gobpf

utilizing eBPF from Go

FOSDEM • 2017-02-05 • michael@kinvolk.io

kinvolk.io

`bpf_trace_printk("hello");`

Michael, Software Engineer at Kinvolk

We mostly work on Linux core system software (kernel, container, etc.)

… and like it!

https://kinvolk.io/about/

We are hiring, too!

What is gobpf?

- library to create, load and use eBPF programs from Go
- bring together eBPF capabilities and Go software

Agenda

Introduction to eBPF

- Berkeley Packet Filter / cBPF
- Programs
- Maps

gobpf

- BPF Compiler Collection (bcc)
- elf
- CI

What is eBPF?

- “bytecode virtual machine” in the Linux kernel
- used for tracing kernel functions, networking, performance analysis, ...

Berkeley Packet Filter / LSF / cBPF

- original use case packet filtering
- based on paper “The BSD Packet Filter: A New Architecture for User-level Packet Capture” from December 19, 1992

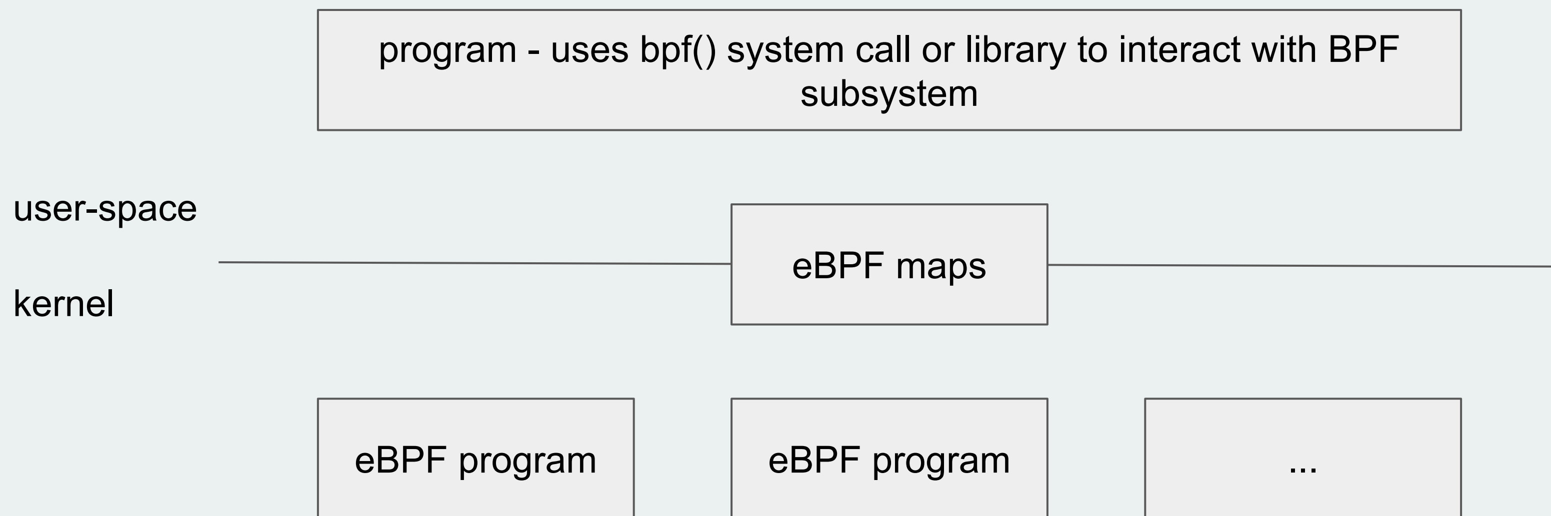
```
sudo tcpdump -p -ni eth0 -d "ip and udp"
(000) ldh      [12]
(001) jeq      #0x800          jt 2    jf 5
(002) ldb      [23]
(003) jeq      #0x11           jt 4    jf 5
(004) ret      #262144
(005) ret      #0
```

https://blog.cloudflare.com/bpf-the-forgotten-bytecode/

Why eBPF?

- enables you to
  - attach to different kernel events
  - do networking (SDN), e.g. packet parsing & modification
  - use bpf as a security mechanism, e.g. filtering
  - ...
- safety guarantee through the bpf verifier
- fast, running in the kernel
  - JIT compiled (for x86-64, arm64, and s390 if enabled, `/proc/sys/net/core/bpf_jit_enable`)

How does it work?



kinvolk.io

eBPF programs

Programs can be attached to ...

- sockets
    - execute eBPF program for each packet
    - https://www.kernel.org/doc/Documentation/networking/filter.txt
    - non-privileged operation contrary to other bpf(2) ops
- kernel tracepoints
    - `linux:include/trace/events/`
    - https://www.kernel.org/doc/Documentation/trace/tracepoints.txt

https://github.com/torvalds/linux/blob/v4.9/include/uapi/linux/bpf.h#L90-L99

eBPF programs

Programs can be attached to ...

- kprobes
  - “a set of handlers placed on a certain instruction address” - https://lwn.net/Articles/132196/
  - pre-handler and a post-handler: kretprobe
  - https://www.kernel.org/doc/Documentation/kprobes.txt
- uprobes
  - user-space counterpart of kprobes

eBPF maps

Maps are a generic data structure to share “data between eBPF kernel programs, and also between kernel and user-space applications.”

- a key/value for a given map can have an arbitrary structure, as specified by the user at map-creation time
- one special map `BPF_MAP_TYPE_PROG_ARRAY` holding file descriptors referring to other eBPF programs
- man (2) bpf notoriously not up-to-date

eBPF maps

- Available may types:
  - `HASH`
  - `ARRAY`
  - `PROG_ARRAY`
  - `PERF_EVENT_ARRAY`
  - `PERCPU_HASH`, `PERCPU_ARRAY`
  - `STACK_TRACE`
  - `CGROUP_ARRAY`

- Linux v4.10 adds `LRU_HASH`

https://github.com/torvalds/linux/blob/v4.9/include/uapi/linux/bpf.h#L78-L88

BPF_MAP_TYPE_PERF_EVENT_ARRAY

- an array of file descriptors (one per cpu, created with perf_event_open(2)) to in-kernel ring buffers containing perf events
- allows sending a lot of events very fast
- user-space program can read asynchronously from mmap'ed memory

Warning: somewhat scary code on next slide

“... to start by coding a struct bpf_insn is starting with difficulty setting Ultra-Violence”

- Brendan Gregg

Example: `fchownat(2)` count kprobe

```
/* Put 0 (the map key) on the stack */
BPF_ST_MEM(BPF_W, BPF_REG_10, -4, 0),
/* Put frame pointer into R2 */
BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),
/* Decrement pointer by four */
BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -4),
/* Put map_fd into R1 */
BPF_LD_MAP_FD(BPF_REG_1, map_fd),
/* Load current count from map into R0 */
BPF_RAW_INSN(BPF_JMP | BPF_CALL, 0, 0, 0, BPF_FUNC_map_lookup_elem),
/* If returned value NULL, skip two instructions and exit */
BPF_JMP_IMM(BPF_JEQ, BPF_REG_0, 0, 2),
/* Put 1 into R1 */
BPF_MOV64_IMM(BPF_REG_1, 1),
/* Increment value by 1 */
BPF_RAW_INSN(BPF_STX | BPF_XADD | BPF_W, BPF_REG_0, BPF_REG_1, 0, 0),
BPF_EXIT_INSN(),
```

https://kinvolk.io/blog/2016/11/introducing-gobpf---using-ebpf-from-go/

The IO Visor Project

“[...] open source project and a community of developers to accelerate the innovation, development, and sharing of virtualized in-kernel IO services for tracing, analytics, monitoring, security and networking functions.”

https://www.iovisor.org/about

https://github.com/iovisor

gobpf

library to create, load and use eBPF programs from Go

- use Cgo + the BPF Compiler Collection (bcc) or
- load and use pre-build elf object files
- https://github.com/iovisor/gobpf

Why gobpf?

- There was no library which does what we need
  - … but the Hover Framework: https://github.com/iovisor/iomodules
- We like Go @ Kinvolk + use it a lot
- We work on Weave Scope, which is written in Go

Why gobpf?

weavescope

- Scope probes (read: agents) need to gather a lot of system data
  - https://github.com/weaveworks/scope/tree/master/probe/endpoint
- Doing that with eBPF is often faster and/or more reliable than e.g. `/proc` or conntrack (Linux connection tracking) parsing

gobpf/bcc

- `import bpf "github.com/iovisor/gobpf/bcc"`
- write eBPF program in C
- load with gobpf

kinvolk.io

gobpf/bcc

- “A modified C language for BPF backends” - https://github.com/iovisor/bcc/blob/master/README.md
- When using bcc, we can rely on bcc helper functions which make it easier to e.g. work with maps: `map.update(&key, &value);`

gobpf/bcc

libbcc.so is the core of bcc and not only a library but also a compiler to translate eBPF programs written in aforementioned modified C language into byte code for bpf(2)

- uses clang + llvm-bpf backend
- allows you to verify program before loading it
- spares you “a kludgy workflow, sometimes involving compiling directly in a linux kernel source tree”

gobpf/bcc

```
const source string = `
#include <uapi/linux/ptrace.h>
#include <bcc/proto.h>

typedef struct {
        u32 pid;
        uid_t uid;
        gid_t gid;
        int ret;
        char filename[256];
} chown_event_t;

BPF_PERF_OUTPUT(chown_events);
BPF_HASH(chowncall, u64, chown_event_t);
```

```
int kprobe__sys_fchownat(
        struct pt_regs *ctx,
        int dfd, const char *filename,
        uid_t uid, gid_t gid, int flag)
{
        u64 pid = bpf_get_current_pid_tgid();
        chown_event_t event = {
                .pid = pid >> 32,
                .uid = uid,
                .gid = gid,
        };
        bpf_probe_read(&event.filename,
                       sizeof(event.filename),
                       (void *)filename);
        chowncall.update(&pid, &event);
        return 0;
}
```


kinvolk.io

gobpf/bcc

```
m := bpf.NewModule(source, []string{})
…
chownKprobe, err := m.LoadKprobe("kprobe__sys_fchownat")
err = m.AttachKprobe("sys_fchownat", chownKprobe)
…
table := bpf.NewTable(0, m)
perfMap, err := bpf.InitPerfMap(table, channel)
…
go func() {
	var event chownEvent
	for {
		data := <-channel
		err := binary.Read(bytes.NewBuffer(data), binary.LittleEndian, &event)
		…
		filename := (*C.char)(unsafe.Pointer(&event.Filename))
		fmt.Printf("uid %d gid %d pid %d called fchownat(2) on %s (return value: %d)\n",
			event.Uid, event.Gid, event.Pid, C.GoString(filename), event.ReturnValue)
	}
}()
```

gobpf/bcc

```
// Gateway function as required with CGO Go >= 1.6
// ...
//export callback_to_go
func callback_to_go(cbCookie unsafe.Pointer, raw unsafe.Pointer, rawSize C.int) {
	callbackData := lookupCallback(uint64(uintptr(cbCookie)))
	receiverChan := callbackData.receiverChan
	go func() {
		receiverChan <- C.GoBytes(raw, rawSize)
	}()
}

…

C.bpf_open_perf_buffer(
		(C.perf_reader_raw_cb)(unsafe.Pointer(C.callback_to_go)),
		unsafe.Pointer(uintptr(callbackDataIndex)), -1, C.int(cpu))
```

Demo: fchownat(2) snoop

gobpf/elf

- `import bpf "github.com/iovisor/gobpf/elf"`
- load + use pre-built elf object

kinvolk.io

gobpf/elf

- relies on elf sections:

```
#define SEC(NAME) __attribute__((section(NAME), used))
...
SEC("kretprobe/tcp_v4_connect")
int kretprobe__tcp_v4_connect(struct pt_regs *ctx)
{
...
```

- pkg debug/elf used to find kprobes + maps, resolve relocatable sections and load data
- https://github.com/weaveworks/tcptracer-bpf

gobpf/elf

```
clang -O2 \
   -emit-llvm \                 # produce LLVM bitcode object file
   -c ebpf.c -o - | \           # write result to stdout
   llc \                        # LLVM static compiler
   -march=bpf \                 # generate eBPF code
   -filetype=obj -o ebpf.o      # write object file
```

gobpf/elf

- allows you to build a single object for multiple kernel versions by setting the kernel version to a constant: `0xFFFFFFFE`
  - gobpf/elf then gets your kernel version from uname(2)
- use *feature* at your own risk; requires careful consideration as
  - in k{ret,}probes, used kernel functions and accessed structs could change
  - those are internal functions / structures not covered by the ABI guarantee

Continuous integration for eBPF programs

- problem: requires modern Linux kernel + root access
- currently, we use semaphoreci.com + custom rkt stage1-kvm images
    - experimental build script for custom stage1 images: https://github.com/kinvolk/stage1-builder

Questions?

https://fosdem.org/2017/schedule/event/go_bpf ← Submit feedback, if you want

https://www.speakerdeck.com/schu

schu

schux00

michael@kinvolk.io