Unifying kfunc and helper defs

LSFMMBPF 2023

David Marchevsky
Software Engineer

TL;DR

Why does the verifier need to know about functions BPF programs can call?

What does it need to know?

How do we currently give it this information? Pros/cons of different approaches?

What parts of current approaches are kludges or implementation details? Can we pick an approach?

Why does the verifier need to know about functions BPF programs can call?

Try writing safe helpers / kfuncs without any guarantees about input or calling context

Or writing BPF progs that know nothing about helper output

The verifier needs to know:

- For each function argument, what verification logic is necessary?
- How about the function's return value? If it returns ptr, can it be NULL?
- Does the function acquire or release any resources? If so, which?
- Any other function-specific verification logic
  - Catchall for anything that can't be expressed more generally
  - Usually hardcoded

Current Approach: bpf_func_proto for helpers

```
const struct bpf_func_proto bpf_map_lookup_elem_proto = {
        .func           = bpf_map_lookup_elem,
        .gpl_only       = false,
        .pkt_access     = true,
        .ret_type       = RET_PTR_TO_MAP_VALUE_OR_NULL,
  /* .ret_type = PTR_MAYBE_NULL | RET_PTR_TO_MAP_VALUE */
        .arg1_type      = ARG_CONST_MAP_PTR,
        .arg2_type      = ARG_PTR_TO_MAP_KEY,
};
```

Pretty standard helper definition

Note that type flag is used to express "maybe NULL"

Base type can be OR'd with type flags to modify verification logic

.

kernel/bpf/helpers.c

Current Approach: bpf_func_proto for helpers

```
static const struct bpf_func_proto bpf_kptr_xchg_proto = {
        .func           = bpf_kptr_xchg,
        .gpl_only       = false,
        .ret_type       = RET_PTR_TO_BTF_ID_OR_NULL,
        .ret_btf_id     = BPF_PTR_POISON,
        .arg1_type      = ARG_PTR_TO_KPTR,
        .arg2_type      = ARG_PTR_TO_BTF_ID_OR_NULL | OBJ_RELEASE,
        .arg2_btf_id    = BPF_PTR_POISON,
};
```

A more complicated proto

Some arg type handling isn't expressed in static, explicit definition. Here, BPF_PTR_POISON arg2 and ret are replaced by user-defined type by verifier.

OBJ_RELEASE type flag marks resource being released

kernel/bpf/helpers.c

OBJ_RELEASE - what about acquire?

```
static bool is_acquire_function(enum bpf_func_id func_id,
        const struct bpf_map *map)
{
  enum bpf_map_type map_type = map ? map->map_type :
BPF_MAP_TYPE_UNSPEC;

  if (func_id == BPF_FUNC_sk_lookup_tcp ||
      func_id == BPF_FUNC_sk_lookup_udp ||
      func_id == BPF_FUNC_skc_lookup_tcp ||
      func_id == BPF_FUNC_ringbuf_reserve ||
      func_id == BPF_FUNC_kptr_xchg)
    return true;
```

func_id == BPF_FUNC_whatever -> “other function-specific verification logic”

kernel/bpf/verifier.c

	Helpers	Kfuncs
Args / Retval	Type and flags expressed via enums in helper proto	
Can return NULL?	PTR_MAYBE_NULL type flag	
Acquire	Function-specific (is_acquire_function and others)	
Release	OBJ_RELEASE type flag	
Other function-specific verification?	meta->func_id == BPF_FUNC_spin_unlock	
Summary	Mostly explicit definition in bpf_func_proto	

Current Approach: BTF for kfuncs

```
__bpf_kfunc struct bpf_cpumask *bpf_cpumask_create(void)
__bpf_kfunc u32 bpf_cpumask_first(const struct cpumask *cpumask)

BTF_ID_FLAGS(func, bpf_cpumask_create, KF_ACQUIRE | KF_RET_NULL)
BTF_ID_FLAGS(func, bpf_cpumask_first, KF_RCU)
```

Pretty standard kfunc definitions

- BTF type of params / retval drives verification logic
- Type isn't always sufficient, so flags exist here too

Kfunc flags used to express “acquire” and “maybe returns NULL”

.

kernel/bpf/cpumask.c

Current Approach: BTF for kfuncs

```
__bpf_kfunc void *bpf_dynptr_slice(const struct bpf_dynptr_kern *ptr, u32 offset,
                                   void *buffer, u32 buffer__szk)
__bpf_kfunc void bpf_obj_drop_impl(void *p__alloc, void *meta__ign)

BTF_ID_FLAGS(func, bpf_obj_drop_impl, KF_RELEASE)
BTF_ID_FLAGS(func, bpf_dynptr_slice, KF_RET_NULL)
```

Some more tricky definitions

__suffixes in param name adjust BTF type-based verification logic
__szk -> this u32 contains size of 'buffer'
__alloc -> object was allocated using bpf_obj_new
__ign -> ignore type-specific logic entirely

Kfunc flag used to express "release" too

.

kernel/bpf/helpers.c

	Helpers	Kfuncs
Args / Retval	Type and flags expressed via enums in helper proto	Look at BTF types of the kfunc __suffixes modify type-specific logic
Can return NULL?	PTR_MAYBE_NULL type flag	KF_RET_NULL kfunc flag
Acquire	Function-specific (is_acquire_function and others)	KF_ACQUIRE kfunc flag
Release	OBJ_RELEASE type flag	KF_RELEASE kfunc flag
Other function-specific verification?	meta->func_id == BPF_FUNC_spin_unlock	btf_id == special_kfunc_list[KF_bpf_rbtree_remove]
Summary	Mostly explicit definition in bpf_func_proto	Mostly implicit based on BTF types, w/ caveats

Problems - function-level vs arg-level properties

```
__bpf_kfunc void bpf_obj_drop_impl(void *p__alloc, void *meta__ign)
BTF_ID_FLAGS(func, bpf_obj_drop_impl, KF_RELEASE)
```

What’s being released?

Verifier looks for arg w/ ref_obj_id != 0, presumably it’s been acquired

What if multiple args have ref_obj_id != 0?

“verifier internal error: more than one arg with ref_obj_id”

Helpers don’t have this issue

```
          .arg2_type     = ARG_PTR_TO_BTF_ID_OR_NULL | OBJ_RELEASE,
```

But they don’t have proper func-level properties either

“Is this helper a release function” -> “Does the helper have an OBJ_RELEASE-flagged arg?”

Problems - __suffixes

```
__bpf_kfunc void bpf_obj_drop_impl(void *p__alloc, void *meta__ign)
BTF_ID_FLAGS(func, bpf_obj_drop_impl, KF_RELEASE)
```

__suffixes are a partial workaround for lack of arg-level flags

If the above function returned void *, how to tag it __alloc?

Problems - Lots of duplicated logic

`check_helper_call` and `check_kfunc_call` in `verifier.c` do the same thing

Where to go from here

What’s the desired end state?

Which parts of current implementations are kludges or implementation details?

I don’t have historical context, so asked Alexei

Desired end state

Strong preference for using information exposed by C language

Why? So path to making any arbitrary kernel function callable from BPF is as short as possible

If we need to annotate functions, ideally we'd do so in a generally-useful way

e.g. `sparse` tool and `__rcu`

Kludge, Historical Artifact, Implementation Detail

`bpf_func_proto:` Predates BTF, doesn’t leverage C type info

`__suffixes:` BTF tags are better

kfunc flags: Implementation detail

`BTF_ID_SET` to expose kfunc: Something like `EXPORT_SYMBOL` would be better

Only thing that’s particularly blessed is use of BTF type info

TODO: Unify function definitions, dedupe `check_{helper,kfunc}_call`

Help? Questions? Opinions?