A Beginner's Guide to eBPF Programming with Go

Liz Rice
VP Open Source Engineering, Aqua Security

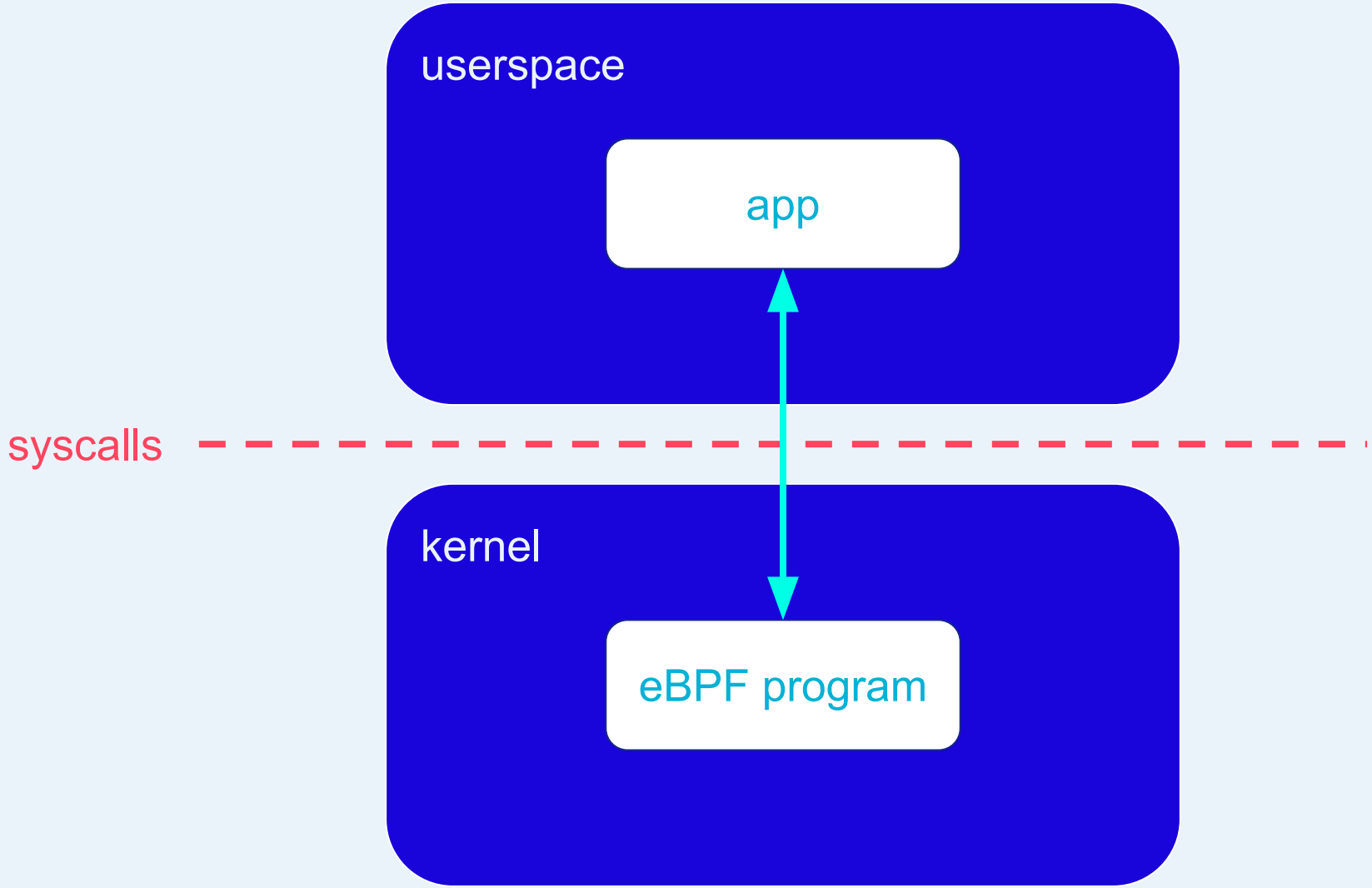
@lizrice

Run custom code in the kernel

man bpf

The bpf() system call performs a range of operations related to extended Berkeley Packet Filters. Extended BPF (or eBPF) is similar to the original ("classic") BPF (cBPF) used to filter network packets.

For both cBPF and eBPF programs, the kernel statically analyzes the programs before loading them, in order to ensure that they cannot harm the running system.



aqua

bpftrace

bpftrace is a high-level tracing language for Linux enhanced Berkeley Packet Filter (eBPF) available in recent Linux kernels (4.x). bpftrace uses LLVM as a backend to compile scripts to BPF-bytecode and makes use of BCC for interacting with the Linux BPF system, as well as existing Linux tracing capabilities: kernel dynamic tracing (kprobes), user-level dynamic tracing (uprobes), and tracepoints. The bpftrace language is inspired by awk and C, and predecessor tracers such as DTrace and SystemTap. bpftrace was created by Alastair Robertson.

To learn more about bpftrace, see the Reference Guide and One-Liner Tutorial.

One-Liners

The following one-liners demonstrate different capabilities:

```
# Files opened by process
bpftrace -e 'tracepoint:syscalls:sys_enter_open { printf("%s %s\n", comm, str(args->filename)); }'

# Syscall count by program
bpftrace -e 'tracepoint:raw_syscalls:sys_enter { @[comm] = count(); }'

# Read bytes by process:
bpftrace -e 'tracepoint:syscalls:sys_exit_read /args->ret/ { @[comm] = sum(args->ret); }'

# Read size distribution by process:
bpftrace -e 'tracepoint:syscalls:sys_exit_read { @[comm] = hist(args->ret); }'

# Show per-second syscall rates:
bpftrace -e 'tracepoint:raw_syscalls:sys_enter { @ = count(); } interval:s:1 { print(@); clear(@);
```

Explore bpf syscalls in bpftrace

```
$ sudo strace -e bpf bpftrace -e
'tracepoint:raw_syscalls:sys_enter { @[comm] = count(); }'
```

```
$ sudo strace -e bpf bpftrace -e
'tracepoint:raw_syscalls:sys_enter { @[comm] = count(); }'

bpf(BPF_MAP_CREATE, {map_type=BPF_MAP_TYPE_ARRAY, key_size=4, value_size=4, max_entrie
bpf(BPF_MAP_CREATE, {map_type=BPF_MAP_TYPE_PERCPU_HASH, key_size=16, value_size=8, max
bpf(BPF_MAP_CREATE, {map_type=BPF_MAP_TYPE_PERCPU_HASH, key_size=16, value_size=8, max
bpf(BPF_MAP_CREATE, {map_type=BPF_MAP_TYPE_PERF_EVENT_ARRAY, key_size=4, value_size=4,
Attaching 1 probe...
bpf(BPF_MAP_UPDATE_ELEM, {map_fd=4, key=0x7ffcdab0dfcc, value=0x7ffcdab0dfd0, flags=BP
bpf(BPF_MAP_UPDATE_ELEM, {map_fd=4, key=0x7ffcdab0dfcc, value=0x7ffcdab0dfd0, flags=BP
bpf(BPF_PROG_LOAD, {prog_type=BPF_PROG_TYPE_TRACEPOINT, insn_cnt=27, insns=0x7f86f16b3
```

```
$ sudo strace -e bpf bpftrace -e
'tracepoint:raw_syscalls:sys_enter { @[comm] = count(); }'

bpf(BPF_MAP_CREATE, {map_type=BPF_MAP_TYPE_ARRAY, key_size=4, value_size=4, max_entrie
bpf(BPF_MAP_CREATE, {map_type=BPF_MAP_TYPE_PERCPU_HASH, key_size=16, value_size=8, max
bpf(BPF_MAP_CREATE, {map_type=BPF_MAP_TYPE_PERCPU_HASH, key_size=16, value_size=8, max
bpf(BPF_MAP_CREATE, {map_type=BPF_MAP_TYPE_PERF_EVENT_ARRAY, key_size=4, value_size=4,
Attaching 1 probe...
bpf(BPF_MAP_UPDATE_ELEM, {map_fd=4, key=0x7ffcdab0dfcc, value=0x7ffcdab0dfd0, flags=BP
bpf(BPF_MAP_UPDATE_ELEM, {map_fd=4, key=0x7ffcdab0dfcc, value=0x7ffcdab0dfd0, flags=BP
bpf(BPF_PROG_LOAD, {prog_type=BPF_PROG_TYPE_TRACEPOINT, insn_cnt=27, insns=0x7f86f16b3

^C

@[vmstats]: 2
@[systemd-journal]: 5
@[sudo]: 7
@[multipathd]: 9
@[containerd]: 10
@[bpftrace]: 16
...
```

eBPF programs & maps

```
bpf(BPF_PROG_LOAD, …)
bpf(BPF_MAP_CREATE, …)
```

man bpf

eBPF programs can be written in a restricted C that is compiled (using the clang compiler) into eBPF bytecode. Various features are omitted from this restricted C, such as loops, global variables, variadic functions, floating-point numbers, and passing structures as function arguments.

man bpf

eBPF programs can be written in a restricted C that is compiled (using the clang compiler) into eBPF bytecode. Various features are omitted from this restricted C, such as loops, global variables, variadic functions, floating-point numbers, and passing structures as function arguments.

[eBPF Helper functions] are used by eBPF programs to interact with the system, or with the context in which they work. For instance, they can be used to print debugging messages...

```
bpf_trace_printk()
bpf_get_current_comm()
bpf_perf_event_output()
...
```

man bpf

Maps are a generic data structure for storage of different types of data. They allow sharing of data between eBPF kernel programs, and also between kernel and user-space applications.

Attaching eBPF to events

eBPF programs are event-driven and are run when the kernel or an application passes a certain hook point. Pre-defined hooks include system calls, function entry/exit, kernel tracepoints, network events, and several others.

If a predefined hook does not exist for a particular need, it is possible to create a kernel probe (kprobe) or user probe (uprobe) to attach eBPF programs almost anywhere in kernel or user applications.

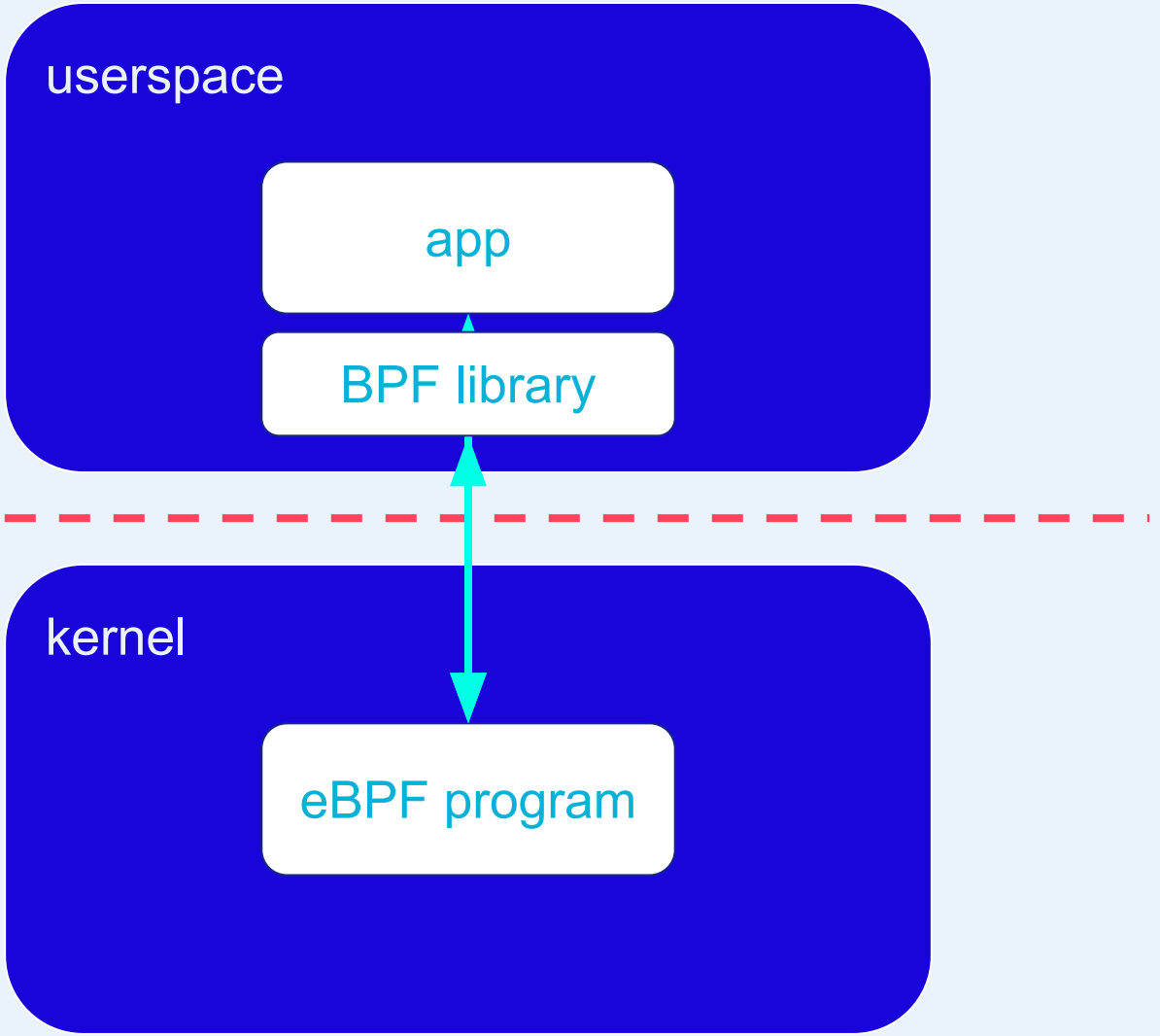
```
$ sudo strace -e bpf,perf_event_open,ioctl bpftrace -e
'tracepoint:raw_syscalls:sys_enter { @[comm] = count(); }'

bpf(BPF_MAP_CREATE, {map_type=BPF_MAP_TYPE_ARRAY, key_size=4, value_size=4, max_entrie
bpf(BPF_MAP_CREATE, {map_type=BPF_MAP_TYPE_PERCPU_HASH, key_size=16, value_size=8, max
...
bpf(BPF_PROG_LOAD, {prog_type=BPF_PROG_TYPE_TRACEPOINT, prog_name="sys_enter", ...
    attach_prog_fd=0}, 120) = 9
perf_event_open({type=PERF_TYPE_TRACEPOINT, size=0, ...) = 8
ioctl(8, PERF_EVENT_IOC_SET_BPF, 9)     = 0
```

Attach custom code to an event

```
bpf(BPF_PROG_LOAD, …) = x
perf_event_open(…) = y
ioctl(y, PERF_EVENT_IOC_SET_BPF, x)
```

How to write eBPF hello world?

written in our choice of language

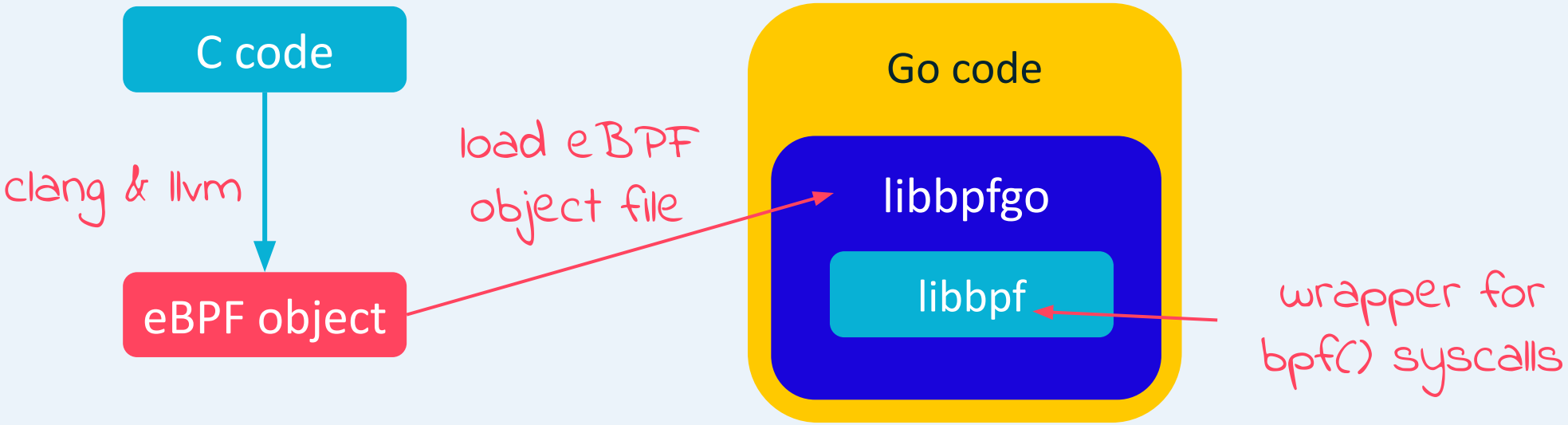


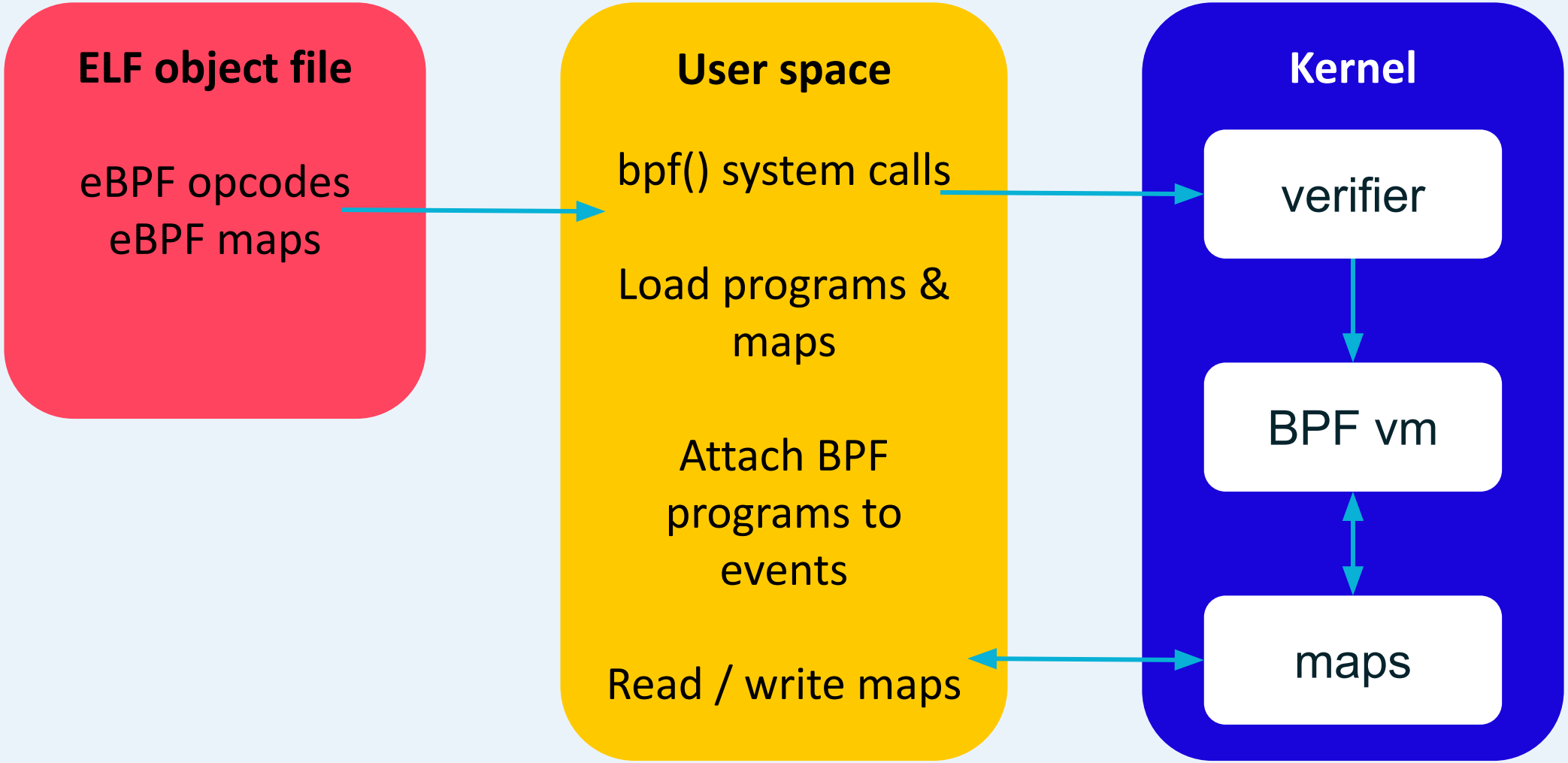
written in C
compiled by clang

aqua

libbpfgo

Golang wrapper around libbpf




ELF object file
eBPF opcodes
eBPF maps
User space
bpf() system calls
Load programs & maps
Attach BPF programs to events
Read / write maps
Kernel
verifier
BPF vm
maps

aqua

```
TARGET := hello
TARGET_BPF := $(TARGET).bpf.o
GO_SRC := $(shell find . -type f -name '*.go')
BPF_SRC := $(shell find . -type f -name '*.bpf.c')
...
.PHONY: all
all: $(TARGET) $(TARGET_BPF)

go_env := CC=clang CGO_CFLAGS="-I $(LIBBPF_HEADERS)" CGO_LDFLAGS="$(LIBBPF_OBJ)"
$(TARGET): $(GO_SRC)
    $(go_env) go build -o $(TARGET)

$(TARGET_BPF): $(BPF_SRC)
    clang -I /usr/include/x86_64-linux-gnu \
        -O2 -c -target bpf \
        -o $@ $<
```

eBPF hello world

```c
SEC("kprobe/sys_execve")
int hello(void *ctx)
{
    bpf_printk("I'm alive!");
    return 0;
}
```


github.com/lizrice/libbpfgo-beginners

```go
func doEbpf() {
    sig := make(chan os.Signal, 1)
    signal.Notify(sig, os.Interrupt)

    b, _ := bpf.NewModuleFromFile("hello.bpf.o")
    defer b.Close()
    b.BPFLoadObject()

    p, _ := bpfModule.GetProgram("hello")
    p.AttachKprobe("__x64_sys_execve")

    go bpf.TracePrint()
    <-sig
}
```

eBPF maps

Maps are a generic data structure for storage of different types of data. They allow sharing of data between eBPF kernel programs, and also between kernel and user-space applications.

Each map type has the following attributes:

* type
* maximum number of elements
* key size in bytes
* value size in bytes

```
BPF_MAP_TYPE_UNSPEC
BPF_MAP_TYPE_HASH
BPF_MAP_TYPE_ARRAY
BPF_MAP_TYPE_PROG_ARRAY
BPF_MAP_TYPE_PERF_EVENT_ARRAY
BPF_MAP_TYPE_PERCPU_HASH
BPF_MAP_TYPE_PERCPU_ARRAY
BPF_MAP_TYPE_STACK_TRACE
BPF_MAP_TYPE_CGROUP_ARRAY
BPF_MAP_TYPE_LRU_HASH
BPF_MAP_TYPE_LRU_PERCPU_HASH
BPF_MAP_TYPE_LPM_TRIE
BPF_MAP_TYPE_ARRAY_OF_MAPS
BPF_MAP_TYPE_HASH_OF_MAPS
BPF_MAP_TYPE_DEVMAP
BPF_MAP_TYPE_SOCKMAP
BPF_MAP_TYPE_CPUMAP
```

bpf_perf_event_output()

Write raw *data* blob into a special BPF perf event held by *map* of type **BPF_MAP_TYPE_PERF_EVENT_ARRAY**.

libbpfgo's PerfBuffer sends these data blobs on a Go channel

```c
BPF_PERF_OUTPUT(events);
SEC("kprobe/sys_execve")
int hello(void *ctx)
{
    u64 data = 1337;
    bpf_perf_event_output(ctx, &events, BPF_F_CURRENT_CPU, &data, sizeof(u64));
    return 0;
}
```

```
func main() {
...
    e := make(chan []byte, 300)
    pb, _ := b.InitPerfBuf("events", e, nil, 1024)
    pb.Start()

    go func() {
        for data := <-e {
            val := binary.LittleEndian.Uint64(data)
            fmt.Printf("data %d\n", data)
        }
    }()

    <-sig
    pb.Stop()
}
```

@lizrice

github.com/lizrice/libbpfgo-beginners

Recreate bpftrace command

```
bpftrace -e 'tracepoint:raw_syscalls:sys_enter { @[comm] = count();}'
```

```c
BPF_PERF_OUTPUT(events);
SEC("raw_tracepoint/sys_enter")
int hello(void *ctx)
{
    char data[100];
    bpf_get_current_comm(&data, 100);
    bpf_perf_event_output(ctx, &events, BPF_F_CURRENT_CPU, &data, 100);
    return 0;
}
```

```
func main() {
...
    prog.AttachRawTracepoint("sys_enter")
...
    c := make(map[string]int, 300)
    go func() {
        for data := range e {
            comm := string(data)
            c[comm]++
        }
    }()
    <-sig
    pb.Stop()
    for comm, n := range c {
        fmt.Printf("%s: %d\n", comm, n)
    }
}
```


github.com/lizrice/libbpfgo-beginners

Thank you

github.com/lizrice/libbpfgo-beginners

@lizrice