NAME

BPF-HELPERS − list of eBPF helper functions

DESCRIPTION

The extended Berkeley Packet Filter (eBPF) subsystem consists in programs written in a pseudo−assembly language, then attached to one of the several kernel hooks and run in reaction of specific events. This framework differs from the older, "classic" BPF (or "cBPF") in several aspects, one of them being the ability to call special functions (or "helpers") from within a program. These functions are restricted to a white−list of helpers defined in the kernel.

These helpers are used by eBPF programs to interact with the system, or with the context in which they work. For instance, they can be used to print debugging messages, to get the time since the system was booted, to interact with eBPF maps, or to manipulate network packets. Since there are several eBPF program types, and that they do not run in the same context, each program type can only call a subset of those helpers.

Due to eBPF conventions, a helper can not have more than five arguments.

Internally, eBPF programs call directly into the compiled helper functions without requiring any foreign−function interface. As a result, calling helpers introduces no overhead, thus offering excellent performance.

This document is an attempt to list and document the helpers available to eBPF developers. They are sorted by chronological order (the oldest helpers in the kernel at the top).

HELPERS

void *bpf_map_lookup_elem(struct bpf_map *_map_, const void *_key_)

Description
Perform a lookup in *map* for an entry associated to *key*.

Return Map value associated to *key*, or **NULL** if no entry was found.

int bpf_map_update_elem(struct bpf_map *_map_, const void *_key_, const void *_value_, u64 _flags_)

Description
Add or update the value of the entry associated to *key* in *map* with *value*. *flags* is one of:

BPF_NOEXIST
The entry for *key* must not exist in the map.

BPF_EXIST
The entry for *key* must already exist in the map.

BPF_ANY
No condition on the existence of the entry for *key*.

Flag value **BPF_NOEXIST** cannot be used for maps of types **BPF_MAP_TYPE_ARRAY** or **BPF_MAP_TYPE_PERCPU_ARRAY** (all elements always exist), the helper would return an error.

Return 0 on success, or a negative error in case of failure.

int bpf_map_delete_elem(struct bpf_map *_map_, const void *_key_)

Description
Delete entry with *key* from *map*.

Return 0 on success, or a negative error in case of failure.

int bpf_map_push_elem(struct bpf_map *_map_, const void *_value_, u64 _flags_)

Description

Push an element *value* in *map*. *flags* is one of:

BPF_EXIST If the queue/stack is full, the oldest element is removed to make room for this.

Return 0 on success, or a negative error in case of failure.

int bpf_probe_read(void *_dst_, u32 _size_, const void *_src_)

Description

For tracing programs, safely attempt to read *size* bytes from address *src* and store the data in *dst*.

Return 0 on success, or a negative error in case of failure.

u64 bpf_ktime_get_ns(void)

Description

Return the time elapsed since system boot, in nanoseconds.

Return Current *ktime*.

int bpf_trace_printk(const char *_fmt_, u32 _fmt_size_, ...)

Description

This helper is a "printk()−like" facility for debugging. It prints a message defined by format *fmt* (of size *fmt_size*) to file */sys/kernel/debug/tracing/trace* from DebugFS, if available. It can take up to three additional **u64** arguments (as an eBPF helpers, the total number of arguments is limited to five).

Each time the helper is called, it appends a line to the trace. The format of the trace is customizable, and the exact output one will get depends on the options set in */sys/kernel/debug/tracing/trace_options* (see also the *README* file under the same directory). However, it usually defaults to something like:

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

In the above:

- **telnet** is the name of the current task.
- **470** is the PID of the current task.
- **001** is the CPU number on which the task is running.
- In **.N..**, each character refers to a set of options (whether irqs are enabled, scheduling options, whether hard/softirqs are running, level of preempt_disabled respectively). **N** means that **TIF_NEED_RESCHED** and **PREEMPT_NEED_RESCHED** are set.
- **419421.045894** is a timestamp.
- **0x00000001** is a fake value used by BPF for the instruction pointer register.
- **<formatted msg>** is the message formatted with *fmt*.

The conversion specifiers supported by *fmt* are similar, but more limited than for printk(). They are **%d**, **%i**, **%u**, **%x**, **%ld**, **%li**, **%lu**, **%lx**, **%lld**, **%lli**, **%llu**, **%llx**, **%p**, **%s**. No modifier (size of field, padding with zeroes, etc.) is available, and the helper will return **−EINVAL** (but print nothing) if it encounters an unknown specifier.

Also, note that **bpf_trace_printk**() is slow, and should only be used for debugging purposes. For this reason, a notice bloc (spanning several lines) is printed to kernel logs and

states that the helper should not be used "for production use" the first time this helper is used (or more precisely, when **trace_printk**() buffers are allocated). For passing values to user space, perf events should be preferred.

Return The number of bytes written to the buffer, or a negative error in case of failure.

u32 bpf_get_prandom_u32(void)

Description

Get a pseudo−random number.

From a security point of view, this helper uses its own pseudo−random internal state, and cannot be used to infer the seed of other random functions in the kernel. However, it is essential to note that the generator used by the helper is not cryptographically secure.

Return A random 32−bit unsigned value.

u32 bpf_get_smp_processor_id(void)

Description

Get the SMP (symmetric multiprocessing) processor id. Note that all programs run with preemption disabled, which means that the SMP processor id is stable during all the execution of the program.

Return The SMP id of the processor running the program.

int bpf_skb_store_bytes(struct sk_buff *_skb_, u32 _offset_, const void *_from_, u32 _len_, u64 _flags_)

Description

Store *len* bytes from address *from* into the packet associated to *skb*, at *offset*. *flags* are a combination of **BPF_F_RECOMPUTE_CSUM** (automatically recompute the checksum for the packet after storing the bytes) and **BPF_F_INVALIDATE_HASH** (set *skb*−>**hash**, *skb*−>**swhash** and *skb*−>**l4hash** to 0).

A call to this helper is susceptible to change the underlaying packet buffer. Therefore, at load time, all checks on pointers previously done by the verifier are invalidated and must be performed again, if the helper is used in combination with direct packet access.

Return 0 on success, or a negative error in case of failure.

int bpf_l3_csum_replace(struct sk_buff *_skb_, u32 _offset_, u64 _from_, u64 _to_, u64 _size_)

Description

Recompute the layer 3 (e.g. IP) checksum for the packet associated to *skb*. Computation is incremental, so the helper must know the former value of the header field that was modified (*from*), the new value of this field (*to*), and the number of bytes (2 or 4) for this field, stored in *size*. Alternatively, it is possible to store the difference between the previous and the new values of the header field in *to*, by setting *from* and *size* to 0. For both methods, *offset* indicates the location of the IP checksum within the packet.

This helper works in combination with **bpf_csum_diff**(), which does not update the checksum in−place, but offers more flexibility and can handle sizes larger than 2 or 4 for the checksum to update.

A call to this helper is susceptible to change the underlaying packet buffer. Therefore, at load time, all checks on pointers previously done by the verifier are invalidated and must be performed again, if the helper is used in combination with direct packet access.

Return 0 on success, or a negative error in case of failure.

int bpf_l4_csum_replace(struct sk_buff *_skb_, u32 _offset_, u64 _from_, u64 _to_, u64 _flags_)

Description

Recompute the layer 4 (e.g. TCP, UDP or ICMP) checksum for the packet associated to *skb*. Computation is incremental, so the helper must know the former value of the header field that was modified (*from*), the new value of this field (*to*), and the number of bytes (2 or 4) for this field, stored on the lowest four bits of *flags*. Alternatively, it is possible to store the difference between the previous and the new values of the header field in *to*, by setting *from* and the four lowest bits of *flags* to 0. For both methods, *offset* indicates the location of the IP checksum within the packet. In addition to the size of the field, *flags* can be added (bitwise OR) actual flags. With **BPF_F_MARK_MANGLED_0**, a null checksum is left untouched (unless **BPF_F_MARK_ENFORCE** is added as well), and for updates resulting in a null checksum the value is set to **CSUM_MANGLED_0** instead. Flag **BPF_F_PSEUDO_HDR** indicates the checksum is to be computed against a pseudo−header.

This helper works in combination with **bpf_csum_diff**(), which does not update the checksum in−place, but offers more flexibility and can handle sizes larger than 2 or 4 for the checksum to update.

A call to this helper is susceptible to change the underlaying packet buffer. Therefore, at load time, all checks on pointers previously done by the verifier are invalidated and must be performed again, if the helper is used in combination with direct packet access.

Return 0 on success, or a negative error in case of failure.

int bpf_tail_call(void *_ctx_, struct bpf_map *_prog_array_map_, u32 _index_)

Description

This special helper is used to trigger a "tail call", or in other words, to jump into another eBPF program. The same stack frame is used (but values on stack and in registers for the caller are not accessible to the callee). This mechanism allows for program chaining, either for raising the maximum number of available eBPF instructions, or to execute given programs in conditional blocks. For security reasons, there is an upper limit to the number of successive tail calls that can be performed.

Upon call of this helper, the program attempts to jump into a program referenced at index *index* in *prog_array_map*, a special map of type **BPF_MAP_TYPE_PROG_ARRAY**, and passes *ctx*, a pointer to the context.

If the call succeeds, the kernel immediately runs the first instruction of the new program. This is not a function call, and it never returns to the previous program. If the call fails, then the helper has no effect, and the caller continues to run its subsequent instructions. A call can fail if the destination program for the jump does not exist (i.e. *index* is superior to the number of entries in *prog_array_map*), or if the maximum number of tail calls has been reached for this chain of programs. This limit is defined in the kernel by the macro **MAX_TAIL_CALL_CNT** (not accessible to user space), which is currently set to 32.

Return 0 on success, or a negative error in case of failure.

int bpf_clone_redirect(struct sk_buff *_skb_, u32 _ifindex_, u64 _flags_)

Description

Clone and redirect the packet associated to *skb* to another net device of index *ifindex*. Both ingress and egress interfaces can be used for redirection. The **BPF_F_INGRESS** value in *flags* is used to make the distinction (ingress path is selected if the flag is present, egress path otherwise). This is the only flag supported for now.

In comparison with **bpf_redirect**() helper, **bpf_clone_redirect**() has the associated cost of duplicating the packet buffer, but this can be executed out of the eBPF program.

Conversely, **bpf_redirect**() is more efficient, but it is handled through an action code where the redirection happens only after the eBPF program has returned.

A call to this helper is susceptible to change the underlaying packet buffer. Therefore, at load time, all checks on pointers previously done by the verifier are invalidated and must be performed again, if the helper is used in combination with direct packet access.

Return 0 on success, or a negative error in case of failure.

u64 bpf_get_current_pid_tgid(void)

Return A 64−bit integer containing the current tgid and pid, and created as such: *current_task***−>tgid << 32 |** *current_task***−>pid**.

u64 bpf_get_current_uid_gid(void)

Return A 64−bit integer containing the current GID and UID, and created as such: *current_gid* **<< 32 |** *current_uid*.

int bpf_get_current_comm(char *****buf***, u32** *size_of_buf***)**

Description

Copy the **comm** attribute of the current task into *buf* of *size_of_buf*. The **comm** attribute contains the name of the executable (excluding the path) for the current task. The *size_of_buf* must be strictly positive. On success, the helper makes sure that the *buf* is NUL−terminated. On failure, it is filled with zeroes.

Return 0 on success, or a negative error in case of failure.

u32 bpf_get_cgroup_classid(struct sk_buff *****skb***)**

Description

Retrieve the classid for the current task, i.e. for the net_cls cgroup to which *skb* belongs.

This helper can be used on TC egress path, but not on ingress.

The net_cls cgroup provides an interface to tag network packets based on a user−provided identifier for all traffic coming from the tasks belonging to the related cgroup. See also the related kernel documentation, available from the Linux sources in file *Documentation/cgroup−v1/net_cls.txt*.

The Linux kernel has two versions for cgroups: there are cgroups v1 and cgroups v2. Both are available to users, who can use a mixture of them, but note that the net_cls cgroup is for cgroup v1 only. This makes it incompatible with BPF programs run on cgroups, which is a cgroup−v2−only feature (a socket can only hold data for one version of cgroups at a time).

This helper is only available is the kernel was compiled with the **CONFIG_CGROUP_NET_CLASSID** configuration option set to "**y**" or to "**m**".

Return The classid, or 0 for the default unconfigured classid.

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

Description

Push a *vlan_tci* (VLAN tag control information) of protocol *vlan_proto* to the packet associated to *skb*, then update the checksum. Note that if *vlan_proto* is different from **ETH_P_8021Q** and **ETH_P_8021AD**, it is considered to be **ETH_P_8021Q**.

A call to this helper is susceptible to change the underlaying packet buffer. Therefore, at load time, all checks on pointers previously done by the verifier are invalidated and must be performed again, if the helper is used in combination with direct packet access.

Return 0 on success, or a negative error in case of failure.

int bpf_skb_vlan_pop(struct sk_buff *_skb_)

Description

Pop a VLAN header from the packet associated to *skb*.

A call to this helper is susceptible to change the underlaying packet buffer. Therefore, at load time, all checks on pointers previously done by the verifier are invalidated and must be performed again, if the helper is used in combination with direct packet access.

Return 0 on success, or a negative error in case of failure.

int bpf_skb_get_tunnel_key(struct sk_buff *_skb_, struct bpf_tunnel_key *_key_, u32 _size_, u64 _flags_)

Description

Get tunnel metadata. This helper takes a pointer *key* to an empty **struct bpf_tunnel_key** of **size**, that will be filled with tunnel metadata for the packet associated to *skb*. The *flags* can be set to **BPF_F_TUNINFO_IPV6**, which indicates that the tunnel is based on IPv6 protocol instead of IPv4.

The **struct bpf_tunnel_key** is an object that generalizes the principal parameters used by various tunneling protocols into a single struct. This way, it can be used to easily make a decision based on the contents of the encapsulation header, "summarized" in this struct. In particular, it holds the IP address of the remote end (IPv4 or IPv6, depending on the case) in *key*->**remote_ipv4** or *key*->**remote_ipv6**. Also, this struct exposes the *key*->**tunnel_id**, which is generally mapped to a VNI (Virtual Network Identifier), making it programmable together with the **bpf_skb_set_tunnel_key**() helper.

Let's imagine that the following code is part of a program attached to the TC ingress interface, on one end of a GRE tunnel, and is supposed to filter out all messages coming from remote ends with IPv4 address other than 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;     // drop packet

if (key.remote_ipv4 != 0x0a000001)
        return TC_ACT_SHOT;     // drop packet

return TC_ACT_OK;               // accept packet
```

This interface can also be used with all encapsulation devices that can operate in "collect metadata" mode: instead of having one network device per specific configuration, the "collect metadata" mode only requires a single device where the configuration can be extracted from this helper.

This can be used together with various tunnels such as VXLan, Geneve, GRE or IP in IP (IPIP).

Return 0 on success, or a negative error in case of failure.

int bpf_skb_set_tunnel_key(struct sk_buff *_skb_, struct bpf_tunnel_key *_key_, u32 _size_, u64 _flags_)

Description

Populate tunnel metadata for packet associated to *skb.* The tunnel metadata is set to the contents of *key*, of *size*. The *flags* can be set to a combination of the following values:

BPF_F_TUNINFO_IPV6
Indicate that the tunnel is based on IPv6 protocol instead of IPv4.

BPF_F_ZERO_CSUM_TX
For IPv4 packets, add a flag to tunnel metadata indicating that checksum computation should be skipped and checksum set to zeroes.

BPF_F_DONT_FRAGMENT
Add a flag to tunnel metadata indicating that the packet should not be fragmented.

BPF_F_SEQ_NUMBER
Add a flag to tunnel metadata indicating that a sequence number should be added to tunnel header before sending the packet. This flag was added for GRE encapsulation, but might be used with other protocols as well in the future.

Here is a typical usage on the transmit path:

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

See also the description of the **bpf_skb_get_tunnel_key**() helper for additional information.

Return 0 on success, or a negative error in case of failure.

u64 bpf_perf_event_read(struct bpf_map *_map_, u64 _flags_)

Description

Read the value of a perf event counter. This helper relies on a *map* of type **BPF_MAP_TYPE_PERF_EVENT_ARRAY**. The nature of the perf event counter is selected when *map* is updated with perf event file descriptors. The *map* is an array whose size is the number of available CPUs, and each cell contains a value relative to one CPU. The value to retrieve is indicated by *flags*, that contains the index of the CPU to look up, masked with **BPF_F_INDEX_MASK**. Alternatively, *flags* can be set to **BPF_F_CURRENT_CPU** to indicate that the value for the current CPU should be retrieved.

Note that before Linux 4.13, only hardware perf event can be retrieved.

Also, be aware that the newer helper **bpf_perf_event_read_value**() is recommended over **bpf_perf_event_read**() in general. The latter has some ABI quirks where error and counter value are used as a return code (which is wrong to do since ranges may overlap). This issue is fixed with **bpf_perf_event_read_value**(), which at the same time provides more features over the **bpf_perf_event_read**() interface. Please refer to the description of **bpf_perf_event_read_value**() for details.

Return The value of the perf event counter read from the map, or a negative error code in case of failure.

int bpf_redirect(u32 _ifindex_, u64 _flags_)

Description

Redirect the packet to another net device of index *ifindex*. This helper is somewhat similar to **bpf_clone_redirect**(), except that the packet is not cloned, which provides

increased performance.

Except for XDP, both ingress and egress interfaces can be used for redirection. The **BPF_F_INGRESS** value in *flags* is used to make the distinction (ingress path is selected if the flag is present, egress path otherwise). Currently, XDP only supports redirection to the egress interface, and accepts no flag at all.

The same effect can be attained with the more generic **bpf_redirect_map**(), which requires specific maps to be used but offers better performance.

Return For XDP, the helper returns **XDP_REDIRECT** on success or **XDP_ABORTED** on error. For other program types, the values are **TC_ACT_REDIRECT** on success or **TC_ACT_SHOT** on error.

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

Description

Retrieve the realm or the route, that is to say the **tclassid** field of the destination for the *skb*. The indentifier retrieved is a user−provided tag, similar to the one used with the net_cls cgroup (see description for **bpf_get_cgroup_classid**() helper), but here this tag is held by a route (a destination entry), not by a task.

Retrieving this identifier works with the clsact TC egress hook (see also **tc−bpf(8)**), or alternatively on conventional classful egress qdiscs, but not on TC ingress path. In case of clsact TC egress hook, this has the advantage that, internally, the destination entry has not been dropped yet in the transmit path. Therefore, the destination entry does not need to be artificially held via **netif_keep_dst**() for a classful qdisc until the *skb* is freed.

This helper is available only if the kernel was compiled with **CONFIG_IP_ROUTE_CLASSID** configuration option.

Return The realm of the route for the packet associated to *skb*, or 0 if none was found.

int bpf_perf_event_output(struct pt_reg ****ctx*, struct bpf_map ****map***, u64** *flags***, void ****data***, u64** *size***)**

Description

Write raw *data* blob into a special BPF perf event held by *map* of type **BPF_MAP_TYPE_PERF_EVENT_ARRAY**. This perf event must have the following attributes: **PERF_SAMPLE_RAW** as **sample_type**, **PERF_TYPE_SOFTWARE** as **type**, and **PERF_COUNT_SW_BPF_OUTPUT** as **config**.

The *flags* are used to indicate the index in *map* for which the value must be put, masked with **BPF_F_INDEX_MASK**. Alternatively, *flags* can be set to **BPF_F_CURRENT_CPU** to indicate that the index of the current CPU core should be used.

The value to write, of *size*, is passed through eBPF stack and pointed by *data*.

The context of the program *ctx* needs also be passed to the helper.

On user space, a program willing to read the values needs to call **perf_event_open**() on the perf event (either for one or for all CPUs) and to store the file descriptor into the *map*. This must be done before the eBPF program can send data into it. An example is available in file *samples/bpf/trace_output_user.c* in the Linux kernel source tree (the eBPF program counterpart is in *samples/bpf/trace_output_kern.c*).

bpf_perf_event_output() achieves better performance than **bpf_trace_printk**() for sharing data with user space, and is much better suitable for streaming data from eBPF programs.

Note that this helper is not restricted to tracing use cases and can be used with programs attached to TC or XDP as well, where it allows for passing data to user space listeners. Data can be:

- Only custom structs,
- Only the packet payload, or
- A combination of both.

Return 0 on success, or a negative error in case of failure.

int bpf_skb_load_bytes(const struct sk_buff ****skb*, u32** *offset***, void ****to***, u32** *len***)**

Description

This helper was provided as an easy way to load data from a packet. It can be used to load *len* bytes from *offset* from the packet associated to *skb*, into the buffer pointed by *to*.

Since Linux 4.7, usage of this helper has mostly been replaced by "direct packet access", enabling packet data to be manipulated with *skb***->data** and *skb***->data_end** pointing respectively to the first byte of packet data and to the byte after the last byte of packet data. However, it remains useful if one wishes to read large quantities of data at once from a packet into the eBPF stack.

Return 0 on success, or a negative error in case of failure.

int bpf_get_stackid(struct pt_reg ****ctx*, struct bpf_map ****map***, u64** *flags***)**

Description

Walk a user or a kernel stack and return its id. To achieve this, the helper needs *ctx*, which is a pointer to the context on which the tracing program is executed, and a pointer to a *map* of type **BPF_MAP_TYPE_STACK_TRACE**.

The last argument, *flags*, holds the number of stack frames to skip (from 0 to 255), masked with **BPF_F_SKIP_FIELD_MASK**. The next bits can be used to set a combination of the following flags:

BPF_F_USER_STACK
Collect a user space stack instead of a kernel stack.

BPF_F_FAST_STACK_CMP
Compare stacks by hash only.

BPF_F_REUSE_STACKID
If two different stacks hash into the same *stackid*, discard the old one.

The stack id retrieved is a 32 bit long integer handle which can be further combined with other data (including other stack ids) and used as a key into maps. This can be useful for generating a variety of graphs (such as flame graphs or off−cpu graphs).

For walking a stack, this helper is an improvement over **bpf_probe_read**(), which can be used with unrolled loops but is not efficient and consumes a lot of eBPF instructions. Instead, **bpf_get_stackid**() can collect up to **PERF_MAX_STACK_DEPTH** both kernel and user frames. Note that this limit can be controlled with the **sysctl** program, and that it should be manually increased in order to profile long user stacks (such as stacks for Java programs). To do so, use:

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

Return The positive or null stack id on success, or a negative error in case of failure.

s64 bpf_csum_diff(__be32 *_from_, u32 _from_size_, __be32 *_to_, u32 _to_size_, __wsum _seed_)

Description

Compute a checksum difference, from the raw buffer pointed by *from*, of length *from_size* (that must be a multiple of 4), towards the raw buffer pointed by *to*, of size *to_size* (same remark). An optional *seed* can be added to the value (this can be cascaded, the seed may come from a previous call to the helper).

This is flexible enough to be used in several ways:

- With *from_size* == 0, *to_size* > 0 and *seed* set to checksum, it can be used when pushing new data.
- With *from_size* > 0, *to_size* == 0 and *seed* set to checksum, it can be used when removing data from a packet.
- With *from_size* > 0, *to_size* > 0 and *seed* set to 0, it can be used to compute a diff. Note that *from_size* and *to_size* do not need to be equal.

This helper can be used in combination with **bpf_l3_csum_replace**() and **bpf_l4_csum_replace**(), to which one can feed in the difference computed with **bpf_csum_diff**().

Return The checksum result, or a negative error code in case of failure.

int bpf_skb_get_tunnel_opt(struct sk_buff *_skb_, u8 *_opt_, u32 _size_)

Description

Retrieve tunnel options metadata for the packet associated to *skb*, and store the raw tunnel option data to the buffer *opt* of *size*.

This helper can be used with encapsulation devices that can operate in "collect metadata" mode (please refer to the related note in the description of **bpf_skb_get_tunnel_key**() for more details). A particular example where this can be used is in combination with the Geneve encapsulation protocol, where it allows for pushing (with **bpf_skb_get_tunnel_opt**() helper) and retrieving arbitrary TLVs (Type−Length−Value headers) from the eBPF program. This allows for full customization of these headers.

Return The size of the option data retrieved.

int bpf_skb_set_tunnel_opt(struct sk_buff *_skb_, u8 *_opt_, u32 _size_)

Description

Set tunnel options metadata for the packet associated to *skb* to the option data contained in the raw buffer *opt* of *size*.

See also the description of the **bpf_skb_get_tunnel_opt**() helper for additional information.

Return 0 on success, or a negative error in case of failure.

int bpf_skb_change_proto(struct sk_buff *_skb_, __be16 _proto_, u64 _flags_)

Description

Change the protocol of the *skb* to *proto*. Currently supported are transition from IPv4 to IPv6, and from IPv6 to IPv4. The helper takes care of the groundwork for the transition, including resizing the socket buffer. The eBPF program is expected to fill the new headers, if any, via **skb_store_bytes**() and to recompute the checksums with **bpf_l3_csum_replace**() and **bpf_l4_csum_replace**(). The main case for this helper is to perform NAT64 operations out of an eBPF program.

Internally, the GSO type is marked as dodgy so that headers are checked and segments are

recalculated by the GSO/GRO engine. The size for GSO target is adapted as well.

All values for *flags* are reserved for future usage, and must be left at zero.

A call to this helper is susceptible to change the underlaying packet buffer. Therefore, at load time, all checks on pointers previously done by the verifier are invalidated and must be performed again, if the helper is used in combination with direct packet access.

Return 0 on success, or a negative error in case of failure.

int bpf_skb_change_type(struct sk_buff ****skb*, u32** *type***)**

Description

Change the packet type for the packet associated to *skb*. This comes down to setting *skb***->pkt_type** to *type*, except the eBPF program does not have a write access to *skb***->pkt_type** beside this helper. Using a helper here allows for graceful handling of errors.

The major use case is to change incoming *skb*s to **PACKET_HOST** in a programmatic way instead of having to recirculate via **redirect**(..., **BPF_F_INGRESS**), for example.

Note that *type* only allows certain values. At this time, they are:

PACKET_HOST
Packet is for us.

PACKET_BROADCAST
Send packet to all.

PACKET_MULTICAST
Send packet to group.

PACKET_OTHERHOST
Send packet to someone else.

Return 0 on success, or a negative error in case of failure.

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

Description

Check whether *skb* is a descendant of the cgroup2 held by *map* of type **BPF_MAP_TYPE_CGROUP_ARRAY**, at *index*.

Return The return value depends on the result of the test, and can be:

- 0, if the *skb* failed the cgroup2 descendant test.
- 1, if the *skb* succeeded the cgroup2 descendant test.
- A negative error code, if an error occurred.

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

Description

Retrieve the hash of the packet, *skb***->hash**. If it is not set, in particular if the hash was cleared due to mangling, recompute this hash. Later accesses to the hash can be done directly with *skb***->hash**.

Calling **bpf_set_hash_invalid**(), changing a packet prototype with **bpf_skb_change_proto**(), or calling **bpf_skb_store_bytes**() with the **BPF_F_INVALIDATE_HASH** are actions susceptible to clear the hash and to trigger a new computation for the next call to **bpf_get_hash_recalc**().

Return The 32−bit hash.

u64 bpf_get_current_task(void)

Return A pointer to the current task struct.

int bpf_probe_write_user(void *_dst_, const void *_src_, u32 _len_)

Description

Attempt in a safe way to write *len* bytes from the buffer *src* to *dst* in memory. It only works for threads that are in user context, and *dst* must be a valid user space address.

This helper should not be used to implement any kind of security mechanism because of TOC−TOU attacks, but rather to debug, divert, and manipulate execution of semi−cooperative processes.

Keep in mind that this feature is meant for experiments, and it has a risk of crashing the system and running programs. Therefore, when an eBPF program using this helper is attached, a warning including PID and process name is printed to kernel logs.

Return 0 on success, or a negative error in case of failure.

int bpf_current_task_under_cgroup(struct bpf_map *_map_, u32 _index_)

Description

Check whether the probe is being run is the context of a given subset of the cgroup2 hierarchy. The cgroup2 to test is held by *map* of type **BPF_MAP_TYPE_CGROUP_ARRAY**, at *index*.

Return The return value depends on the result of the test, and can be:

- 0, if the *skb* task belongs to the cgroup2.
- 1, if the *skb* task does not belong to the cgroup2.
- A negative error code, if an error occurred.

int bpf_skb_change_tail(struct sk_buff *_skb_, u32 _len_, u64 _flags_)

Description

Resize (trim or grow) the packet associated to *skb* to the new *len*. The *flags* are reserved for future usage, and must be left at zero.

The basic idea is that the helper performs the needed work to change the size of the packet, then the eBPF program rewrites the rest via helpers like **bpf_skb_store_bytes**(), **bpf_l3_csum_replace**(), **bpf_l3_csum_replace**() and others. This helper is a slow path utility intended for replies with control messages. And because it is targeted for slow path, the helper itself can afford to be slow: it implicitly linearizes, unclones and drops offloads from the *skb*.

A call to this helper is susceptible to change the underlaying packet buffer. Therefore, at load time, all checks on pointers previously done by the verifier are invalidated and must be performed again, if the helper is used in combination with direct packet access.

Return 0 on success, or a negative error in case of failure.

int bpf_skb_pull_data(struct sk_buff *_skb_, u32 _len_)

Description

Pull in non−linear data in case the *skb* is non−linear and not all of *len* are part of the linear section. Make *len* bytes from *skb* readable and writable. If a zero value is passed for *len*, then the whole length of the *skb* is pulled.

This helper is only needed for reading and writing with direct packet access.

For direct packet access, testing that offsets to access are within packet boundaries (test on *skb*−>**data_end**) is susceptible to fail if offsets are invalid, or if the requested data is in non−linear parts of the *skb*. On failure the program can just bail out, or in the case of a non−linear buffer, use a helper to make the data available. The **bpf_skb_load_bytes**() helper is a first solution to access the data. Another one consists in using **bpf_skb_pull_data** to pull in once the non−linear parts, then retesting and eventually access the data.

At the same time, this also makes sure the *skb* is uncloned, which is a necessary condition for direct write. As this needs to be an invariant for the write part only, the verifier detects writes and adds a prologue that is calling **bpf_skb_pull_data()** to effectively unclone the *skb* from the very beginning in case it is indeed cloned.

A call to this helper is susceptible to change the underlaying packet buffer. Therefore, at load time, all checks on pointers previously done by the verifier are invalidated and must be performed again, if the helper is used in combination with direct packet access.

Return 0 on success, or a negative error in case of failure.

s64 bpf_csum_update(struct sk_buff *_skb_, __wsum _csum_)

Description

Add the checksum *csum* into *skb*−>**csum** in case the driver has supplied a checksum for the entire packet into that field. Return an error otherwise. This helper is intended to be used in combination with **bpf_csum_diff**(), in particular when the checksum needs to be updated after data has been written into the packet through direct packet access.

Return The checksum on success, or a negative error code in case of failure.

void bpf_set_hash_invalid(struct sk_buff *_skb_)

Description

Invalidate the current *skb*−>**hash**. It can be used after mangling on headers through direct packet access, in order to indicate that the hash is outdated and to trigger a recalculation the next time the kernel tries to access this hash or when the **bpf_get_hash_recalc**() helper is called.

int bpf_get_numa_node_id(void)

Description

Return the id of the current NUMA node. The primary use case for this helper is the selection of sockets for the local NUMA node, when the program is attached to sockets using the **SO_ATTACH_REUSEPORT_EBPF** option (see also **socket(7)**), but the helper is also available to other eBPF program types, similarly to **bpf_get_smp_processor_id**().

Return The id of current NUMA node.

int bpf_skb_change_head(struct sk_buff *_skb_, u32 _len_, u64 _flags_)

Description

Grows headroom of packet associated to *skb* and adjusts the offset of the MAC header accordingly, adding *len* bytes of space. It automatically extends and reallocates memory as required.

This helper can be used on a layer 3 *skb* to push a MAC header for redirection into a layer 2 device.

All values for *flags* are reserved for future usage, and must be left at zero.

A call to this helper is susceptible to change the underlaying packet buffer. Therefore, at load time, all checks on pointers previously done by the verifier are invalidated and must

be performed again, if the helper is used in combination with direct packet access.

Return 0 on success, or a negative error in case of failure.

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

Description

Adjust (move) *xdp_md***->data** by *delta* bytes. Note that it is possible to use a negative value for *delta*. This helper can be used to prepare the packet for pushing or popping headers.

A call to this helper is susceptible to change the underlaying packet buffer. Therefore, at load time, all checks on pointers previously done by the verifier are invalidated and must be performed again, if the helper is used in combination with direct packet access.

Return 0 on success, or a negative error in case of failure.

int bpf_probe_read_str(void ****dst*, int** *size***, const void ****unsafe_ptr***)**

Description

Copy a NUL terminated string from an unsafe address *unsafe_ptr* to *dst*. The *size* should include the terminating NUL byte. In case the string length is smaller than *size*, the target is not padded with further NUL bytes. If the string length is larger than *size*, just *size*−1 bytes are copied and the last byte is set to NUL.

On success, the length of the copied string is returned. This makes this helper useful in tracing programs for reading strings, and more importantly to get its length at runtime. See the following snippet:

```
SEC("kprobe/sys_open")
void bpf_sys_open(struct pt_regs *ctx)
        char buf[PATHLEN]; // PATHLEN is defined to 256
        int res = bpf_probe_read_str(buf, sizeof(buf),
                                     ctx->di);

        // Consume buf, for example push it to
        // userspace via bpf_perf_event_output(); we
        // can use res (the string length) as event
        // size, after checking its boundaries.
```

In comparison, using **bpf_probe_read()** helper here instead to read the string would require to estimate the length at compile time, and would often result in copying more memory than necessary.

Another useful use case is when parsing individual process arguments or individual environment variables navigating *current***->mm->arg_start** and *current***->mm->env_start**: using this helper and the return value, one can quickly iterate at the right offset of the memory area.

Return On success, the strictly positive length of the string, including the trailing NUL character. On error, a negative value.

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

Description

If the **struct sk_buff** pointed by *skb* has a known socket, retrieve the cookie (generated by the kernel) of this socket. If no cookie has been set yet, generate a new cookie. Once generated, the socket cookie remains stable for the life of the socket. This helper can be useful for monitoring per socket networking traffic statistics as it provides a unique socket identifier per namespace.

Return A 8−byte long non−decreasing number on success, or 0 if the socket field is missing inside *skb*.

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

Description
Equivalent to bpf_get_socket_cookie() helper that accepts *skb*, but gets socket from **struct bpf_sock_addr** contex.

Return A 8−byte long non−decreasing number.

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

Description
Equivalent to bpf_get_socket_cookie() helper that accepts *skb*, but gets socket from **struct bpf_sock_ops** contex.

Return A 8−byte long non−decreasing number.

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

Return The owner UID of the socket associated to *skb*. If the socket is **NULL**, or if it is not a full socket (i.e. if it is a time−wait or a request socket instead), **overflowuid** value is returned (note that **overflowuid** might also be the actual UID value for the socket).

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

Description
Set the full hash for *skb* (set the field *skb***−>hash**) to value *hash*.

Return

int bpf_setsockopt(struct bpf_sock_ops ****bpf_socket*, int** *level***, int** *optname***, char ****optval***, int** *optlen***)**

Description
Emulate a call to **setsockopt()** on the socket associated to *bpf_socket*, which must be a full socket. The *level* at which the option resides and the name *optname* of the option must be specified, see **setsockopt(2)** for more information. The option value of length *optlen* is pointed by *optval*.

This helper actually implements a subset of **setsockopt()**. It supports the following *level*s:

- **SOL_SOCKET**, which supports the following *optname*s: **SO_RCVBUF**, **SO_SNDBUF**, **SO_MAX_PACING_RATE**, **SO_PRIORITY**, **SO_RCVLOWAT**, **SO_MARK**.
- **IPPROTO_TCP**, which supports the following *optname*s: **TCP_CONGESTION**, **TCP_BPF_IW**, **TCP_BPF_SNDCWND_CLAMP**.
- **IPPROTO_IP**, which supports *optname* **IP_TOS**.
- **IPPROTO_IPV6**, which supports *optname* **IPV6_TCLASS**.

Return 0 on success, or a negative error in case of failure.

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

Description
Grow or shrink the room for data in the packet associated to *skb* by *len_diff*, and according to the selected *mode*.

There is a single supported mode at this time:

- **BPF_ADJ_ROOM_NET**: Adjust room at the network layer (room space is added or removed below the layer 3 header).

All values for *flags* are reserved for future usage, and must be left at zero.

A call to this helper is susceptible to change the underlaying packet buffer. Therefore, at load time, all checks on pointers previously done by the verifier are invalidated and must be performed again, if the helper is used in combination with direct packet access.

Return 0 on success, or a negative error in case of failure.

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

Description

Redirect the packet to the endpoint referenced by *map* at index *key*. Depending on its type, this *map* can contain references to net devices (for forwarding packets through other ports), or to CPUs (for redirecting XDP frames to another CPU; but this is only implemented for native XDP (with driver support) as of this writing).

All values for *flags* are reserved for future usage, and must be left at zero.

When used to redirect packets to net devices, this helper provides a high performance increase over **bpf_redirect**(). This is due to various implementation details of the underlying mechanisms, one of which is the fact that **bpf_redirect_map**() tries to send packet as a "bulk" to the device.

Return **XDP_REDIRECT** on success, or **XDP_ABORTED** on error.

int bpf_sk_redirect_map(struct bpf_map ****map*, u32** *key***, u64** *flags***)**

Description

Redirect the packet to the socket referenced by *map* (of type **BPF_MAP_TYPE_SOCKMAP**) at index *key*. Both ingress and egress interfaces can be used for redirection. The **BPF_F_INGRESS** value in *flags* is used to make the distinction (ingress path is selected if the flag is present, egress path otherwise). This is the only flag supported for now.

Return **SK_PASS** on success, or **SK_DROP** on error.

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

Description

Add an entry to, or update a *map* referencing sockets. The *skops* is used as a new value for the entry associated to *key*. *flags* is one of:

BPF_NOEXIST
The entry for *key* must not exist in the map.

BPF_EXIST
The entry for *key* must already exist in the map.

BPF_ANY
No condition on the existence of the entry for *key*.

If the *map* has eBPF programs (parser and verdict), those will be inherited by the socket being added. If the socket is already attached to eBPF programs, this results in an error.

Return 0 on success, or a negative error in case of failure.

int bpf_xdp_adjust_meta(struct xdp_buff ****xdp_md*, int** *delta***)**

Description

Adjust the address pointed by *xdp_md***->data_meta** by *delta* (which can be positive or negative). Note that this operation modifies the address stored in *xdp_md***->data**, so the latter must be loaded only after the helper has been called.

The use of *xdp_md*->**data_meta** is optional and programs are not required to use it. The rationale is that when the packet is processed with XDP (e.g. as DoS filter), it is possible to push further meta data along with it before passing to the stack, and to give the guarantee that an ingress eBPF program attached as a TC classifier on the same device can pick this up for further post−processing. Since TC works with socket buffers, it remains possible to set from XDP the **mark** or **priority** pointers, or other pointers for the socket buffer. Having this scratch space generic and programmable allows for more flexibility as the user is free to store whatever meta data they need.

A call to this helper is susceptible to change the underlaying packet buffer. Therefore, at load time, all checks on pointers previously done by the verifier are invalidated and must be performed again, if the helper is used in combination with direct packet access.

Return 0 on success, or a negative error in case of failure.

int bpf_perf_event_read_value(struct bpf_map ***map*, u64** *flags*, **struct bpf_perf_event_value** ***buf*, u32** *buf_size*)

Description

Read the value of a perf event counter, and store it into *buf* of size *buf_size*. This helper relies on a *map* of type **BPF_MAP_TYPE_PERF_EVENT_ARRAY**. The nature of the perf event counter is selected when *map* is updated with perf event file descriptors. The *map* is an array whose size is the number of available CPUs, and each cell contains a value relative to one CPU. The value to retrieve is indicated by *flags*, that contains the index of the CPU to look up, masked with **BPF_F_INDEX_MASK**. Alternatively, *flags* can be set to **BPF_F_CURRENT_CPU** to indicate that the value for the current CPU should be retrieved.

This helper behaves in a way close to **bpf_perf_event_read**() helper, save that instead of just returning the value observed, it fills the *buf* structure. This allows for additional data to be retrieved: in particular, the enabled and running times (in *buf*->**enabled** and *buf*->**running**, respectively) are copied. In general, **bpf_perf_event_read_value**() is recommended over **bpf_perf_event_read**(), which has some ABI issues and provides fewer functionalities.

These values are interesting, because hardware PMU (Performance Monitoring Unit) counters are limited resources. When there are more PMU based perf events opened than available counters, kernel will multiplex these events so each event gets certain percentage (but not all) of the PMU time. In case that multiplexing happens, the number of samples or counter value will not reflect the case compared to when no multiplexing occurs. This makes comparison between different runs difficult. Typically, the counter value should be normalized before comparing to other experiments. The usual normalization is done as follows.

```
normalized_counter = counter * t_enabled / t_running
```

Where t_enabled is the time enabled for event and t_running is the time running for event since last normalization. The enabled and running times are accumulated since the perf event open. To achieve scaling factor between two invocations of an eBPF program, users can can use CPU id as the key (which is typical for perf array usage model) to remember the previous value and do the calculation inside the eBPF program.

Return 0 on success, or a negative error in case of failure.

int bpf_perf_prog_read_value(struct bpf_perf_event_data ***ctx*, struct bpf_perf_event_value** ***buf*, u32** *buf_size*)

Description

For en eBPF program attached to a perf event, retrieve the value of the event counter associated to *ctx* and store it in the structure pointed by *buf* and of size *buf_size*. Enabled and running times are also stored in the structure (see description of helper **bpf_perf_event_read_value**() for more details).

Return 0 on success, or a negative error in case of failure.

int bpf_getsockopt(struct bpf_sock_ops ****bpf_socket*, int** *level***, int** *optname***, char ****optval***, int** *optlen***)**

Description

Emulate a call to **getsockopt()** on the socket associated to *bpf_socket*, which must be a full socket. The *level* at which the option resides and the name *optname* of the option must be specified, see **getsockopt(2)** for more information. The retrieved value is stored in the structure pointed by *opval* and of length *optlen*.

This helper actually implements a subset of **getsockopt()**. It supports the following *level*s:

- **IPPROTO_TCP**, which supports *optname* **TCP_CONGESTION**.
- **IPPROTO_IP**, which supports *optname* **IP_TOS**.
- **IPPROTO_IPV6**, which supports *optname* **IPV6_TCLASS**.

Return 0 on success, or a negative error in case of failure.

int bpf_override_return(struct pt_reg ****regs*, u64** *rc***)**

Description

Used for error injection, this helper uses kprobes to override the return value of the probed function, and to set it to *rc*. The first argument is the context *regs* on which the kprobe works.

This helper works by setting setting the PC (program counter) to an override function which is run in place of the original probed function. This means the probed function is not run at all. The replacement function just returns with the required value.

This helper has security implications, and thus is subject to restrictions. It is only available if the kernel was compiled with the **CONFIG_BPF_KPROBE_OVERRIDE** configuration option, and in this case it only works on functions tagged with **ALLOW_ERROR_INJECTION** in the kernel code.

Also, the helper is only available for the architectures having the CONFIG_FUNCTION_ERROR_INJECTION option. As of this writing, x86 architecture is the only one to support this feature.

Return

int bpf_sock_ops_cb_flags_set(struct bpf_sock_ops ****bpf_sock*, int** *argval***)**

Description

Attempt to set the value of the **bpf_sock_ops_cb_flags** field for the full TCP socket associated to *bpf_sock_ops* to *argval*.

The primary use of this field is to determine if there should be calls to eBPF programs of type **BPF_PROG_TYPE_SOCK_OPS** at various points in the TCP code. A program of the same type can change its value, per connection and as necessary, when the connection is established. This field is directly accessible for reading, but this helper must be used for updates in order to return an error if an eBPF program tries to set a callback that is not supported in the current kernel.

The supported callback values that *argval* can combine are:

- **BPF_SOCK_OPS_RTO_CB_FLAG** (retransmission time out)
- **BPF_SOCK_OPS_RETRANS_CB_FLAG** (retransmission)
- **BPF_SOCK_OPS_STATE_CB_FLAG** (TCP state change)

Here are some examples of where one could call such eBPF program:

- When RTO fires.
- When a packet is retransmitted.
- When the connection terminates.
- When a packet is sent.
- When a packet is received.

Return Code **−EINVAL** if the socket is not a full TCP socket; otherwise, a positive number containing the bits that could not be set is returned (which comes down to 0 if all bits were set as required).

int bpf_msg_redirect_map(struct sk_msg_buff ****msg*, struct bpf_map ****map***, u32** *key***, u64** *flags***)**

Description

This helper is used in programs implementing policies at the socket level. If the message *msg* is allowed to pass (i.e. if the verdict eBPF program returns **SK_PASS**), redirect it to the socket referenced by *map* (of type **BPF_MAP_TYPE_SOCKMAP**) at index *key*. Both ingress and egress interfaces can be used for redirection. The **BPF_F_INGRESS** value in *flags* is used to make the distinction (ingress path is selected if the flag is present, egress path otherwise). This is the only flag supported for now.

Return **SK_PASS** on success, or **SK_DROP** on error.

int bpf_msg_apply_bytes(struct sk_msg_buff ****msg*, u32** *bytes***)**

Description

For socket policies, apply the verdict of the eBPF program to the next *bytes* (number of bytes) of message *msg*.

For example, this helper can be used in the following cases:

- A single **sendmsg**() or **sendfile**() system call contains multiple logical messages that the eBPF program is supposed to read and for which it should apply a verdict.
- An eBPF program only cares to read the first *bytes* of a *msg*. If the message has a large payload, then setting up and calling the eBPF program repeatedly for all bytes, even though the verdict is already known, would create unnecessary overhead.

When called from within an eBPF program, the helper sets a counter internal to the BPF infrastructure, that is used to apply the last verdict to the next *bytes*. If *bytes* is smaller than the current data being processed from a **sendmsg**() or **sendfile**() system call, the first *bytes* will be sent and the eBPF program will be re−run with the pointer for start of data pointing to byte number *bytes* **+ 1**. If *bytes* is larger than the current data being processed, then the eBPF verdict will be applied to multiple **sendmsg**() or **sendfile**() calls until *bytes* are consumed.

Note that if a socket closes with the internal counter holding a non−zero value, this is not a problem because data is not being buffered for *bytes* and is sent as it is received.

Return

int bpf_msg_cork_bytes(struct sk_msg_buff ****msg*, u32** *bytes***)**

Description

For socket policies, prevent the execution of the verdict eBPF program for message *msg* until *bytes* (byte number) have been accumulated.

This can be used when one needs a specific number of bytes before a verdict can be assigned, even if the data spans multiple **sendmsg**() or **sendfile**() calls. The extreme case would be a user calling **sendmsg**() repeatedly with 1−byte long message segments. Obviously, this is bad for performance, but it is still valid. If the eBPF program needs *bytes* bytes to validate a header, this helper can be used to prevent the eBPF program to be called again until *bytes* have been accumulated.

Return

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

Description

For socket policies, pull in non−linear data from user space for *msg* and set pointers *msg***−>data** and *msg***−>data_end** to *start* and *end* bytes offsets into *msg*, respectively.

If a program of type **BPF_PROG_TYPE_SK_MSG** is run on a *msg* it can only parse data that the (**data**, **data_end**) pointers have already consumed. For **sendmsg**() hooks this is likely the first scatterlist element. But for calls relying on the **sendpage** handler (e.g. **sendfile**()) this will be the range (**0**, **0**) because the data is shared with user space and by default the objective is to avoid allowing user space to modify data while (or after) eBPF verdict is being decided. This helper can be used to pull in data and to set the start and end pointer to given values. Data will be copied if necessary (i.e. if data was not linear and if start and end pointers do not point to the same chunk).

A call to this helper is susceptible to change the underlaying packet buffer. Therefore, at load time, all checks on pointers previously done by the verifier are invalidated and must be performed again, if the helper is used in combination with direct packet access.

All values for *flags* are reserved for future usage, and must be left at zero.

Return 0 on success, or a negative error in case of failure.

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

Description

Bind the socket associated to *ctx* to the address pointed by *addr*, of length *addr_len*. This allows for making outgoing connection from the desired IP address, which can be useful for example when all processes inside a cgroup should use one single IP address on a host that has multiple IP configured.

This helper works for IPv4 and IPv6, TCP and UDP sockets. The domain (*addr***−>sa_family**) must be **AF_INET** (or **AF_INET6**). Looking for a free port to bind to can be expensive, therefore binding to port is not permitted by the helper: *addr***−>sin_port** (or **sin6_port**, respectively) must be set to zero.

Return 0 on success, or a negative error in case of failure.

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

Description

Adjust (move) *xdp_md***−>data_end** by *delta* bytes. It is only possible to shrink the packet as of this writing, therefore *delta* must be a negative integer.

A call to this helper is susceptible to change the underlaying packet buffer. Therefore, at

load time, all checks on pointers previously done by the verifier are invalidated and must be performed again, if the helper is used in combination with direct packet access.

Return 0 on success, or a negative error in case of failure.

int bpf_skb_get_xfrm_state(struct sk_buff *_skb_, u32 _index_, struct bpf_xfrm_state *_xfrm_state_, u32 _size_, u64 _flags_)

Description

Retrieve the XFRM state (IP transform framework, see also **ip−xfrm(8)**) at *index* in XFRM "security path" for *skb*.

The retrieved value is stored in the **struct bpf_xfrm_state** pointed by *xfrm_state* and of length *size*.

All values for *flags* are reserved for future usage, and must be left at zero.

This helper is available only if the kernel was compiled with **CONFIG_XFRM** configuration option.

Return 0 on success, or a negative error in case of failure.

int bpf_get_stack(struct pt_regs *_regs_, void *_buf_, u32 _size_, u64 _flags_)

Description

Return a user or a kernel stack in bpf program provided buffer. To achieve this, the helper needs *ctx*, which is a pointer to the context on which the tracing program is executed. To store the stacktrace, the bpf program provides *buf* with a nonnegative *size*.

The last argument, *flags*, holds the number of stack frames to skip (from 0 to 255), masked with **BPF_F_SKIP_FIELD_MASK**. The next bits can be used to set the following flags:

BPF_F_USER_STACK
Collect a user space stack instead of a kernel stack.

BPF_F_USER_BUILD_ID
Collect buildid+offset instead of ips for user stack, only valid if **BPF_F_USER_STACK** is also specified.

bpf_get_stack() can collect up to **PERF_MAX_STACK_DEPTH** both kernel and user frames, subject to sufficient large buffer size. Note that this limit can be controlled with the **sysctl** program, and that it should be manually increased in order to profile long user stacks (such as stacks for Java programs). To do so, use:

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

Return A non−negative value equal to or less than *size* on success, or a negative error in case of failure.

int bpf_skb_load_bytes_relative(const struct sk_buff *_skb_, u32 _offset_, void *_to_, u32 _len_, u32 _start_header_)

Description

This helper is similar to **bpf_skb_load_bytes**() in that it provides an easy way to load *len* bytes from *offset* from the packet associated to *skb*, into the buffer pointed by *to*. The difference to **bpf_skb_load_bytes**() is that a fifth argument *start_header* exists in order to select a base offset to start from. *start_header* can be one of:

BPF_HDR_START_MAC
Base offset to load data from is *skb*'s mac header.

BPF_HDR_START_NET
Base offset to load data from is *skb*'s network header.

In general, "direct packet access" is the preferred method to access packet data, however, this helper is in particular useful in socket filters where *skb***->data** does not always point to the start of the mac header and where "direct packet access" is not available.

Return 0 on success, or a negative error in case of failure.

int bpf_fib_lookup(void *_ctx_, struct bpf_fib_lookup *_params_, int _plen_, u32 _flags_)

Description
Do FIB lookup in kernel tables using parameters in *params*. If lookup is successful and result shows packet is to be forwarded, the neighbor tables are searched for the nexthop. If successful (ie., FIB lookup shows forwarding and nexthop is resolved), the nexthop address is returned in ipv4_dst or ipv6_dst based on family, smac is set to mac address of egress device, dmac is set to nexthop mac address, rt_metric is set to metric from route (IPv4/IPv6 only), and ifindex is set to the device index of the nexthop from the FIB lookup.

plen argument is the size of the passed in struct. *flags* argument can be a combination of one or more of the following values:

BPF_FIB_LOOKUP_DIRECT
Do a direct table lookup vs full lookup using FIB rules.

BPF_FIB_LOOKUP_OUTPUT
Perform lookup from an egress perspective (default is ingress).

ctx is either **struct xdp_md** for XDP programs or **struct sk_buff** tc cls_act programs.

Return

- < 0 if any input argument is invalid
- 0 on success (packet is forwarded, nexthop neighbor exists)
- > 0 one of **BPF_FIB_LKUP_RET_** codes explaining why the packet is not forwarded or needs assist from full stack

int bpf_sock_hash_update(struct bpf_sock_ops_kern *_skops_, struct bpf_map *_map_, void *_key_, u64 _flags_)

Description
Add an entry to, or update a sockhash *map* referencing sockets. The *skops* is used as a new value for the entry associated to *key*. *flags* is one of:

BPF_NOEXIST
The entry for *key* must not exist in the map.

BPF_EXIST
The entry for *key* must already exist in the map.

BPF_ANY
No condition on the existence of the entry for *key*.

If the *map* has eBPF programs (parser and verdict), those will be inherited by the socket being added. If the socket is already attached to eBPF programs, this results in an error.

Return 0 on success, or a negative error in case of failure.

int bpf_msg_redirect_hash(struct sk_msg_buff *_msg_, struct bpf_map *_map_, void *_key_, u64 _flags_)

Description

This helper is used in programs implementing policies at the socket level. If the message *msg* is allowed to pass (i.e. if the verdict eBPF program returns **SK_PASS**), redirect it to the socket referenced by *map* (of type **BPF_MAP_TYPE_SOCKHASH**) using hash *key*. Both ingress and egress interfaces can be used for redirection. The **BPF_F_INGRESS** value in *flags* is used to make the distinction (ingress path is selected if the flag is present, egress path otherwise). This is the only flag supported for now.

Return **SK_PASS** on success, or **SK_DROP** on error.

int bpf_sk_redirect_hash(struct sk_buff *_skb_, struct bpf_map *_map_, void *_key_, u64 _flags_)

Description

This helper is used in programs implementing policies at the skb socket level. If the sk_buff *skb* is allowed to pass (i.e. if the verdeict eBPF program returns **SK_PASS**), redirect it to the socket referenced by *map* (of type **BPF_MAP_TYPE_SOCKHASH**) using hash *key*. Both ingress and egress interfaces can be used for redirection. The **BPF_F_INGRESS** value in *flags* is used to make the distinction (ingress path is selected if the flag is present, egress otherwise). This is the only flag supported for now.

Return **SK_PASS** on success, or **SK_DROP** on error.

int bpf_lwt_push_encap(struct sk_buff *_skb_, u32 _type_, void *_hdr_, u32 _len_)

Description

Encapsulate the packet associated to *skb* within a Layer 3 protocol header. This header is provided in the buffer at address *hdr*, with *len* its size in bytes. *type* indicates the protocol of the header and can be one of:

BPF_LWT_ENCAP_SEG6
IPv6 encapsulation with Segment Routing Header (**struct ipv6_sr_hdr**). *hdr* only contains the SRH, the IPv6 header is computed by the kernel.

BPF_LWT_ENCAP_SEG6_INLINE
Only works if *skb* contains an IPv6 packet. Insert a Segment Routing Header (**struct ipv6_sr_hdr**) inside the IPv6 header.

A call to this helper is susceptible to change the underlaying packet buffer. Therefore, at load time, all checks on pointers previously done by the verifier are invalidated and must be performed again, if the helper is used in combination with direct packet access.

Return 0 on success, or a negative error in case of failure.

int bpf_lwt_seg6_store_bytes(struct sk_buff *_skb_, u32 _offset_, const void *_from_, u32 _len_)

Description

Store *len* bytes from address *from* into the packet associated to *skb*, at *offset*. Only the flags, tag and TLVs inside the outermost IPv6 Segment Routing Header can be modified through this helper.

A call to this helper is susceptible to change the underlaying packet buffer. Therefore, at load time, all checks on pointers previously done by the verifier are invalidated and must be performed again, if the helper is used in combination with direct packet access.

Return 0 on success, or a negative error in case of failure.

int bpf_lwt_seg6_adjust_srh(struct sk_buff *_skb_, u32 _offset_, s32 _delta_)

Description

Adjust the size allocated to TLVs in the outermost IPv6 Segment Routing Header contained in the packet associated to *skb*, at position *offset* by *delta* bytes. Only offsets after the segments are accepted. *delta* can be as well positive (growing) as negative (shrinking).

A call to this helper is susceptible to change the underlaying packet buffer. Therefore, at load time, all checks on pointers previously done by the verifier are invalidated and must be performed again, if the helper is used in combination with direct packet access.

Return 0 on success, or a negative error in case of failure.

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

Description

Apply an IPv6 Segment Routing action of type *action* to the packet associated to *skb*. Each action takes a parameter contained at address *param*, and of length *param_len* bytes. *action* can be one of:

SEG6_LOCAL_ACTION_END_X
End.X action: Endpoint with Layer−3 cross−connect. Type of *param*: **struct in6_addr**.

SEG6_LOCAL_ACTION_END_T
End.T action: Endpoint with specific IPv6 table lookup. Type of *param*: **int**.

SEG6_LOCAL_ACTION_END_B6
End.B6 action: Endpoint bound to an SRv6 policy. Type of param: **struct ipv6_sr_hdr**.

SEG6_LOCAL_ACTION_END_B6_ENCAP
End.B6.Encap action: Endpoint bound to an SRv6 encapsulation policy. Type of param: **struct ipv6_sr_hdr**.

A call to this helper is susceptible to change the underlaying packet buffer. Therefore, at load time, all checks on pointers previously done by the verifier are invalidated and must be performed again, if the helper is used in combination with direct packet access.

Return 0 on success, or a negative error in case of failure.

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

Description

This helper is used in programs implementing IR decoding, to report a successfully decoded key press with *scancode*, *toggle* value in the given *protocol*. The scancode will be translated to a keycode using the rc keymap, and reported as an input key down event. After a period a key up event is generated. This period can be extended by calling either **bpf_rc_keydown**() again with the same values, or calling **bpf_rc_repeat**().

Some protocols include a toggle bit, in case the button was released and pressed again between consecutive scancodes.

The *ctx* should point to the lirc sample as passed into the program.

The *protocol* is the decoded protocol number (see **enum rc_proto** for some predefined values).

This helper is only available is the kernel was compiled with the **CONFIG_BPF_LIRC_MODE2** configuration option set to "**y**".

Return

int bpf_rc_repeat(void ****ctx*)**

Description

This helper is used in programs implementing IR decoding, to report a successfully decoded repeat key message. This delays the generation of a key up event for previously generated key down event.

Some IR protocols like NEC have a special IR message for repeating last button, for when a button is held down.

The *ctx* should point to the lirc sample as passed into the program.

This helper is only available is the kernel was compiled with the **CONFIG_BPF_LIRC_MODE2** configuration option set to "**y**".

Return

uint64_t bpf_skb_cgroup_id(struct sk_buff *_skb_)

Description

Return the cgroup v2 id of the socket associated with the *skb*. This is roughly similar to the **bpf_get_cgroup_classid**() helper for cgroup v1 by providing a tag resp. identifier that can be matched on or used for map lookups e.g. to implement policy. The cgroup v2 id of a given path in the hierarchy is exposed in user space through the f_handle API in order to get to the same 64−bit id.

This helper can be used on TC egress path, but not on ingress, and is available only if the kernel was compiled with the **CONFIG_SOCK_CGROUP_DATA** configuration option.

Return The id is returned or 0 in case the id could not be retrieved.

u64 bpf_skb_ancestor_cgroup_id(struct sk_buff *_skb_, int _ancestor_level_)

Description

Return id of cgroup v2 that is ancestor of cgroup associated with the *skb* at the *ancestor_level*. The root cgroup is at *ancestor_level* zero and each step down the hierarchy increments the level. If *ancestor_level* == level of cgroup associated with *skb*, then return value will be same as that of **bpf_skb_cgroup_id**().

The helper is useful to implement policies based on cgroups that are upper in hierarchy than immediate cgroup associated with *skb*.

The format of returned id and helper limitations are same as in **bpf_skb_cgroup_id**().

Return The id is returned or 0 in case the id could not be retrieved.

u64 bpf_get_current_cgroup_id(void)

Return A 64−bit integer containing the current cgroup id based on the cgroup within which the current task is running.

void* get_local_storage(void *_map_, u64 _flags_)

Description

Get the pointer to the local storage area. The type and the size of the local storage is defined by the *map* argument. The *flags* meaning is specific for each map type, and has to be 0 for cgroup local storage.

Depending on the BPF program type, a local storage area can be shared between multiple instances of the BPF program, running simultaneously.

A user should care about the synchronization by themself. For example, by using the **BPF_STX_XADD** instruction to alter the shared data.

Return A pointer to the local storage area.

int bpf_sk_select_reuseport(struct sk_reuseport_md *_reuse_, struct bpf_map *_map_, void *_key_, u64 _flags_)

Description

Select a **SO_REUSEPORT** socket from a **BPF_MAP_TYPE_REUSEPORT_ARRAY** *map*. It checks the selected socket is matching the incoming request in the socket buffer.

Return 0 on success, or a negative error in case of failure.

struct bpf_sock *bpf_sk_lookup_tcp(void *_ctx_, struct bpf_sock_tuple *_tuple_, u32 _tuple_size_, u64 _netns_, u64 _flags_)

Description

Look for TCP socket matching *tuple*, optionally in a child network namespace *netns*. The return value must be checked, and if non−**NULL**, released via **bpf_sk_release**().

The *ctx* should point to the context of the program, such as the skb or socket (depending on the hook in use). This is used to determine the base network namespace for the lookup.

tuple_size must be one of:

sizeof(*tuple***−>ipv4)**
Look for an IPv4 socket.

sizeof(*tuple***−>ipv6)**
Look for an IPv6 socket.

If the *netns* is a negative signed 32−bit integer, then the socket lookup table in the netns associated with the *ctx* will will be used. For the TC hooks, this is the netns of the device in the skb. For socket hooks, this is the netns of the socket. If *netns* is any other signed 32−bit value greater than or equal to zero then it specifies the ID of the netns relative to the netns associated with the *ctx*. *netns* values beyond the range of 32−bit integers are reserved for future use.

All values for *flags* are reserved for future usage, and must be left at zero.

This helper is available only if the kernel was compiled with **CONFIG_NET** configuration option.

Return Pointer to **struct bpf_sock**, or **NULL** in case of failure. For sockets with reuseport option, the **struct bpf_sock** result is from **reuse−>socks**[] using the hash of the tuple.

struct bpf_sock *bpf_sk_lookup_udp(void *_ctx_, struct bpf_sock_tuple *_tuple_, u32 _tuple_size_, u64 _netns_, u64 _flags_)

Description

Look for UDP socket matching *tuple*, optionally in a child network namespace *netns*. The return value must be checked, and if non−**NULL**, released via **bpf_sk_release**().

The *ctx* should point to the context of the program, such as the skb or socket (depending on the hook in use). This is used to determine the base network namespace for the lookup.

tuple_size must be one of:

sizeof(*tuple***−>ipv4)**
Look for an IPv4 socket.

sizeof(*tuple***−>ipv6)**
Look for an IPv6 socket.

If the *netns* is a negative signed 32−bit integer, then the socket lookup table in the netns associated with the *ctx* will will be used. For the TC hooks, this is the netns of the device in the skb. For socket hooks, this is the netns of the socket. If *netns* is any other signed

32−bit value greater than or equal to zero then it specifies the ID of the netns relative to the netns associated with the *ctx*. *netns* values beyond the range of 32−bit integers are reserved for future use.

All values for *flags* are reserved for future usage, and must be left at zero.

This helper is available only if the kernel was compiled with **CONFIG_NET** configuration option.

Return Pointer to **struct bpf_sock**, or **NULL** in case of failure. For sockets with reuseport option, the **struct bpf_sock** result is from **reuse−>socks**[] using the hash of the tuple.

int bpf_sk_release(struct bpf_sock *_sock_)

Description

Release the reference held by *sock*. *sock* must be a non−**NULL** pointer that was returned from **bpf_sk_lookup_xxx**().

Return 0 on success, or a negative error in case of failure.

int bpf_map_pop_elem(struct bpf_map *_map_, void *_value_)

Description

Pop an element from *map*.

Return 0 on success, or a negative error in case of failure.

int bpf_map_peek_elem(struct bpf_map *_map_, void *_value_)

Description

Get an element from *map* without removing it.

Return 0 on success, or a negative error in case of failure.

int bpf_msg_push_data(struct sk_buff *_skb_, u32 _start_, u32 _len_, u64 _flags_)

Description

For socket policies, insert *len* bytes into *msg* at offset *start*.

If a program of type **BPF_PROG_TYPE_SK_MSG** is run on a *msg* it may want to insert metadata or options into the *msg*. This can later be read and used by any of the lower layer BPF hooks.

This helper may fail if under memory pressure (a malloc fails) in these cases BPF programs will get an appropriate error and BPF programs will need to handle them.

Return 0 on success, or a negative error in case of failure.

int bpf_msg_pop_data(struct sk_msg_buff *_msg_, u32 _start_, u32 _pop_, u64 _flags_)

Description

Will remove *pop* bytes from a *msg* starting at byte *start*. This may result in **ENOMEM** errors under certain situations if an allocation and copy are required due to a full ring buffer. However, the helper will try to avoid doing the allocation if possible. Other errors can occur if input parameters are invalid either due to *start* byte not being valid part of *msg* payload and/or *pop* value being to large.

Return 0 on success, or a negative error in case of failure.

int bpf_rc_pointer_rel(void *_ctx_, s32 _rel_x_, s32 _rel_y_)

Description

This helper is used in programs implementing IR decoding, to report a successfully decoded pointer movement.

The *ctx* should point to the lirc sample as passed into the program.

This helper is only available is the kernel was compiled with the **CONFIG_BPF_LIRC_MODE2** configuration option set to "**y**".

Return

EXAMPLES

Example usage for most of the eBPF helpers listed in this manual page are available within the Linux kernel sources, at the following locations:

- *samples/bpf/*
- *tools/testing/selftests/bpf/*

LICENSE

eBPF programs can have an associated license, passed along with the bytecode instructions to the kernel when the programs are loaded. The format for that string is identical to the one in use for kernel modules (Dual licenses, such as "Dual BSD/GPL", may be used). Some helper functions are only accessible to programs that are compatible with the GNU Privacy License (GPL).

In order to use such helpers, the eBPF program must be loaded with the correct license string passed (via **attr**) to the **bpf**() system call, and this generally translates into the C source code of the program containing a line similar to the following:

```
char ____license[] __attribute__((section("license"), used)) = "GPL";
```

IMPLEMENTATION

This manual page is an effort to document the existing eBPF helper functions. But as of this writing, the BPF sub−system is under heavy development. New eBPF program or map types are added, along with new helper functions. Some helpers are occasionally made available for additional program types. So in spite of the efforts of the community, this page might not be up−to−date. If you want to check by yourself what helper functions exist in your kernel, or what types of programs they can support, here are some files among the kernel tree that you may be interested in:

- *include/uapi/linux/bpf.h* is the main BPF header. It contains the full list of all helper functions, as well as many other BPF definitions including most of the flags, structs or constants used by the helpers.
- *net/core/filter.c* contains the definition of most network−related helper functions, and the list of program types from which they can be used.
- *kernel/trace/bpf_trace.c* is the equivalent for most tracing program−related helpers.
- *kernel/bpf/verifier.c* contains the functions used to check that valid types of eBPF maps are used with a given helper function.
- *kernel/bpf/* directory contains other files in which additional helpers are defined (for cgroups, sockmaps, etc.).

Compatibility between helper functions and program types can generally be found in the files where helper functions are defined. Look for the **struct bpf_func_proto** objects and for functions returning them: these functions contain a list of helpers that a given program type can call. Note that the **default:** label of the **switch ... case** used to filter helpers can call other functions, themselves allowing access to additional helpers. The requirement for GPL license is also in those **struct bpf_func_proto**.

Compatibility between helper functions and map types can be found in the **check_map_func_compatibility**() function in file *kernel/bpf/verifier.c*.

Helper functions that invalidate the checks on **data** and **data_end** pointers for network processing are listed in function **bpf_helper_changes_pkt_data**() in file *net/core/filter.c*.

SEE ALSO

bpf(2), **cgroups**(7), **ip**(8), **perf_event_open**(2), **sendmsg**(2), **socket**(7), **tc−bpf**(8)

COLOPHON

This page is part of release 5.02 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man−pages/.