BPF Graph Collections + Verifier Changes

LSFMMBPF 2023

David Marchevsky
Software Engineer

Meta

TL;DR

What’s so bad about BPF maps?

How do new-style data structures look?

What interesting verifier changes were necessary to implement these data structures?

Motivation: sched_ext

LWN’s “The extensible scheduler class” is great summary

Tejun: “Why can’t this look like normal kernel code?”

Problem with maps: unfamiliarity

Some data structures fit naturally into the map API (e.g. HASH, ARRAY), others less so

Programs interacting with the latter group of data structures can be hard to parse for kernel programmers without previous BPF experience, and unwieldy to use otherwise

Interaction with data structures is a big part of familiarity / understandability

Problem with maps: inflexibility

When the map API was developed BPF programs were less complex than they are now, and there were constraints on BPF programs that are no longer relevant

- e.g. `max_elems` to declare size ahead of time

When new data structures are added to the BPF environment, this tends to lead to a “square peg / round hole” problem, as not all generic map helpers are relevant or convenient for many kinds of data structures

- e.g. what should `bpf_map_delete_elem` do for a bloom filter map?
- How can a rbtree map support custom comparators for adding a node to a tree when `bpf_map_update_elem` provides no such facility?

Problem with maps: inflexibility

Maps and the (generic or custom) helper functions which manipulate them are UAPI

- As a result our ability to back out of suboptimal architecture or implementation decisions is limited. This raises the cost of adding a new helper or map.

We now have kfuncs (“unstable helpers”), BTF, and kptrs, so the above limitation is self-imposed

- See Alexei’s LPC 2022 presentation for more on this

Object lifetime tied to map lifetime

BPF linked list and rbtree

- kfuncs for interaction
- use BPF any-context allocator
  - No more `max_elems`, allocate nodes yourself and put them in the collection
- "Intrusive" nodes - define your own struct w/ `bpf_{list, rb}_node` field
- Locking exposed to BPF program writer - grab spinlock yourself

Show me the code: Definition

```
struct node_data {
  long key;
  long data;
  struct bpf_rb_node node;
};

#define private(name) SEC(".data." #name) __hidden
__attribute__((aligned(8)))
private(A) struct bpf_spin_lock glock;
private(A) struct bpf_rb_root groot __contains(node_data, node);
```

User-defined type w/ bpf_rb_node

spin_lock can’t be mmap’d to userspace, hence private section

rb_root in same section (map_value, really) as spin_lock -> lock protects the tree

__contains -> BTF tag that ties tree to node type

selftests/bpf/progs/rbtree.c

Show me the code: Allocate some nodes

```
long rbtree_first_and_remove(void *ctx)
{
   struct bpf_rb_node *res = NULL;
   struct node_data *n, *m, *o;

   n = bpf_obj_new(typeof(*n));
   if (!n)
      return 1;
   n->key = 3;

   m = bpf_obj_new(typeof(*m));
   if (!m)
      goto err_out;
   m->key = 5;

   o = bpf_obj_new(typeof(*o));
   if (!o)
      goto err_out;
   o->key = 1;
```

bpf_obj_new is a wrapper around BPF allocator

selftests/bpf/progs/rbtree.c

Show me the code: Add nodes to tree

```
bpf_spin_lock(&glock);
bpf_rbtree_add(&groot, &n->node, less);
bpf_rbtree_add(&groot, &m->node, less);
bpf_rbtree_add(&groot, &o->node, less);

res = bpf_rbtree_first(&groot);
if (!res) {
   bpf_spin_unlock(&glock);
   return 2;
}

o = container_of(res, struct node_data, node);

res = bpf_rbtree_remove(&groot, &o->node);
bpf_spin_unlock(&glock);
```

Verifier will reject programs which don't hold lock assoc'd w/ tree

rbtree_add passes ownership of node's lifetime to the tree

Looks more/less like normal kernel code

selftests/bpf/progs/rbtree.c

Show me the code: Shared ownership for nodes*

```
struct node_data {
   long key;
   long list_data;
   struct bpf_rb_node r;
   struct bpf_list_node l;
   struct bpf_refcount ref;
};
```

bpf_refcount implements shared ownership

selftests/bpf/progs/rbtree.c

Show me the code: Shared ownership for nodes*

```
struct node_data *n, *m;

n = bpf_obj_new(typeof(*n));
if (!n)
  return -1;

m = bpf_refcount_acquire(n);
m->key = 123;
m->list_data = 456;

bpf_spin_lock(lock);
if (bpf_rbtree_add(root, &n->r, less)) { /* snip */ }
bpf_spin_unlock(lock);

bpf_spin_lock(lock);
if (bpf_list_push_front(head, &m->l)) { /* snip */ }
bpf_spin_unlock(lock);
```

bpf_refcount_acquire -> bump refcount

selftests/bpf/progs/refcounted_kptr.c

Interesting Verifier Changes

`bpf_obj_new, bpf_obj_drop`

- Give me a typed object (type can be user-defined)

`btf_field` and `btf_record`

- Does this user-defined type contain any special fields? (spin_lock, rb node)
- If so, where?

Strong and weak references for local kptrs

- n.b.: called "owning" "non-owning" in code currently
- Express ownership / lack of ownership over kptr's lifetime

Shared ownership w/ `bpf_refcount*`

- Integration w/ `bpf_obj_new`, `bpf_obj_drop`

Can you use any of these to ease implementation of your idea?