Multiple XDP programs per interface: Status and outstanding issues

Toke Høiland-Jørgensen

Linux Plumbers Conference - Networking and BPF Summit
August 2020

Outline

- Problem statement
- Currently implemented solution
- Outstanding issues and discussion

Problem statement

Why do we need more than one XDP program on each interface?

Why do we need multiple XDP programs

There is only one XDP hook per netdev, so an application that wants to use XDP has to own the XDP hook.

But what if a user wants to install more than one such application?

For example - it should be possible to run all these at the same time:

- XDP-based DDOS protection
- XDP-accelerated IDS (e.g., Suricata)
- Custom XDP program

This will make it more attractive to add XDP support.

Prior state of the art

In most large deployments of XDP, all programs are written by the same people inside an org.

But even here, we have seen a need for running multiple programs:

- Katran xdp_root
- Cloudflare xdpdump

Both rely on tail calls – i.e., earlier programs need to know about later ones.

Design goals

From my talk at LPC 2019:

> *High-level goal: execute multiple eBPF programs in a single XDP hook.*
>
> *With the following features:*
>
> 1. Arbitrary execution order
>   - *Must be possible to change the order dynamically*
>   - *Execution chain can depend on program return code*
> 2. *Should work without modifying the programs themselves*

The solution

What works today, and how does it work?

New kernel features used for multiprog

- BPF `freplace` function replacement
  - Load one BPF program to replace a function in another
- Atomic replace of XDP programs
  - Supply expected existing program FD when attaching XDP program
  - Or use `bpf_link` XDP attachment

With this, we can build multi-prog support in userspace at BPF program load time.

The XDP dispatcher

```
static const struct xdp_dispatcher_config {
        __u8 num_progs_enabled;
        __u32 chain_call_actions[10];     /* bitmask of actions to chain call */
        __u32 run_prios[10];              /* priority (for sorting programs in execution order) */
} conf = {}; /* populated at load */

int prog0(struct xdp_md *ctx) { return XDP_PASS; } /* repeat for prog0()..prog9() functions */

SEC("xdp/dispatcher")
int xdp_dispatcher(struct xdp_md *ctx)
{
        int ret;

        /* handle prog0 */
        if (conf.num_progs_enabled < 1) /* for verifier dead code elimination */
                goto out;
        ret = prog0(ctx);
        if (!((1U << ret) & conf.chain_call_actions[0]))
                return ret;
        /* end prog0 - repeat for prog1..prog9 */
out:
        return XDP_PASS;
}
```

Loading the dispatcher

```c
int load_dispatcher(int num_progs, struct xdp_dispatcher_config *config)
{
  struct bpf_object *obj;
  struct bpf_map *map;

  obj = bpf_object__open("xdp-dispatcher.o");
  map = bpf_map__next(NULL, obj); /* map backing global data in BPF prog */

  config->num_progs_enabled = num_progs;
  for (int i = 0; i < num_progs; i++) {
    if (config->chain_call_cations[i])
      continue; /* already set, should be the common case */

    /* defaults - in reality, get from actual programs, see later slide */
    config->chain_call_actions[i] = (1U << XDP_PASS);
    config->run_prios[i] = 50;
  }

  bpf_map__set_initial_value(map, &config, sizeof(config));

  bpf_object__load(obj);
  return bpf_program__fd(bpf_object__find_prog_by_idx(obj, 0));
}
```

Attaching component program (single prog)

```c
int attach_prog_to_dispatcher(struct bpf_object *bpf_obj)
{
  struct bpf_program *bpf_prog; struct xdp_dispatcher_config config = {};
  int dispatcher_fd, link_fd, num_progs = 1;

  bpf_prog = bpf_object__find_program_by_idx(bpf_obj, 0);
  dispatcher_fd = load_dispatcher(num_progs, &config);

  /* link program into dispatcher */
  bpf_program__set_attach_target(bpf_prog, dispatcher_fd, "prog0");
  bpf_program__set_type(bpf_prog, BPF_PROG_TYPE_EXT);
  bpf_object__load(bpf_obj);
  link_fd = bpf_raw_tracepoint_open(NULL, bpf_program__fd(bpf_prog));

  /* pin link */
  bpf_obj_pin(bpf_program__fd(bpf_prog), "/sys/fs/bpf/xdp/dispatch-IFINDEX-DID/prog0-prog");
  bpf_obj_pin(link_fd, "/sys/fs/bpf/xdp/dispatch-IFINDEX-DID/prog0-link");

  /* now the dispatcher_fd is ready to be attached to the interface */
  return dispatcher_fd;
}
```

Adding another program (doesn't work yet)

```c
int attach_second_program(int old_dispatcher_fd, int new_prog_fd)
{
  struct xdp_dispatcher_config old_config = {};
  int map_fd, prog_fds[2] = { -1, new_prog_fd };
  __u32 map_key = 0;
  char buf[100];

  map_fd = get_map_from_prog_id(old_dispatcher);
  bpf_map_lookup_elem(map_fd, &map_key, &old_config);

  sprintf(buf, "/sys/fs/xdp/dispatch-%d-%d/prog0-prog", ifindex, get_prog_id(old_dispatcher_fd));
  prog_fds[0] = bpf_object_get(buf);

  /* determine order of progs - old prog prio from old_config, new from prog BTF */
  sort_by_run_prio(&prog_fds, &old_config);
  new_dispatcher_fd = load_dispatcher(2, &old_config);

  /* support for this is still missing from the kernel (see later slide) */
  bpf_raw_tracepoint_open(NULL, prog_fds[0], new_dispatcher_fd, get_btf_id("prog0"));
  bpf_raw_tracepoint_open(NULL, prog_fds[1], new_dispatcher_fd, get_btf_id("prog1"));

  return new_dispatcher_fd;
}
```

Attaching to an interface

```
int attach_to_interface(int ifindex, struct bpf_object *bpf_obj)
{
    int err, new_dispatcher_fd, old_dispatcher_id, old_dispatcher_fd = -1, xdp_flags = 0;

retry:
    old_dispatcher_id = get_prog_id_from_ifindex(ifindex);
    if (old_dispatcher_id) {
        struct bpf_program *prog = bpf_object__find_program_by_idx(bpf_obj, 0);
        old_dispatcher_fd = bpf_prog_get_fd_by_id(old_dispatcher_id);
        new_dispatcher_fd = attach_second_program(old_dispatcher_fd, bpf_program__fd(prog));
    } else {
        xdp_flags = XDP_FLAGS_UPDATE_IF_NOEXIST;
        new_dispatcher_fd = attach_prog_to_dispatcher(bpf_obj);
    }

    /* atomic replace of old dispatcher (or none) with new */
    DECLARE_LIBBPF_OPTS(bpf_xdp_set_link_opts, opts, .old_fd = old_dispatcher_fd);
    err = bpf_set_link_xdp_fd_opts(ifindex, new_dispatcher_fd, xdp_flags, &opts);
    if (err && errno == EEXIST)
        goto retry; /* replaced since we queried ifindex, start over */

    return err;
}
```

Determining program order and actions

BPF programs encode priority and chain call actions in BTF.

```c
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>
#include <xdp/xdp_helpers.h> /* #define XDP_RUN_CONFIG(f) _CONCAT(_,f) SEC(".xdp_run_config") */

struct {
        __uint(priority, 10);
        __uint(XDP_PASS, 1);
} XDP_RUN_CONFIG(xdp_pass); /* from xdp_helpers.h - struct name + section, like BTF map def */

SEC("prog")
int xdp_pass(struct xdp_md *ctx)
{
    return XDP_PASS;
}

char _license[] SEC("license") = "GPL";
```

These serve as defaults when loading programs onto an interface.

The libxdp library

The libxdp library encapsulates all this:

```c
int main()
{
  struct xdp_program *prog;
  int err;

  /* load from file: */
  prog = xdp_program__open_file("my-program.o", "section_name", NULL);
  /* ...or, if using custom libbpf loading, create from BPF obj: */
  prog = xdp_program__from_bpf_obj(my_obj, "section_name");

  /* optionally modify XDP program metadata before load */
  xdp_program__set_run_prio(prog, 100);
  xdp_program__set_chain_call_enabled(prog, XDP_PASS, true);

  /* load and attach program */
  err = xdp_program__attach(prog, IFINDEX, XDP_MODE_NATIVE, 0);

  xdp_program__close(prog); /* frees memory, program stays attached */
  return err ? EXIT_FAILURE : EXIT_SUCCESS;
}
```

Working example

Loading multiple programs at once with xdp-loader works:

```
# xdp-loader load testns xdp_*.o
# xdp-loader status
sudo ./xdp-loader status
CURRENT XDP PROGRAM STATUS:

Interface        Prio  Program name      Mode     ID   Tag               Chain actions
--------------------------------------------------------------------------------------
lo               <no XDP program>
eth0             <no XDP program>
testns                 xdp_dispatcher   native   176  d51e469e988d81da
 =>              10    xdp_pass                  181  3b185187f1855c4c  XDP_PASS
 =>              50    xdp_drop                  186  57cd311f2e27366b  XDP_PASS
```

However, still can't load them one at a time.

Outstanding issues

Missing kernel features (soon to be resolved)

- Attaching freplace programs in multiple places
  - Attach existing progs to new dispatcher, then atomically replace on interface
  - WiP (by me)
- Not quite equivalence between replacing/replaced programs
  - Verifier doesn't treat freplace programs exactly like parents
  - WiP (by Udip Pant)

More fundamental issues with using freplace

Using freplace presents a few issues:

- Programs must be loaded as freplace (can't change after load)
  - Option to "promote" one XDP program to freplace another?
- XDP programs can't use freplace themselves
  - We are "squatting" on a potentially useful feature
- Only supported on x86_64
  - Can't use freplace at all on non-x86_64!

Are these acceptable, and/or can they be resolved?

How to ensure userspace coordination?

Doing multi-prog this way means userspace applications must agree on:

- Structure of dispatcher program
- How to obtain references for component progs/bpf_links (pinning path)
- Format of BPF program metadata (prio + chain call actions)
- Synchronisation primitives (locking / atomic replace semantics)

This is a protocol for cooperative multiprog operation. Libxdp is an implementation of this protocol.

Can we achieve consensus on this?

The need for pinning (and cleaning up)

Regular (non-multiprog) XDP programs stay attached after load.

To replicate this, libxdp currently pins all component programs, which has a few issues:

- Tied to a specific `bpffs` instance (problem with namespaces)
- No automatic cleanup when interface disappears

How do we resolve this?

One idea: Andrii suggested “sticky” bpf_links that share lifetime with the object they attach to.

Other issues? Questions?

- xdp-loader and libxdp: https://github.com/xdp-project/xdp-tools
- See also https://xdp-project.net