BPF struct_ops

new features driven by sched_ext in last few months

Kui-Feng Lee

Related Projects

- In last few months, struct_ops were driven by it’s applications a lot.
- sched_ext is very active
- Fuse-BPF
- BPF qdisc

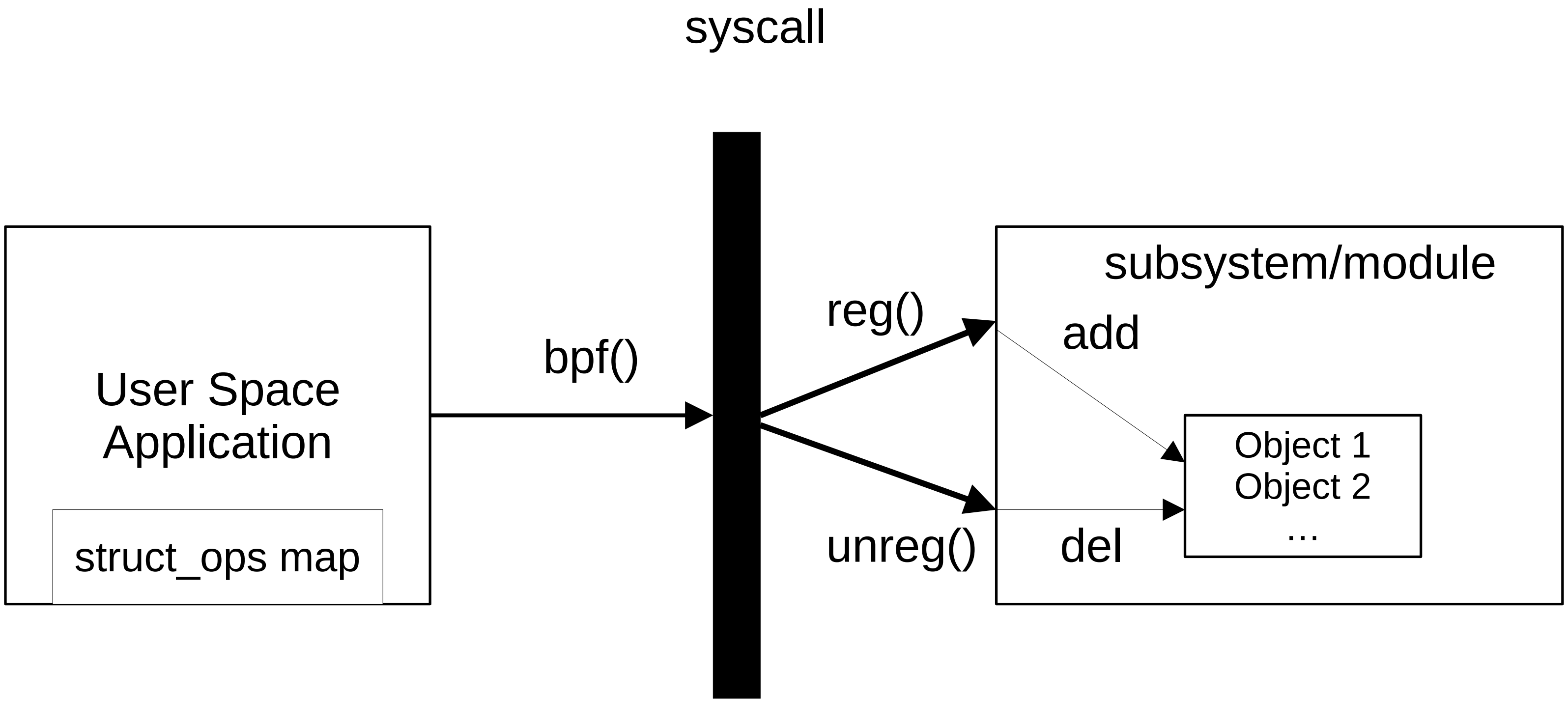
Quick Introduction

- With struct_ops, you, as a module or a subsystem, can call operators of an interface. And, the interface has been implemented in BPF as struct_ops maps.

```
struct dummy_ops {
    int (*add)(int v1, int v2);
    int (*sub)(int v1, int v2);
}
```

```
Int dummy_ops__reg(void *kdata)
{
    struct dummy_ops *ops = kdata;
    Int v;

    if (ops->add) {
        v = ops->add(7, 8);
        if (v != 15)
            return -EINVAL;
    }
    if (ops->sub) {
        v = ops->sub(7, 8);
        if (v != -1)
            return -EINVAL;
    }
    return 0;
}
```



syscall

User Space
Application

struct_ops map

bpf()

reg()

unreg()

subsystem/module

add

del

Object 1
Object 2
…

New Features

- sched_ext has driven a lot of new features of struct_ops
- Last few months
  - Shadow variables
  - Null arguments
  - Large number of programs (operators)
  - Use struct_ops from kernel modules
  - Epoll & link detachment
  - … more

Shadow variables

- Previous, the following were not allowed through skeletons
  - Change the values of data fields
  - Assign functions to operators

```
struct dummy_ops {
  int flags;
  int (*start)(void);
};
```

```
SEC(“.struct_ops.link”)
struct dummy_ops my_ops = {
  .flags = 0x10,
  .start = (void*)&my_start,
};
```

What subsystems should do

```
struct dummy_ops {
    int flags;
    int (*start)(void);
};
```

```
int dummy_ops_init_member(const struct btf_type *t,
    const struct btf_member *member,
     void *kdata, const void *udata)
{
    if (member->offset = offsetof(struct dummy_ops, flags) * 8) {
        ((struct dummy_ops *)kdata)->flags =
                   ((struct dummy_ops *)udata)->flags;
        return 1;
    }
    return 0;
}


struct dummy_ops my_dummy_ops = {
    ……
    .init_member = dummy_ops_init_member,
    …...
};
```

What user space should do

```
/* dummy_ops_prog.c */
int first_start(void) { … }
int second_start(void) { … }

SEC(“.struct_ops.link”)
struct dummy_ops dummy_1 = {
    .flags = 0x10,
    .start = &first_start
};

/* loader.c */
skel = dummy_ops_prog__open();
skel->struct_ops.dummy_1->flags |= 0x3;
skel->struct_ops.dummy_1->start = skel->progs.second_start;
err = struct_ops_module__load(skel);
```

Null arguments

- All arguments were trusted previously.
- Passing a null pointer to a struct_ops operator might cause a crash.

Annotate arguments

- You can annotate an argument as nullable to pass a null pointer.
- The verifier enforces BPF programs check the pointer before accessing the buffer.

What subsystems should do

```
struct bpf_testmod_ops {
    ……
    int (*test_maybe_null)(int, struct task_struct *),
    ……
};
```

cfi stub

```
int bpf_testmod_ops__test_maybe_null(int dummy,
    struct task_struct *task__nullable)
{
    return 0;
}

struct bpf_testmod_ops __bpf_testmod_ops = {
    ……
    .test_maybe_null = bpf_testmod_ops__test_maybe_null,
    ……
};

struct bpf_struct_ops testmod_ops = {
    ……
    .cfi_stubs = &__bpf_testmod_ops,
    ……
};
```

BPF Program

```
int maybe_null_op(int dummy, struct task_struct *task) {
    …...
    if (task)
        use_pid(task->pid);
    …...
}
```

Large number of programs

- All trampolines of operators in a struct_ops map should be in a memory page.
- You could have less than 20 operators with x86_64 platform.
- Now, it supports up to 8 pages for trampolines of a struct_ops map.

struct_ops from modules

- Kernel modules can now define their struct_ops types and receives struct_ops objects of these types.
- selftests/bpf/bpf_testmod.c is a good example.

```
struct bpf_struct_ops bpf_bpf_testmod_ops = {
        .verifier_ops = &bpf_testmod_verifier_ops,
        .init = bpf_testmod_ops_init,
        .init_member = bpf_testmod_ops_init_member,
        .reg = bpf_dummy_reg,
        .unreg = bpf_dummy_unreg,
        .cfi_stubs = &__bpf_testmod_ops,
        .name = "bpf_testmod_ops",
        .owner = THIS_MODULE,
};

static int bpf_testmod_init(void)
{
            ......
        ret = register_bpf_struct_ops(&bpf_bpf_testmod_ops, bpf_testmod_ops);
             ......
}
```

Compatibility

- APIs/types evolve over time.
- struct_ops types may add operators or arguments.

Extra arguments

- Add one or more arguments to an existing operator
- Run an old implementation with a new kernel
- The signature has been changed
- The verifier checks behavior, not signature

```
/* v1 */
struct player {
   int (*play)(int track),
}

/* v2 */
struct player {
  int (*play)(int track, int volume),
}
```

New operators

- Add new operators to an existing struct_ops type.
- A type in the kernel has more fields/operators than the corresponding types in BPF programs.
- Libbpf would reset these additional fields/operators to 0s before loading the struct_ops map.
- Libbpf would ignore zeroed additional fields absent in the kernel (values are 0s)

```
/* player_v1.c */
struct player {
  int (*play)(int track);
};

SEC(“.struct_ops.link”)
struct player player_old = {
  .player = (void *)player_play,
};

/* player_v2.c */
struct player {
  int (*play)(int track);
  int (*stop)(void);
};

SEC(“.struct_ops.link”)
struct player player_new = {
  .player = (void *)player_play,
  .stop = NULL,
};
```

- Load player_v1.c with the v2 kernel
- Load player_v2.c with the v1 kernel if stop is NULL

Types with suffices

- Libbpf would skip the suffices in the pattern “___XXXX” (3 underlines)
- “player___v1” and “player___v2” would be mapped to “player” in the kernel.
- Enable developers to has multiple definitions for the same struct_ops type
- Thanks to Eduard Zingerman

```
struct player_v1 {
  int (*play)(int track);
};

struct player_v2 {
  int (*play)(int track);
  int (*stop)(void);
};
```

```
SEC(“.struct_ops.link”)
struct player_v1 player_old = {
  .player = (void *)player_play,
};

SEC(“.struct_ops.link”)
struct player_v2 player_new = {
  .player = (void *)player_play,
  .stop = (void *)player_stop,
};
```

What is on the way

Epoll

- Send EPOLLHUP if a struct_ops link has been detached.
- Why?
  - Modules & subsystems may proactively deactivate struct_ops objects registered to them.
  - User space programs may want to know the deactivation.

Epoll with detachment

- You can detach a struct_ops link from user space programs
- Kernel modules or subsystems can detach a struct_ops link as well. (deactivate a struct_ops object)

What subsystems should do

- Receive an additional argument from reg()/update()/unreg().
  - A pointer to a bpf link

```
struct bpf_struct_ops {
        ……
        int (*reg)(void *kdata, struct bpf_link *link);
        void (*unreg)(void *kdata, struct bpf_link *link);
        int (*update)(void *kdata, void *old_kdata, struct bpf_link *link);
        ……
};
```

```
__bpf_kfunc int bpf_dummy_do_link_detach(void)
{
        struct bpf_link *link;
        int ret = -ENOENT;

        spin_lock(&detach_lock);
        link = link_to_detach;
        /* Make sure the link is still valid by increasing its refcnt */
        if (link && IS_ERR(bpf_link_inc_not_zero(link)))
                link = NULL;
        spin_unlock(&detach_lock);

        if (link) {
                ret = link->ops->detach(link);
                bpf_link_put(link);
        }

        return ret;
}
```

What user space progs should do

```
skel = struct_ops_detach__open_and_load();
link = bpf_map__attach_struct_ops(skel->maps.testmod_do_detach);
fd = bpf_link__fd(link);

epollfd = epoll_create1(0);
ev.events = EPOLLHUP;
ev.data.fd = fd;
err = epoll_ctl(epollfd, EPOLL_CTL_ADD, fd, &ev);
if (!ASSERT_OK(err, "epoll_ctl"))
        goto cleanup;

/* Wait for EPOLLHUP */
nfds = epoll_wait(epollfd, events, 2, 500);
```

Questions?