнительные системные вызовы, задействованные в eBPF

Напомним, что до сих пор вы видели системные вызовы `bpf()`, которые добавляют данные BTF, программу и карты, а также данные карты в ядро. Следующее, что показывает вывод утилиты `strace`, относится к настройке буфера Perf.

> В оставшейся части этой главы мы относительно глубоко погружаемся в последовательности системных вызовов, возникающие при использовании буферов `Perf`, кольцевых буферов BPF, `kprobes` и итераций карт. Не все программы eBPF должны делать это, поэтому, если вы спешите или считаете, что это слишком подробно, не стесняйтесь переходить к краткому изложению главы в её конце. Я не обижусь!

Инициализация буфера Perf

Вы уже видели вызовы `bpf(BPF_MAP_UPDATE_ELEM)`, которые добавляют записи (элементы) в `config` карту. Далее в выводе были показаны несколько вызовов, которые выглядят подобным образом:

```
bpf(BPF_MAP_UPDATE_ELEM, {map_fd=4, key=0xffffa7842490, value=0xffffa7a2b410,
flags=BPF_ANY}, 128) = 0
```

Они очень похожи на вызовы, определяющие записи `config` карты, за исключением того, что в этом случае файловый дескриптор карты равен 4, что представляет карту `output` буфера Perf.

Как и прежде, ключ и значение являются указателями, поэтому вы не сможете определить числовые значения ключа или значения из этого вывода утилиты `strace`. Я вижу, что такой системный вызов повторяется четыре раза с одинаковыми значениями для всех параметров, хотя нет никакого способа узнать изменились ли значения, содержащиеся по указателям, в промежутках между каждым вызовом. Глядя на эти вызовы `bpf()` с кодом команды `BPF_MAP_UPDATE_ELEM`, мы остаёмся без ответа на некоторые вопросы о том, как настраивается и используется буфер:

---

7 Здесь проявляется неоднозначность русскоязычной терминологии: и reference и link — и то и другое это «ссылка». Для сохранения смысла текста мы будем link упоминать как «связь». (Прим. пер.)

- Почему четыре вызова `BPF_MAP_UPDATE_ELEM`? Связано ли это как-то с тем, что `output` карта создавалась максимум с четырьмя записями?

- После этих четырех экземпляров `BPF_MAP_UPDATE_ELEM` в выводе `strace` больше вообще не появляются системные вызовы `bpf()`. Это может показаться немного странным, потому что карта и предназначена для того, чтобы программа eBPF могла записывать данные каждый раз когда она активируется, и вы видели что данные действительно отображаются кодом пользовательского пространства. Эти данные явно не извлекаются из карты с помощью системных вызовов `bpf()`, так как же они получены?

Вы также еще не видели никаких доказательств того, как программа eBPF привязывается к событию `kprobe`, которое ее активирует. Чтобы получить объяснение всех этих проблем, мне нужно, чтобы утилита `strace` показала ещё несколько других системных вызовов при запуске этого примера, например так:

```
# strace -e bpf,perf_event_open,ioctl,ppoll ./hello-buffer-config.py
```

Для краткости я буду игнорировать те вызовы `ioctl()`, которые конкретно не связаны с функциональностью eBPF в этом примере.

Присоединение к событиям kprobe

Вы видели, что файловый дескриптор 6 был назначен для представления приветствия программы eBPF после ее загрузки в ядро. Чтобы прикрепить программу eBPF к событию, вам также потребуется файловый дескриптор, представляющий это конкретное событие. Следующая строка из вывода strace показывает создание дескриптора файла для execve() kprobe:

```
perf_event_open({type=0x6 /* PERF_TYPE_??? */, ...},...) = 7
```

Согласно (https://man7.org/linux/man-pages/man2/perf_event_open.2.html) справочной странице системного вызова `perf_event_open()`[8], он «создает дескриптор файла, который позволяет измерять информацию о производительности». Из вывода[9] видно, что `strace` просто не знает, как интерпретировать параметр `type` со значением 6, но если вы внимательно изучите эту справочную страницу, она описывает, как Linux поддерживает динамические типы Performance Measurement Unit[10]:

> *... для каждого экземпляра PMU есть подкаталог в /sys/bus/event_source/devices. В каждом таком подкаталоге есть файл type, содержимое которого представляет собой целое число которое можно использовать в поле типа.*

Совершенно достаточно для того, что если вы заглянете в этот каталог, то вы найдете файл `kprobe/type`:

```
$ cat /sys/bus/event_source/devices/kprobe/type
6
```

Исходя из этого, вы можете увидеть, что вызов `perf_event_open()` имеет `type`, установленный в значение 6, чтобы указать что это и есть тип `kprobe` события `Perf`.

---

8 `man perf_event_open` (Прим. пер.)

9 Трассировки системных вызовов.

10 PMU — единиц измерения производительности.

К сожалению, `strace` не выводит подробностей, которые убедительно показали бы что именно `kprobe` подключен к системному вызову `execve()`, но я надеюсь, что здесь уже достаточно свидетельств, чтобы убедить вас в том что возвращаемый здесь файловый дескриптор именно это и представляет.

Код возврата от `perf_event_open()` равен 7, и он представляет файловый дескриптор для `kprobe` события `Perf`, и вы знаете, что файловый дескриптор 6 представляет программу `hello` eBPF. Страница руководства для `perf_event_open()` также объясняет, как использовать `ioctl()` для создания связи между ними:

> *PERF_EVENT_IOC_SET_BPF [...] позволяет присоединить программу Berkeley Packet Filter (BPF) к существующему событию точки трассировки kprobe. Аргумент представляет собой дескриптор файла программы BPF, созданный предыдущим системным вызовом bpf(2).*

Это и объясняет следующий системный вызов `ioctl()`, который вы увидите в выводе `strace`, с аргументами, относящимися сразу к двум файловым дескрипторам:

```
ioctl(7, PERF_EVENT_IOC_SET_BPF, 6) = 0
```

Там же присутствует еще один вызов `ioctl()`, который включает событие `kprobe`:

```
ioctl(7, PERF_EVENT_IOC_ENABLE, 0) = 0
```

Согласно этому месту, программа eBPF и должна активироваться всякий раз, когда на этой машине выполняется функция `execve()`.

Настройка и чтение событий Perf

Я уже упоминала, что вижу четыре вызова `bpf(BPF_MAP_UPDATE_ELEM)`, связанных с выводом буфера Perf. При трассировке дополнительных системных вызовов вывод `strace` показывает четыре фрагмента подобных вот такому:

```
perf_event_open({type=PERF_TYPE_SOFTWARE, size=0 /* PERF_ATTR_SIZE_??? */,
config=PERF_COUNT_SW_BPF_OUTPUT, ...}, -1, X, -1, PERF_FLAG_FD_CLOEXEC) = Y

ioctl(Y, PERF_EVENT_IOC_ENABLE, 0) = 0

bpf(BPF_MAP_UPDATE_ELEM, {map_fd=4, key=0xffffa7842490, value=0xffffa7a2b410,
flags=BPF_ANY}, 128) = 0
```

Я использовала здксь X, чтобы указать место где в выводимых данные показывают значения 0, 1, 2 и 3, соответственно, в четырех экземплярах этого вызова. Обратившись к справочной странице системного вызова `perf_event_open()`, вы увидите, что это номер CPU[11], а поле перед ним — `pid` или идентификатор процесса. Из той же справочной страницы:

> *pid == -1 и процессор >= 0*
>
> *Это включает в измерения все процессы/потоки на указанном процессоре.*

---

11 Номер ядра — поле с именем CPU в выводе команды: `lscpu -e`, см. https://linux-ru.ru/viewtopic.php?f=18&t=7028

Тот факт, что эта последовательность повторяется четыре раза, соответствует четырем ядрам процессора в моем ноутбуке. Это, наконец, объяснение того, почему в карте с именем `output` буфера Perf четыре записи: по одной для каждого ядра ЦП (см. главу «Создание карт» выше). Это также объясняет «массивную» (как массив, array) составляющую в имени типа карты `BPF_MAP_TYPE_PERF_EVENT_ARRAY`, поскольку карта представляет не просто один кольцевой буфер Perf, а массив буферов — по одному для каждого ядра.

Если вы пишете программы eBPF, вам не нужно беспокоиться о таких деталях как обработка количества ядер, потому как об этом позаботится за вас любая из библиотек eBPF, обсуждающихся в главе 10, но я думаю, что это интересный аспект работы системных вызовов, которые вы видите при использовании утилиты `strace` для этой программы.

Каждый вызов `perf_event_open()` возвращает значение дескриптора файла, который я в выводе выше представила как Y; они имеют значения конкретно 8, 9, 10 и 11. Системные вызовы `ioctl()` разрешают вывод `Perf` для каждого из этих файловых дескрипторов. Системные вызовы `bpf()` с командой `BPF_MAP_UPDATE_ELEM` устанавливают запись карты так, чтобы она указывала на кольцевой буфер Perf для каждого ядра CPU, чтобы указать куда он может отправлять свои данные. Затем код пользовательского пространства может использовать `ppoll()` для всех четырех[12] файловых дескрипторов выходного потока, чтобы он мог получить выходные данные, в зависимости от того, какое ядро запускает программу `hellp` eBPF для любого заданного события `execve()` `kprobe`. Вот системный вызов `ppoll()`:

```
ppoll([{fd=8, events=POLLIN}, {fd=9, events=POLLIN}, {fd=10, events=POLLIN},
{fd=11, events=POLLIN}], 4, NULL, NULL, 0) = 1 ([{fd=8, revents=POLLIN}])
```

Как вы увидите, если попытаетесь запустить пример программы самостоятельно, эти вызовы `ppoll()` блокируются до тех пор, пока не появится хоть что-то для чтения из одного из (любого) этих файловых дескрипторов. Вы не увидите кода возврата, выведенного на экран, до тех пор пока что-то не вызовет `execve()`, что заставит программу eBPF записать данные, которые программа пользовательского пространства извлекает с помощью этого вызова `ppoll()`.

В главе 2 я упомянула, что если у вас стоит ядро версии 5.8 или выше, кольцевые буферы самого BPF теперь предпочтительнее буферов Perf. Давайте взглянем на модифицированную версию того же примера кода, в котором используется кольцевой буфер BPF.

Кольцевые буфера

Как указано в документации ядра сейчас, кольцевые буферы eBPF предпочтительнее буферов Perf отчасти из соображений производительности, но, также, и для обеспечения сохранения порядка данных, даже если данные передаются разными ядрами CPU. Здесь есть только один буфер, общий для всех ядер. Чтобы преобразовать `hello-buffer-config.py` для использования кольцевого буфера eBPF, требуется не так и много изменений. В прилагаемом репозитории GitHub вы найдете этот пример как `Chapter4/hello-ring-buffer-config.py`. В Таблице 4-2 показаны различия.

`hello-buffer-config.py`	`hello-ring-buffer-config.py`
BPF_PERF_OUTPUT(output);	BPF_RINGBUF_OUTPUT(output, 1);

12 Только для данного конкретного случая: по числу CPU в общем виде. (Прим. пер.)

output.perf_submit(ctx, &data, sizeof(data));	output.ringbuf_output(&data, sizeof(data), 0);
b["output"].	b["output"].
open_perf_buffer(print_event)	open_ring_buffer(print_event)
b.perf_buffer_poll()	b.ring_buffer_poll()

Таблица 4-2. Различия между примером кода BCC с использованием буфера Perf и кольцевого буфера eBPF

Как и следовало ожидать, поскольку эти изменения относятся только к `output` буферу, системные вызовы, связанные с загрузкой программы и карты `config`, а также с присоединением программы к событию `kprobe`, все остаются без изменений.

Системный вызов `bpf()`, создающий карту `output` кольцевого буфера eBPF, выглядит следующим образом:

```
bpf(BPF_MAP_CREATE, {map_type=BPF_MAP_TYPE_RINGBUF, key_size=0, value_size=0,
max_entries=4096, ... map_name="output", ...}, 128) = 4
```

Основное отличие в выводе утилиты `strace` заключается в том, что теперь в нем нет следов последовательности из четырех различных системных вызовов `perf_event_open()`, `ioctl()` и `bpf(BPF_MAP_UPDATE_ELEM)`, которые вы наблюдали во время настройки буфера Perf. Для кольцевого буфера eBPF есть только один файловый дескриптор, общий для всех ядер процессора.

На момент написания этого текста BCC использовался механизм `ppoll()`, который я показала ранее для буферов Perf, но используется и более новый механизм `epoll()` для ожидания данных из кольцевого буфера. Давайте воспользуемся этим утверждением как возможностью понять разницу между `ppoll()` и `epoll()`.

В примере с буфером Perf я показала как `hello-buffer-config.py` производит системный вызов `ppoll()`, например:

```
ppoll([{fd=8, events=POLLIN}, {fd=9, events=POLLIN}, {fd=10, events=POLLIN},
{fd=11, events=POLLIN}], 4, NULL, NULL, 0) = 1 ([{fd=8, revents=POLLIN}])
```

Обратите внимание, что здесь передается целый набор файловых дескрипторов: 8, 9, 10 и 11, из которых процесс пользовательского пространства хотел бы получить данные. Каждый раз, когда это событие опроса возвращает какие-то данные, необходимо сделать еще один вызов `ppoll()`, чтобы снова настроить всё это на один и тот же набор файловых дескрипторов. При использовании `epoll()` набор файловых дескрипторов управляется в объекте ядра. Вы можете увидеть это в следующей последовательности связанных с `epoll()` системных вызовов, когда `hello-ring-buffer-config.py` настраивает доступ к выходному кольцевому буферу. Во-первых, программа пользовательского пространства запрашивает создание нового экземпляра для вызова `epoll()` в ядре:

```
epoll_create1(EPOLL_CLOEXEC) = 8
```

Это возвращает файловый дескриптор 8. Затем выполняется вызов `epoll_ctl()`, который указывает ядру добавить файловый дескриптор 4 (`output` буфер) к набору файловых дескрипторов в этом экземпляре `epoll`:

```
epoll_ctl(8, EPOLL_CTL_ADD, 4, {events=EPOLLIN, data={u32=0, u64=0}}) = 0
```

Программа пользовательского пространства использует `epoll_pwait()` для ожидания пока данные не станут доступны в кольцевом буфере. Этот вызов возвращается только тогда, когда данные станут доступными:

```
epoll_pwait(8, [{events=EPOLLIN, data={u32=0, u64=0}}], 1, -1, NULL, 8) = 1
```

Естественно, если вы пишете код с использованием такой среды, как BCC (или `libbpf`, или любой другой из библиотек, которые я опишу позже в этой книге), вам действительно не нужно знать эти базовые подробности о том, как ваше приложение в пользовательском пространстве получает информацию от ядра, через Perf или кольцевые буферы eBPF. Надеюсь, вам было интересно заглянуть под крышку, чтобы увидеть, как все это работает.

Однако вы вполне можете столкнуться с тем, что пишется код, который обращается к карте из пользовательского пространства, и может быть полезно увидеть пример того, как это достигается. Ранее в этой главе я использовал утилиту `bpftool` для изучения содержимого карты `config`. Поскольку это утилита, работающая в пользовательском пространстве, давайте воспользуемся `strace`, чтобы посмотреть, какие системные вызовы она выполняет для получения этой информации.

Чтение информации из карты

Следующая команда покажет выдержку из системных вызовов `bpf()`, которые выполняет утилита `bpftool` при чтении содержимого карты `config`:

```
# strace -e bpf bpftool map dump name config
```

Как вы увидите, последовательность состоит из двух основных шагов:

- Перебрать все карты в поисках любой из них именем `config`.
- Если найдена соответствующая карта, то перебрать все элементы этой карты.

Поиск карты

Вывод начинается с повторяющейся последовательности похожих вызовов, так как `bpftool` просматривает все карты в поисках любой с именем `config`.

```
bpf(BPF_MAP_GET_NEXT_ID, {start_id=0,...}, 12) = 0
bpf(BPF_MAP_GET_FD_BY_ID, {map_id=48...}, 12) = 3
bpf(BPF_OBJ_GET_INFO_BY_FD, {info={bpf_fd=3, ...}}, 16) = 0
bpf(BPF_MAP_GET_NEXT_ID, {start_id=48, ...}, 12) = 0
bpf(BPF_MAP_GET_FD_BY_ID, {map_id=116, ...}, 12) = 3
bpf(BPF_OBJ_GET_INFO_BY_FD, {info={bpf_fd=3...}}, 16) = 0
```

1. Вызов с `BPF_MAP_GET_NEXT_ID` получает идентификатор следующей карты после значения, указанного в параметре `start_id`.
2. Вызов с `BPF_MAP_GET_FD_BY_ID` возвращает дескриптор файла для указанного идентификатора карты.
3. Вызов с `BPF_OBJ_GET_INFO_BY_FD` извлекает информацию об объекте (в данном случае о карте), на который ссылается файловый дескриптор. Эта информация включает имя объекта, чтобы `bpftool` мог проверить, является ли это той картей, которую он ищет.

4. Последовательность повторяется с шага 1, получая идентификатор следующей карты после последней найденной карты.

Осуществляется группа из этих трех системных вызовов для каждой карты, загруженной в ядро, и здесь вы также должны увидеть что значения, используемые для `start_id` и `map_id`, соответствуют идентификаторам этих карт. Повторение шаблона заканчивается когда больше не остается карт для просмотра, в результате чего вызов с кодом команды `BPF_MAP_GET_NEXT_ID` возвращает значение `ENOENT`, например:

```
bpf(BPF_MAP_GET_NEXT_ID, {start_id=133,...}, 12) = -1 ENOENT (No such file or directory)
```

Если соответствующая карта найдена, `bpftool` сохраняет её файловый дескриптор чтобы по нему можно было считывать элементы из этой карты.

Чтение элементов карты

К этой точке `bpftool` имеет файловый дескриптор ссылающийся на карту (карты) из которых он собирается читать. Давайте посмотрим на последовательность системных вызовов для чтения этой информации:

```
bpf(BPF_MAP_GET_NEXT_KEY, {map_fd=3, key=NULL,
next_key=0xaaaaf7a63960}, 24) = 0
bpf(BPF_MAP_LOOKUP_ELEM, {map_fd=3, key=0xaaaaf7a63960,
value=0xaaaaf7a63980, flags=BPF_ANY}, 32) = 0
[{
        "key": 0,
        "value": {
            "message": "Hey root!"
        }
bpf(BPF_MAP_GET_NEXT_KEY, {map_fd=3, key=0xaaaaf7a63960,
next_key=0xaaaaf7a63960}, 24) = 0
bpf(BPF_MAP_LOOKUP_ELEM, {map_fd=3, key=0xaaaaf7a63960,
value=0xaaaaf7a63980, flags=BPF_ANY}, 32) = 0
    },{
        "key": 501,
        "value": {
        "message": "Hi user 501!"
        }
bpf(BPF_MAP_GET_NEXT_KEY, {map_fd=3, key=0xaaaaf7a63960,
next_key=0xaaaaf7a63960}, 24) = -1 ENOENT (No such file or directory)
    }
]
+++ exited with 0 +++
```

1. Прежде всего, приложению необходимо найти валидный ключ, присутствующий в карте. Это делается с помощью варианта `BPF_MAP_GET_NEXT_KEY` системного вызова `bpf()`. Ключевой аргумент является указателем на карту, и системный вызов вернет следующий валидный ключ после этого. Передавая указатель `NULL`, приложение запрашивает первый валидный ключ в карте. Ядро записывает ключ в место, указанное указателем `next_key`.

2. Получив ключ, приложение запрашивает ассоциированное значение, которое даёт записанное в памяти местоположение, указанное этим значением.

3. В этом месте `bpftool` имеет содержимое первой пары ключ-значение и выводит эту информацию на экран.

4. Теперь `bpftool` переходит к следующему ключу в карте, извлекает его значение и выводит эту пару ключ-значение на экран.

5. Следующий вызов с `BPF_MAP_GET_NEXT_KEY` возвращает `ENOENT`, чтобы указать, что в карте больше нет записей.

6. На этом `bpftool` завершает вывод, записываемый на экран, и завершает работу.

Обратите внимание, что здесь `bpftool` был назначен на файловый дескриптор 3, соответствующий карте `config`. Это та же самая карта, на которую ссылается `hello-buffer-config.py` с файловым дескриптором 4. Как я уже упоминала, файловые дескрипторы зависят от конкретного процесса. Этот анализ того как ведет себя `bpftool` показывает как программа пользовательского пространства может перебирать доступные карты и пары ключ-значение, хранящиеся в карте.

Итоги

В этой главе вы увидели, как код пользовательского пространства использует системный вызов `bpf()` для загрузки программ и карт eBPF. Вы видели программы и карт, созданные с помощью команд `BPF_PROG_LOAD` и `BPF_MAP_CREATE`.

Вы узнали, что ядро отслеживает количество ссылок на программы и карты eBPF, освобождая их, когда счетчик ссылок падает до нуля. Вы также познакомились с концепциями закрепления объектов BPF в файловой системе и использования связей BPF для создания дополнительных ссылок.

Вы видели пример использования `BPF_MAP_UPDATE_ELEM` для создания записей в карте из пользовательского пространства. Существуют аналогичные команды — `BPF_MAP_LOOKUP_ELEM` и `BPF_MAP_DELETE_ELEM` — для извлечения и удаления значений из карты. Существует также команда `BPF_MAP_GET_NEXT_KEY` для поиска следующего ключа, присутствующего в карте. Вы можете использовать это для перебора всех допустимых записей.

Вы видели примеры программ пользовательского пространства, использующих `perf_event_open()` и `ioctl()` для присоединения программ eBPF к событиям `kprobe`. Метод подключения может сильно отличаться для других типов программ eBPF, а некоторые из них даже используют для этого системный вызов `bpf()`. Например, есть системный вызов `bpf(BPF_PROG_ATTACH)`, который можно используется для присоединения программ `cgroup`, а `bpf(BPF_RAW_TRACEPOINT_OPEN)` — для необработанных точек трассировки (см. упражнение 5 в конце этой главы).

Я также показала, как вы можете использовать `BPF_MAP_GET_NEXT_ID`, `BPF_MAP_GET_FD_BY_ID` и `BPF_OBJ_GET_INFO_BY_FD` для поиска карт (и других объектов), хранящихся в ядре. Есть и другие команды `bpf()`, которые я не рассматривала в этой главе, но того, что вы здесь увидели, достаточно, чтобы получить хороший обзор. Вы также видели некоторые данные BTF, загружаемые в ядро, и я упомянула, что `bpftool`

использует эту информацию чтобы понять формат структур данных, для того чтобы она могла их красиво распечатать. Я еще не объяснила, как выглядят эти данные BTF или как они используются для обеспечения переносимости программ eBPF между версиями ядра. Это будет в следующей главе.

Упражнения

Вот несколько вещей, которые вы можете попробовать, если хотите глубже изучить системный вызов `bpf()`:

1. Убедитесь, что поле `insn_cnt` из системного вызова `BPF_PROG_LOAD` соответствует количеству инструкций, которые выводятся, если вы выгружаете транслированный байт-код eBPF для этой программы с помощью `bpftool`. (Это задокументировано на справочной странице для системного вызова `bpf()`: https://man7.org/linux/man-pages/man2/bpf.2.html)

2. Запустите два экземпляра примера программы так, чтобы было две карты с именем `config`. Если вы запустите: `bpftool map dump name config`, то вывод будет включать информацию о двух разных картах, а также об их содержимом. Запустите это под `strace` и проследите за использованием различных файловых дескрипторов через вывод системного вызова. Вы видите, где он извлекает информацию о карте и где он извлекает пары ключ-значение, хранящиеся в ней?

3. Используйте: `bpftool map update` для изменения карты `config` во время работы одной из программ-примеров. Используйте sudo -u <имя пользователя> <команда> для того чтобы убедиться, что эти изменения конфигурации подхватываются программой eBPF.

4. Во время работы `hello-buffer-config.py` используйте `bpftool`, чтобы закрепить программу в файловой системе BPF, например:

   # bpftool prog pin name hello /sys/fs/bpf/hi

   Закройте запущенную программу и проверьте, загружена ли программа `hello` в ядро, используя список программ через `bpftool`. Вы можете очистить ссылку, удалив закрепление с помощью `rm /sys/fs/bpf/hi`.

5. Присоединение к необработанной точке трассировки значительно проще на уровне системного вызова, чем присоединение к `kprobe`, поскольку оно просто включает системный вызов `bpf()`. Попробуйте преобразовать `hello-buffer-config.py`, чтобы он присоединялся к необработанной точке трассировки для `sys_enter`, используя макрос `RAW_TRACEPOINT_PROBE` BCC (если вы выполняли упражнения из главы 2, у вас уже есть некоторая подходящая программа, которую вы можете использовать). Вам не нужно явно прикреплять программу в коде Python, так как BCC позаботится об этом за вас. Запустив это под `strace`, вы должны увидеть системный вызов, похожий на этот:

   ```
   bpf(BPF_RAW_TRACEPOINT_OPEN, {raw_tracepoint={name="sys_enter",
   prog_fd=6}}, 128) = 7
   ```

   Точка трассировки в ядре имеет имя `sys_enter`, и к ней привязывается программа eBPF с файловым дескриптором 6. Отныне всякий раз, когда выполнение в ядре достигает этой точки трассировки, оно запускает программу eBPF.

6. Запустите приложение `opensnoop` из набора инструментов `libbpf` BCC (https://github.com/iovisor/bcc/tree/master/libbpf-tools). Этот инструмент устанавливает некоторые связи BPF, которые вы можете увидеть с помощью `bpftool`, например:

```
# bpftool link list
116: perf_event prog 1849
        bpf_cookie 0
        pids opensnoop(17711)
117: perf_event prog 1851
        bpf_cookie 0
        pids opensnoop(17711)
```

Убедитесь что идентификаторы программ (1849 и 1851 в моем примере вывода здесь) совпадают с выводом из списка загруженных программ eBPF:

```
# bpftool prog list
...
1849: tracepoint name tracepoint__syscalls__sys_enter_openat
        tag 8ee3432dcd98ffc3 gpl run_time_ns 95875 run_cnt 121
        loaded_at 2023-01-08T15:49:54+0000 uid 0
        xlated 240B jited 264B memlock 4096B map_ids 571,568
        btf_id 710
        pids opensnoop(17711)
1851: tracepoint name tracepoint__syscalls__sys_exit_openat
        tag 387291c2fb839ac6 gpl run_time_ns 8515669 run_cnt 120
        loaded_at 2023-01-08T15:49:54+0000 uid 0
        xlated 696B jited 744B memlock 4096B map_ids 568,571,569
        btf_id 710
        pids opensnoop(17711)
```

7. Во время работы `opensnoop` попробуйте закрепить одну из этих ссылок с командой: `bpftool link pin id 116 /sys/fs/bpf/mylink` (используя один из идентификаторов связей, которые вы видите выше в списке ссылок `bpftool`). Вы должны увидеть, что даже после завершения работы `opensnoop` и ссылка, и соответствующая программа остаются загруженными в ядре.

8. Если вы перейдете к примерам кода для главы 5, вы найдете там версию `hello-buffer-config.py`, написанную с использованием библиотеки `libbpf`. Эта библиотека автоматически устанавливает связь BPF на программу, которую она загружает в ядро. Используйте `strace` для проверки системных вызовов `bpf()`, которые она делает, и просмотрите системные вызовы `bpf(BPF_LINK_CREATE)`.