ELF relocation for static data in BPF

Joe Stringer, Daniel Borkmann
Cilium.io

Linux Plumbers 2018, BPF MC, Vancouver, Nov 15, 2018

BPF workflow in Cilium

- Event → Golang daemon → C header regeneration → LLVM → BPF loader → BPF verifier → BPF JIT → Atomic program replacement
- Auto-generated C header contains:
  - Configuration data as constants
  - Defines for changing logic in program code on demand (ifdefs), e.g.:
    - Emission of debug data / drop monitor / flow tracing via perf RB
    - Aggregation level for perf RB
    - Accounting / switch between global or local BPF conntrack

BPF workflow in Cilium

- Upsides of 'full' program regeneration:
  - Emitted BPF instructions *always* optimized to C code
  - Full flexibility and build workflow rather straight forward
- Downsides:
  - Daemon needs to shell out for compilation, no reasonable library integration due to unstable LLVM API
  - Runtime dependency on clang and LLVM in Cilium container
  - Runtime cost for every endpoint (e.g. container), especially when regeneration triggered for *all* endpoints $\leftarrow$ biggest pain point

Where do we want to be (ideally)?

- Invocation of clang and LLVM only once at *build* time, *not* runtime
- Minimal runtime overhead on program data and logic updates
- Potential steps (short- and mid-term):
  1. Optimize program config data updates
  2. Optimize program logic updates
  3. Move entire BPF program as template into golang binary at build
     - Dynamic program 'assembly' out of golang
     - No LLVM, ELF parsing at runtime anymore

Step 1: config data updates

- Move all config into BPF array map
- Pros:
  - Potentially possible with older kernels
  - No LLVM, verification, JIT, etc needed
- But outweighed by cons:
  - Runtime overhead for map lookup
  - Needs map in map for atomic config updates
  - Requires significant program rewrites
  - Nightmare with verifier complexity explosion

Step 1: config data updates

- Proposal: move config as static data into ELF file
  - Assumes config updates $>$ logic updates
  - ELF becomes template (sort of)
  - Regeneration only rewrites bytes in ELF data section, then reloads program into kernel for atomic replace
  - Piggybacking on LLVM generating relocation entries (similar to maps)

Step 1: config data updates

```
#include <linux/bpf.h>
#include <stdint.h>

#ifndef __section
# define __section(NAME)                  \
   __attribute__((section(NAME), used))
#endif
#ifndef __fetch
# define __fetch(x) (__u32)(&(x))
#endif

__u32 foo = 42;

int __main(struct __sk_buff *skb)
{
        skb->mark = __fetch(foo);
        return 0;
}

char __license[] __section("license") = "";
```

Step 1: config data updates

```
$ clang -O2 -Wall -target bpf -c test.c -o test.o

# readelf -S test.o
[...]
  [ 4] .data             PROGBITS [...]        <-- the one to modify
       0000000000000004  0000000000000000  WA       0     0     4
[...]

$ readelf -r test.o
Relocation section '.rel.text' at offset 0xd0 contains 1 entries:
  Offset          Info           Type           Sym. Value    Sym. Name
000000000000  000300000001 unrecognized: 1       0000000000000000 foo

$ readelf -s test.o
Symbol table '.symtab' contains 4 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT    5 __license
     2: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT    2 __main
     3: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT    4 foo
```

Step 1: config data updates

```
static int bpf_apply_relo_glob(struct bpf_elf_ctx *ctx,
                               struct bpf_elf_prog *prog,
                               GElf_Rel *relo, GElf_Sym *sym)
{
        __u32 insn_off = relo->r_offset / sizeof(struct bpf_insn);
        int *data;

        if (insn_off >= prog->insns_num)
                return -EINVAL;

        data = ctx->glo_data->d_buf + sym->st_value;
        prog->insns[insn_off].imm = *data;
        return 0;
}
```

Step 1: config data updates

```
# tc qdisc  add dev lo clsact
# tc filter add dev lo ingress bpf da obj test.o sec .text verb

Prog section '.text' loaded (5)!
 - Type:         3
 - Instructions: 5 (0 over limit)
 - License:

Verifier analysis:

0: (18) r2 = 0x2a                      <-- foo (42)
2: (63) *(u32 *)(r1 +8) = r2
3: (b7) r0 = 0
4: (95) exit
processed 4 insns (limit 131072), stack depth 0

#
```

Step 1: config data updates

- Prototype: iproute2.git
- Limitations:
  - Only works for simple data that fits into imm field
  - Ugly macro hackery could help a way around this for old kernels
  - Overall not too feasible for complex structs / arrays though

Step 1: config data updates

```
#include <linux/bpf.h>
#include <stdint.h>

#ifndef __section
# define __section(NAME)                  \
   __attribute__((section(NAME), used))
#endif

__u8 foo[4] = { 0, 1, 2, 3 };

int __main(struct __sk_buff *skb)
{
        __builtin_memcpy(&skb->mark, foo, sizeof(foo));
        return 0;
}

char __license[] __section("license") = "";
```

Step 1: config data updates

```
# tc qdisc  add dev lo clsact
# tc filter add dev lo ingress bpf da obj test.o sec .text verb

Prog section '.text' rejected: Permission denied (13)!
 - Type:         3
 - Instructions: 15 (0 over limit)
 - License:

Verifier analysis:

0: (18) r2 = 0x3020100                 <-- foo
2: (71) r3 = *(u8 *)(r2 +1)
R2 invalid mem access 'inv'

Error fetching program/map!
Unable to load program

#
```

Step 1: config data updates

- Steps from here:
  1. Extend program load → BPF loader copies global data into kernel
  2. Prog-local buffer sits in `prog->aux->global.{data,size}`
  3. Address known at load time → verifier rewrites special LD_IMM_DW
  4. `src_reg = BPF_PSEUDO_PROG_BUFF`, `imm = sym->st_value`
  5. Generalizing `PTR_TO_MAP_VALUE` for generic reuse of size limit
  6. Buffer could be RO or RW (e.g. in combination with 'BPF spinlocks')
- Prototype: bpf.git, iproute2.git

Step 1: config data updates

```
# tc filter add dev lo ingress bpf da obj test.o sec .text verb

[...]
 0: (18) r2 = 0xffff9dda8be5fdc8       <-- program's data section
 2: (71) r3 = *(u8 *)(r2 +1)
  R1=ctx(id=0,off=0,imm=0) R2_w=map_value(id=0,off=0,ks=0,vs=4,imm=0) [...]
 3: (67) r3 <<= 8
 4: (71) r4 = *(u8 *)(r2 +0)
  R1=ctx(id=0,off=0,imm=0) R2_w=map_value(id=0,off=0,ks=0,vs=4,imm=0) [...]
 5: (4f) r3 |= r4
 6: (71) r4 = *(u8 *)(r2 +2)
  R1=ctx(id=0,off=0,imm=0) R2_w=map_value(id=0,off=0,ks=0,vs=4,imm=0) [...]
 7: (71) r2 = *(u8 *)(r2 +3)
  R1=ctx(id=0,off=0,imm=0) R2_w=map_value(id=0,off=0,ks=0,vs=4,imm=0) [...]
 8: (67) r2 <<= 8
 9: (4f) r2 |= r4
10: (67) r2 <<= 16
11: (4f) r2 |= r3
12: (63) *(u32 *)(r1 +8) = r2
13: (b7) r0 = 0
14: (95) exit
processed 14 insns (limit 131072), stack depth 0
```

Step 2: prog logic updates (rough sketch)

- Basic idea: static keys for BPF programs
- LLVM built-in for having static key-like branch from C code
  1. Compiler moves this cold code path out of line
  2. Enabled: forward jump to region, fixed backward jump to origin
  3. Disabled: patched out forward jump offset to 0 for fall-through
  4. BTF could describe offsets for correlation, dumped via bpftool
- BPF syscall command to trigger patching: prog fd + BPF insn offset
  - BPF insn: single `off:16` replacement from `struct bpf_insn`
  - JIT could re-JIT and atomically replace `prog->bpf_func` address
    - JITing of BPF subprogs needs to be moved out of verifier
- Prototype: wip