Defragmenting Drive C

Defragmenting the loader landscape

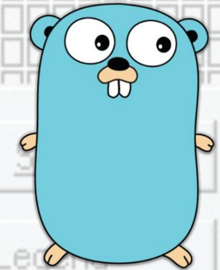
Lorenz Bauer & Timo Beckers

Cloudflare Isovalent

Linux Plumbers 2021

Defragmenting file system...

20% Complete



Agenda

Part 1: Introducing cilium/ebpf + Community Use Cases

Part 2: Proposal: UAPI bindings

Part 3: Proposal: Integrating ELF loaders

Introducing cilium/ebpf

What is it, and what can it do?

What is cilium/ebpf?

- eBPF library in pure Go: github.com/cilium/ebpf
- Maintained by Cloudflare and Cilium developers
- Easy to distribute
  - No dependency on C compiler
  - Great cross-compilation support
- Focus on “fat userspace” eBPF for long-running programs
- MIT licensed

WHYYYY?

Go has a fat runtime:

- C FFI (aka CGo) is slower than syscalls and harder to distribute
- Makes binaries bigger
- Go runtime profiler cannot see into C-land

Community Use Cases

Who uses it and what for?

runc + containerd

A container runtime used by Docker and Kubernetes.

Use package `asm` to assemble BPF cgroupv2 device filter at runtime, package `link` to attach them to cgroups.

github.com/containerd/cgroups

github.com/opencontainers/runc

DataDog: datadog-agent

Enables workload observability.

Use package `asm` to edit BPF at runtime, package `perf` to read perf_event_ring.

https://github.com/DataDog/datadog-agent

https://github.com/DataDog/ebpf-manager

Intel: CRI Resource Manager

Go-based Prometheus exporter to monitor a container's AVX512 activity using FPU Tracepoints.

The metrics are used to schedule more latency-sensitive workloads away from heavy AVX512 users, who tend to be noisy neighbours.

github.com/intel/cri-resource-manager

intel

Palantir: trace getaddrinfo

Use `bpf2go` to build standalone tools that can execute anywhere, `link` to trace libc's `getaddrinfo` in all containers on the platform.

See the talk below for the how & why:

User-space Probing

- Potential rejected solution - Monitoring packets directly
  - - High complexity solutions
  - - Unknown performance implications
  - + Captures 100% of DNS traffic
  - + Operates a level below containers
- Attaching Uprobes to the `libc` library
  - 90+% of our user traffic is Java/Python and is using `libc` for DNS resolution
    - The clients are managed by many teams. Uprobes allow us to monitor the clients with out involving the teams
  - We can attach uprobe to function `getaddrinfo` to get an accurate representation of our DNS systems
  - Good support for uprobe management using `cilium/ebpf`

Palantir

4:59

Using user-space tracing to solve DNS problems
by Andrius Grabauskas
https://youtu.be/0RDp1IPxbg0

Palantir

Hetzner Cloud: limit conntrack entries

Count and limit conntrack entries per VM on hypervisors using kprobes.

`bpf2go` allows for deploying a Go daemon with pre-compiled eBPF programs embedded. The built-in `link.Kprobe()` API manages kprobes easily out of the box.

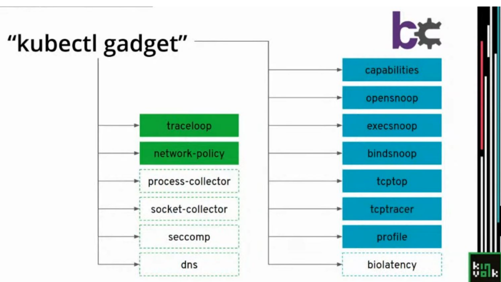
HETZNER

Microsoft: Inspektor Gadget

Collection of tools similar to BCC to debug and inspect Kubernetes applications.

Using bpf2go, CO-RE, BPF iterators.

github.com/kinvolk/inspektor-gadget



eCHO episode 19 - Inspektor Gadget
with Marga Manterola
https://youtu.be/RZ2qNm_vlUc

Cloudflare

Control plane for XDP-based DDoS mitigation / L4 load balancer / sk_lookup, fair share rate limiter in unprivileged socket filter.

Use most of the functionality!

github.com/cloudflare/rakelimit

Cilium

Container Network Interface implementation with fine grained network ACL and observability.

Manage map persistence, receive datapath notifications via perf ring, generate programs on the fly using the assembler. Planning to replace iproute2 and bpftool with `cilium/ebpf`.

github.com/cilium/cilium

UAPI bindings

How can we simplify uapi/bpf.h for code gen?

Generating Go types for BPF syscalls

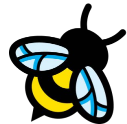
- Maintaining bpf(2) syscall bindings by hand = 😞
- Instead: read vmlinux BTF, write out Go types and functions

```
$ go doc . | grep MapLookupElem
func MapLookupElem(attr *MapLookupElemAttr) error
type MapLookupElemAttr struct{ ... }
```

Challenges

- No `union` type
- Runtime uses GC
  - Pointers need special treatment
  - No explicit control over heap / stack allocation

Unnamed types / fields



```
/* flags for BPF_MAP_UPDATE_ELEM command */
enum {
      BPF_ANY             = 0,
      BPF_NOEXIST         = 1,
      BPF_EXIST           = 2,
      BPF_F_LOCK          = 4,
};

union bpf_attr {
       struct {
               __u32   map_type;
               __u32   key_size;
               __u32   value_size;
               __u32   max_entries;
               ...
       };
...
```

Unnamed types / fields

```
/* flags for BPF_MAP_UPDATE_ELEM command */
enum {
        BPF_ANY         = 0,
        BPF_NOEXIST     = 1,
        BPF_EXIST       = 2,
        BPF_F_LOCK      = 4,
};

union bpf_attr {
        struct {
                __u32   map_type;
                __u32   key_size;
                __u32   value_size;
                __u32   max_entries;
                ...
        };
...
```

=>

```
enum bpf_map_update_flags {
        BPF_ANY         = 0,
        BPF_NOEXIST     = 1,
        BPF_EXIST       = 2,
        BPF_F_LOCK      = 4,
};

union bpf_attr {
        struct {
                __u32   map_type;
                __u32   key_size;
                __u32   value_size;
                __u32   max_entries;
                ...
        } map_create;
...
```

Preprocessor macros

```c
#define BPF_F_QUERY_EFFECTIVE   (1U << 0)

/* Flags for BPF_PROG_TEST_RUN */

#define BPF_F_TEST_RUN_ON_CPU   (1U << 0)
```

=>

```c
enum bpf_prog_query_flags {
        BPF_F_QUERY_EFFECTIVE = (1U << 0),
};

enum bpf_prog_run_flags {
        BPF_F_TEST_RUN_ON_CPU = (1U << 0),
};
```

Invisible pointers

```c
struct { /* used by BPF_OBJ_* commands */
        __aligned_u64   pathname;
        __u32           bpf_fd;
        __u32           file_flags;
};
```

=>

```c
struct {
        __bpf_md_ptr(char *, pathname);
        __u32           bpf_fd;
        __u32           file_flags;
};

// equivalent to

struct {
        union {
            char *pathname;
            __u64 :64;
        } __attribute__((aligned(8)));
        __u32   bpf_fd;
        __u32   file_flags;
};
```

Field overloading

```c
struct { /* used by BPF_*_GET_*_ID */
   union {
        __u32        start_id;
        __u32        prog_id;
        ...
   };
   __u32        next_id;
   __u32        open_flags;
};
```

=>

```c
struct { // BPF_PROG_GET_FD_BY_ID
   __u32 prog_id;
} prog_get_fd_by_id;

struct { // BPF_MAP_GET_FD_BY_ID
   __u32 map_id;
   __u32 next_id; // ignored!
   __u32 open_flags;
} map_get_fd_by_id;

struct { // BPF_(PROG|MAP|...)_GET_NEXT_ID
   __u32 start_id;
   __u32 next_id;
} obj_get_next_id;
```

Recap: how to simplify bpf.h for robots (and humans)

- Use enums instead of macros
- Give types / fields a name
  - Anonymous unions are OK
- Use __bpf_md_ptr throughout
- Have one bpf_attr field per bpf_cmd
- All of these can be backwards compatible

Q&A

Integrating ELF loaders

How can we align “second-parties” with libbpf?

The Loader Landscape

Existing native libraries today:

- libbpf
- cilium/ebpf (Go)
- aya-rs (Rust)
- eBPF for Windows (cannot use GPL?)

Features still land in libbpf first, but no straightforward way for other projects to verify compliance.

How could we align these projects better?

At LPC 2019, it was mentioned often that libbpf is the ‘spec’. Now how do we codify this?

Instead of a vague ‘spec’, let’s write **tests!**

Background

Compiler communicates with a loader through **ELF binaries**.

Existing loaders like libbpf, bpftool and iproute2 (now also libbpf-based) expect ELF binaries to follow certain **conventions**.

Basic example, ELF **section names**:

- legacy maps → `maps`
- BTF maps → `.maps`
- program attach type → section name prefixes

As time went on, the eBPF feature set (and the ELF's contents) grew more complex, and the loader had to follow suit.

Background

- ➔ How do we verify if ELFs are correctly parsed and none of its contents were ignored? (e.g. when a new ELF section is introduced)
- ➔ How do we ensure the loader stays compatible with older and newer kernels and LLVM versions?
- ➔ How do we verify compliance with libbpf?
- ➔ How can the Linux project maintain control over the toolchain and keep everything compatible?
    - ◆ eBPF programs that only work with loader X or Y is not an ideal situation
    - ◆ More loaders == more ecosystem fragmentation. Or does it?

Selftests!

The Linux kernel ships with a set of selftests to verify if compiler, loader and kernel behave as expected.

Building them results in userspace programs and BPF ELF objects:

xdping	← userspace executable
xdping_kern.o	← BPF ELF loaded by userspace executable
xdping.c	← userspace source
xdping_kern.c	← BPF source

Problem 1: selftests have complex userspace programs

Besides BPF programs themselves, selftests also require **specific userspace code** to load and exercise said BPF ELFs.

- Each userspace program is specific to a particular BPF program.
- The tests are often complex and set up arbitrary netdevs, addresses, etc.
- There are simply **too many** to replicate and maintain in other languages, especially by smaller teams.
- **No way to verify** if loader's resulting bytecode is correct even when accepted by the verifier. (some CO-RE offset might be off, etc.)
- Unfortunately, selftests also exist to exercise behaviour of the kernel itself, which is not wanted/needed for a loader project.

Problem 2: BPF ELFs can be intentionally invalid

Some selftest ELFs are intentionally invalid for negative testing purposes, but are not named consistently to reflect this.

Today, cilium/ebpf ignores errors loading known-invalid ones.
This is not ideal, as we need to filter based on object name (fragile):

```
// Some files like btf__core_reloc_arrays___err_too_small.o
// trigger an error on purpose. Use the name to infer whether
// the test should succeed.
var valid bool
switch name {
case "btf__core_reloc_type_id___missing_targets.o",
	"btf__core_reloc_flavors__err_wrong_name.o":
	valid = false
```

Another approach is needed

In `cilium/ebpf`, the selftest BPF ELFs are parsed and loaded, but **never executed**.

There might be subtle bugs that we currently cannot uncover.

→ ‘Unit’ tests are needed that target the (userspace) loader infra in isolation.

Suggestion: Test BPF loaders using BPF!

The common factor in all eBPF loaders is… BPF! Can we write BPF programs that test themselves at runtime?

Single ELF containing BPF prog(s) that:

- Contain useful lineinfo BTF for clear verifier output (using comments / macros)
- Contains assertions:
    - Reading map values
    - Has the right map access been relocated here?
    - Check if CO-RE relocations have been executed using known-good values (verifier OK, but is the offset correct?)
- Generate error feedback by verifier or BPF_PROG_RUN

Suggestion: Minimal Userspace

Minimal userspace program that is:

- devoid of any ‘business’ logic and only interacts with simple libbpf APIs
- easy to port to other platforms / languages

It should be limited to high-level tasks:

- Find ELFs on disk (e.g. a glob)
- Invoke the loader (e.g. libbpf) that loads progs, maps, BTF, … and performs the necessary transformations
- Call PROG_RUN on all programs in the ELF
- Succeed if test run returns 0

Examples of eBPF runtime tests

cilium/ebpf contains a few eBPF runtime tests today:

Static data relocation

```
volatile const unsigned int uneg                 = -1;
volatile const int neg                           = -2;
static volatile const unsigned int static_uneg = -3;
static volatile const int static_neg            = -4;

__section("socket/3") int data_sections() {
	if (uneg != (unsigned int)-1)
		return __LINE__;

	if (neg != -2)
		return __LINE__;
```

CO-RE relocations

```
#define local_id_not_zero(expr) \
	({ \
		if (bpf_core_type_id_local(expr) == 0) { \
			return __LINE__; \
		} \
	})

__section("socket_filter/type_ids") int type_ids() {
	local_id_not_zero(int);
	local_id_not_zero(struct { int frob; });
	// etc...
```

Next Steps

An independent loader test suite is needed, separate from other end-to-end kernel / eBPF selftests. Suggestion:

→ Submit a new ‘test-loader’ test suite to Linux tree.

→ Agree (iteratively) on a common set of behaviours all loaders must implement.

Example: PROG_ARRAY auto-population

For declaring tail call maps more intuitively in BTF map definitions, cilium/ebpf allows using the .values mechanism to specify PROG_ARRAY contents directly.

Inspired by libbpf's map-in-map declarations, but not yet supported in libbpf.

```
struct {
    __uint(type, BPF_MAP_TYPE_PROG_ARRAY);
    __uint(key_size, sizeof(uint32_t));
    __uint(max_entries, 10);
    __array(values, int ());
} prog_array_init __section(".maps") = {
    .values = {
        [1] = &tail_1,
        [5] = &tail_foo,
    },
};
```

Accepting this behaviour into the loader test suite signifies a **contract / agreement** for all loaders to implement it.

Other Thoughts

- Easier to bootstrap new loaders / libraries:
  - Minimum requirement is ELF parsing, implementing PROG_LOAD and PROG_RUN syscalls.
  - Iterate from there to support more and more of the test suite incrementally.
- Potentially **reduce** fragmentation in the ecosystem due to having a clear common goal, with fewer (or, ideally, none) deviations from libbpf behaviour.

Q&A