Трассировка событий ядра с помощью eBPF

Наблюдение за Linux из пользовательского пространства
Практическая работа

Михаил Пожидаев

Институт прикладной математики и компьютерных наук ТГУ

2025 г.

eBPF

Программы, выполняющиеся внутри ядра

eBPF позволяет загрузить небольшую программу в ядро Linux и подключить её к определённому событию.
Программа eBPF выполняется внутри ядра, но не является обычным модулем ядра.
Она не может выполнять произвольные операции.
До загрузки в ядро программа проверяется специальным верификатором.
Пользовательская программа загружает eBPF-код, получает события и отображает их.

Главная идея

Собрать сведения о работе ядра, отфильтровать события в ядре и передать только нужные данные в пользовательское пространство.

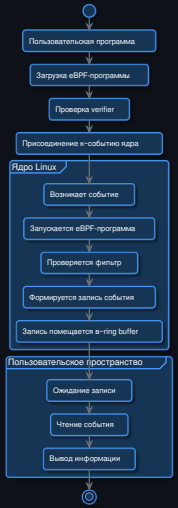
Архитектура программы

Ядро и пользовательское пространство

Программа состоит из двух частей.
Первая часть содержит eBPF-программу.
Она подключается к событию ядра и формирует записи для передачи наружу.
Вторая часть работает как обычный пользовательский процесс.
Она загружает eBPF-программу, настраивает фильтры, получает события и освобождает ресурсы.

Архитектура eBPF-приложения

Обмен событиями между ядром и пользовательским процессом



Какие события наблюдать

Точки подключения eBPF

Для лабораторной работы удобно наблюдать запуск процессов.
В этом случае можно использовать tracepoint `sched:sched_process_exec`.
Другой вариант — наблюдать открытие файлов через tracepoint `syscalls:sys_enter_openat`.
Также можно отслеживать системные вызовы, входящие в ядро.
Тип события определяет состав дополнительных данных.

Для первого знакомства лучше выбрать tracepoint.

У него стабильнее интерфейс и меньше зависимость от внутренних имён функций ядра.

Tracepoint

Стабильная точка наблюдения

Tracepoint — это заранее определённая точка трассировки внутри ядра.
Ядро вызывает подключённые к ней обработчики при возникновении соответствующего события.
Список доступных tracepoint можно посмотреть в отладочной файловой системе.
Список событий планировщика находится в каталоге
`/sys/kernel/debug/tracing/events/sched`.
Tracepoint обычно безопаснее для учебной работы, чем привязка к адресу внутренней функции ядра.

```
sudo mount -t debugfs none /sys/kernel/debug
sudo ls /sys/kernel/debug/tracing/events
sudo ls /sys/kernel/debug/tracing/events/sched
```

kprobe и uprobes

Другие способы подключения

`kprobe` позволяет подключиться к адресу или имени функции ядра.
Такой способ даёт больше возможностей, но сильнее зависит от версии и конфигурации ядра.
`uprobe` используется для наблюдения за функциями пользовательской программы.
Например, можно отслеживать вызов функции в конкретной библиотеке.
Для основной реализации рекомендуется использовать tracepoint.
Использование `kprobe` или `uprobe` можно оставить дополнительным вариантом.

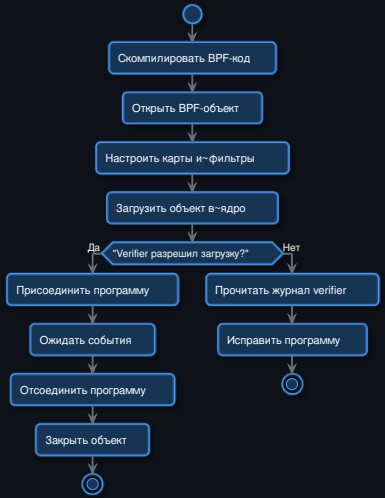
Жизненный цикл eBPF-программы

Загрузка и освобождение

Сначала пользовательская программа открывает объект eBPF.
Затем libbpf загружает программу и карты в ядро.
Verifier проверяет программу.
После успешной проверки программа присоединяется к выбранному событию.
Во время работы пользовательская программа читает данные из буфера.
При завершении программа отсоединяет обработчик и освобождает объект.

Жизненный цикл объекта

От исходного кода до события



Пользовательская программа

Библиотека libbpf

Для взаимодействия с eBPF удобно использовать библиотеку `libbpf`.
Она предоставляет функции для открытия, загрузки и присоединения BPF-объектов.
Она также упрощает работу с картами и буферами событий.
Обычно eBPF-программа компилируется отдельно в объектный файл.
Пользовательская программа затем загружает этот объект во время запуска.
Для реального приложения каждый шаг должен проверять код возврата.

```
obj = example_bpf__open();
if (!obj)
    return 1;

if (example_bpf__load(obj))
    return 1;

if (example_bpf__attach(obj))
    return 1;

/* Настройка ring buffer */
/* Ожидание событий */

example_bpf__destroy(obj);
```

BPF-программа

Минимальная функция обработки события

BPF-программа получает контекст события от ядра.
Из контекста она извлекает нужные поля.
Затем программа формирует запись и передаёт её в пользовательское пространство.
Функция должна завершиться быстро.
Она не должна выполнять длительные циклы или ожидать пользовательский процесс.

```
SEC("tracepoint/sched/sched_process_exec")
int handle_exec(void *ctx)
{
    /* Получить данные события */
    /* Проверить фильтр */
    /* Записать событие в ring buffer */

    return 0;
}
```

Данные события

Что передавать пользователю

Для каждой записи нужно определить собственную структуру данных.
Структура должна содержать только необходимые поля.
Размеры массивов должны быть ограничены.
Нельзя передавать в событии указатель на память ядра.
Указатель перестанет быть доступен после завершения eBPF-программы.

```
struct event {
    __u32 pid;
    __u32 tid;
    __u32 uid;
    __u64 timestamp;
    char comm[16];
    char filename[256];
};
```

Идентификаторы процесса

PID, TGID и UID

В Linux процесс может состоять из нескольких потоков.
Идентификатор потока называется `TID`.
Идентификатор группы потоков обычно называют `PID` или `TGID`.
Для однопоточной программы эти значения часто совпадают.
Идентификатор пользователя процесса можно получить из поля `uid`.
Для диагностики полезно передавать и PID, и TID.

Имя процесса

Получение значения comm

Ядро хранит короткое имя текущей задачи в поле `comm`.
Обычно его длина ограничена 16 байтами вместе с завершающим нулём.
Имя `comm` не обязательно совпадает с полным именем исполняемого файла.
Для длинного имени файла нужно отдельно получать данные из контекста конкретного tracepoint.
Строки необходимо копировать с учётом ограничения размера буфера.

Ring buffer

Передача событий в пользовательское пространство

`ring buffer` — это кольцевой буфер событий между ядром и пользовательским процессом.
BPF-программа резервирует место в буфере.
Затем она записывает туда структуру события и отправляет её пользователю.
Пользовательская программа извлекает записи по мере их появления.

```
event = bpf_ringbuf_reserve(
    &events,
    sizeof(*event),
    0
);

if (!event)
    return 0;

event->pid = pid;

bpf_ringbuf_submit(event, 0);
```

Освобождение записи

submit и discard

После успешного резервирования запись необходимо завершить.
Если запись заполнена корректно, используется `bpf_ringbuf_submit()`.
Если продолжать обработку невозможно, запись нужно отменить через `bpf_ringbuf_discard()`.
Нельзя оставить зарезервированную запись без завершения.

```
event = bpf_ringbuf_reserve(
    &events,
    sizeof(*event),
    0
);

if (!event)
    return 0;

if (error) {
    bpf_ringbuf_discard(event, 0);
    return 0;
}

bpf_ringbuf_submit(event, 0);
```

Потеря событий

Что происходит при переполнении

Ring buffer имеет конечный размер.
Если свободного места недостаточно, резервирование записи завершается неудачей.
Событие в этом случае не передаётся пользовательской программе.
Потерянные события необходимо учитывать отдельным счётчиком.
В пользовательской программе также нужно учитывать ошибки чтения буфера.

```
event = bpf_ringbuf_reserve(
    &events,
    sizeof(*event),
    0
);

if (!event) {
    __sync_fetch_and_add(&lost_events, 1);
    return 0;
}
```

Карты eBPF

Общее состояние программ

Карты позволяют хранить данные, доступные eBPF-программе и пользовательской программе. С их помощью можно реализовать счётчики, фильтры и таблицы соответствий.
Примеры карт:

- `BPF_MAP_TYPE_ARRAY` — массив фиксированного размера.
- `BPF_MAP_TYPE_HASH` — таблица ключей и значений.
- `BPF_MAP_TYPE_PERCPU_ARRAY` — отдельные значения для каждого CPU.
- `BPF_MAP_TYPE_RINGBUF` — буфер событий.

Карты существуют внутри ядра и не являются обычными указателями.

Счётчики

Сколько событий действительно обработано

Минимально нужно учитывать несколько величин.
Количество событий, обнаруженных eBPF-программой.
Количество событий, прошедших фильтр.
Количество событий, переданных пользователю.
Количество потерянных событий.
Количество событий, отброшенных пользовательской программой.
Для простого счётчика можно использовать массив из одного элемента.
При высокой нагрузке полезно использовать per-CPU карту, чтобы уменьшить конкуренцию между процессорами.

Фильтрация в ядре

Не передавать лишние события

Фильтрация должна выполняться внутри eBPF-программы.
Это уменьшает нагрузку на ring buffer.
Это уменьшает количество переходов между ядром и пользовательским пространством.
Параметры фильтра удобно хранить в карте.

```
if (event_uid != target_uid)
    return 0;

if (pid != target_pid)
    return 0;
```

Настройка фильтра

Изменение параметров без перекомпиляции

Пользовательская программа может записать параметры фильтра в BPF-карту.
eBPF-программа читает эти параметры при обработке события.
Например, карта может содержать один ключ и идентификатор процесса.
Фильтр можно изменить без пересборки и повторной загрузки eBPF-программы.

```
__u32 key = 0;
__u32 target_pid = 1234;

bpf_map_update_elem(
    map_fd,
    &key,
    &target_pid,
    BPF_ANY
);
```

Проверка verifier

Почему ядро доверяет eBPF-коду

Перед загрузкой verifier анализирует все возможные пути выполнения eBPF-программы.
Он проверяет границы доступа к памяти.
Он контролирует типы указателей.
Он ограничивает циклы и время выполнения.
Он запрещает недопустимые вызовы вспомогательных функций.
Если программа может обратиться к невалидной памяти или бесконечно выполнять инструкции, загрузка будет отклонена.

```
char buffer[16];

buffer[index] = value;
```

Ошибка проверки

Почему verifier отклоняет программу

Такой код может быть отклонён, если verifier не может доказать, что `index` находится в диапазоне от 0 до 15.
Границы массива нужно проверять явно.

Помощники eBPF

Безопасные операции ядра

eBPF-программа не может вызывать произвольные функции ядра.
Для разрешённых операций используются BPF helper functions.
С их помощью можно получить идентификатор текущего процесса.
Можно получить имя текущей задачи.
Можно прочитать ограниченный фрагмент строки.
Можно записать данные в карту или ring buffer.
Доступные помощники зависят от типа программы и версии ядра.

Ограничения eBPF-программы

Это не обычный C-код

Нельзя использовать произвольное выделение памяти через `malloc()`.
Нельзя обращаться к пользовательскому указателю без специального помощника.
Нельзя разыменовывать указатель на память ядра без разрешённого механизма доступа.
Нельзя выполнять бесконечный цикл.
Нельзя вызывать функции стандартной библиотеки C.
Можно использовать только разрешённые типы данных и помощники.
Код должен быть коротким и завершаться за ограниченное время.

Чтение событий

Ожидание без активного опроса

Пользовательская программа должна ждать события с минимальным потреблением процессорного времени.
Для этого используется функция `ring_buffer__poll()`.
Положительное значение обычно означает количество обработанных записей.
Нулевое значение означает, что за указанный период записи не поступили.

```
while (!exiting) {
    err = ring_buffer__poll(
        ring_buffer,
        1000
    );

    if (err < 0)
        break;
}
```

Обработчик записи

Callback пользовательской программы

При появлении события libbpf вызывает зарегистрированную функцию.
Указатель `data` действителен только в течение вызова callback.
Нельзя сохранять его для последующего использования без копирования.

```
static int handle_event(
    void *ctx,
    void *data,
    size_t data_size)
{
    const struct event *event;

    event = data;

    printf(
        "pid=%u uid=%u comm=%s\n",
        event->pid,
        event->uid,
        event->comm
    );

    return 0;
}
```

Безопасное завершение

Обработка SIGINT

Трассировка должна завершаться по нажатию Ctrl+C.
Обработчик сигнала не должен отсоединять eBPF-программу.
Он не должен освобождать карты, буферы или вызывать функции libbpf.
Достаточно изменить флаг типа `volatile sig_atomic_t`.
Основной цикл заметит флаг и выполнит освобождение ресурсов в обычном контексте.

```c
static volatile sig_atomic_t exiting;

static void sigint_handler(int signal)
{
    exiting = 1;
}
```

Обработка ошибок

Проверять нужно каждый этап

Необходимо проверять открытие BPF-объекта.
Необходимо проверять загрузку программы.
Необходимо проверять результат работы verifier.
Необходимо проверять присоединение к tracepoint.
Необходимо проверять создание ring buffer.
Необходимо проверять чтение событий.
При ошибке нужно освободить уже созданные ресурсы.
Порядок освобождения должен быть обратным порядку создания.

Сборка BPF-программы

clang и ядровые заголовки

Для компиляции BPF-кода используется `clang` с целью `bpf`.
BPF-программа обычно компилируется с отладочной информацией.
Отладочная информация помогает libbpf и инструментам ядра работать с типами данных.
Для сборки также потребуются заголовочные файлы ядра.

```
clang \
    -O2 \
    -g \
    -target bpf \
    -c tsulab.bpf.c \
    -o tsulab.bpf.o
```

Пользовательская программа компилируется обычным компилятором C.
Точный набор библиотек зависит от дистрибутива и способа установки libbpf.
Проверить наличие библиотеки можно через `pkg-config`.

```
cc \
    -O2 \
    -g \
    tsulab.c \
    -o tsulab \
    -lbpf \
    -lelf \
    -lz

pkg-config --cflags --libs libbpf
```

BTF и CO-RE

Переносимость между версиями ядра

BTF содержит описание типов данных, используемых ядром.
CO-RE означает Compile Once, Run Everywhere.
С помощью BTF и CO-RE одна BPF-программа может работать на разных версиях ядра.
Без CO-RE программа может зависеть от конкретных заголовочных файлов.
Проверить наличие BTF можно командой `ls -l /sys/kernel/btf/vmlinux`.
Если этот файл отсутствует, возможности переносимой сборки могут быть ограничены.

Проверка окружения

Подготовка системы

Перед началом работы нужно проверить версию ядра.
Нужно проверить наличие BPF-файловой системы.
При необходимости её можно подключить.
Нужно проверить наличие отладочной файловой системы.
Некоторые операции требуют прав суперпользователя или специальных capabilities.

```
uname -a

mount | grep bpf

sudo mount -t bpf bpf /sys/fs/bpf

mount | grep debugfs

ls -l /sys/kernel/btf/vmlinux
```

Запуск трассировки

Наблюдение за событиями процессов

После сборки пользовательская программа запускается с выбранным фильтром.
Для проверки запуска событий удобно выполнить несколько команд.
После завершения нужно нажать Ctrl+C и проверить итоговую статистику.

```
sudo ./tsulab --pid 1234

sudo ./tsulab --uid 1000

/bin/true
/bin/echo test
/usr/bin/id
```

Наблюдение за открытием файлов

Эксперимент с openat

Если выбран tracepoint системного вызова `openat`, можно запускать команды, читающие файлы.
Для каждого события полезно выводить имя процесса и имя открываемого файла.
Нужно отдельно проверить случай ошибки открытия файла.
Например, можно обратиться к несуществующему пути.
Событие входа в системный вызов возникает независимо от того, завершится операция успешно или с ошибкой.

```
cat /etc/hostname
cat /proc/meminfo
head /var/log/syslog
```

Потеря событий

Нагрузка на ring buffer

Чтобы проверить обработку переполнения, нужно создать большое количество событий.
Размер ring buffer можно временно уменьшить.
Пользовательская программа должна показать количество потерянных событий.
Результат зависит от скорости процессора, планировщика и скорости обработки callback.

```
for i in $(seq 1 100000); do
    /bin/true
done
```

В конце работы нужно вывести:

- количество обнаруженных событий.
- количество событий, прошедших фильтр.
- количество переданных событий.
- количество обработанных событий.
- количество потерянных событий.
- количество событий, отброшенных фильтром.

Если количество обнаруженных событий нельзя получить напрямую, нужно явно описать используемое приближение.
Счётчики должны храниться в картах eBPF или вычисляться по полученным событиям.

Типичные ошибки

Что проверять при отладке

Фильтрация только в пользовательской программе нарушает смысл задания.
Передача указателя вместо содержимого строки приводит к ошибке доступа.
Использование неограниченного массива может привести к отказу verifier.
Забытый `bpf_ringbuf_discard()` оставляет зарезервированную запись.
Обращение к данным callback после его завершения является ошибкой.
Освобождение BPF-объекта до отсоединения программы приводит к ошибке завершения.
Обработчик сигнала не должен вызывать функции libbpf.

Проверка результата

Что нужно показать

Нужно показать события, возникающие при запуске процессов или открытии файлов.
Нужно продемонстрировать работу фильтра по PID, UID или имени программы.
Нужно показать, что фильтрация выполняется в ядре.
Нужно искусственно создать большое количество событий.
Нужно показать обработку переполнения буфера.
Нужно завершить программу по SIGINT.
После завершения не должно оставаться загруженных программ, карт или ссылок.

Вопросы для защиты

Что нужно понимать

Что такое eBPF и чем он отличается от обычного модуля ядра?
Зачем нужен verifier?
Какие операции запрещены eBPF-программе?
Чем отличаются tracepoint, kprobe и uprobe?
Для чего используются карты eBPF?
Чем отличаются `ring buffer` и `perf buffer`?
Что произойдёт при переполнении ring buffer?
Почему фильтрацию выгодно выполнять в ядре?
Почему указатель на данные callback нельзя сохранить?
Почему обработчик `SIGINT` не должен отсоединять eBPF-программу?
Какие ресурсы необходимо освободить при завершении программы?

Спасибо за внимание!

Домашняя страница:
https://marigostra.ru

Электронная почта:
msp@luwrain.org

Telegram:
@Marigostra