bpf-helpers - функции-помощники Linux

Оглавление

Подсистема расширенных фильтров Беркли (extended Berkeley Packet Filter или eBPF) состоит из программ на языке псевдоассемблера, которые присоединяются к одной из нескольких ловушек в ядре (kernel hook) и работают при определённых событиях. Эта модель отличается об прежней «классической» модели фильтров ("classic" BPF или cBPF) в нескольких аспектах, одним из которых является возможность вызывать из программы специальные функции-помощники (helper). Набор доступных функций определяется списком (white-list) вспомогательных функций в ядре.

Эти функции применяются программами eBPF для взаимодействия с системой или в контексте работы программы. Например, они могут служить для вывода отладочных сообщений, получения времени, прошедшего с момента загрузки системы, взаимодействия с отображениями eBPF или манипуляций с сетевыми пакетами. Поскольку имеются разные типы программ eBPF и они работают в разном контексте, каждому типу программ доступна лишь часть вспомогательных функций.

По принятым в eBPF соглашениям вспомогательная функция не может принимать более 5 аргументов.

Программы eBPF напрямую вызывают скомпилированные функции-помощники, не требующие внешнего функционального интерфейса. В результате вызов этих функций не создаёт дополнительных издержек, что обеспечивает высокую производительность.

В этом документе приведён список вспомогательных функций, доступных разработчикам программ eBPF. Функции отсортированы в хронологическом порядке их добавления в ядро (начиная с наиболее старых).

bpf_map_lookup_elem

```
void *bpf_map_lookup_elem(struct bpf_map *map, const void *key)
```

Поиск записи в отображении map по заданному ключу key. Если запись не найдена, функция возвращает NULL.

bpf_map_update_elem

```
long bpf_map_update_elem(struct bpf_map *map, const void *key, const void *value, u64 flags)
```

Добавляет или обновляет запись в отображении map, заданном ключом key. Значения флагов указаны ниже.

BPF_NOEXIST
Записи для ключа key не должно присутствовать в отображении map.

BPF_EXIST
Запись для ключа key должна присутствовать в отображении map.

BPF_ANY
Условие наличия записи для key не задано.

Значение флага BPF_NOEXIST не может использоваться для отображений типа BPF_MAP_TYPE_ARRAY или BPF_MAP_TYPE_PERCPU_ARRAY (все элементы существуют всегда), иначе функция будет возвращать ошибку.

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_map_delete_elem

```
long bpf_map_delete_elem(struct bpf_map *map, const void *key)
```

Удаляет из отображения map запись с ключом key.

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_probe_read

```
long bpf_probe_read(void *dst, u32 size, const void *unsafe_ptr)
```

В программе трассировки (безопасно) пытается прочитать size байтов из памяти ядра по указателю unsafe_ptr и сохранить их в dst.

Обычно вместо этой функции применяется Generally, use bpf_probe_read() или bpf_probe_read_kernel().

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_ktime_get_ns

```
u64 bpf_ktime_get_ns(void)
```

Возвращает время, прошедшее с момента загрузки системы, в наносекундах. Время, в течение которого система была приостановлена (suspend), не учитывается, см. clock_gettime(CLOCK_MONOTONIC).

Функция возвращает текущее значение ktime.

bpf_trace_printk

```
long bpf_trace_printk(const char *fmt, u32 fmt_size, ...)
```

Это предназначенная для отладки функция в стиле printk(), выводящая сообщение в формате fmt (размером fmt_size) в файл /sys/kernel/debug/tracing/trace отладочной файловой системы DebugFS, если он доступен. Функция может принимать до 3 дополнительных аргументов u64 (в соответствии с общим ограничением eBPF в 5 аргументов).

При каждом вызове функции она добавляет строку в файл trace. Строки отбрасываются при открытии файла /sys/kernel/debug/tracing/trace. Чтобы предотвратить это используется /sys/kernel/debug/tracing/trace_pipe. Формат трассировки можно настраивать с помощью опций, заданных в файле /sys/kernel/debug/tracing/trace_options (см. файл README в каталоге /sys/kernel/debug/tracing). По умолчанию выводятся строки вида

```
telnet-470   [001] .N.. 419421.045894: 0x00000001: <formatted msg>
```

Здесь telnet указывает имя текущей задачи, 470 - её идентификатор (PID), 001 - номер процессора (CPU) на котором выполняется задача. В последовательности .N.. каждый символ относится к установке опций (включение прерываний, планирование, работа аппаратных и программных прерываний, уровень preempt_disabled). N в данном случае говорит

об установке TIF_NEED_RESCHED и PREEMPT_NEED_RESCHED. Значение 419421.045894 указывает метку времени, 0x00000001 - фиктивное значение, используемое BPF для регистра указателя инструкций. <formatted msg> содержит сообщение в формате fmt.

Спецификаторы преобразования для fmt похожи на применяемые в printk(), но включают лишь %d, %i, %u, %x, %ld, %li, %lu, %lx, %lld, %lli, %llu, %llx, %p, %s. Модификаторы вывода (размер поля, заполнение нулями и т. п.) не поддерживаются, а при наличии непонятного спецификатора функция возвращает лишь -EINVAL (ничего не печатая).

Функция bpf_trace_printk() достаточно медленная и применять её следует лишь при отладке. Поэтому в журналах ядра выводится блок уведомления (для нескольких строк), указывающий, что эту функцию не следует применять в рабочих системах (production use), при её первом использовании (точнее, при выделении буферов trace_printk()). Для передачи значений в пользовательское пространство следует отдавать предпочтение событиям perf.

Функция возвращает число записанных в буфер байтов, а при отказе - отрицательное значение.

bpf_get_prandom_u32

`u32 bpf_get_prandom_u32(void)`

Возвращает псевдослучайное значение. С точки зрения безопасности эта функция использует своё псевдослучайное внутреннее состояние и не может служить для получения затравки (seed) других псевдослучайных функций в ядре. Важно понимать, что используемый функцией генератор на является криптографически защищенным.

Функция возвращает случайное 32-битовое число без знака.

bpf_get_smp_processor_id

`u32 bpf_get_smp_processor_id(void)`

Определяет идентификатор процессора SMP (symmetric multiprocessing). Отметим, что все программы работают с выключенным вытеснением (preemption), поэтому идентификатор процессора SMP не меняется в процессе выполнения программы.

Функция возвращает SMP id для процессора, на котором выполняется программа.

bpf_skb_store_bytes

`long bpf_skb_store_bytes(struct sk_buff *skb, u32 offset, const void *from, u32 len, u64 flags)`

Сохраняет len байтов, указанных адресом from, в пакете, связанном с skb, по указанному смещению offset. Параметр flags указывает комбинацию BPF_F_RECOMPUTE_CSUM (автоматический пересчет контрольной суммы пакета после записи байтов) и BPF_F_INVALIDATE_HASH (установка для skb->hash, skb->swhash и skb->l4hash значений 0).

Вызов этой функции может менять содержимое базового буфера пакетов, поэтому при загрузке все проверки указателей, выполненные ранее, становятся недействительными и должны быть повторены, если функция применяется в сочетании с прямым доступом к пакету.

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_l3_csum_replace

`long bpf_l3_csum_replace(struct sk_buff *skb, u32 offset, u64 from, u64 to, u64 size)`

Пересчитывает контрольную сумму сетевого уровня L3 (например, IP) для пакета, связанного с skb. Расчёт выполняется инкрементно, поэтому функция должна знать размер (size, 2 или 4) и прежнее значение изменяемого поля (from) заголовка, заменяемого новым значением (to). Возможно сохранение разности между прежним и новым значением в поле to путём установки from=0 и size=0. В обоих случаях параметр offset указывает местоположение поля IP checksum в пакете.

Эта функция работает в сочетании с bpf_csum_diff(), которая не обновляет контрольную сумму на месте, обеспечивает большую гибкость и может работать с полями контрольной суммы другого размера (не только 2 и 4).

Вызов этой функции может менять содержимое базового буфера пакетов, поэтому при загрузке все проверки указателей, выполненные ранее, становятся недействительными и должны быть повторены, если функция применяется в сочетании с прямым доступом к пакету.

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_l4_csum_replace

`long bpf_l4_csum_replace(struct sk_buff *skb, u32 offset, u64 from, u64 to, u64 flags)`

Пересчитывает контрольную сумму транспортного уровня L4 (например, TCP, UDP, ICMP) для пакета, связанного с skb. Расчёт выполняется инкрементно, поэтому функция должна знать размер (4 младших бита поля flags) и прежнее значение изменяемого поля (from) заголовка, заменяемого новым значением (to). Возможно сохранение разности между прежним и новым значением в поле to путём установки from=0 и и сброса (0) 4 младших битов поля flags. В обоих случаях параметр offset указывает местоположение поля контрольной суммы в пакете. В дополнение к размеру поле flags может содержать фактические флаги (побитовая операция OR). При установке BPF_F_MARK_MANGLED_0 нулевая контрольная сумма не пересчитывается (если нет также флага BPF_F_MARK_ENFORCE), а для обновлений, приводящих к нулевой контрольной сумме взамен устанавливается значение CSUM_MANGLED_0. Флаг BPF_F_PSEUDO_HDR указывает, что контрольная сумма рассчитывается заново для псевдозаголовка.

Эта функция работает в сочетании с bpf_csum_diff(), которая не обновляет контрольную сумму на месте, обеспечивает большую гибкость и может работать с полями контрольной суммы другого размера (не только 2 и 4).

Вызов этой функции может менять содержимое базового буфера пакетов, поэтому при загрузке все проверки указателей, выполненные ранее, становятся недействительными и должны быть повторены, если функция применяется в сочетании с прямым доступом к пакету.

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_tail_call

`long bpf_tail_call(void *ctx, struct bpf_map *prog_array_map, u32 index)`

Эта специальная функция служит для перехода в другую программу eBPF. Используется тот же кадр стека (значения стека и регистров для вызывающей программы недоступны вызываемой). Этот механизм позволяет создавать цепочки программ для увеличения максимального числа доступных инструкция eBPF или выполнения заданных программ по условию. Из соображений безопасности число последовательных вызовов (длина цепочки) ограничивается.

При вызове этой функции программа пытается перейти в программу, указанную параметром index в специальном отображении prog_array_map типа BPF_MAP_TYPE_PROG_ARRAY и передаёт указатель контекста ctx.

При успешном вызове ядро сразу же выполняет первую инструкцию вызванной программы. Это не вызов функции и управление в предыдущую (вызвавшую) программу не возвращается. Отказ при вызове не влияет на вызывающую программу и она просто продолжает работу. Причиной отказа может быть отсутствие вызываемой программы (т. е. значение index больше числа элементов в prog_array_map) или достигнута максимальная длина цепочки программ, задаваемая в ядре макросом MAX_TAIL_CALL_CNT (недоступен из пользовательского пространства). В настоящее время используется значение 32.

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_clone_redirect

`long bpf_clone_redirect(struct sk_buff *skb, u32 ifindex, u64 flags)`

Клонирует и перенаправляет пакет, связанный с skb, в другое сетевое устройство с индексом ifindex. Для перенаправления можно использовать входные и выходные интерфейсы в зависимости от наличия флага BPF_F_INGRESS в параметре flags (установка флага задает входной интерфейс, сброс - выходной). Иные флаги функция в настоящее время не поддерживает.

По сравнению с функцией bpf_redirect() использование bpf_clone_redirect() вызывает дополнительные издержки на дублирование буфера пакета, но это происходит вне программы eBPF. Функция bpf_redirect() более эффективна, но обрабатывается кодом действия и перенаправление происходит только после возврата из программы eBPF.

Вызов этой функции может менять содержимое базового буфера пакетов, поэтому при загрузке все проверки указателей, выполненные ранее, становятся недействительными и должны быть повторены, если функция применяется в сочетании с прямым доступом к пакету.

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_get_current_pid_tgid

`u64 bpf_get_current_pid_tgid(void)`

Возвращает 64-битовое целое число, содержащее текущие значения tgid и pid в форме

`current_task->tgid << 32 | current_task->pid.`

bpf_get_current_uid_gid

`u64 bpf_get_current_uid_gid(void)`

Возвращает 64-битовое целое число, содержащее текущие значения GID и UIDв форме

`current_gid << 32 | current_uid.`

bpf_get_current_comm

`long bpf_get_current_comm(void *buf, u32 size_of_buf)`

Копирует атрибут comm текущей задачи в buf размером size_of_buf. Атрибут comm содержит имя исполняемого файла (без пути) текущей задачи. Значение size_of_buf должно быть положительным. При успешном выполнении функции строка buf завршается NUL-символом, при отказе заполняется нулями.

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_get_cgroup_classid

`u32 bpf_get_cgroup_classid(struct sk_buff *skb)`

Извлекает значение classid для текущей задачи, т. е. для контрольной группы (cgroup) net_cls, к которой относится skb. Эту функцию можно применять на выходном пути TC[1], но она непригодна для входного пути.

Группа управления net_cls обеспечивает интерфейс для маркировки сетевых пакетов на основе предоставленного пользователем идентификатора для всего трафика от задач, относящихся к соответствующей cgroup [1].

В ядре Linux имеется две версии cgroup - v1 и v2. Обе версии доступны для пользователей и могут применяться совместно, но следует отметить, что net_cls относится лишь к v1. Это делает функцию несовместимой с программами BPF, работающими только с v2 (сокет может включать в каждый момент данные лишь одной версии cgroup).

Эта функция доступна в ядре с опцией конфигурации CONFIG_CGROUP_NET_CLASSID, имеющей значение y или m.

Функция возвращает classid или значение 0 для принятого по умолчанию classid.

bpf_skb_vlan_push

`long bpf_skb_vlan_push(struct sk_buff *skb, __be16 vlan_proto, u16 vlan_tci)`

Вталкивает vlan_tci (данные управления тега VLAN) протокола vlan_proto в пакет, связанный с skb, а затем обновляет контрольную сумму. Если vlan_proto отличается от ETH_P_8021Q и ETH_P_8021AD, предполагается ETH_P_8021Q.

[1]Traffic Control - управление трафиком.

Вызов этой функции может менять содержимое базового буфера пакетов, поэтому при загрузке все проверки указателей, выполненные ранее, становятся недействительными и должны быть повторены, если функция применяется в сочетании с прямым доступом к пакету.

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_skb_vlan_pop

long bpf_skb_vlan_pop(struct sk_buff *skb)

Выталкивает заголовок VLAN из пакета, связанного с skb.

Вызов этой функции может менять содержимое базового буфера пакетов, поэтому при загрузке все проверки указателей, выполненные ранее, становятся недействительными и должны быть повторены, если функция применяется в сочетании с прямым доступом к пакету.

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_skb_get_tunnel_key

long bpf_skb_get_tunnel_key(struct sk_buff *skb, struct bpf_tunnel_key *key, u32 size, u64 flags)

Возвращает метаданные туннеля, принимая указатель key на пустую структуру bpf_tunnel_key размером size, в которую помещаются метаданные туннеля для пакета, связанного с skb. В поле flags может помещаться флаг BPF_F_TUNINFO_IPV6, указывающий, что туннель использует протокол IPv6, а не IPv4.

Структура bpf_tunnel_key является объектом, обобщающим основные параметры, используемые разными туннельными протоколами. Её можно использовать для упрощения принятия решения на основе заголовка инкапсуляции. В частности, структура включает IP-адрес удалённой стороны (IPv4 или IPv6) в поле key->remote_ipv4 или key->remote_ipv6. Кроме того, структура раскрывает параметр key->tunnel_id, который обычно отображается на идентификатор виртуальной сети (Virtual Network Identifier или VNI), что позволяет программировать его с помощью функции bpf_skb_set_tunnel_key().

Представим, что приведённый ниже фрагмент кода является частью программы, связанной со входным интерфейсом TC на одном конце туннеля GRE, и должен отфильтровывать все сообщения с адресов IPv4 на другом конце туннеля, отличающихся от 10.0.0.1

```
int ret;
struct bpf_tunnel_key key = {};

ret = bpf_skb_get_tunnel_key(skb, &key, sizeof(key), 0);
if (ret < 0)
        return TC_ACT_SHOT;     // отбросить пакет

if (key.remote_ipv4 != 0x0a000001)
        return TC_ACT_SHOT;     // отбросить пакет

return TC_ACT_OK;               // воспринять пакет
```

Этот интерфейс можно использовать также с устройствами инкапсуляции, которые могут работать в режиме сбора метаданных, получая конфигурацию от этой функции.

Функция подходит для работы с такими туннелями, как VXLan, Geneve, GRE, IPIP.

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_skb_set_tunnel_key

long bpf_skb_set_tunnel_key(struct sk_buff *skb, struct bpf_tunnel_key *key, u32 size, u64 flags)

Заполняет структуру метаданных туннеля bpf_tunnel_key для пакета, связанного с skb. Структура указывается ключом key и имеет размер size. Поле flags может содержать комбинацию перечисленных ниже флагов.

BPF_F_TUNINFO_IPV6
: Указывает, что туннель использует протокол IPv6, а не IPv4.

BPF_F_ZERO_CSUM_TX
: Для пакетов IPv4 дабавляет флаг метаданных, указывающий, что расчёт контрольной суммы следует пропустить, заполнив поле контрольной суммы нулями.

BPF_F_DONT_FRAGMENT
: Добавляет флаг метаданных, указывающий, что пакет не следует фрагментировать.

BPF_F_SEQ_NUMBER
: Добавляет флаг метаданных, указывающий, что в заголовок туннеля перед отправкой пакета следует добавить порядковый номер. Этот флаг был добавлен для инкапсуляции GRE, но может применяться и другими протоколами.

Ниже представлен пример использовани на выходном пути.

```
struct bpf_tunnel_key key;
        заполнение key ...
bpf_skb_set_tunnel_key(skb, &key, sizeof(key), 0);
bpf_clone_redirect(skb, vxlan_dev_ifindex, 0);
```

См. также описание функции bpf_skb_get_tunnel_key().

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_perf_event_read

u64 bpf_perf_event_read(struct bpf_map *map, u64 flags)

Считывает значение счётчика событий perf. Эта функция основана на отображении map типа BPF_MAP_TYPE_PERF_EVENT_ARRAY. Характер счётчика событий perf выбирается при обновлении map с помощью

файловых дескрипторов событий perf. Отображение map является массивом, размер которого определяется числом доступных CPU, и каждый элемент содержит значение для одного процессора. Извлекаемое значение указывается параметром flags, который содержит индекс интересующего CPU с маской BPF_F_INDEX_MASK. В поле flags можно поместить значение BPF_F_CURRENT_CPU, указывающее извлечение счётчика для текущего CPU.

Отметим, что в Linux с ядрами до 4.13 доступны лишь аппаратные события perf.

В общем случае рекомендует использовать более новую функцию bpf_perf_event_read_value(), поскольку bpf_perf_event_read() имеет некоторые особенности ABI, когда в качестве кода возврата используется ошибка и значение счётчика (это некорректно, поскольку диапазоны могут перекрываться). Проблема решена в функции bpf_perf_event_read_value(), которая также предоставляет больше возможностей.

Функция возвращает значение счётчика событий perf из отображения или отрицательное значение в случае ошибки.

bpf_redirect

```
long bpf_redirect(u32 ifindex, u64 flags)
```

Перенаправляет пакет на другое сетевое устройство с индексом ifindex. Эта функция похожа на bpf_clone_redirect(), но не клонирует пакет, что обеспечивает более высокую производительность.

За исключением XDP[1], для перенаправления могут использоваться входные и выходные интерфейсы в зависимости от наличия флага BPF_F_INGRESS в параметре flags (установленный флаг задаёт входной интерфейс, сброшенный - выходной). XDP в настоящее время поддерживает лишь перенаправление в выходной интерфейс и не принимает никаких флагов.

Такой же эффект можно получить с помощью функции bpf_redirect_map(), использующей отображение BPF для сохранения цели перенаправления вместо её прямого указания функции-помощнику.

Для XDP функция возвращает XDP_REDIRECT при успехе или XDP_ABORTED при ошибке. Для иных типов программ при успехе возвращается TC_ACT_REDIRECT, при ошибке - TC_ACT_SHOT.

bpf_get_route_realm

```
u32 bpf_get_route_realm(struct sk_buff *skb)
```

Извлекает область (realm) маршрута, т. е. поле tclassid для получателя в skb. Извлекаемый идентификатор является предоставленным пользователем тегом, подобным применяемому с net_cls cgroup (см. bpf_get_cgroup_classid()), но относящимся к маршруту (записи для адресата), а не к задаче.

Извлечение этого идентификатора работает с выходной ловушкой TC clsact (см. man 8 tc-bpf) или с классическими дисциплинами выходных очередей qdisc, не на входном пути TC. Вариант с выходной ловушкой TC clsact имеет преимущество в том, что запись для получателя ещё не отброшена на пути передачи, поэтому её не требуется удерживать искусственно через netif_keep_dst() как в классической дисциплине qdisc, пока skb не будет освобождён.

Эта функция доступна лишь с ядрами, собранными с опцией CONFIG_IP_ROUTE_CLASSID.

Функция возвращает область маршрута для пакета, связанного с skb, или 0, если ничего не найдено.

bpf_perf_event_output

```
long bpf_perf_event_output(void *ctx, struct bpf_map *map, u64 flags, void *data, u64 size)
```

Записывает блок необработанных данных (blob) в специальное событие BPF в отображении map типа BPF_MAP_TYPE_PERF_EVENT_ARRAY. Это событие должно иметь атрибуты PERF_SAMPLE_RAW sample_type, PERF_TYPE_SOFTWARE type и PERF_COUNT_SW_BPF_OUTPUT.

Аргумент flags служит для указания индекса в map, по которому должно размещаться значение, с применением маски BPF_F_INDEX_MASK. В качестве flags можно указать BPF_F_CURRENT_CPU для указания использования индекса текущего ядра CPU.

Записываемое значение размером size передаётся через стек eBPF и указывается параметром data.

Программе-помощнику должен также передаваться контекст ctx.

В пользовательском пространстве программе, желающей прочитать значения, нужно вызвать функцию perf_event_open() для события perf (на одном или всех CPU) и сохранить дескриптор файла в map. Это должно выполняться до того, как программа eBPF сможет передавать туда данные. Пример этого представлен в файле samples/bpf/trace_output_user.c дерева исходных кодов ядра Linux (программа eBPF находится в файле samples/bpf/trace_output_kern.c).

Функция bpf_perf_event_output() обеспечивает лучшую производительность, нежели bpf_trace_printk(), для совместного использования данных с пространством пользователей и гораздо лучше подходит для потоковых данных из программ eBPF.

Эта функция не ограничивается трассировкой и может применяться с программами, присоединёнными к TC или XDP, где она позволяет передавать данные получателям в пользовательском пространстве. Данными могут быть настраиваемые (custom) структуры и/или содержимое пакета (payload).

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_skb_load_bytes

```
long bpf_skb_load_bytes(const void *skb, u32 offset, void *to, u32 len)
```

Функция обеспечивает простой способ загрузки данных из пакета и может служить для загрузки len байтов, начиная со смещения offset, из пакета, связанного с skb, в буфер, указанный параметром to.

[1] eXpress Data Path - «экспресс-путь» передачи данных.

Начиная с ядра Linux 4.7 использование этой функции было в основном заменено прямым доступом к пакетам, позволяющим манипулировать данными пакета с skb->data и skb->data_end, указывающими первый байт и байт, следующий за последним. Однако функция остаётся полезной, если нужно однократно считать большой объем данных из пакета в стек eBPF.

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_get_stackid

`long bpf_get_stackid(void *ctx, struct bpf_map *map, u64 flags)`

Возвращает идентификатор стека пользовательского пространства или ядра. Для этого функции нужен указатель на контекст ctx, в котором выполняется программа трассировки, а также указатель на отображение map типа BPF_MAP_TYPE_STACK_TRACE. Аргумент flags указывает число пропускаемых кадров стека (0 — 255) с маской BPF_F_SKIP_FIELD_MASK. Может применяться комбинация указанных ниже флагов.

BPF_F_USER_STACK
Обращение к пользовательскому стеку, а не к стеку ядра.

BPF_F_FAST_STACK_CMP
Сравнение стеков только по хэш-значениям.

BPF_F_REUSE_STACKID
Если два хэш-значения дают одно значение stackid, более старый стек отбрасывается.

Получаемый идентификатор стека является целочисленным 32-битовым дескриптором, который можно комбинировать с другими данными (включая идентификаторы других стеков) и использовать в качестве ключей отображений. Это может быть полезно для создания различных графов (например, графы кадров или off-cpu).

Для обхода стека эта функция лучше, чем bpf_probe_read(), которую можно применять с развёрнутыми (unrolled) циклами, но она менее эффективна и использует много инструкций eBPF. Однако bpf_get_stackid() может собирать до PERF_MAX_STACK_DEPTH кадров ядра и пользовательских кадров. Отметим, что этот предел задаёт sysctl и его можно увеличить вручную для профилирования длинных пользовательских стеков (таких как стеки программ Java). Для этого служит команда вида

`# sysctl kernel.perf_event_max_stack=<new value>`

Функция возвращает положительный или нулевой (null) идентификатор стека при успешном выполнении и отрицательное значение в случае ошибки.

bpf_csum_diff

`s64 bpf_csum_diff(__be32 *from, u32 from_size, __be32 *to, u32 to_size, __wsum seed)`

Рассчитывает разность контрольных сумм необработанных (raw) буферов - указанного from, размером from_size и указанного to, размером to_size (размеры буферов должны быть кратны 4). К значению может добавляться необязательный параметр seed (это можно делать каскадно, принимая результат предыдущего вызова функции).

Функцию можно применять разными способами.

- С from_size == 0, to_size > 0 и seed со значением контрольной суммы при добавлении данных.
- С from_size > 0, to_size == 0 и seed со значением контрольной суммы при удалении данных из пакета.
- С from_size > 0, to_size > 0 и seed = 0 для расчёта разности. Отметим, что размеры from_size и to_size могут различаться.

Эту функцию можно применять вместе с bpf_l3_csum_replace() и bpf_l4_csum_replace(), которым можно передать разность, определённую bpf_csum_diff().

Функция возвращает результат расчёта или отрицательный код ошибки при отказе.

bpf_skb_get_tunnel_opt

`long bpf_skb_get_tunnel_opt(struct sk_buff *skb, void *opt, u32 size)`

Извлекает метаданные опций туннеля для пакета, связанного с skb, и сохраняет необработанные данные в буфере opt размера size.

Эту функцию можно использовать с устройствами инкапсуляции, которые могут работать в режиме сбора метаданных (см. bpf_skb_get_tunnel_key()). Примером такого использования является сочетание с протоколом инкапсуляции Geneve, где функция позволяет вталкивать (с помощью bpf_skb_get_tunnel_key()) и извлекать произвольные TLV[1] из программы eBPF. Это позволяет полностью настраивать заголовки.

Функция возвращает размер найденный данных опций.

bpf_skb_set_tunnel_opt

`long bpf_skb_set_tunnel_opt(struct sk_buff *skb, void *opt, u32 size)`

Устанавливает метаданные опций туннеля для пакета, связанного с skb, на основе данных из необработанного буфера opt размером size. Дополнительные сведения приведены в описании функции bpf_skb_get_tunnel_opt().

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_skb_change_proto

`long bpf_skb_change_proto(struct sk_buff *skb, __be16 proto, u64 flags)`

Меняет для skb значение протокола на proto. В настоящее время можно заменить IPv4 на IPv6 и обратно. Функция выполняет подготовку к замене, включая изменение размера буфера сокета. Предполагается, что программа eBPF заполнит новые заголовки (при наличии) с помощью skb_store_bytes() и пересчитает контрольные суммы с помощью

[1]Type-Length-Value - тип, размер, значение.

bpf_l3_csum_replace() и bpf_l4_csum_replace(). Основным назначением этой функции являются операции NAT64 из программы eBPF.

Тип GSO внутренне помечен как рискованный, поэтому заголовки проверяются, а сегменты пересчитываются машиной GSO/GRO. Размер для цели GSO адаптируется.

Значения параметра flags зарезервированы на будущее и поле должно иметь значение 0.

Вызов этой функции может менять содержимое базового буфера пакетов, поэтому при загрузке все проверки указателей, выполненные ранее, становятся недействительными и должны быть повторены, если функция применяется в сочетании с прямым доступом к пакету.

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_skb_change_type

```
long bpf_skb_change_type(struct sk_buff *skb, u32 type)
```

Меняет тип (адресата) для пакета, связанного с skb. Действие функции к установке для skb->pkt_type значения type. Программа eBPF не имеет другого доступа для записи в skb->pkt_type, кроме этой функции. Функция помогает изящно обрабатывать ошибки.

Основным применением является замена входящих skb на PACKET_HOST программным путём вместо рециркуляции через redirect(..., BPF_F_INGRESS), например.

Для type в настоящее время разрешены лишь указанные ниже значения.

PACKET_HOST
Пакет для нас (для данного хоста).

PACKET_BROADCAST
Передать пакет всем.

PACKET_MULTICAST
Передать пакет в группу.

PACKET_OTHERHOST
Передать пакето кому-либо иному.

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_skb_under_cgroup

```
long bpf_skb_under_cgroup(struct sk_buff *skb, struct bpf_map *map, u32 index)
```

Проверяет, является ли skb потомком группы управления cgroup2, содержащейся в отображении map типа BPF_MAP_TYPE_CGROUP_ARRAY по индексу index.

Возвращаемое значение определяется результатом проверки:

- 0, если skb не является наследником cgroup2;
- 1, если skb является наследником cgroup2;
- отрицательное значение при возникновении ошибки.

bpf_get_hash_recalc

```
u32 bpf_get_hash_recalc(struct sk_buff *skb)
```

Извлекает хэш пакета skb->hash. Если значение не установлено, например, в результате изменения пакета (mangling), хэш рассчитывается заново. Последующий доступ к хэш-значению возможен напрямую через skb->hash.

Вызов bpf_set_hash_invalid(), смена прототипа пакета с помощью bpf_skb_change_proto() или вызов bpf_skb_store_bytes() с BPF_F_INVALIDATE_HASH могут приводить к очистке хэша и запуску нового расчёта для следующего вызова bpf_get_hash_recalc().

Функция возвращает 32-битовое хэш-значение.

bpf_get_current_task

```
u64 bpf_get_current_task(void)
```

Возвращает указатель на текущую структуру задачи (task).

bpf_probe_write_user

```
long bpf_probe_write_user(void *dst, const void *src, u32 len)
```

Пытается безопасным способом записать len байтов из буфера src в память dst. Функция работает только для потоков (thread) пользовательского контекста, а параметр dst должен указывать действительный в пользовательском пространстве адрес.

Эту функцию не следует применять для реализации каких-либо механизмов защиты из-за атак TOC-TOU, она предназначена, скорее, для отладки и управления исполнением полукооперативными процессами.

Следует помнить, что функция экспериментальная и с ней связан риск отказов системы и работающих программ. Поэтому при подключении программы eBPF, использующей эту функцию, в журнал ядра заносится предупреждения с указанием PID и имени процесса.

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_current_task_under_cgroup

```
long bpf_current_task_under_cgroup(struct bpf_map *map, u32 index)
```

Проверяет запущен ли тест (probe) в контексте данного подмножества иерархии cgroup2, содержащейся в отображении map типа BPF_MAP_TYPE_CGROUP_ARRAY по индексу index.

Возвращаемое значение определяется результатом проверки:

- 0, если задача *skb* относится к cgroup2;
- 1, если задача *skb* не относится к cgroup2;
- отрицательное значение при возникновении ошибки.

bpf_skb_change_tail

```
long bpf_skb_change_tail(struct sk_buff *skb, u32 len, u64 flags)
```

Меняет (расширяет или сокращает) размер пакета, связанного с skb, на новое значение len. Поле flags является резервным и должно иметь значение 0.

Основная идея состоит в изменении функцией размера пакета, после чего программа eBPF переписывает остальное с помощью функций bpf_skb_store_bytes(), bpf_l3_csum_replace(), bpf_l4_csum_replace() и др. Эта функция служит утилитой медленного пути, предназначенной для откликов с управляющими сообщениями, поэтому сама является достаточно медленной, неявно линеаризуя, деклонируя и отбрасывая выгрузку из skb.

Вызов этой функции может менять содержимое базового буфера пакетов, поэтому при загрузке все проверки указателей, выполненные ранее, становятся недействительными и должны быть повторены, если функция применяется в сочетании с прямым доступом к пакету.

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_skb_pull_data

```
long bpf_skb_pull_data(struct sk_buff *skb, u32 len)
```

Извлекает нелинейные данные в случае, когда skb является нелинейным и не весь размер len относится к линейной части. Функция делает len буфера skb доступными для чтения и записи, при len=0 вытягивается skb целиком.

Эта функция нужна лишь для чтения или записи при прямом доступе к пакету.

При непосредственном доступе к пакету проверка того, что заданное для доступа смещение не выводит за границы пакета (проверка skb->data_end), может давать отрицательный результат, если смещение некорректно или запрошенные данные относятся к нелинейной части skb. В таком случае программа может выйти из строя или (в случае нелинейного буфера) использовать вспомогательную функцию для обеспечения доступности данных. Первым решением является вызов bpf_skb_load_bytes(), а другое заключается в использовании bpf_skb_pull_data для извлечения в один приём нелинейных частей с последующей проверкой и доступом к данным.

При этом гарантируется, что skb не будет клонироваться, что является необходимым условием для прямой записи. Поскольку инвариантность нужна лишь для записи, проверка обнаруживает запись и добавляет пролог, вызывающий bpf_skb_pull_data() для эффективного деклонирования skb в самом начале, если буфер действительно клонирован.

Вызов этой функции может менять содержимое базового буфера пакетов, поэтому при загрузке все проверки указателей, выполненные ранее, становятся недействительными и должны быть повторены, если функция применяется в сочетании с прямым доступом к пакету.

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_csum_update

```
s64 bpf_csum_update(struct sk_buff *skb, __wsum csum)
```

Добавляет csum к skb->csum, если драйвер ввёл в это поле контрольную сумму всего пакета. В противном случае возвращается ошибка. Эта функция предназначена для использования вместе с bpf_csum_diff(), в частности, для обновления контрольной суммы после записи данных в пакет при прямом доступе к нему.

Функция возвращает контрольную сумму при успешном выполнении и отрицательный код в случа ошибки.

bpf_set_hash_invalid

```
void bpf_set_hash_invalid(struct sk_buff *skb)
```

делает недействительным текущий хэш skb->hash. Это может использоваться после изменения (mangling) заголовков путём прямого доступа к пакету, чтобы указать несоответствие хэша и запустить его повторный расчёт при следующей попытке ядра получить доступ к этому хэш-значению или при вызове bpf_get_hash_recalc().

bpf_get_numa_node_id

```
long bpf_get_numa_node_id(void)
```

Возвращает идентификатор текущего узла NUMA[1]. Основным назначением функции является выбор сокетов для локального узла NUMA при подключении программы к сокетам с опцией SO_ATTACH_REUSEPORT_EBPF (см. man 7 socket), но она доступна и другим типам программ, подобно bpf_get_smp_processor_id().

[1]Non-Uniform Memory Access - неунифицированный доступ к памяти, когда время доступа зависит от расположения памяти относительно процессора.

bpf_skb_change_head

```
long bpf_skb_change_head(struct sk_buff *skb, u32 len, u64 flags)
```

Увеличивает пространство пакета, связанного с skb, на len байтов и корректирует должным образом смещение заголовка MAC при необходимости автоматически расширяя выделенную память.

Эту функцию можно использовать с L3 skb при вталкивания заголовка MAC для перенаправления на устройство L2.

Значения для флагов являются резервными и поле flags должно иметь значение 0.

Вызов этой функции может менять содержимое базового буфера пакетов, поэтому при загрузке все проверки указателей, выполненные ранее, становятся недействительными и должны быть повторены, если функция применяется в сочетании с прямым доступом к пакету.

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_xdp_adjust_head

```
long bpf_xdp_adjust_head(struct xdp_buff *xdp_md, int delta)
```

Перемещает данные xdp_md->data на delta байтов (значение delta может быть отрицательным). Функция может применяться для подготовки пакета к вталкиванию или выталкиванию заголовков.

Вызов этой функции может менять содержимое базового буфера пакетов, поэтому при загрузке все проверки указателей, выполненные ранее, становятся недействительными и должны быть повторены, если функция применяется в сочетании с прямым доступом к пакету.

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_probe_read_str

```
long bpf_probe_read_str(void *dst, u32 size, const void *unsafe_ptr)
```

Копирует строку с NUL-символом в конце из небезопасного адреса ядра unsafe_ptr в dst. Обычно вместо этой функции применяется bpf_probe_read_user_str() или bpf_probe_read_kernel_str().

При успешном копировании функция возвращает размер скопированной строки с учётом NUL-символа, а при ошибке - отрицательное значение.

bpf_get_socket_cookie

```
u64 bpf_get_socket_cookie(struct sk_buff *skb)
```

Если структура sk_buff, указанная skb, имеет известный сокет, функция извлекает (созданное ядром) значение cookie этого сокета. При отсутствии cookie создаётся новое значение, которое сохраняется в течение срока действия сокета. Функция может быть полезна для отслеживания статистики сетевого трафика на уровне сокета, поскольку она возвращает глобальный идентификатор сокета, который можно считать уникальным.

Функция возвращает 8-байтовое неубывающее значение при успехе или 0, если поля сокета нет в skb.

bpf_get_socket_cookie

```
u64 bpf_get_socket_cookie(struct bpf_sock_addr *ctx)
```

Эквивалентна функции bpf_get_socket_cookie(), получающей skb, но берет сокет из контекста структуры bpf_sock_addr.

Функция возвращает 8-байтовое неубывающее значение.

bpf_get_socket_cookie

```
u64 bpf_get_socket_cookie(struct bpf_sock_ops *ctx)
```

Эквивалентна функции bpf_get_socket_cookie(), получающей skb, но берет сокет из контекста структуры bpf_sock_ops.

Функция возвращает 8-байтовое неубывающее значение.

bpf_get_socket_uid

```
u32 bpf_get_socket_uid(struct sk_buff *skb)
```

Возвращает UID владельца сокета, связанного с skb. Для сокета NULL или неполного сокета (т. е. в состоянии time-wait или сокета запроса) возвращается значение overflowuid, которое может быть и фактическим значением UID для сокета.

bpf_set_hash

```
long bpf_set_hash(struct sk_buff *skb, u32 hash)
```

Устанавливает для полного хэша skb (поле skb->hash) значение hash.

Функция возвращает 0.

bpf_setsockopt

```
long bpf_setsockopt(void *bpf_socket, int level, int optname, void *optval, int optlen)
```

Эмулирует вызов setsockopt() на сокете, связанном с bpf_socket, который должен быть полным сокетом. Требуется указать уровень, где размещается опция и имя optname (см. man 2 setsockopt). Значение опции размером optlen указывает параметр optval.

Параметр bpf_socket имеет одно из указанных ниже значений.

- структура bpf_sock_ops для BPF_PROG_TYPE_SOCK_OPS.
- структура bpf_sock_ops для BPF_CGROUP_INET4_CONNECT и BPF_CGROUP_INET6_CONNECT.

Эта функция фактически реализует часть setsockopt() и поддерживает указанные ниже уровни.

SOL_SOCKET
Поддерживает значения optname: SO_RCVBUF, SO_SNDBUF, SO_MAX_PACING_RATE, SO_PRIORITY, SO_RCVLOWAT, SO_MARK, SO_BINDTODEVICE, SO_KEEPALIVE.

IPPROTO_TCP
Поддерживает значения optname: TCP_CONGESTION, TCP_BPF_IW, TCP_BPF_SNDCWND_CLAMP, TCP_SAVE_SYN, TCP_KEEPIDLE, TCP_KEEPINTVL, TCP_KEEPCNT, TCP_SYNCNT, TCP_USER_TIMEOUT.

IPPROTO_IP
Поддерживает optname IP_TOS.

IPPROTO_IPV6
Поддерживает optname IPV6_TCLASS.

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_skb_adjust_room

`long bpf_skb_adjust_room(struct sk_buff *skb, s32 len_diff, u32 mode, u64 flags)`

Расширяет или сокращает пространство для данных в пакет, связанном с skb на len_diff байтов в соответствии с режимом mode.

По умолчанию функция сбрасываю любой выгруженный идентификатор контрольной суммы skb в CHECKSUM_NONE. Этого можно избежать установкой описанного ниже флага.

BPF_F_ADJ_ROOM_NO_CSUM_RESET
Не сбрасывать выгруженные данные контрольной суммы skb в CHECKSUM_NONE.

В настоящее время поддерживается два режиме, указанных ниже.

BPF_ADJ_ROOM_MAC
Корректировка пространства на канальном уровне (размер пространства изменяется ниже заголовка L2).

BPF_ADJ_ROOM_NET
Корректировка пространства на сетевом уровне (размер пространства изменяется ниже заголовка L3).

Ниже перечислены поддерживаемые флаги.

BPF_F_ADJ_ROOM_FIXED_GSO
Не менять gso_size. Настройка mss в этом случае недоступна для дейтаграмм.

BPF_F_ADJ_ROOM_ENCAP_L3_IPV4 и BPF_F_ADJ_ROOM_ENCAP_L3_IPV6
Резервируется пространство для туннельного заголовка с корректировкой смещений skb и других полей.

BPF_F_ADJ_ROOM_ENCAP_L4_GRE и BPF_F_ADJ_ROOM_ENCAP_L4_UDP
Используются с флагом ENCAP_L3 для задания типа туннеля.

BPF_F_ADJ_ROOM_ENCAP_L2(len)
Используется с флагами ENCAP_L3/L4 для уточнения типа туннеля, len указывает размер внутреннего заголовка is MAC.

Вызов этой функции может менять содержимое базового буфера пакетов, поэтому при загрузке все проверки указателей, выполненные ранее, становятся недействительными и должны быть повторены, если функция применяется в сочетании с прямым доступом к пакету.

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_redirect_map

`long bpf_redirect_map(struct bpf_map *map, u32 key, u64 flags)`

Перенаправляет пакет конечной точке, указанной индексом key в отображении map. В зависимости от типа отображение map может указывать сетевые устройства (для пересылки пакета через другой порт) или CPU (для пересылки кадров XDP в другой CPU[1]).

Два младших бита параметра flags служат для возврата кода завершения в случае ошибки. Это сделано для того, чтобы можно было возвращать один из кодов завершения программы XDP (вплоть до XDP_TX), выбранный вызывающей стороной. Старшие биты параметра flags должны быть сброшены (0).

Похожая функция bpf_redirect() обеспечивает перенаправление по ifindex, не требуя для этого отображения.

Функция возвращает XDP_REDIRECT при успехе или код завершения программы в двух младших битах параметра flags в случае ошибки.

bpf_sk_redirect_map

`long bpf_sk_redirect_map(struct sk_buff *skb, struct bpf_map *map, u32 key, u64 flags)`

Перенаправляет пакет в сокет, указанный map (типа BPF_MAP_TYPE_SOCKMAP), по индексу key. Для перенаправления можно использовать входные и выходные интерфейсы, значение BPF_F_INGRESS в поле flags позволяет различать их (наличие флага указывает входной путь, отсутствие - выходной). Другие флаги не поддерживаются.

Функция возвращает SK_PASS при успехе, SK_DROP в случае ошибки.

bpf_sock_map_update

`long bpf_sock_map_update(struct bpf_sock_ops *skops, struct bpf_map *map, void *key, u64 flags)`

Добавляет запись в отображение map, указывающее сокет, или обновляет его. Параметр skops используется в качестве нового значения записи, связанной с key. Поле flags принимает одно из указанных ниже значений.

BPF_NOEXIST
Запись для key должна присутствовать в map.

[1]Это реализовано пока лишь для естественного XDP (с поддержкой драйвера)

BPF_EXIST
Запись для key уже имеется в map.
BPF_ANY
Условия по наличию записи для key не заданы.
Если *map* имеет программы eBPF (parser и verdict), они будут наследоваться добавляемым сокетом. Если сокет уже связан с программами eBPF, это приведёт к ошибке.

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_xdp_adjust_meta

```
long bpf_xdp_adjust_meta(struct xdp_buff *xdp_md, int delta)
```
Изменяет адрес, указанный xdp_md->data_meta на величину delta (может быть положительной и отрицательной). Отметим, что эта операция меняет адрес, хранящийся в xdp_md->data, поэтому он должен загружаться только после вызова функции.

Использование xdp_md->data_meta необязательно и от программ не требуется использовать его. Причина этого заключается в том, что при обработке пакета в XDP (например, фильтром DoS) возможно вталкивание дополнительных метаданных до передачи в стек с гарантией того, что входная программа eBPF, связанная с классификатором TC на том же устройстве, может получить их для последующей постобработки. Поскольку TC работает с буферами сокетов, остаётся возможность установки из XDP указателей метки или приоритета, а также других указателей для буфера сокетов. Универсальность и программируемость этого пространства данных обеспечивает большую гибкость, поскольку пользователь может сохранять любые нужные ему метаданные.

Вызов этой функции может менять содержимое базового буфера пакетов, поэтому при загрузке все проверки указателей, выполненные ранее, становятся недействительными и должны быть повторены, если функция применяется в сочетании с прямым доступом к пакету.

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_perf_event_read_value

```
long bpf_perf_event_read_value(struct bpf_map *map, u64 flags, struct bpf_perf_event_value *buf, u32 buf_size)
```
Считывает значение счётчика событий perf и сохраняет его в buf размером buf_size. Эта функция основана на отображении map типа BPF_MAP_TYPE_PERF_EVENT_ARRAY. Характер счётчика событий perf выбирается при обновлении map с помощью файловых дескрипторов событий perf. Отображение map является массивом, размер которого определяется числом доступных CPU, и каждый элемент содержит значение для одного процессора. Извлекаемое значение указывается параметром flags, который содержит индекс интересующего CPU с маской BPF_F_INDEX_MASK. В поле flags можно поместить значение BPF_F_CURRENT_CPU, указывающее извлечение счётчика для текущего CPU.

Эта функция себя подобно bpf_perf_event_read(), но вместо возврата наблюдаемого значения помещает его в структуру buf. Это позволяет извлечь дополнительные данные, в частности, копируется время включения (enable) и работы (running) в buf->enabled и buf->running. В общем случае рекомендуется применять bpf_perf_event_read_value(), а не bpf_perf_event_read(), у которой имеются проблемы с ABI и меньше возможностей.

Получаемые значения представляют интерес, поскольку аппаратные счётчики PMU[1] имеют ограниченные ресурсы. Когда число открытых на основе PMU событий perf превышает число доступных счётчиков, ядро мультиплексирует эти события, чтобы каждому отдавалась некая доля времени PMU (но не всё). В случае мультиплексирования число выборок или значение счётчика не будут отражать ситуацию, которая была бы без мультиплексирования. Это осложняет сравнение разных запусков. Обычно значение счётчика следует нормализовать перед сравнением с другими экспериментами. Нормализация обычно имеет вид

```
normalized_counter = counter * t_enabled / t_running
```
где t_enabled - время, выделенное для события, а t_running - время, прошедшее для события с момента последней нормализации. Время включения и работы аккумулируется с момента события perf. Для масштабирования между двумя вызовами программы eBPF пользователи могут применять идентификатор CPU в качестве ключа (это типично для модели использования массива perf), чтобы запомнить прежнее значение и выполнить расчёт в программе eBPF.

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_perf_prog_read_value

```
long bpf_perf_prog_read_value(struct bpf_perf_event_data *ctx, struct bpf_perf_event_value *buf, u32 buf_size)
```
Для программы eBPF, присоединённой к событию perf, функция извлекает значение счётчика событий, связанных с ctx, и сохраняет его в структуре размера buf_size, указанной buf. Время включения и работы также сохраняются в структуре (см. bpf_perf_event_read_value().

Функция возвращает 0 при успешном выполнении и отрицательное значение при ошибке.

bpf_getsockopt

```
long bpf_getsockopt(void *bpf_socket, int level, int optname, void *optval, int optlen)
```
Эмулирует вызов getsockopt() на сокете, связанном с bpf_socket, который должен быть полным сокетом. Должен быть задан уровень level, на котором размещается опция, и имя optname для опции (см. man 2 getsockopt). Извлеченное значение сохраняется в структуре размера optlen, указанной opval.

Параметру bpf_socket следует иметь одно из указанных ниже значений.

- структура bpf_sock_ops для BPF_PROG_TYPE_SOCK_OPS.

[1]Performance Monitoring Unit - модуль мониторинга производительности.

- структура bpf_sock_ops для BPF_CGROUP_INET4_CONNECT и BPF_CGROUP_INET6_CONNECT.

Эта функция фактически реализует часть setsockopt() и поддерживает указанные ниже уровни.

IPPROTO_TCP
Поддерживает optname TCP_CONGESTION.
IPPROTO_IP
Поддерживает optname IP_TOS.
IPPROTO_IPV6
Поддерживает optname IPV6_TCLASS.
При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_override_return

`long bpf_override_return(struct pt_regs *regs, u64 rc)`
Функция служит для вставки ошибок и использует kprobe для замены возвращаемого значения проверяемой функции, устанавливая код rc. Первым аргументом является контекст regs, в котором работает kprobe.

Эта функция работает путём установки программного счётчика (program counter или PC) на функцию переопределения, которая работает вместо проверяемой функции, которая в результате просто не вызывается. Функция подмены возвращает запрошенное значение.

Эта функция влияет на безопасность, поэтому для неё заданы ограничения. Функция доступна лишь в ядре, собранном с опцией CONFIG_BPF_KPROBE_OVERRIDE, и работает лишь для функций, имеющих тег ALLOW_ERROR_INJECTION в коде ядра.

Функция доступна лишь для архитектуры, поддерживающей опцию CONFIG_FUNCTION_ERROR_INJECTION (на момент написания этого документа эту функцию поддерживала лишь архитектура x86.

Функция возвращает 0.

bpf_sock_ops_cb_flags_set

`long bpf_sock_ops_cb_flags_set(struct bpf_sock_ops *bpf_sock, int argval)`
Пытается установить в поле bpf_sock_ops_cb_flags полного сокета TCP, связанного с bpf_sock_ops значение argval.

Основное назначение этой функции — определять, следует ли вызывать программы eBPF типа BPF_PROG_TYPE_SOCK_OPS в разных точках кода TCP. Программа одного типа может при необходимости менять значение при каждом соединении, когда соединение уже организовано. Это поле доступно для чтения напрямую, но для обновления должна использоваться данная функция, чтобы возвращалась ошибка, если программа eBPF пытается организовать обратный вызов (callback), не поддерживаемый текущим ядром.

Параметр argval является набором флагов из числа указанных ниже.

- BPF_SOCK_OPS_RTO_CB_FLAG (тайм-аут повтора передачи).
- BPF_SOCK_OPS_RETRANS_CB_FLAG (повтор передачи).
- BPF_SOCK_OPS_STATE_CB_FLAG (смена состояния TCP).
- BPF_SOCK_OPS_RTT_CB_FLAG (в каждом интервале RTT).

Поэтому данную функцию можно применять для очистки флага callback путём сброса (0) соответствующего бита, например, для отключения обратного вызова RTO

`bpf_sock_ops_cb_flags_set(bpf_sock, bpf_sock->bpf_sock_ops_cb_flags & ~BPF_SOCK_OPS_RTO_CB_FLAG)`
Ниже приведено несколько примеров, где можно было бы вызвать такую программу eBPF.

- Срабатывание таймера RTO.
- Повторная передача пакета.
- Разрыв соединения.
- Передача пакета.
- Получение пакета.

Функция возвращает код -EINVAL, если сокет не является полным сокетом TCP, в иных случаях возвращается положительное число, содержащее биты, которые не удалось установить (0, если установлены все биты).

bpf_msg_redirect_map

`long bpf_msg_redirect_map(struct sk_msg_buff *msg, struct bpf_map *map, u32 key, u64 flags)`
Эта функция применяется в программах, реализующих правила на уровне сокета. Если сообщение msg можно пропустить (вердикт eBPF возвращает SK_PASS), оно перенаправляется в сокет, указанный map (типа BPF_MAP_TYPE_SOCKMAP) по индексу key. Для перенаправления можно использовать входные и выходные интерфейсы, наличие BPF_F_INGRESS в поле flags входной путь, отсутствие - выходной). Другие флаги не поддерживаются.

Функция возвращает SK_PASS при успехе, SK_DROP в случае ошибки.

bpf_msg_apply_bytes

`long bpf_msg_apply_bytes(struct sk_msg_buff *msg, u32 bytes)`
Для правил сокетов применяет вердикт программы eBPF к следующим bytes байтов сообщения msg.

Эту функцию можно применять, например, в следующих случаях.

- Один системный вызов sendmsg() или sendfile() содержит несколько логических сообщений, которые программа eBPF должна прочитать и вынести вердикт.
- Программе eBPF нужно прочитать лишь первые bytes байтов msg. Если содержимое сообщения велико, неоднократная организация и вызов программы eBPF для всех байтов ведут к значительным издержкам, даже если вердикт уже известен.

При вызове из программы eBPF функция устанавливает внутренний счётчик инфраструктуры BPF, который служит для применения последнего вердикта к следующим bytes байтов. Если значение bytes меньше текущего размера данных, обрабатываемых вызовом sendmsg() или sendfile(), первые bytes байтов будут переданы, а программа eBPF будет запушена снова с указателем на начало данных bytes + 1. Если bytes больше текущего размера обрабатываемых данных, вердикт eBPF будет применяться к нескольким вызовам sendmsg() или sendfile(), пока не будет обработано bytes байтов.

Если сокет закрывается с отличным от 0 внутренним счётчиком, это не создаёт проблемы, поскольку данные не буферизуются по bytes, а передаются по мере их получения.

Функция возвращает 0.

bpf_msg_cork_bytes

```
long bpf_msg_cork_bytes(struct sk_msg_buff *msg, u32 bytes)
```

Для правил сокетов функция предотвращает исполнение вердикта программы eBPF для сообщения msg, пока не будет накоплено bytes байтов.

Это можно использовать в случаях, когда требуется определённое число байтов для вынесения вердикта, даже если данные охватывают несколько вызовов sendmsg() или sendfile(). Крайним случаем будет повторяющийся вызов пользователем sendmsg() с 1-байтовыми сегментами сообщения. Очевидно, что это негативно влияет на производительность, но не запрещено. Если программе eBPF нужны bytes байтов для проверки заголовка, можно использовать эту функцию, чтобы программе eBPF не вызывалась снова, пока не будет накоплено bytes байтов.

Функция возвращает 0.

bpf_msg_pull_data

```
long bpf_msg_pull_data(struct sk_msg_buff *msg, u32 start, u32 end, u64 flags)
```

Для правил сокетов функция извлекает нелинейные данные из пользовательского пространства для msg, а также устанавливает указатели msg->data и msg->data_end на смещения start и end байтов в msg, respectively.

Если программа типа BPF_PROG_TYPE_SK_MSG запущена для msg, она может анализировать лишь данные, которые указатели (data, data_end) уже использовали. Для ловушек sendmsg() это будет, вероятно, первый элемент списка рассеяния (scatterlist). Однако для вызовов, полагающихся на обработчик sendpage (например, sendfile()), это будет диапазон (0, 0), поскольку данные используются совместно с пользовательским пространством и принятой по умолчанию целью является исключение возможности изменения данных пользователем, пока не будет принят вердикт eBPF. Эту функцию можно использовать для извлечения данных и установки указателей на начало и конец. Данные при необходимости будут копироваться (если они были нелинейны, а указатели начала и конца относятся к разным блокам).

Вызов этой функции может менять содержимое базового буфера пакетов, поэтому при загрузке все проверки указателей, выполненные ранее, становятся недействительными и должны быть повторены, если функция применяется в сочетании с прямым доступом к пакету.

Флаги зарезервированы на будущее и поле flags должно иметь значение 0.

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_bind

```
long bpf_bind(struct bpf_sock_addr *ctx, struct sockaddr *addr, int addr_len)
```

Привязывает сокет, указанный ctx к адресу addr размером addr_len. Это позволяет организовывать исходящие соединения с нужного адреса IP, что может быть полезно, например, когда всем процесса из cgroup следует использовать один адрес IP на хосте с несколькими адресами.

Функция работает с сокетами IPv4, IPv6, TCP, UDP, а поле addr->sa_family должно иметь значение AF_INET или AF_INET6. Рекомендуется указывать порт 0 (sin_port или sin6_port), что обеспечивает поведение в стиле IP_BIND_ADDRESS_NO_PORT и позволяет ядру эффективно выбирать свободный порт, пока квартет адресов и портов (4-tuple) уникален. Передача ненулевого значения может приводить к снижению производительности.

Функция возвращает 0 при успешном выполнении и отрицательное значение в случае ошибки.

bpf_xdp_adjust_tail

```
long bpf_xdp_adjust_tail(struct xdp_buff *xdp_md, int delta)
```

Смещает xdp_md->data_end на delta байтов. Функция позволяет увеличивать и сокращать «хвост» пакета. Для сокращения применяется отрицательное значение delta.

Вызов этой функции может менять содержимое базового буфера пакетов, поэтому при загрузке все проверки указателей, выполненные ранее, становятся недействительными и должны быть повторены, если функция применяется в сочетании с прямым доступом к пакету.

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_skb_get_xfrm_state

```
long bpf_skb_get_xfrm_state(struct sk_buff *skb, u32 index, struct bpf_xfrm_state *xfrm_state, u32 size, u64 flags)
```

Извлекает состояние XFRM (модель преобразований IP, см. man 8 ip-xfrm) с индексом index в «пути защиты» XFRM для skb. Извлеченное сохраняется в структуре bpf_xfrm_state размера size, указанной xfrm_state.

Флаги зарезервированы на будущее и поле flags должно иметь значение 0.

Функция доступна лишь для ядер, собранных с опцией CONFIG_XFRM.

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_get_stack

```
long bpf_get_stack(void *ctx, void *buf, u32 size, u64 flags)
```

Возвращает стек пользователя или ядра в предоставленный программой bpf буфер. Для этого функции нужен параметр ctx, указывающий контекст, в котором выполняется программа трассировки. Для сохранения трассировки стека программа bpf предоставляет буфер buf с неотрицательным размером size. Аргумент flags указывает число пропускаемых кадров стека (0 - 255) с маской BPF_F_SKIP_FIELD_MASK. В поле можно указать приведённые ниже флаги.

BPF_F_USER_STACK
Работать со стеком пользователя, а не ядра.

BPF_F_USER_BUILD_ID
Собирать buildid+offset вместо ips для пользовательского стека (применимо лишь с флагом BPF_F_USER_STACK).

Функция может собирать да PERF_MAX_STACK_DEPTH кадров из стека пользователя или ядра при наличии достаточно большого буфера. Этим ограничением может управлять программа sysctl и задавать предел вручную для работы с длинным пользовательским стеком (например, в программах Java). Команда установки значения имеет вид

```
# sysctl kernel.perf_event_max_stack=<new value>
```

Функция возвращает неотрицательное значение, не превышающее size, при успешном выполнении и отрицательное значение в случае ошибки.

bpf_skb_load_bytes_relative

```
long bpf_skb_load_bytes_relative(const void *skb, u32 offset, void *to, u32 len, u32 start_header)
```

Эта функция похожа на bpf_skb_load_bytes() и обеспечивает простой способ загрузить len байтов со смещением offset из пакета, связанного с skb, в буфер, указанный to. Отличие от bpf_skb_load_bytes() состоит в использовании пятого аргумента start_header, служащего для указания базового смещения для начала отсчёта. Параметр start_header может принимать одно из указанных ниже значений.

BPF_HDR_START_MAC
Базой смещения для загрузки данных является заголовок канального уровня в skb.

BPF_HDR_START_NET
Базой смещения для загрузки данных является заголовок сетевого уровня в skb.

В общем случае прямой доступ к пакету является предпочтительным методом доступа к данным, однако эта функция полезна, в частности, в фильтрах сокетов, где skb->data не всегда указывает начало заголовка MAC и прямой доступ к пакету невозможен.

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_fib_lookup

```
long bpf_fib_lookup(void *ctx, struct bpf_fib_lookup *params, int plen, u32 flags)
```

Выполняет поиск в FIB таблиц ядра с использованием параметров params. Если поиск удался и результат указывает, что кадр пересылается, выполняется поиск в таблице соседей для определения следующего узла (nexthop). При успехе (поиск в FIB указал пересылку и nexthop найден) в структуре bpf_fib_lookup возвращается адрес nexthop в ipv4_dst или ipv6_dst в зависимости от семейства адресов, smac указывает MAC-адрес выходного устройства, dmac - MAC-адрес nexthop, rt_metric - метрику маршрута (только для IPv4 и IPv6), а ifindex указывает индекс устройства nexthop из поиска в FIB. Аргумент plen указывает размер данных, передаваемых в структуре, а поле flags может включать 1 или 2 показанных ниже флага.

BPF_FIB_LOOKUP_DIRECT
Выполнять прямой поиск в таблице вместо полного поиска с применением правил FIB.

BPF_FIB_LOOKUP_OUTPUT
Выполнять поиск с точки зрения выхода (по умолчанию ищется с точки зрения входа).

Параметр ctx содержит структуру xdp_md для программ XDP или sk_buff для программ tc cls_act.

Функция возвращает отрицательное значение, если любой из аргументов не пригоден, 0 при успехе (пакет пересылается, nexthop существует) или один из кодов BPF_FIB_LKUP_RET_, указывающих, почему пакет не был переслан и нужно содействие всего стека.

bpf_sock_hash_update

```
long bpf_sock_hash_update(struct bpf_sock_ops *skops, struct bpf_map *map, void *key, u64 flags)
```

Добавляет или обновляет отображение sockhash, указывающее сокеты. Параметр skops служит новым значением записи, связанной с ключом key. Поле flags содержит одно из указанных ниже значений.

BPF_NOEXIST
Запись для key должна присутствовать в map.

BPF_EXIST
Запись для key уже имеется в map.

BPF_ANY
Условия по наличию записи для key не заданы.

Если map имеет программы eBPF (parser и verdict), они будут наследоваться добавляемым сокетом. Если сокет уже связан с программами eBPF, это приведёт к ошибке.

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_msg_redirect_hash

long bpf_msg_redirect_hash(struct sk_msg_buff *msg, struct bpf_map *map, void *key, u64 flags)

Эта функция применяется в программах, реализующих правила на уровне сокета. Если сообщение msg можно пропустить (вердикт eBPF возвращает SK_PASS), оно перенаправляется в сокет, указанный map (типа BPF_MAP_TYPE_SOCKHASH) по индексу key. Для перенаправления можно использовать входные и выходные интерфейсы, наличие BPF_F_INGRESS в поле flags входной путь, отсутствие - выходной). Другие флаги не поддерживаются.

Функция возвращает SK_PASS при успехе, SK_DROP в случае ошибки.

bpf_sk_redirect_hash

long bpf_sk_redirect_hash(struct sk_buff *skb, struct bpf_map *map, void *key, u64 flags)

Эта функция применяется в программах, реализующих правила на skb уровня сокета. Если skb можно пропустить (вердикт eBPF возвращает SK_PASS), буфер перенаправляется в сокет, указанный map (типа BPF_MAP_TYPE_SOCKHASH) по индексу key. Для перенаправления можно использовать входные и выходные интерфейсы, наличие BPF_F_INGRESS в поле flags входной путь, отсутствие - выходной). Другие флаги не поддерживаются.

Функция возвращает SK_PASS при успехе, SK_DROP в случае ошибки.

bpf_lwt_push_encap

long bpf_lwt_push_encap(struct sk_buff *skb, u32 type, void *hdr, u32 len)

Инкапсулирует пакет, связанный с skb, в протокол L3 с использованием заголовка из буфера по адресу hdr размером len байтов. Параметр type указывает протокол и может принимать одно из указанных ниже значений.

BPF_LWT_ENCAP_SEG6
Инкапсуляция IPv6 с использованием заголовка SRH[1] (struct ipv6_sr_hdr). Буфер hdr содержит лишь заголовок SRH, а заголовок IPv6 добавляет ядро.

BPF_LWT_ENCAP_SEG6_INLINE
Работает лишь с skb, содержащим пакет IPv6, и вставляет заголовок SRH (struct ipv6_sr_hdr) внутрь заголовка IPv6.

BPF_LWT_ENCAP_IP
Инкапсуляция IP (GRE, GUE, IPIP и т. п.). Внешним заголовком должен быть заголовок IPv4 или IPv6, за которым могут следовать дополнительные заголовки с общим размером всех добавляемых впереди заголовков до LWT_BPF_MAX_HEADROOM байтов. Отметим, что при skb_is_gso(skb) = true можно добавить не более 2 заголовков и при наличии внутреннего заголовка это должен быть заголовок GRE или UDP/GUE.

Типы BPF_LWT_ENCAP_SEG6* могут использовать программы BPF типа BPF_PROG_TYPE_LWT_IN, а тип BPF_LWT_ENCAP_IP - программы типов BPF_PROG_TYPE_LWT_IN и BPF_PROG_TYPE_LWT_XMIT.

Вызов этой функции может менять содержимое базового буфера пакетов, поэтому при загрузке все проверки указателей, выполненные ранее, становятся недействительными и должны быть повторены, если функция применяется в сочетании с прямым доступом к пакету.

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_lwt_seg6_store_bytes

long bpf_lwt_seg6_store_bytes(struct sk_buff *skb, u32 offset, const void *from, u32 len)

Сохраняет len байтов из адреса from в пакете, связанном с skb, по смещению offset. Эта функция может применяться лишь для изменения флагов, тега и TLV во внешнем заголовке IPv6 SRH.

Вызов этой функции может менять содержимое базового буфера пакетов, поэтому при загрузке все проверки указателей, выполненные ранее, становятся недействительными и должны быть повторены, если функция применяется в сочетании с прямым доступом к пакету.

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_lwt_seg6_adjust_srh

long bpf_lwt_seg6_adjust_srh(struct sk_buff *skb, u32 offset, s32 delta)

Корректирует размер, выделенный для TLV во внешнем заголовке IPv6 SRH пакета, связанного с skb, в позиции offset на delta байтов. Принимаются лишь смещения после сегментов, а значение delta может быть положительным (расширение) или отрицательным (сокращение).

Вызов этой функции может менять содержимое базового буфера пакетов, поэтому при загрузке все проверки указателей, выполненные ранее, становятся недействительными и должны быть повторены, если функция применяется в сочетании с прямым доступом к пакету.

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

[1]Segment Routing Header - заголовок маршрутизации по сегментам.

bpf_lwt_seg6_action

```
long bpf_lwt_seg6_action(struct sk_buff *skb, u32 action, void *param, u32 param_len)
```

Применяет действие IPv6 Segment Routing типа action к пакету, связанному с skb. Каждое действие содержит параметры размером param_len байтов, размещённые по адресу param. Действием может быть 1 из указанных ниже.

SEG6_LOCAL_ACTION_END_X
End.X - конечная точка с L3 cross-connect, param указывает структуру in6_addr.

SEG6_LOCAL_ACTION_END_T
End.T - конечная точка с конкретным результатом поиска в таблице IPv6, param указывает int.

SEG6_LOCAL_ACTION_END_B6
End.B6 - конечная точка привязана к правилу SRv6, param указывает структуру ipv6_sr_hdr.

SEG6_LOCAL_ACTION_END_B6_ENCAP
End.B6.Encap - конечная точка привязана к правилу инкапсуляции SRv6, param указывает структуру ipv6_sr_hdr.

Вызов этой функции может менять содержимое базового буфера пакетов, поэтому при загрузке все проверки указателей, выполненные ранее, становятся недействительными и должны быть повторены, если функция применяется в сочетании с прямым доступом к пакету.

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_rc_repeat

```
long bpf_rc_repeat(void *ctx)
```

Эта функция применяется в программах, реализующих декодирование IR[1], для информирования об успешном декодировании повторяющегося сообщения от кнопки (key). Это задерживает генерацию события key up (отпускание кнопки) для созданного ранее события key down (нажатие кнопки).

Некоторые протоколы IR, такие как NEC, включают специальное сообщение IR для повтора последнего нажатия кнопки, когда кнопка удерживается.

Параметру ctx следует указывать на выборку, переданную в программу.

Функция работает только с ядрами, собранными с опцией конфигурации CONFIG_BPF_LIRC_MODE2 = y.

Функция возвращает 0.

bpf_rc_keydown

```
long bpf_rc_keydown(void *ctx, u32 protocol, u64 scancode, u32 toggle)
```

Эта функция применяется в программах, реализующих декодирование IR, для информирования об успешном декодировании нажатия кнопки со скан-кодом scancode, имеющей значение toggle в данном протоколе protocol. Скан-код транслируется в код кнопки с помощью отображения rc и сообщаться на вход как нажатие кнопки. По истечении определённого времени генерируется событие key up. Этот период можно расширить повторным вызовом bpf_rc_keydown() с теми же значениями или вызовом bpf_rc_repeat().

Некоторые протоколы используют бит переключения (toggle) на случай, когда кнопка была отпущена и нажата снова в интервале между сканированием.

В параметре ctx следует указывать выборку lirc, переданную в программу, protocol - декодированный номер протокола (см. предопределённые значения в enum rc_proto).

Функция работает только с ядрами, собранными с опцией конфигурации CONFIG_BPF_LIRC_MODE2 = y.

Функция возвращает 0.

bpf_skb_cgroup_id

```
u64 bpf_skb_cgroup_id(struct sk_buff *skb)
```

Возвращает идентификатор cgroup v2 для сокета, связанного с skb. Функция похожа на bpf_get_cgroup_classid() для cgroup v1, предоставляя идентификатор, которых может применяться для сопоставления или поиска в отображении, например, для реализации правил. Идентификатор cgroup v2 данного пути в иерархии раскрывается в пользовательское пространство через f_handle API, чтобы получить тот же 64-битовый идентификатор.

Эту функцию можно применять на выходном пути TC, но она непригодна для входного пути. Функция доступна лишь для ядер, собранных с конфигурационной опцией CONFIG_SOCK_CGROUP_DATA.

Функция возвращает значение идентификатор или 0, если идентификатор не найден.

bpf_get_current_cgroup_id

```
u64 bpf_get_current_cgroup_id(void)
```

Функция возвращает 64-битовое целое число, содержащее текущий идентификатор cgroup с которой работает текущая задача.

bpf_get_local_storage

```
void *bpf_get_local_storage(void *map, u64 flags)
```

Возвращает указатель на локальную область хранения, тип и размер которой задаёт аргумент map. Назначение параметра flags зависит от типа отображения и для локального хранилища должно иметь значение 0.

В зависимости от типа программы BPF локальное хранилище может совместно использоваться несколькими экземплярами программы BPF, работающими одновременно. Пользователю следует озаботиться вопросами синхронизации, например, используя инструкцию BPF_STX_XADD для изменения общих данных.

[1] Infrared - инфракрасный.

bpf_sk_select_reuseport

```
long bpf_sk_select_reuseport(struct sk_reuseport_md *reuse, struct bpf_map *map, void *key, u64 flags)
```

Выбирает сокет SO_REUSEPORT из отображения map BPF_MAP_TYPE_REUSEPORT_ARRAY, проверяя его на соответствие входящему запросу.

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_skb_ancestor_cgroup_id

```
u64 bpf_skb_ancestor_cgroup_id(struct sk_buff *skb, int ancestor_level)
```

Возвращает идентификатор cgroup v2, являющийся предком cgroup, связанной с skb на уровне ancestor_level. Корнем cgroup является ancestor_level = 0 и каждый шаг вниз по иерархии инкрементирует уровень. Если ancestor_level совпадает с уровнем cgroup, связанной с skb, возвращаемое значение будет совпадать с результатом bpf_skb_cgroup_id().

Функция полезна для реализации правил, основанных на cgroup, которые в иерархии выше cgroup, связанной с skb.

Возвращаемый формат и ограничения функции совпадают с заданными для bpf_skb_cgroup_id().

Функция возвращает идентификатор или 0, если идентификатор не найден.

bpf_sk_lookup_tcp

```
struct bpf_sock *bpf_sk_lookup_tcp(void *ctx, struct bpf_sock_tuple *tuple, u32 tuple_size, u64 netns, u64 flags)
```

Отыскивает сокет TCP, соответствующий tuple, с возможностью указать также дочернее простанство имён сети netns. Найденное значение должно проверяться и в случае непустого (не NULL) значения освобождаться через bpf_sk_release(). Параметру ctx следует указывать контекст программы, такой как skb или socket (в зависимости от применяемой ловушки). Это позволяет определить базовое пространство сетевых имён для поиска. Параметр tuple_size должен принимать значение sizeof(tuple->ipv4) для поиска сокета IPv4 или sizeof(tuple->ipv6) для сокета IPv6.

Если параметр netns является отрицательным 32-битовым целым числом со знаком, для поиска используется таблица в netns, связанном с ctx. Для ловушек TC это netns устройства в skb, для ловушек сокетов - netns сокета. Иные 32-битовые значения netns указывают идентификатор netns относительно пространства netns, связанного с ctx. Значения netns, выходящие за рамки 32 битов, зарезервированы на будущее. Все значения флагов являются резервными и поле flags должно иметь значение 0.

Функция доступна только для ядер, собранных с опцией CONFIG_NET.

Функция возвращает указатель на структуру bpf_sock или NULL в случае ошибки. Для сокетов с опцией reuseport структура bpf_sock берётся из reuse->socks[] с использованием хэш-значения кортежа адресов и портов (tuple).

bpf_sk_lookup_udp

```
struct bpf_sock *bpf_sk_lookup_udp(void *ctx, struct bpf_sock_tuple *tuple, u32 tuple_size, u64 netns, u64 flags)
```

Отыскивает сокет UDP, соответствующий tuple, с возможностью указать также дочернее простанство имён сети netns. Найденное значение должно проверяться и в случае непустого (не NULL) значения освобождаться через bpf_sk_release(). Параметру ctx следует указывать контекст программы, такой как skb или socket (в зависимости от применяемой ловушки). Это позволяет определить базовое пространство сетевых имён для поиска. Параметр tuple_size должен принимать значение sizeof(tuple->ipv4) для поиска сокета IPv4 или sizeof(tuple->ipv6) для сокета IPv6.

Если параметр netns является отрицательным 32-битовым целым числом со знаком, для поиска используется таблица в netns, связанном с ctx. Для ловушек TC это netns устройства в skb, для ловушек сокетов - netns сокета. Иные 32-битовые значения netns указывают идентификатор netns относительно пространства netns, связанного с ctx. Значения netns, выходящие за рамки 32 битов, зарезервированы на будущее. Все значения флагов являются резервными и поле flags должно иметь значение 0.

Функция доступна только для ядер, собранных с опцией CONFIG_NET.

Функция возвращает указатель на структуру bpf_sock или NULL в случае ошибки. Для сокетов с опцией reuseport структура bpf_sock берётся из reuse->socks[] с использованием хэш-значения кортежа адресов и портов (tuple).

bpf_sk_release

```
long bpf_sk_release(struct bpf_sock *sock)
```

Освобождает ссылку, указанную параметром sock, который должен быть ненулевым (не NULL) указателем, возвращенным bpf_sk_lookup_xxx().

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_map_push_elem

```
long bpf_map_push_elem(struct bpf_map *map, const void *value, u64 flags)
```

Вталкивает значение элемента value в отображение map. Поле flags содержит BPF_EXIST (если очередь или стек заполнены, самый старый элемент удаляется для освобождения места).

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_map_pop_elem

```
long bpf_map_pop_elem(struct bpf_map *map, void *value)
```

Выталкивает элемент из отображения map.

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_map_peek_elem

```
long bpf_map_peek_elem(struct bpf_map *map, void *value)
```

Извлекает элемент отображения map, не удаляя его.

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_msg_push_data

```
long bpf_msg_push_data(struct sk_msg_buff *msg, u32 start, u32 len, u64 flags)
```

Для правил сокета вставляет len байтов в msg по смещению start. Программа типа BPF_PROG_TYPE_SK_MSG, работающая с msg, может захотеть добавить туда метаданные или опции, которые позднее можно прочитать и использовать в любой ловушке BPF нижележащего уровня.

Эта функция может вызывать ошибку при незватке памяти (отказ fails) и программа BPF будет получать соответствующее сообщение, которое она должна обработать.

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_msg_pop_data

```
long bpf_msg_pop_data(struct sk_msg_buff *msg, u32 start, u32 len, u64 flags)
```

Функция удаляет len из msg, начиная с байта start. Это может вызывать ошибку ENOMEM, если требуется выделение и копирование по причине заполнения кольцевого буфера. Однако функция будет пытаться избежать выделения памяти, если это возможно. Другие ошибки могут возникать, если непригодны входные параметры, например, байт start не является действительной частью данных msg payload или выталкиваемое значение слишком велико.

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_rc_pointer_rel

```
long bpf_rc_pointer_rel(void *ctx, s32 rel_x, s32 rel_y)
```

Функция применяется в программах, реализующих декодирование IR, для информирования об успешном декодировании перемещения указателя. Параметру ctx следует указывать выборку lirc, передаваемую в программу.

Функция доступна лишь для ядер, собранных с опцией конфигурации CONFIG_BPF_LIRC_MODE2 = y.

Функция возвращает 0.

bpf_spin_lock

```
long bpf_spin_lock(struct bpf_spin_lock *lock)
```

Устанавливает блокировку (spinlock), представленную указателем lock, которая хранится как часть значения отображения. Снятие блокировки позволяет без опаски изменить остальные поля в этом значении. Блокировка затем может (и должна) быть снята вызовом функции bpf_spin_unlock(lock).

С блокировками в программах BPF связано множество ограничений, указанных ниже.

- Объекты bpf_spin_lock могут находиться лишь внутри отображений типов BPF_MAP_TYPE_HASH и BPF_MAP_TYPE_ARRAY (список может быть расширен в будущем).
- Обязательно описание отображения в формате BTF.
- Программа BPF может выполнять в каждый момент времени лишь 1 блокировку, поскольку наличие нескольких может приводить к зависанию (dead lock).
- На элемент отображения разрешается лишь 1 структура bpf_spin_lock.
- При снятой блокировке вызовы (из BPF в BPF или функцию-помощник) не разрешены.
- Вызовы инструкций BPF_LD_ABS и BPF_LD_IND не разрешены внутри области с блокировкой.
- Программа BPF **должна** вызывать функцию bpf_spin_unlock() для снятия блокировки на всех путях вызова до завершения своей работы.
- Программа BPF может обращаться к структуре bpf_spin_lock только через функции bpf_spin_lock() и bpf_spin_unlock(). Загрузка или сохранение данных с поле lock структуры bpf_spin_lock в отображении не допускается.
- Для использования функции bpf_spin_lock() BTF-описание значения отображения должно быть структурой, включающей поле anyname на верхнем уровне. Вложенная блокировка внутри другой структуры не разрешена.
- Поле lock структуры bpf_spin_lock в отображении должно быть выровнено по границе 4 байтов в значении.
- Вызов BPF_MAP_LOOKUP_ELEM не копирует поле bpf_spin_lock в пользовательское пространство.
- Вызов BPF_MAP_UPDATE_ELEM или обновление из программы BPF не обновляет поле bpf_spin_lock.
- bpf_spin_lock не может размещаться в стеке или внутри сетевого пакета (только в значениях отображений).
- bpf_spin_lock доступна только пользователю root.
- Программы трассировки и фильтров сокета не могут применять bpf_spin_lock() из-за недостаточных проверок вытеснения (в будущем это может быть разрешено).
- bpf_spin_lock не разрешается во внутренних (вложенных) отображениях (map-in-map).

Функция возвращает 0.

bpf_spin_unlock

```
long bpf_spin_unlock(struct bpf_spin_lock *lock)
```

Снимает блокировку lock, установленную функций bpf_spin_lock(lock).

Функция возвращает 0.

bpf_sk_fullsock

```
struct bpf_sock *bpf_sk_fullsock(struct bpf_sock *sk)
```

Получает указатель на структуру bpf_sock для обеспечения возможности доступа ко всем её полям.

Функция возвращает указатель на структуру bpf_sock при успехе и NULL в случае ошибки.

bpf_tcp_sock

```
struct bpf_tcp_sock *bpf_tcp_sock(struct bpf_sock *sk)
```

Получает указатель на структуру bpf_tcp_sock по указателю на структуру bpf_sock.

Функция возвращает указатель на структуру bpf_tcp_sock при успехе и NULL в случае ошибки.

bpf_skb_ecn_set_ce

```
long bpf_skb_ecn_set_ce(struct sk_buff *skb)
```

Устанавливает в поле ECN заголовка IP значение CE[1], если текущим значением является ECT[2]. В ином случае не делает ничего. Функция работает с IPv6 и IPv4.

Функция возвращает 1, если флаг CE установлен или уже был в заголовке, 0 - в ином случае.

bpf_get_listener_sock

```
struct bpf_sock *bpf_get_listener_sock(struct bpf_sock *sk)
```

Возвращает указатель на структуру bpf_sock в состоянии TCP_LISTEN. Вызов bpf_sk_release() не нужен и не разрешен.

Функция возвращает на структуру bpf_sock при успехе и NULL в случае ошибки.

bpf_skc_lookup_tcp

```
struct bpf_sock *bpf_skc_lookup_tcp(void *ctx, struct bpf_sock_tuple *tuple, u32 tuple_size, u64 netns, u64 flags)
```

Отыскивает сокет TCP, соответствующий tuple, с возможностью указать также дочернее простанство имён сети netns. Найденное значение должно проверяться и в случае непустого (не NULL) значения освобождаться через bpf_sk_release().

Эта функция идентична bpf_sk_lookup_tcp(), но возвращает также сокеты timewait и request. Для доступа к полной структуре служат функции bpf_sk_fullsock() и bpf_tcp_sock().

Функция доступна только для ядер, собранных с опцией CONFIG_NET.

Функция возвращает указатель на структуру bpf_sock или NULL в случае ошибки. Для сокетов с опцией reuseport структура bpf_sock берётся из reuse->socks[] с использованием хэш-значения кортежа адресов и портов (tuple).

bpf_tcp_check_syncookie

```
long bpf_tcp_check_syncookie(struct bpf_sock *sk, void *iph, u32 iph_len, struct tcphdr *th, u32 th_len)
```

Проверяет, содержится ли в iph и th действительное подтверждение SYN cookie ACK для слушающего сокета в sk. Параметр iph указывает начало заголовка IPv4 или IPv6, а iph_len содержит sizeof(struct iphdr) или (struct ip6hdr). Параметр th указывает начало заголовка TCP, а th_len содержит sizeof(struct tcphdr).

Функция возвращает 0, если iph и th содержат действительное подтверждение SYN cookie ACK, и отрицательное значение ошибки в ином случае.

bpf_sysctl_get_name

```
long bpf_sysctl_get_name(struct bpf_sysctl *ctx, char *buf, size_t buf_len, u64 flags)
```

Получает имя sysctl в /proc/sys/ и копирует его в предоставленный программой буфер buf размером buf_len. Если размер буфера отличается от 0, он всегда завершается NUL-символом. Если flags = 0, копируется полное имя (например, net/ipv4/tcp_mem), а значение BPF_F_SYSCTL_BASE_NAME позволяет копировать лишь базовое имя (например, tcp_mem).

Функция возвращает число скопированных символов (без учёта завершающего NUL) или -E2BIG, если буфер оказался мал (он все равно будет включать часть имени).

bpf_sysctl_get_current_value

```
long bpf_sysctl_get_current_value(struct bpf_sysctl *ctx, char *buf, size_t buf_len)
```

Получает текущее значение sysctl, как оно представлено в /proc/sys (включая перевод строки и т. п.), и копирует его целиком в предоставленный программой буфер buf размером buf_len. Если размер буфера отличается от 0, он всегда завершается NUL-символом.

Функция возвращает число скопированных символов (без учёта завершающего NUL), -E2BIG, если буфер оказался мал (он все равно будет включать часть имени), или -EINVAL, если текущее значение было недоступно, например потому, что sysctl не инициализирована и чтение возвратило -EIO.

[1]Congestion Encountered - наблюдается перегрузка.
[2]ECN Capable Transport - транспорт с поддержкой ECN.

bpf_sysctl_get_new_value

```
long bpf_sysctl_get_new_value(struct bpf_sysctl *ctx, char *buf, size_t buf_len)
```

Получает новое значение, записываемое пользовательским пространством в sysctl (до фактической записи), и копирует его как строку в предоставленный программой буфер buf размером buf_len. Пользовательское пространство может записать новое значение не в начало файла.

Если размер буфера отличается от 0, он всегда завершается NUL-символом.

Функция возвращает число скопированных символов (без учёта завершающего NUL), -E2BIG, если буфер оказался мал (он все равно будет включать часть имени), или -EINVAL, если sysctl читается.

bpf_sysctl_set_new_value

```
long bpf_sysctl_set_new_value(struct bpf_sysctl *ctx, const char *buf, size_t buf_len)
```

Переопределяет новое значение, записываемое пользовательским пространством в sysctl, значением, предоставленным программой в буфере buf размером buf_len. Содержимому buf следует быть строкой такой же формы, какую пользовательское пространство предоставило для записи в sysctl. Пользовательское пространство может записывать в файл не с начала. Для переопределения всего значения sysctl следует установить в файле позицию 0.

Функция возвращает 0 при успехе, -E2BIG, если значение buf_len слишком велико, и -EINVAL, если sysctl читается.

bpf_strtol

```
long bpf_strtol(const char *buf, size_t buf_len, u64 flags, long *res)
```

Преобразует начальную часть строки из буфера buf размером buf_len в длинное целое число с заданной базой и сохраняет результат в res. Строка может содержать в начале произвольное число пробелов (см. man 3 isspace), за которыми может следовать 1 символ '-'. Пять младших битов параметра flags указывают базу для числа, атальные биты флагов не используются. База должна иметь значение 8, 10, 16 или 0, чтобы определять её автоматически, подобно strtol в пользовательском пространстве.

Функция возвращает число извлеченных символов, которое должно быть положительным и не больше buf_len. Если не были найдены пригодные цифры или представлена неподдерживаемая база, возвращается ошибка -EINVAL, если значение выходит за пределы диапазона - -ERANGE.

bpf_strtoul

```
long bpf_strtoul(const char *buf, size_t buf_len, u64 flags, unsigned long *res)
```

Преобразует начальную часть строки из буфера buf размером buf_len в длинное целое число без знака с заданной базой и сохраняет результат в res. Строка может содержать в начале произвольное число пробелов (см. man 3 isspace). Пять младших битов параметра flags указывают базу для числа, атальные биты флагов не используются. База должна иметь значение 8, 10, 16 или 0, чтобы определять её автоматически, подобно strtoul в пользовательском пространстве.

Функция возвращает число извлеченных символов, которое должно быть положительным и не больше buf_len. Если не были найдены пригодные цифры или представлена неподдерживаемая база, возвращается ошибка -EINVAL, если значение выходит за пределы диапазона - -ERANGE.

bpf_sk_storage_get

```
void *bpf_sk_storage_get(struct bpf_map *map, struct bpf_sock *sk, void *value, u64 flags)
```

Получает локальное хранилище bpf (bpf-local-storage) из sk.

Логически это можно считать получением значения из map с sk в качестве ключа. С этой точки зрения действие функции не сильно отличается от bpf_map_lookup_elem(map, &sk), но данная функция требует, чтобы ключ был полным сокетом, а отображение жолжно иметь тип BPF_MAP_TYPE_SK_STORAGE.

Значение сохраняется локально в sk, а не в map. Отображение map используется как «тип» bpf-local-storage. По этому типу (т. е. map) выполняется поиск среди локальных хранилишщ, содержащихся в sk.

Может использоваться необязательное поле flags (BPF_SK_STORAGE_GET_F_CREATE) для создания нового bpf-local-storage, если хранилища ещё нет. Параметр value может использоваться вместе с BPF_SK_STORAGE_GET_F_CREATE для задания начального значения bpf-local-storage. При value = NULL новое хранилише bpf-local-storage инициализируется нулем.

Функция возвращает A bpf-local-storage pointer is returned on success. NULL if not found or there was an error in adding a new bpf-local-storage.

bpf_sk_storage_delete

```
long bpf_sk_storage_delete(struct bpf_map *map, struct bpf_sock *sk)
```

Удаляет bpf-local-storage из sk.

Функция возвращает 0 при успеше и -ENOENT, если хранилище bpf-local-storage не найдено.

bpf_send_signal

```
long bpf_send_signal(u32 sig)
```

Передаёт сигнал sig процессу текущей задачи. Сигнал может доставляться любому потоку (thread) процесса.

Функция возвращает 0 при успешном выполнении или постановке в очередь. Ошибка -EBUSY возвращается, если рабочая очередь под nmi заполнена, -EINVAL при непригодности sig, -EPERM, если нет полномочий на отправку sig, -EAGAIN, если программа bpf может повторить попытку.

bpf_tcp_gen_syncookie

```
s64 bpf_tcp_gen_syncookie(struct bpf_sock *sk, void *iph, u32 iph_len, struct tcphdr *th, u32 th_len)
```

Пытается выдать SYN cookie для пакета с соответствующими заголовками IP/TCP (iph и th), на слушающем сокете в sk. Параметр iph указывает начало заголовка IPv4 или IPv6, а iph_len содержит sizeof(struct iphdr) или sizeof(struct ip6hdr). Параметр th указывает начало заголовка TCP, а th_len - его размер.

При успешном выполнении функция возвращает в 32 младших битах созданное значение SYN cookie, в следующих 16 - значение MSS для этого cookie, а последние 16 битов не используются. При отказе возвращается одно из приведённых ниже значений.

- -EINVAL - SYN cookie не удалось выдать из-за ошибки.
- -ENOENT - SYN cookie не следует выдавать (нет SYN flood).
- -EOPNOTSUPP - конфигурация ядра не разрешает SYN cookie.
- -EPROTONOSUPPORT - версия IP в пакет отличается от 4 и 6.

bpf_skb_output

```
long bpf_skb_output(void *ctx, struct bpf_map *map, u64 flags, void *data, u64 size)
```

Записывает необработанные данные data (blob) в специальное событие BPF, удерживаемое отображением map типа BPF_MAP_TYPE_PERF_EVENT_ARRAY. Это собтие должно имет атрибут PERF_SAMPLE_RAW как sample_type, PERF_TYPE_SOFTWARE как type и PERF_COUNT_SW_BPF_OUTPUT как config.

Параметр flags служит для указания индекса в map, по которому нужно поместить значение, с применением маски BPF_F_INDEX_MASK. Как вариант, можно установить BPF_F_CURRENT_CPU, чтобы указать использование индекса текущего CPU. Значение для записи (размером size) передаётся через стек eBPF и указывается параметром data. Параметр ctx является указателем на структуру sk_buff в ядре.

Эта функция похожа на bpf_perf_event_output(), но ограничена программами BPF raw_tracepoint.

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_probe_read_user

```
long bpf_probe_read_user(void *dst, u32 size, const void *unsafe_ptr)
```

Пытается безопасно считать size байтов по адресу unsafe_ptr в пользовательском пространстве и сохранить их в dst.

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_probe_read_kernel

```
long bpf_probe_read_kernel(void *dst, u32 size, const void *unsafe_ptr)
```

Пытается безопасно считать size байтов по адресу unsafe_ptr в пространстве ядра и сохранить их в dst.

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_probe_read_user_str

```
long bpf_probe_read_user_str(void *dst, u32 size, const void *unsafe_ptr)
```

Копирует строку с NUL-символом в конце с небезопасного пользовательского адреса unsafe_ptr в dst. В параметре size следует учитывать завершающий NUL-байт. Если размер строки меньше size, целевой объект не дополняется байтами NUL. Если размер строки больше size, из неё копируется size-1 байтов и к ним добавляется в конце NUL-байт.

При успешном выполнении возвращается размер скопированной строки, что делает эту функцию полезной в программах трассировки для чтения строк и, что более важно, получения их размера в процессе работы. Пример приведён ниже.

```
SEC("kprobe/sys_open")
void bpf_sys_open(struct pt_regs *ctx)
{
        char buf[PATHLEN]; // Для PATHLEN задано значение 256
        int res = bpf_probe_read_user_str(buf, sizeof(buf), ctx->di
        // Извлекается buf, например, переносом в пользовательское
        // пространство через bpf_perf_event_output(); результат res
        // (размер строки) может применяться как размер события
        // после проверки его границ.
}
```

При использовании bpf_probe_read_user() потребовалось бы указывать размер строки во время компиляции, что часто приводило бы к копированию излишнего объёма памяти.

Другим полезным вариантом применения является анализ отдельных аргументов процесса или переменных среды при перемещении по current->mm->arg_start и current->mm->env_start. Используя значение этой функции можно быстро выполнить операцию с правильным смещением области памяти.

Функция возвращает при успехе размер строки с учётом символа NUL или отрицательное значение при ошибке.

bpf_probe_read_kernel_str

```
long bpf_probe_read_kernel_str(void *dst, u32 size, const void *unsafe_ptr)
```

Копирует строку с NUL-символом в конце с небезопасного адреса ядра unsafe_ptr в dst. Семантика этой функции не отличается от семантики bpf_probe_read_user_str().

Функция возвращает при успехе размер строки с учётом символа NUL или отрицательное значение при ошибке.

bpf_tcp_send_ack

```
long bpf_tcp_send_ack(void *tp, u32 rcv_nxt)
```

Передаёт TCP ACK (tcp-ack). Параметр tp указывает структуру ядра tcp_sock, rcv_nxt - номер ack_seq для отправки.

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_send_signal_thread

```
long bpf_send_signal_thread(u32 sig)
```

Передаёт сигнал sig для потока, связанного с текущей задачей.

Функция возвращает 0 при успешной передаче или постановке в очередь. Ошибка -EBUSY возвращается, если очередь под nmi заполнена, -EINVAL, если сигнал sig недействителен, -EPERM, если нет полномочий на отправку sig, и -EAGAIN, если программа bpf может повторить попытку.

bpf_jiffies64

```
u64 bpf_jiffies64(void)
```

Возвращает 64-битовое число jiffy.

bpf_read_branch_records

```
long bpf_read_branch_records(struct bpf_perf_event_data *ctx, void *buf, u32 size, u64 flags)
```

Для программы eBPF, связанной с событием perf event, функция извлекает записи ветвей (struct perf_branch_entry), связанной с ctx, и сохраняет их в буфере, указанном параметром buf и имеющем размер до size байтов.

В поле flags можно указать значение BPF_F_GET_BRANCH_RECORDS_SIZE, чтобы функция возвращала число байтов, требуемых для сохранения всех записей ветвей. В этом случае можно указать buf = NULL.

Функция при успешном завершении возвращает число байтов, записанных в buf. При ошибке возвращается отрицательный код:

- -EINVAL, если аргументы недействительны или значение size не кратно sizeof(struct perf_branch_entry);
- -ENOENT, если архитектура не поддерживает записи ветвей.

bpf_get_ns_current_pid_tgid

```
long bpf_get_ns_current_pid_tgid(u64 dev, u64 ino, struct bpf_pidns_info *nsdata, u32 size)
```

При успешном выполнении функция возвращает 0, а значения pid и tgid, видимые из текущего пространства имён, возвращаются в структуре nsdata.

Функция возвращает 0 в случае успеха или один из отрицательных кодов ошибок, указанных ниже:

- -EINVAL, если представленные dev и inum не соответствуют dev_t и inode с nsfs текущей задачи или при преобразовании dev в dev_t потеряны старшие биты;
- -ENOENT, если pidns не существует для текущей задачи.

bpf_xdp_output

```
long bpf_xdp_output(void *ctx, struct bpf_map *map, u64 flags, void *data, u64 size)
```

Записывает необработанные данные data (blob) в специальное событие BPF, содержащееся в map типа BPF_MAP_TYPE_PERF_EVENT_ARRAY. Это событие perf должно иметь атрибуты - PERF_SAMPLE_RAW как sample_type, PERF_TYPE_SOFTWARE как type, PERF_COUNT_SW_BPF_OUTPUT как config.

Параметр flags служит для указания индекса в map, по которому нужно поместить значение, с применением маски BPF_F_INDEX_MASK. Как вариант, можно установить BPF_F_CURRENT_CPU, чтобы указать использование индекса текущего CPU. Значение для записи (размером size) передаётся через стек eBPF и указывается параметром data. Параметр ctx является указателем на структуру xdp_buff в ядре.

Эта функция похода на bpf_perf_event_output(), но ограничена программами BPF raw_tracepoint.

При успешном выполнении функция возвращает 0, при отказе - отрицательное значение.

bpf_get_netns_cookie

```
u64 bpf_get_netns_cookie(void *ctx)
```

Извлекает cookie (создано ядром) из сетевого пространства имён, в которым связан параметр ctx. Значение cookie сетевого пространства имён не меняется в течение срока действия и является глобальным идентификатором, который можно считать уникальным. Если ctx = NULL, функция возвращает cookie для исходного сетевого пространства имён. Само значение cookie очень похоже на возвращаемое функцией bpf_get_socket_cookie(), но относится к пространству имён сети, а не сокета.

Функция возвращает 8-байтовое неинтерпретируемое число.

bpf_get_current_ancestor_cgroup_id

```
u64 bpf_get_current_ancestor_cgroup_id(int ancestor_level)
```

Возвращает идентификатор cgroup v2, которая является предком cgroup, связанной с текущей задачей, на уровне ancestor_level. Корнем cgroup является ancestor_level 0 и каждый шаг вниз по иерархии увеличивает уровень. Если ancestor_level совпадает с уровнем cgroup, связанной с текущей задачей, возвращаемое значение будет совпадать с результатом вызова bpf_get_current_cgroup_id().

Функция полезна для реализации правил, основанных на cgroup, которые в иерархии выше cgroup, непосредственно связанной с текущей задачей.

Формат возвращаемого идентификатора и ограничения функции те же, что и для bpf_get_current_cgroup_id().

Функция возвращает значение идентификатора или 0, если идентификатор не извлечён.

bpf_sk_assign

```
long bpf_sk_assign(struct sk_buff *skb, struct bpf_sock *sk, u64 flags)
```

Функция перезагружается в зависимости от типа программы BPF (описание пригодно для BPF_PROG_TYPE_SCHED_CLS и BPF_PROG_TYPE_SCHED_ACT).

Функция присваивает значение sk переменной skb. В сочетании с подходящей конфигурацией маршрутизации для приёма пакета в направлении сокета это приведёт к длставке skb в указанный сокет. Последующее перенаправление skb через bpf_redirect(), bpf_clone_redirect() или иные методы вне BPF могут помешать доставке в сокет.

Эта операция действительна лишь на входном пути TC.

Аргумент flags должен иметь значение 0.

Функция возвращает 0 при успехе и отрицательный код при ошибке:

- -EINVAL, если заданные флаги не поддерживаются;
- -ENOENT, если сокет недоступен для назначения;
- -ENETUNREACH, если сокет недоступен (ошбка netns);
- -EOPNOTSUPP, если операция не поддерживается, например, вызов извне входа TC;
- -ESOCKTNOSUPPORT, если тип сокета не поддержтвается (reuseport).

bpf_sk_assign

```
long bpf_sk_assign(struct bpf_sk_lookup *ctx, struct bpf_sock *sk, u64 flags)
```

Функция перезагружается в зависимости от типа программы BPF (описание пригодно для BPF_PROG_TYPE_SK_LOOKUP).

Функция выбирает sk как результат поиска сокета.

Для успешного выполнения операции сокет должен быть совместим с описанием пакета, представленным объектом ctx.

Протокол L4 (IPPROTO_TCP или IPPROTO_UDP) должен совпадать точно. Семейства адресов IP (AF_INET или AF_INET6) должны быть совместимыми, т. е. для сокетов IPv6, поддерживающих не только v6, могут быть выбраны пакеты IPv4.

Выбирать можно только слушающие сокеты TCP (listener) и не подключенные сокеты UDP. Параметр sk может иметь значение NULL для сброса прежнего выбора.

Аргумент flags может включать сочетание указанных ниже флагов.

- BPF_SK_LOOKUP_F_REPLACE для переопределения прежнего выбора сокета, который могла сделать программа BPF, работавшая ранее.
- BPF_SK_LOOKUP_F_NO_REUSEPORT для пропуска распределения нагрузки в группе reuseport для выбранного сокета.

При успехе ctx->sk будет указывать выбранный сокет.

Функция возвращает 0 при успехе и отрицательный код при ошибке:

- -EAFNOSUPPORT, если семейство адресов сокета (sk->family) несовместимо с семейством пакета (ctx->family);
- -EEXIST, если сокет уже выбран (возможно, другой программой), а флаг BPF_SK_LOOKUP_F_REPLACE не задан;
- -EINVAL, если заданные флаги не поддерживаются;
- -EPROTOTYPE, если протокол L4 в сокете (sk->protocol) не соответствует протоколу в пакете (ctx->protocol);
- -ESOCKTNOSUPPORT, если сокет не находится в разрешенном состоянии (прослущивающий сокет TCP или отсоединенный сокет UDP).

bpf_ktime_get_boot_ns

```
u64 bpf_ktime_get_boot_ns(void)
```

Возвращает время, прошедшее с момента загрузки системы (в наносекундах) - текущее значение ktime, включая время, когда система была приостановлена (см. clock_gettime(CLOCK_BOOTTIME)).

bpf_seq_printf

```
long bpf_seq_printf(struct seq_file *m, const char *fmt, u32 fmt_size, const void *data, u32 data_len)
```

Функция использует seq_file seq_printf() для вывода строки формата. Параметр m представляет seq_file, fmt и fmt_size служат для строки формата, data и data_len - аргументы строки формата. Параметр data - это массив u64 и соответствующие значения строки формата сохраняются в массиве. Для строк и указаталей, где происходит

обращение к указанным элементам в массиве data сохраняются лишь указатели. Поле data_len указывает ращмер данных в байтах.

Для чтения памяти ядра требуются форматы %s, %p{i,l}{4,6}. При таком чтении могут возникать ошибки из-за неверного адреса или верного адреса, требующего существенного отказа в памяти. Если отказ при чтении памяти ядра возникает, строка %s будет пустой, а адрес ip для %p{i,l}{4,6} будет иметь значение 0. Отсутствие возврата ошибки в программу bpf согласуется с поведением функции bpf_trace_printk().

Функция возвращает 0 при успешном выполнении и отрицательный код при ошибке:

- -EBUSY, если буфер копирования памяти для CPU занят (можно повторить попытку, вернув 1 из программы bpf);
- -EINVAL, если аргумент недействителен или fmt не поддерживается (недействителен);
- -E2BIG, если fmt содержит слишком много указателей формата;
- -EOVERFLOW, если возникает переполнение (попытка повторяется с тем же объектом).

bpf_seq_write

`long bpf_seq_write(struct seq_file *m, const void *data, u32 len)`

Функция использует seq_file seq_write() для записи данных. Параметр m представляет seq_file, data и len - данные для записи.

Функция возвращает 0 при успешном выполнении и отрицательный код при ошибке: -EOVERFLOW, если возникает переполнение (попытка повторяется с тем же объектом).

bpf_sk_cgroup_id

`u64 bpf_sk_cgroup_id(struct bpf_sock *sk)`

Функция возвращает идентификатор cgroup v2 сокета sk. Параметр sk должен быть непустым (не NULL) указателем на полный сокет, например, значением, возвращённым функцией bpf_sk_lookup_tcp(), bpf_sk_lookup_udp(), bpf_sk_fullsock() и т. п. Формат возвращаемого идентификатора совпадает с возвращаемым функцией bpf_skb_cgroup_id().

Эта функция доступна лишь для ядер, собранных с опцией CONFIG_SOCK_CGROUP_DATA.

Функция возвращает найденный идентификатор или 0, если идентификатор извлечь не удалось.

bpf_sk_ancestor_cgroup_id

`u64 bpf_sk_ancestor_cgroup_id(struct bpf_sock *sk, int ancestor_level)`

Функция возвращает идентификатор cgroup v2, являющийся предком cgroup, связанной с sk на уровне ancestor_level. Корнем cgroup является ancestor_level 0 и каждый шаг вниз по иерархии увеличивает уровень. Если ancestor_level совпадает с уровнем cgroup, связанной с текущей задачей, возвращаемое значение будет совпадать с результатом вызова bpf_sk_cgroup_id().

Функция полезна для реализации правил, основанных на cgroup, которые в иерархии выше cgroup, связанной с sk.

Формат возвращаемого идентификатора и ограничения функции те же, что и для bpf_sk_cgroup_id().

Функция возвращает найденный идентификатор или 0, если идентификатор извлечь не удалось.

bpf_ringbuf_output

`long bpf_ringbuf_output(void *ringbuf, void *data, u64 size, u64 flags)`

Копирует size байтов и data в кольцевой буфер ringbuf. При наличии BPF_RB_NO_WAKEUP в поле flags, уведомление о доступности новых данных не передаётся, флаг BPF_RB_FORCE_WAKEUP задаёт безусловную передачу такого уведомления.

Функция возвращает 0 при успехе и отрицательный код при ошибке.

bpf_ringbuf_reserve

`void *bpf_ringbuf_reserve(void *ringbuf, u64 size, u64 flags)`

Резервирует size байтов содержимого (payload) в кольцевом буфере ringbuf.

Возвращает действительный указатель на size байтов доступной памяти или NULL, если выделить память не удалось.

bpf_ringbuf_submit

`void bpf_ringbuf_submit(void *data, u64 flags)`

Представляет зарезервированную выборку кольцевого буфера, указанную параметром data. При наличии BPF_RB_NO_WAKEUP в поле flags, уведомление о доступности новых данных не передаётся, флаг BPF_RB_FORCE_WAKEUP задаёт безусловную передачу такого уведомления.

Функция всегда успешна и ничего не возвращает.

bpf_ringbuf_discard

`void bpf_ringbuf_discard(void *data, u64 flags)`

Отбрасывает зарезервированную выборку кольцевого буфера, указанную параметром data. При наличии BPF_RB_NO_WAKEUP в поле flags, уведомление о доступности новых данных не передаётся, флаг BPF_RB_FORCE_WAKEUP задаёт безусловную передачу такого уведомления.

Функция всегда успешна и ничего не возвращает.

bpf_ringbuf_query

```
u64 bpf_ringbuf_query(void *ringbuf, u64 flags)
```

Запрашивает характеристики представленного кольцевого буфера, указываемые битами поля flags:

BPF_RB_AVAIL_DATA
объём ещё не потреблённых данных;
BPF_RB_RING_SIZE
размер кольцевого буфера;
BPF_RB_CONS_POS
позиция потребителя (может переходить через «разрыв» кольца);
BPF_RB_PROD_POS
позиция поставщика (может переходить через «разрыв» кольца).

Возвращенные данные являются просто «моменталиным снимком» фактических значений и могут быть неточны, поэтому функцию следует применять лишь для усиления эвристики и подготовки отчётов, а не для точного расчета.

Функция возвращает запрошенное значение или 0, если поле flags не распознано.

bpf_csum_level

```
long bpf_csum_level(struct sk_buff *skb, u64 level)
```

Меняет уровень контрольной суммы skb на 1 вверх или вниз, или сбрасывает уровень в none, чтобы стек проверил контрольную сумму. Этот уровень применим к протоколам TCP, UDP, GRE, SCTP, FCOE. Например, декапсуляция | ETH | IP | UDP | GUE | IP | TCP | into | ETH | IP | TCP | через функцию bpf_skb_adjust_room() с передачей флага BPF_F_ADJ_ROOM_NO_CSUM_RESET потребует вызова bpf_csum_level() с BPF_CSUM_LEVEL_DEC, поскольку удаляется заголовок UDP. Аналогично, для инкапсуляции обратно может вызываться bpf_csum_level() с BPF_CSUM_LEVEL_INC, если skb всё ещё предназначен для обработки вышележащими уровнями стека, а не просто отправки на выход tc.

В настоящее время поддерживается 3 установки уровня:

BPF_CSUM_LEVEL_INC
повысить skb->csum_level для skb с CHECKSUM_UNNECESSARY;
BPF_CSUM_LEVEL_DEC
понизить skb->csum_level для skb с CHECKSUM_UNNECESSARY;
BPF_CSUM_LEVEL_RESET
сбросить skb->csum_level в 0 и установить CHECKSUM_NONE для проверки контрольной суммы стеком№
BPF_CSUM_LEVEL_QUERY
нет операций, возвращается текущее значение skb->csum_level.

Функция возвращает 0 при успехе и отрицательный код в случае ошибки. При установке BPF_CSUM_LEVEL_QUERY возвращается текущее значение skb->csum_level или код ошибки -EACCES, если для skb не установлено CHECKSUM_UNNECESSARY.

bpf_skc_to_tcp6_sock

```
struct tcp6_sock *bpf_skc_to_tcp6_sock(void *sk)
```

Динамически преобразует указатель sk в указатель tcp6_sock.

Функция возвращает sk, если преобразование действительно, и NULL в ином случае.

bpf_skc_to_tcp_sock

```
struct tcp_sock *bpf_skc_to_tcp_sock(void *sk)
```

Динамически преобразует указатель sk в указатель tcp_sock.

Функция возвращает sk, если преобразование действительно, и NULL в ином случае.

bpf_skc_to_tcp_timewait_sock

```
struct tcp_timewait_sock *bpf_skc_to_tcp_timewait_sock(void *sk)
```

Динамически преобразует указатель sk в указатель tcp_timewait_sock.

Функция возвращает sk, если преобразование действительно, и NULL в ином случае.

bpf_skc_to_tcp_request_sock

```
struct tcp_request_sock *bpf_skc_to_tcp_request_sock(void *sk)
```

Динамически преобразует указатель sk в указатель tcp_request_sock.

Функция возвращает sk, если преобразование действительно, и NULL в ином случае.

bpf_skc_to_udp6_sock

```
struct udp6_sock *bpf_skc_to_udp6_sock(void *sk)
```

Динамически преобразует указатель sk в указатель udp6_sock.

Функция возвращает sk, если преобразование действительно, и NULL в ином случае.

bpf_get_task_stack

```
long bpf_get_task_stack(struct task_struct *task, void *buf, u32 size, u64 flags)
```

Возвращает стек пользователя или ядра в буфере, пердоставленном программой bpf. Для этого функции нужен параметр task, являющийся действительным указателем на структуру task_struct. Для сохранения трассировки стека программа bpf предоставляет буфер buf с неотрицательным размером. Аргумент flags указывает число пропускаемых кадров стека (0 -255) с маской BPF_F_SKIP_FIELD_MASK. Остальные биты могут служить для указанных ниже флагов.

BPF_F_USER_STACK

Работать со стеком пользователя, а не ядра.

BPF_F_USER_BUILD_ID

Собирать buildid+offset вместо ips для пользовательского стека (применимо лишь с флагом BPF_F_USER_STACK).

Функция может собирать да PERF_MAX_STACK_DEPTH кадров из стека пользователя или ядра при наличии достаточно большого буфера. Этим ограничением может управлять программа sysctl и задавать предел вручную для работы с длинным пользовательским стеком (например, в программах Java). Команда установки значения имеет вид

```
# sysctl kernel.perf_event_max_stack=<new value>
```

Функция возвращает неотрицательное значение, не превышающее size, при успешном выполнении и отрицательное значение в случае ошибки.

Примеры

Примеры использования большинства функций-помощников eBPF, описанных здесь, доступны в файлах исходного кода ядра Linux, размещённых в каталогах samples/bpf/ и tools/testing/selftests/bpf/.

Реализация

Этот документ является попыткой описать имеющиеся функции-помощники (helper) eBPF. Подсистема BPF активно развивается с добавлением новых программ eBPF, типов отображений, а также функций-помощников. Для проверки наличия функций-помощников в конкретном ядре или получения сведений о поддерживаемых программах следует обращаться к указанным ниже файлам в дереве исходных кодов ядра.

- include/uapi/linux/bpf.h - основной заголовок BPF, содержащий полный список функций-помощников и другие определения BPF, включая большинство флагов, структур и констант.
- net/core/filter.c содержит определения большинства относящихся к сети функций-помощников и список типов программ, из которых эти функции можно вызывать.
- kernel/trace/bpf_trace.c то же, что и предыдущий, но для функций трассировки.
- kernel/bpf/verifier.c - функции, используемые для проверки использования с данной функцией помощником лишь действительные типы отображений eBPF.
- Каталог kernel/bpf/ содержит файлы с определениями дополнительных функций-помощников (для cgroup, sockmap и т. п.).
- Для проверки доступности функций можно использовать утилиту bpftool, например, bpftool feature probe (см. man 8 bpftool-feature). Ключевое слово unprivileged позволяет увидеть функции, доступные непривилегированным пользователям.

Совместимость функций-помощников с разными типами программ обычно описана в файлах с определением функций помощников (объекты struct bpf_func_proto и возвращающие их функции).

Совместимость функций-помощников с типами отображений моно найти в функции check_map_func_compatibility() в файле kernel/bpf/verifier.c.

Функции-помощники, отменяющие результаты проверки указателей data и data_end для сетевой обработки, перечислены в функции bpf_helper_changes_pkt_data() в файле net/core/filter.c.

Литература

[1] Файл Documentation/admin-guide/cgroup-v1/net_cls.rst в дереве исходных кодов ядра Linux.

Перевод на русский язык

Николай Малых

nmalykh@protokols.ru