eBPF

for the rest of us

Federico Paolinelli - Red Hat

About me

Telco Network Team @ Red Hat

Contributed to:

- Athens
- KubeVirt
- SR-IOV Network Operator
- OPA Gatekeeper
- OVN-Kubernetes
- CNI Plugins
- MetalLB

hachyderm.io/@fedepaol
@fedepaol
fedepaol@gmail.com

eBPF is now mature

What is eBPF?

What is eBPF?

- Extended berkeley packet filter
- Same thing as Javascript was to browsers
- Allows us to change the behavior of the Linux kernel

What is eBPF?

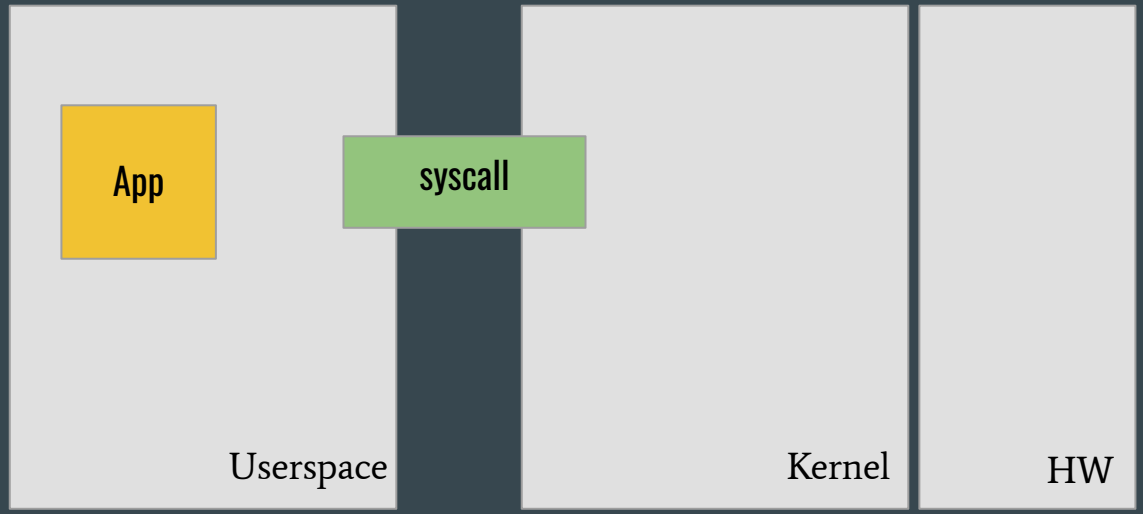
- Extended berkeley packet filter
- Same thing as Javascript was to browsers
- Allows us to change the behavior of the Linux kernel **in a safe and controlled way**

eBPF programs can be **loaded dynamically**

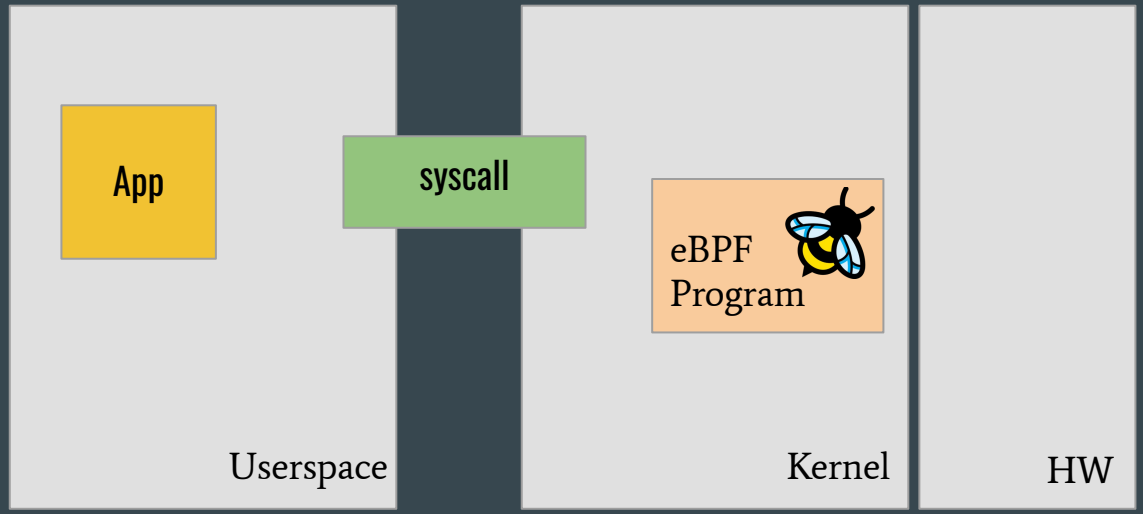
eBPF is
event driven

Why do we care?

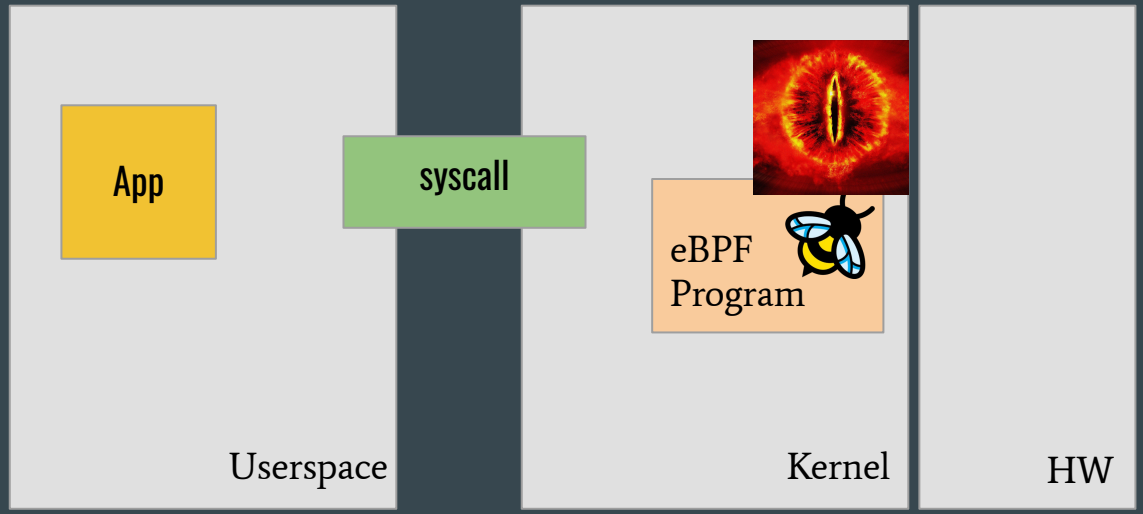
Regular Kernel Interaction



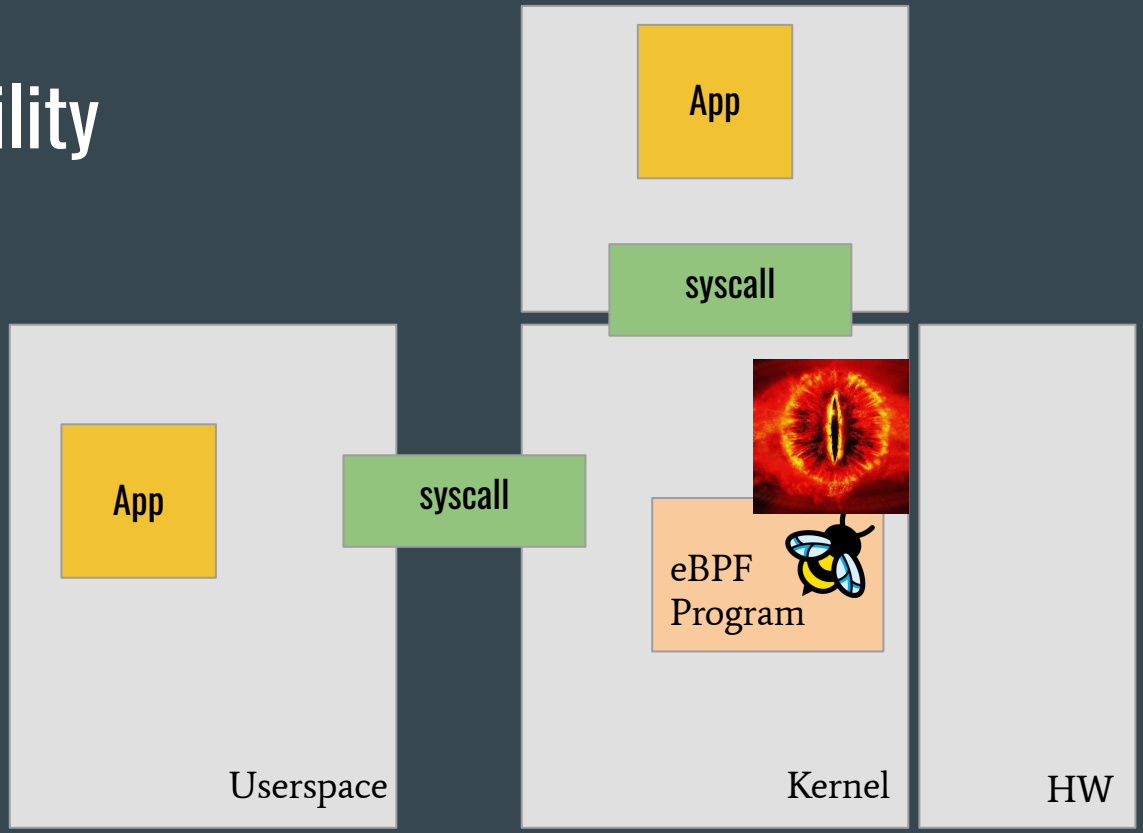
eBPF programs live in the kernel



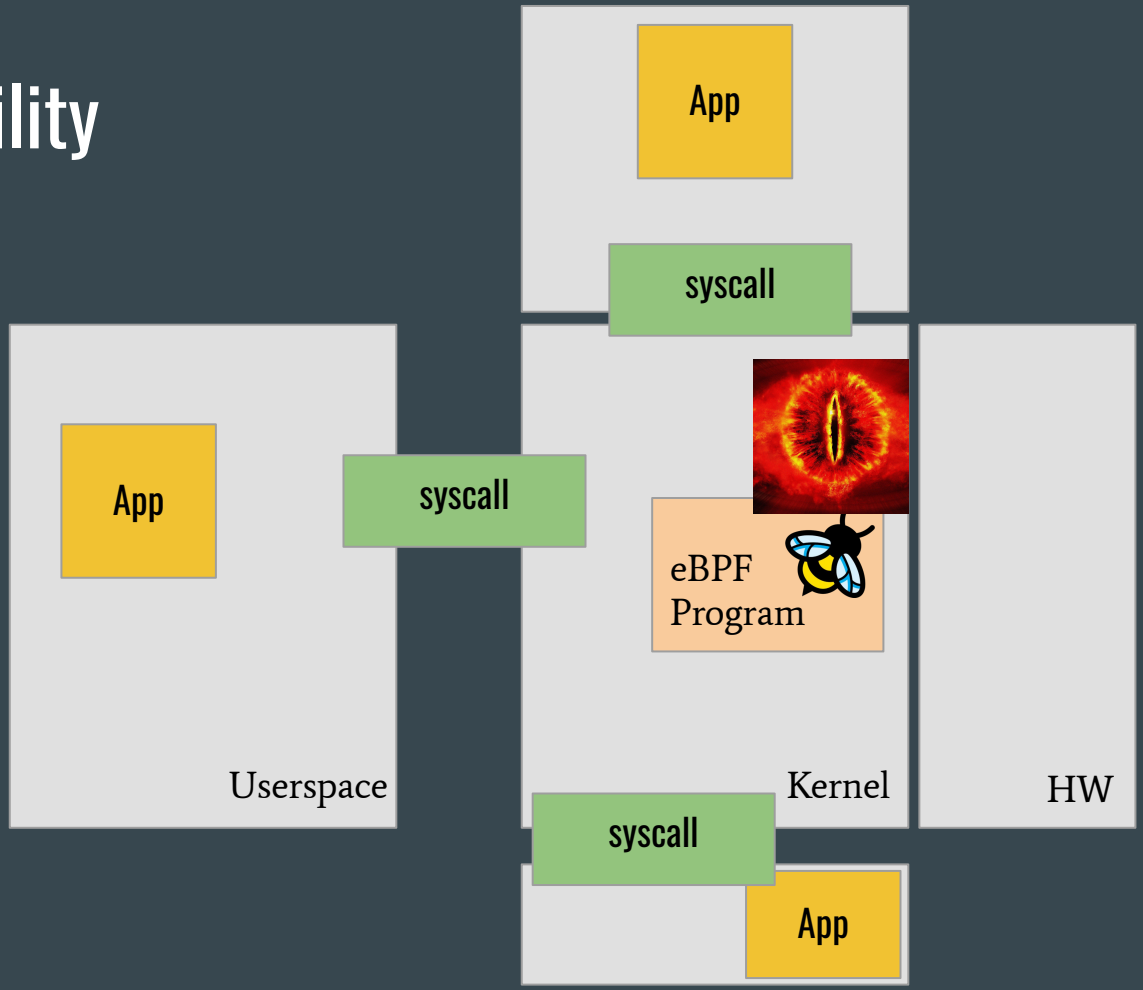
Observability



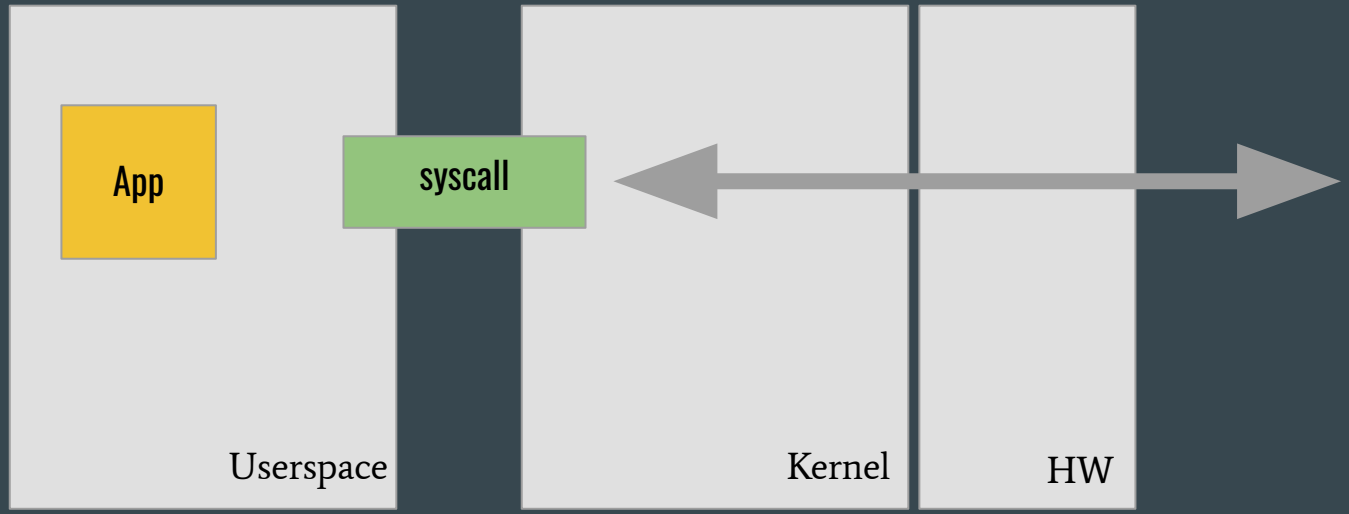
Observability



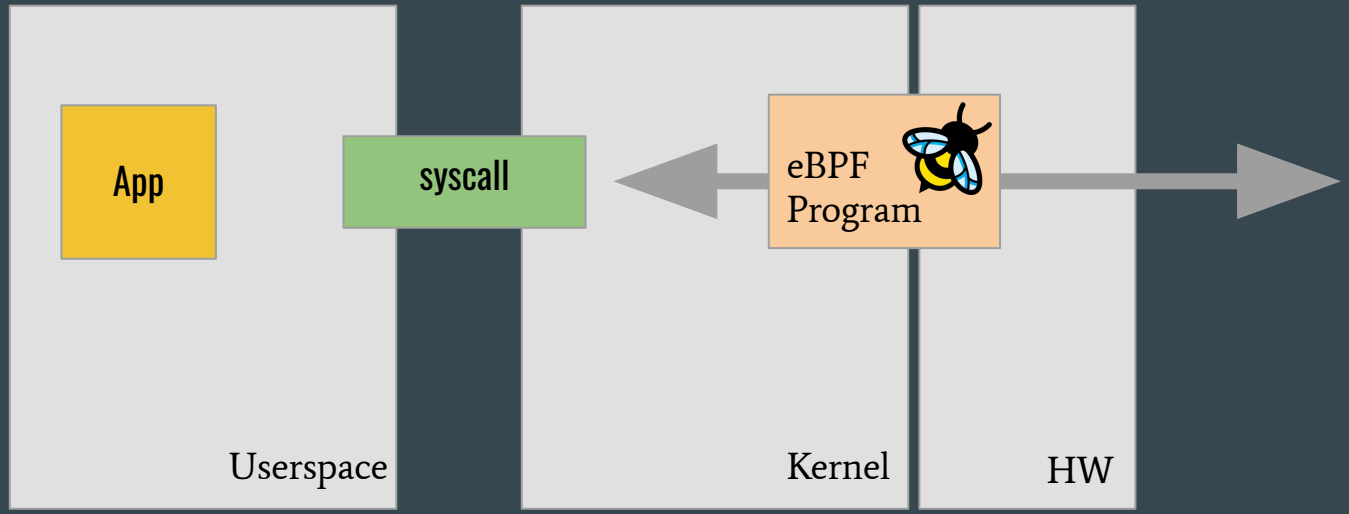
Observability



Networking



Networking



What are the alternatives?

- Linux kernel modules
- Contributing the feature to the kernel

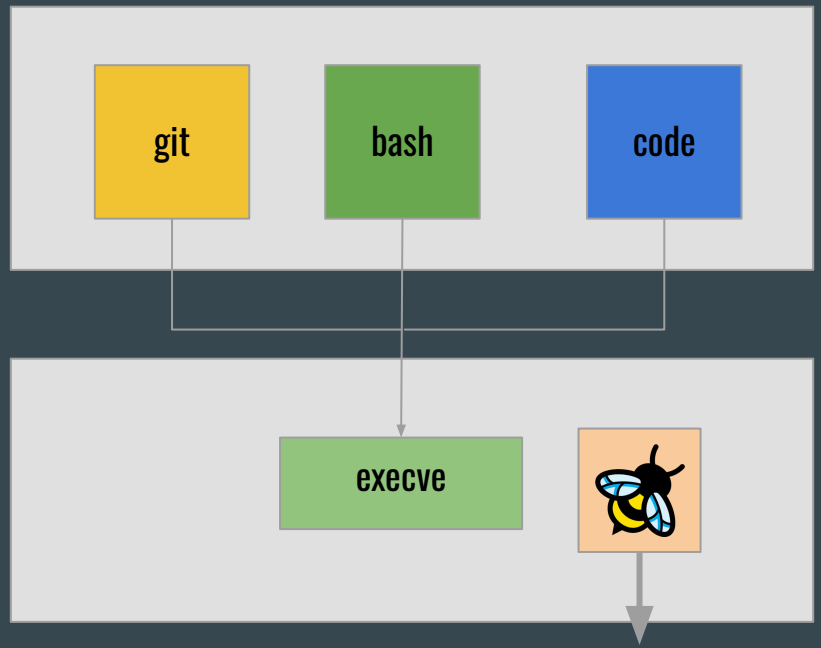
More use cases later

Let’s see an example

Disclaimer!

There will be C code

```c
SEC("kprobe/sys_execve")
int kprobe_execve() {
    bpf_printk("called!");
    return 0;
}
```



/sys/kernel/debug/tracing/trace_pipe

```
git-9348     [000] ...21  2534.887840: bpf_trace_printk: called!
git-9351     [000] ...21  2534.891143: bpf_trace_printk: called!
tail-9354    [000] ...21  2534.894813: bpf_trace_printk: called!
git-9355     [001] ...21  2534.894813: bpf_trace_printk: called!
```

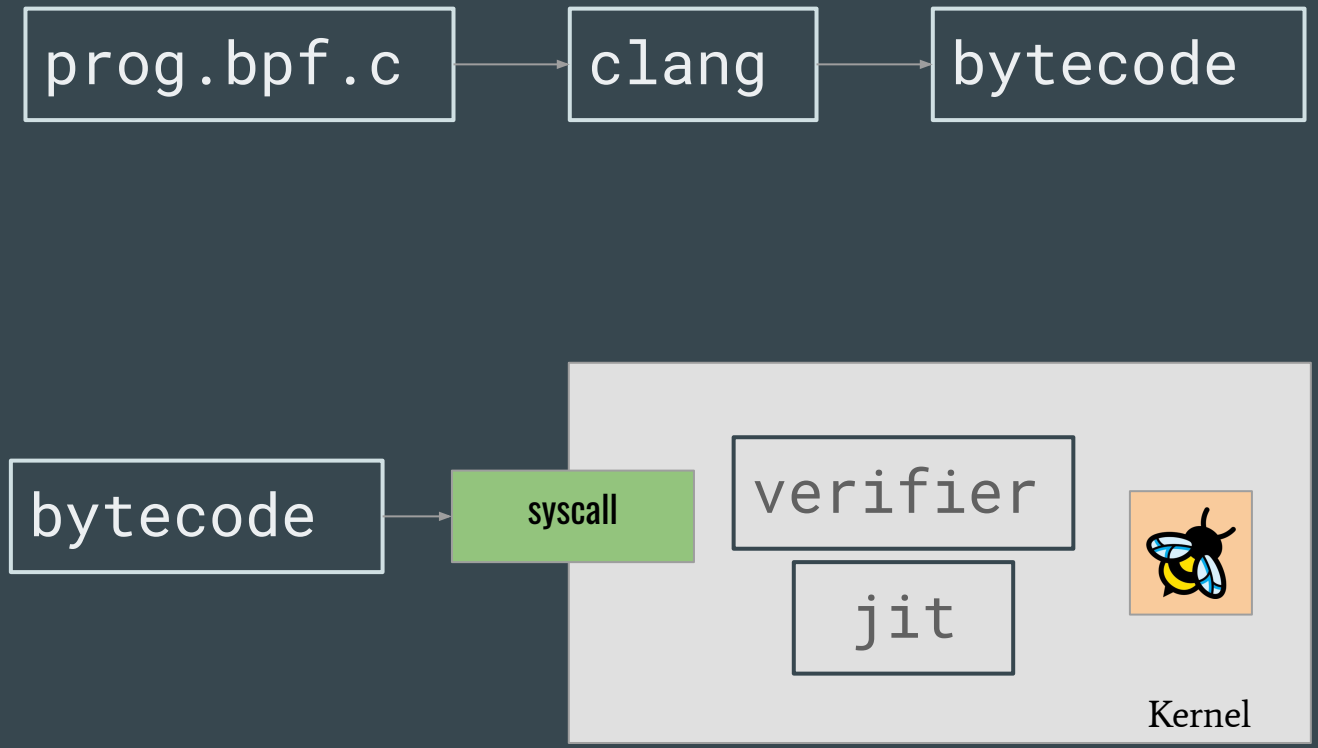
Do I really want something like this running in my kernel?

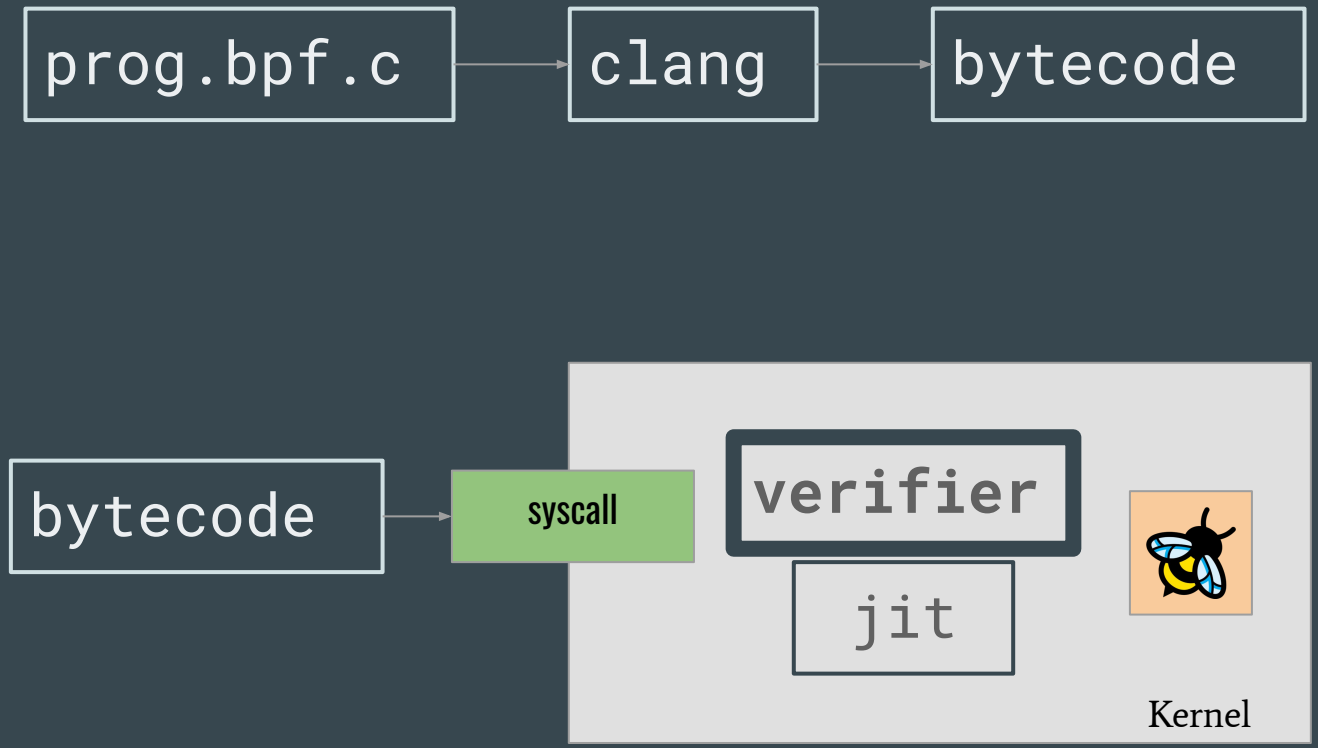
in order to load an eBPF program,
you need

PERMISSIONS

eBPF Verifier


prog.bpf.c
clang
bytecode


prog.bpf.c
clang
bytecode
bytecode
syscall
verifier
jit
Kernel


prog.bpf.c
clang
bytecode
bytecode
syscall
verifier
jit
Kernel

The eBPF verifier checks for

- infinite loops
- uninitialized variables
- memory access out of allowed bounds
- program size (below 4096 instructions)
- program complexity
- allowed calls

eBPF program structure

```c
#include "vmlinux.h"
#include "bpf/bpf_helpers.h"

struct
{
  __uint(type, BPF_MAP_TYPE_ARRAY);
  __uint(max_entries, MAX_MAP_ENTRIES);
  __type(key, __u32);
  __type(value, struct arguments);
} xdp_params_array SEC(".maps");

SEC("xdp")
int xdp_prog_func(struct xdp_md *ctx)
{
   bpf_printk("called");
   // access the map
  return XDP_TX;
}
```

The program

```
SEC("xdp")
int xdp_prog_func(struct xdp_md *ctx)
{
  bpf_printk("called");
  return XDP_TX;
}
```

The program type

```
SEC("xdp")
int xdp_prog_func(struct xdp_md *ctx)
{
  bpf_printk("called");
  return XDP_TX;
}
```

The context parameter

```
SEC("xdp")
int xdp_prog_func(struct xdp_md *ctx)
{
  bpf_printk("called");
  return XDP_TX;
}
```

The return value

```
SEC("xdp")
int xdp_prog_func(struct xdp_md *ctx)
{
  bpf_printk("called");
  return XDP_TX;
}
```


https://flic.kr/p/H4D7j
eBPF Helpers

- *This framework differs from the older, "classic" BPF (or "cBPF") in several aspects, one of them being the ability to call special functions (or "helpers") from within a program*
- *These helpers are used by eBPF programs to interact with the system, or with the context in which they work*
- *each program type can only call a subset of those helpers*

from man bpf-helpers

BPF Helpers

```
SEC("kprobe/sys_execve")
int kprobe_execve() {
    char comm[20];
    bpf_get_current_comm(comm, sizeof(comm));
    bpf_printk("execve: %s\n", comm);
    return 0;
}
```

Description

Copy the comm attribute of the current task into buf of size_of_buf. The comm attribute contains the name of the executable (excluding the path) for the current task. The size_of_buf must be strictly positive

BPF Helpers

```
SEC("kprobe/sys_execve")
int kprobe_execve() {
    char comm[20];
    bpf_get_current_comm(comm, sizeof(comm));
    bpf_printk("execve: %s\n", comm);
    return 0;
}
```

Description

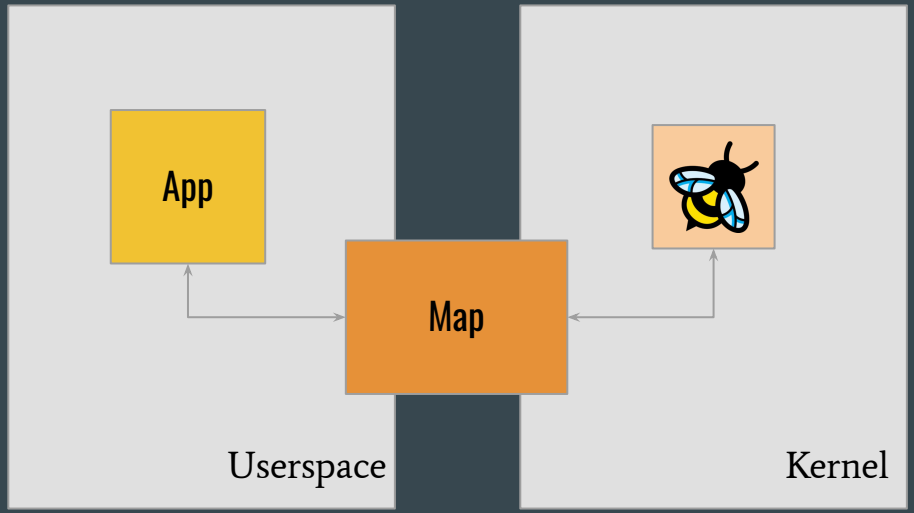
Copy the comm attribute of the current task into buf of size_of_buf. The comm attribute contains the name of the executable (excluding the path) for the current task. The size_of_buf must be strictly positive

Description

Copy the comm attribute of the current task into buf of size_of_buf. The comm attribute contains the name of the executable (excluding the path) for the current task. The size_of_buf must be strictly positive


MAR DEL ZUR
Maps

eBPF maps



Maps are for...

- The only way to have userspace and eBPF programs communicate
- Configuration
- Saving state / share data between programs
- Sending data to userspace

Type of maps

- Array
- HashMap
- LRU
- Perf / Ring Buffer
- SocketMap
- ...

Maps - Hash Map

```
struct command {
    u8 cmd[64];
};

struct{
    __uint(type, BPF_MAP_TYPE_HASH);
    __type(key, struct command);
    __type(value, struct action);
    __uint(max_entries, 1024);
} action_cmd_map SEC(".maps");
```

Maps - Hash Map

```
SEC("kprobe/sys_execve")
int kprobe_execve() {
    struct command key;
    memset(&key, 0, sizeof(key));
    bpf_get_current_comm(&key->cmd, sizeof(key->cmd));

    args = (struct action *)bpf_map_lookup_elem(&action_cmd_map, &key);
    if (!args) {
        return 0;
    }

    // do stuff
    return 0;
}
```

Maps - Hash Map

```
SEC("kprobe/sys_execve")
int kprobe_execve() {
    struct command key;
    memset(&key, 0, sizeof(key));
    bpf_get_current_comm(&key->cmd, sizeof(key->cmd));

    args = (struct action *)bpf_map_lookup_elem(&action_cmd_map, &key);
    if (!args) {
        return 0;
    }

    // do stuff
    return 0;
}
```

Maps - Ring buffer

```
struct {
    __uint(type, BPF_MAP_TYPE_RINGBUF);
    __uint(max_entries, 1 << 24);
} ring_buffer SEC(".maps");


SEC("kprobe/sys_openat")
int BPF_KPROBE(kprobe_openat, struct pt_regs *regs) {
    struct event *event = 0;
    event = bpf_ringbuf_reserve(&ring_buffer, sizeof(struct event), 0);

    // fill event
    bpf_ringbuf_submit(event, 0);
    return 0;
}
```

Maps - Ring buffer

```
struct {
    __uint(type, BPF_MAP_TYPE_RINGBUF);
    __uint(max_entries, 1 << 24);
} ring_buffer SEC(".maps");

SEC("kprobe/sys_openat")
int BPF_KPROBE(kprobe_openat, struct pt_regs *regs) {
    struct event *event = 0;
    event = bpf_ringbuf_reserve(&ring_buffer, sizeof(struct event), 0);

    // fill event
    bpf_ringbuf_submit(event, 0);
    return 0;
}
```

Maps - Ring buffer

```
struct {
    __uint(type, BPF_MAP_TYPE_RINGBUF);
    __uint(max_entries, 1 << 24);
} ring_buffer SEC(".maps");

SEC("kprobe/sys_openat")
int BPF_KPROBE(kprobe_openat, struct pt_regs *regs) {
    struct event *event = 0;
    event = bpf_ringbuf_reserve(&ring_buffer, sizeof(struct event), 0);

    // fill event
    bpf_ringbuf_submit(event, 0);
    return 0;
}
```

Maps - Ring buffer

```
struct {
    __uint(type, BPF_MAP_TYPE_RINGBUF);
    __uint(max_entries, 1 << 24);
} ring_buffer SEC(".maps");

SEC("kprobe/sys_openat")
int BPF_KPROBE(kprobe_openat, struct pt_regs *regs) {
    struct event *event = 0;
    event = bpf_ringbuf_reserve(&ring_buffer, sizeof(struct event), 0);

    // fill event
    bpf_ringbuf_submit(event, 0);
    return 0;
}
```

Portability


Kernel


Kernel


Kernel

Portability

- Different kernels might have different data layouts
- The program is not aware of the memory layout
- How can we make the same artifact run on different kernels without recompiling?

CO-RE

BPF CO-RE (Compile Once – Run Everywhere) is a modern approach to writing portable BPF applications that can run on multiple kernel versions and configurations without modifications and runtime source code compilation on the target machine.

from nakryiko.com/posts/bpf-core-reference-guide/

BTF - BPF type format

- Kind of metadata, describing the program
- A program has BTF information associated to it (i.e. which fields it wants to read)
- The kernel comes with BTF information (i.e. where each field is)

BPF Loader

- When an eBPF program is loaded matches the program's BTF information and the kernel's BTF information
- Provides the offset to the program
- The kernel doesn't care

```c
#include "vmlinux.h"
#include "bpf/bpf_helpers.h"

struct
{
  __uint(type, BPF_MAP_TYPE_ARRAY);
  __uint(max_entries, MAX_MAP_ENTRIES);
  __type(key, __u32);
  __type(value, struct arguments);
} xdp_params_array SEC(".maps");

SEC("xdp")
int xdp_prog_func(struct xdp_md *ctx)
{
   bpf_printk("called");
  return XDP_TX;
}
```

```c
#include "vmlinux.h"
#include "bpf/bpf_helpers.h"

struct
{
  __uint(type, BPF_MAP_TYPE_ARRAY);
  __uint(max_entries, MAX_MAP_ENTRIES);
  __type(key, __u32);
  __type(value, struct arguments);
} xdp_params_array SEC(".maps");

SEC("xdp")
int xdp_prog_func(struct xdp_md *ctx)
{
   bpf_printk("called");
  return XDP_TX;
}
```

vmlinux.h

```
bpftool btf dump file /sys/kernel/btf/vmlinux format c > vmlinux.h
```

vmlinux.h

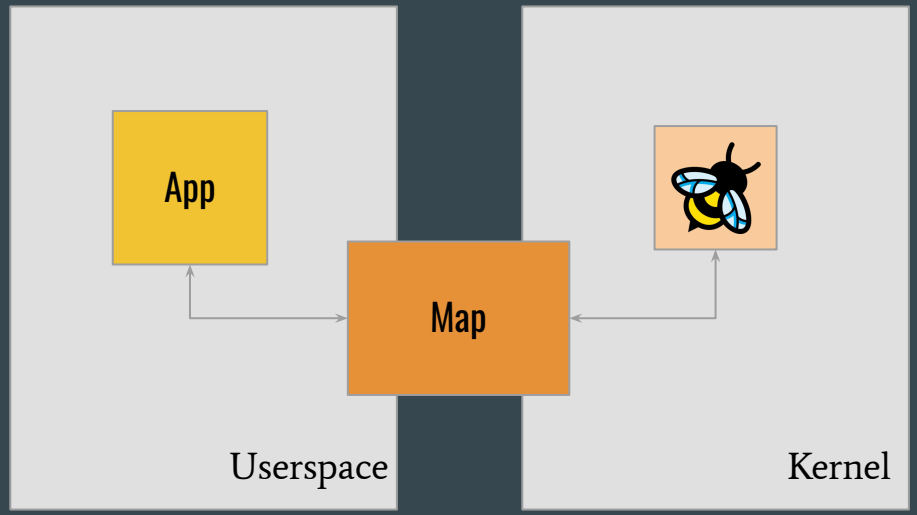
```
bpftool btf dump file /sys/kernel/btf/vmlinux format c > vmlinux.h
```

```
#include "bpf/bpf_helpers.h"
```

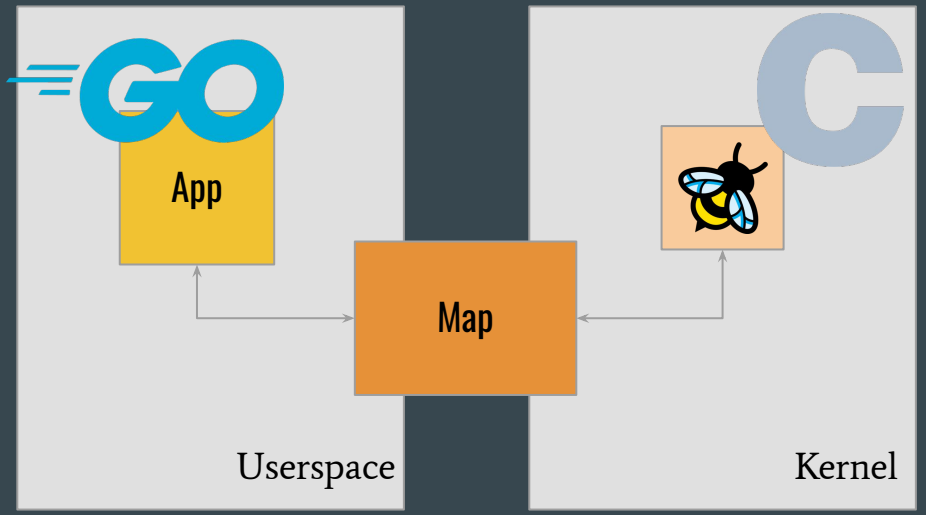
What does it have to do with

GO?

With Go



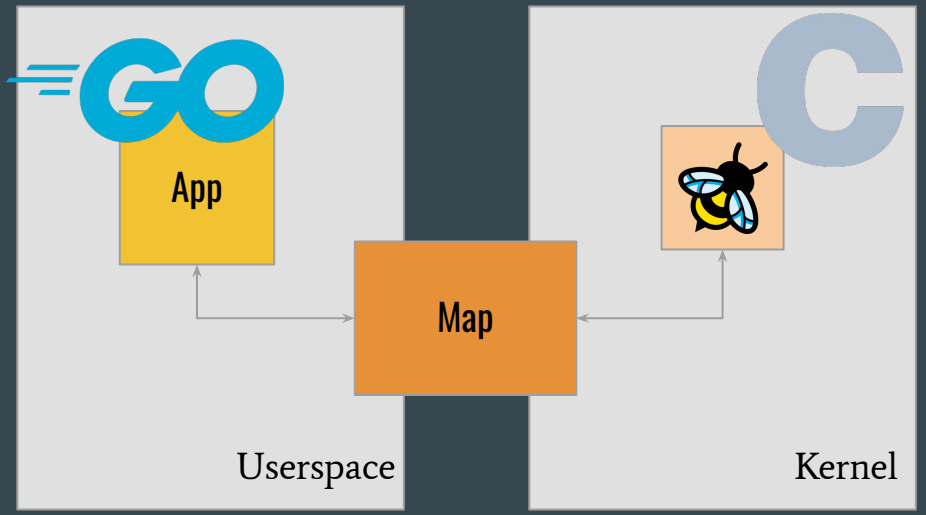
With Go



Cilium EBPF github.com/cilium/ebpf

LibBPF Go github.com/aquasecurity/libbpfgo

With Go



Cilium EBPF github.com/cilium/ebpf

LibBPF Go github.com/aquasecurity/libbpfgo

cilium/ebpf go

bpf2go tool

- compiles C code and generates eBPF elf file
- embeds the eBPF elf file in the single go binary
- provides references to the eBPF maps / programs that are accessible from Go
- generates Go equivalent objects of C structs

bpf2go tool

```
bpf2go -type arguments lb ebpf/xdp_lb.c -- -I ./include
```

bpf2go tool

```
bpf2go -type arguments lb ebpf/xdp_lb.c -- -I ./include
```

bpf2go tool

```
bpf2go -type arguments lb ebpf/xdp_lb.c -- -I ./include
```

bpf2go tool

```
bpf2go -type arguments lb ebpf/xdp_lb.c -- -I ./include
```

bpf2go tool

```
struct arguments
{
  __u8 dst_mac[6];
  __u32 daddr;
  __u32 saddr;
  __u32 vip;
};
```

```
type lbArguments struct {
    DstMac [6]uint8
    _      [2]byte
    Daddr  uint32
    Saddr  uint32
    Vip    uint32
}
```

bpf2go tool

```
struct arguments
{
  __u8 dst_mac[6];
  __u32 daddr;
  __u32 saddr;
  __u32 vip;
};
```

```
type lbArguments struct {
    DstMac [6]uint8
    _      [2]byte
    Daddr  uint32
    Saddr  uint32
    Vip    uint32
}
```

bpf2go tool

```
objs := lbObjects{}
if err := loadLbObjects(&objs, nil); err != nil {
    log.Fatalf("loading objects: %s", err)
}
defer objs.Close()
```

bpf2go tool

```
struct {
  __uint(type, BPF_MAP_TYPE_ARRAY);
  __uint(max_entries, MAX_MAP_ENTRIES);
  __type(key, __u32);
  __type(value, struct arguments);
} xdp_params_array SEC(".maps");
```

```
args := lbArguments{
    Daddr:  intDest,
    Saddr:  intSrc,
    DstMac: macArray,
    Vip:    intVip,
}

objs.XdpParamsArray.Put(uint32(0), args)

objs.XdpParamsArray.Lookup(uint32(0), &args)
```

bpf2go tool

```
struct {
  __uint(type, BPF_MAP_TYPE_ARRAY);
  __uint(max_entries, MAX_MAP_ENTRIES);
  __type(key, __u32);
  __type(value, struct arguments);
} xdp_params_array SEC(".maps");
```

```
args := lbArguments{
    Daddr:  intDest,
    Saddr:  intSrc,
    DstMac: macArray,
    Vip:    intVip,
}

objs.XdpParamsArray.Put(uint32(0), args)

objs.XdpParamsArray.Lookup(uint32(0), &args)
```

bpf2go tool

```
struct {
  __uint(type, BPF_MAP_TYPE_ARRAY);
  __uint(max_entries, MAX_MAP_ENTRIES);
  __type(key, __u32);
  __type(value, struct arguments);
} xdp_params_array SEC(".maps");
```

```
args := lbArguments{
    Daddr:  intDest,
    Saddr:  intSrc,
    DstMac: macArray,
    Vip:    intVip,
}

objs.XdpParamsArray.Put(uint32(0), args)

objs.XdpParamsArray.Lookup(uint32(0), &args)
```

ebpf go

- allows to attach an eBPF program to the corresponding hook
- utilities for consuming perf / ring buffers
- kernel features discovery

What can we do with eBPF?

Kernel inspection

Kernel inspection

- Kprobe / Kretprobe
- Tracepoints
- Perf Events
- ...

KProbes

KProbe / KRetprobe

- Entry / exit of any kernel function
- API not guaranteed
- Syscalls are reasonably stable

Checking who is opening a given file

```
SEC("kprobe/sys_openat")
int BPF_KPROBE(kprobe_openat, struct pt_regs *regs) {
    struct event *event = 0;
    event = bpf_ringbuf_reserve(&ring_buffer, sizeof(struct event), 0);
    if (!event) {
        return 0;
    }
    char *pathname;
    pathname = (char*) PT_REGS_PARM2_CORE(regs);
    bpf_probe_read_str(&event->path, sizeof(event->path), (void *) pathname);

    event->pid = bpf_get_current_pid_tgid() >> 32;
    bpf_get_current_comm(&event->command, sizeof(event->command));
    bpf_ringbuf_submit(event, 0);
    return 0;
}
```

Checking who is opening a given file

```
SEC("kprobe/sys_openat")
int BPF_KPROBE(kprobe_openat, struct pt_regs *regs) {
    struct event *event = 0;
    event = bpf_ringbuf_reserve(&ring_buffer, sizeof(struct event), 0);
    if (!event) {
        return 0;
    }
    char *pathname;
    pathname = (char*) PT_REGS_PARM2_CORE(regs);
    bpf_probe_read_str(&event->path, sizeof(event->path), (void *) pathname);

    event->pid = bpf_get_current_pid_tgid() >> 32;
    bpf_get_current_comm(&event->command, sizeof(event->command));
    bpf_ringbuf_submit(event, 0);
    return 0;
}
```

Checking who is opening a given file

```
SEC("kprobe/sys_openat")
int BPF_KPROBE(kprobe_openat, struct pt_regs *regs) {
    struct event *event = 0;
    event = bpf_ringbuf_reserve(&ring_buffer, sizeof(struct event), 0);
    if (!event) {
        return 0;
    }
    char *pathname;
    pathname = (char*) PT_REGS_PARM2_CORE(regs);
    bpf_probe_read_str(&event->path, sizeof(event->path), (void *) pathname);

    event->pid = bpf_get_current_pid_tgid() >> 32;
    bpf_get_current_comm(&event->command, sizeof(event->command));
    bpf_ringbuf_submit(event, 0);
    return 0;
}
```

Checking who is opening a given file

```
SEC("kprobe/sys_openat")
int BPF_KPROBE(kprobe_openat, struct pt_regs *regs) {
    struct event *event = 0;
    event = bpf_ringbuf_reserve(&ring_buffer, sizeof(struct event), 0);
    if (!event) {
        return 0;
    }
    char *pathname;
    pathname = (char*) PT_REGS_PARM2_CORE(regs);
    bpf_probe_read_str(&event->path, sizeof(event->path), (void *) pathname);

    event->pid = bpf_get_current_pid_tgid() >> 32;
    bpf_get_current_comm(&event->command, sizeof(event->command));
    bpf_ringbuf_submit(event, 0);
    return 0;
}
```

Kill who is opening a given file

```
SEC("kprobe/sys_openat")
int BPF_KPROBE(kprobe_openat, struct pt_regs *regs) {
    struct file_path p;
    memset(&p, 0, sizeof(p));

    char *pathname;
    pathname = (char *)PT_REGS_PARM2_CORE(regs);
    bpf_probe_read_str(p.pp,  sizeof(p.pp), pathname);

    struct action *args = 0;
    args = (struct action *)bpf_map_lookup_elem(&action_file_map, &p);
    if (!args) {
        return 0;
    }
    if (args->kill) {
        bpf_send_signal(9);
    }
    return 0;
}
```

Kill who is opening a given file

```
SEC("kprobe/sys_openat")
int BPF_KPROBE(kprobe_openat, struct pt_regs *regs) {
    struct file_path p;
    memset(&p, 0, sizeof(p));

    char *pathname;
    pathname = (char *)PT_REGS_PARM2_CORE(regs);
    bpf_probe_read_str(p.pp,  sizeof(p.pp), pathname);

    struct action *args = 0;
    args = (struct action *)bpf_map_lookup_elem(&action_file_map, &p);
    if (!args) {
        return 0;
    }
    if (args->kill) {
        bpf_send_signal(9);
    }
    return 0;
}
```

Tracepoints

Tracepoints

- Specific hooks in the kernel
- Guaranteed to be stable
- BTF generated data structures for parameters

Tracepoints

```
more /sys/kernel/tracing/available_events | grep skb
tcp:tcp_retransmit_skb
udp:udp_fail_queue_rcv_skb
net:netif_receive_skb_list_exit
net:netif_receive_skb_exit
net:netif_receive_skb_list_entry
net:netif_receive_skb_entry
net:netif_receive_skb
skb:skb_copy_datagram_iovec
skb:consume_skb
skb:kfree_skb
```

Tracepoints

```
/*
 * Tracepoint for free an sk_buff:
 */
TRACE_EVENT(kfree_skb,

    TP_PROTO(struct sk_buff *skb, void *location,
         enum skb_drop_reason reason),

    TP_ARGS(skb, location, reason),

    TP_STRUCT__entry(
        __field(void *,        skbaddr)
        __field(void *,        location)
        __field(unsigned short,    protocol)
        __field(enum skb_drop_reason,    reason)
    ),

    TP_printk("skbaddr=%p protocol=%u location=%pS reason: %s",
          __entry->skbaddr, __entry->protocol, __entry->location,
          __print_symbolic(__entry->reason,
                   DEFINE_DROP_REASON(FN, FNe)))
);
```

include/trace/events/skb.h

Tracepoints

```
/*
 * Tracepoint for free an sk_buff:
 */
TRACE_EVENT(kfree_skb,

    TP_PROTO(struct sk_buff *skb, void *location,
         enum skb_drop_reason reason),

    TP_ARGS(skb, location, reason),

    TP_STRUCT__entry(
        __field(void *,        skbaddr)
        __field(void *,        location)
        __field(unsigned short,    protocol)
        __field(enum skb_drop_reason,    reason)
    ),

    TP_printk("skbaddr=%p protocol=%u location=%pS reason: %s",
          __entry->skbaddr, __entry->protocol, __entry->location,
          __print_symbolic(__entry->reason,
                   DEFINE_DROP_REASON(FN, FNe)))
);
```

trace_kfree_skb

include/trace/events/skb.h

Checking what packets are being dropped

```
SEC("tp_btf/skb/kfree_skb")
int kfree_skb(struct trace_event_raw_kfree_skb *args) {
    struct sk_buff skb;
    __builtin_memset(&skb, 0, sizeof(skb));

    bpf_probe_read(&skb, sizeof(struct sk_buff), args->skbaddr);
    struct sock *sk = skb.sk;
    enum skb_drop_reason reason = args->reason;
    /* handle the event… */
}
```

Checking what packets are being dropped

```
SEC("tp_btf/skb/kfree_skb")
int kfree_skb(struct trace_event_raw_kfree_skb *args) {
    struct sk_buff skb;
    __builtin_memset(&skb, 0, sizeof(skb));

    bpf_probe_read(&skb, sizeof(struct sk_buff), args->skbaddr);
    struct sock *sk = skb.sk;
    enum skb_drop_reason reason = args->reason;
    /* handle the event… */
}
```

Userspace inspection!

UProbe / URetprobe

- same as kprobe / kretprobe but for userspace
- must be attached to the binary
- even less stability guarantees
- might be useful for well known, stable libraries

Intercepting SSL_write calls

```
SEC("uprobe/SSL_write")
int probe_entry_SSL_write(struct pt_regs *ctx)
{
    u64 current_pid_tgid = bpf_get_current_pid_tgid();
    u32 pid = current_pid_tgid >> 32;

    const char *buf = (const char *)PT_REGS_PARM2(ctx);
    struct active_ssl_buf active_ssl_buf_t;
    active_ssl_buf_t.buf = (uintptr_t)buf;
    bpf_map_update_elem(&active_ssl_write_args_map, &current_pid_tgid,
                        &active_ssl_buf_t, BPF_ANY);

    return 0;
}
```

Intercepting SSL_write calls

```
SEC("uprobe/SSL_write")
int probe_entry_SSL_write(struct pt_regs *ctx)
{
    u64 current_pid_tgid = bpf_get_current_pid_tgid();
    u32 pid = current_pid_tgid >> 32;

    const char *buf = (const char *)PT_REGS_PARM2(ctx);
    struct active_ssl_buf active_ssl_buf_t;
    active_ssl_buf_t.buf = (uintptr_t)buf;
    bpf_map_update_elem(&active_ssl_write_args_map, &current_pid_tgid,
                        &active_ssl_buf_t, BPF_ANY);

    return 0;
}
```

Intercepting SSL_write calls

```
SEC("uretprobe/SSL_write")
int probe_ret_SSL_write(struct pt_regs *ctx)
{
    struct active_ssl_buf *active_ssl_buf_t =
        bpf_map_lookup_elem(&active_ssl_write_args_map,
&current_pid_tgid);
    const char *buf;
    bpf_probe_read(&buf, sizeof(const char *), &active_ssl_buf_t->buf);
    process_SSL_data(ctx, current_pid_tgid, buf);
    bpf_map_delete_elem(&active_ssl_write_args_map, &current_pid_tgid);
    return 0;
}
```

Intercepting SSL_write calls

```
SEC("uretprobe/SSL_write")
int probe_ret_SSL_write(struct pt_regs *ctx)
{
    struct active_ssl_buf *active_ssl_buf_t =
        bpf_map_lookup_elem(&active_ssl_write_args_map,
&current_pid_tgid);
    const char *buf;
    bpf_probe_read(&buf, sizeof(const char *), &active_ssl_buf_t->buf);
    process_SSL_data(ctx, current_pid_tgid, buf);
    bpf_map_delete_elem(&active_ssl_write_args_map, &current_pid_tgid);
    return 0;
}
```

Intercepting SSL_write calls

```
SEC("uretprobe/SSL_write")
int probe_ret_SSL_write(struct pt_regs *ctx)
{
    struct active_ssl_buf *active_ssl_buf_t =
        bpf_map_lookup_elem(&active_ssl_write_args_map,
&current_pid_tgid);
    const char *buf;
    bpf_probe_read(&buf, sizeof(const char *), &active_ssl_buf_t->buf);
    process_SSL_data(ctx, current_pid_tgid, buf);
    bpf_map_delete_elem(&active_ssl_write_args_map, &current_pid_tgid);
    return 0;
}
```


Networking

Networking

- packet filtering
- packet manipulation
- socket redirection

XDP - Express Data Path

- hook for ingress packets only
- very early in the stack
- can pass to the linux kernel, drop, redirect
- pointer-fu to navigate the frame manually

XDP

```
SEC("xdp")
int xdp_only_tcp(struct xdp_md *ctx) {
  void *data = (void *)(long)ctx->data;
  void *data_end = (void *)(long)ctx->data_end;
  struct ethhdr *eth = data;
  __u32 eth_proto;
  eth_proto = eth->h_proto;
  iph = data + sizeof(struct ethhdr);
  if ((iph + 1) > data_end) {
    return XDP_DROP;
  }

  if (iph->protocol != IPPROTO_TCP) {
    return XDP_DROP;
  }
  return XDP_PASS;
}
```

XDP

```
SEC("xdp")
int xdp_only_tcp(struct xdp_md *ctx) {
  void *data = (void *)(long)ctx->data;
  void *data_end = (void *)(long)ctx->data_end;
  struct ethhdr *eth = data;
  __u32 eth_proto;
  eth_proto = eth->h_proto;
  iph = data + sizeof(struct ethhdr);
  if ((iph + 1) > data_end) {
    return XDP_DROP;
  }

  if (iph->protocol != IPPROTO_TCP) {
    return XDP_DROP;
  }
  return XDP_PASS;
}
```

XDP

```
SEC("xdp")
int xdp_only_tcp(struct xdp_md *ctx) {
  void *data = (void *)(long)ctx->data;
  void *data_end = (void *)(long)ctx->data_end;
  struct ethhdr *eth = data;
  __u32 eth_proto;
  eth_proto = eth->h_proto;
  iph = data + sizeof(struct ethhdr);
  if ((iph + 1) > data_end) {
    return XDP_DROP;
  }

  if (iph->protocol != IPPROTO_TCP) {
    return XDP_DROP;
  }
  return XDP_PASS;
}
```

XDP

```
SEC("xdp")
int xdp_only_tcp(struct xdp_md *ctx) {
  void *data = (void *)(long)ctx->data;
  void *data_end = (void *)(long)ctx->data_end;
  struct ethhdr *eth = data;
  __u32 eth_proto;
  eth_proto = eth->h_proto;
  iph = data + sizeof(struct ethhdr);
  if ((iph + 1) > data_end) {
    return XDP_DROP;
  }

  if (iph->protocol != IPPROTO_TCP) {
    return XDP_DROP;
  }
  return XDP_PASS;
}
```

XDP

```
SEC("xdp")
int xdp_only_tcp(struct xdp_md *ctx) {
  void *data = (void *)(long)ctx->data;
  void *data_end = (void *)(long)ctx->data_end;
  struct ethhdr *eth = data;
  __u32 eth_proto;
  eth_proto = eth->h_proto;
  iph = data + sizeof(struct ethhdr);
  if ((iph + 1) > data_end) {
    return XDP_DROP;
  }

  if (iph->protocol != IPPROTO_TCP) {
    return XDP_DROP;
  }
  return XDP_PASS;
}
```

TC - Traffic Control Acton

- packet filtering
- packet manipulation
- works with ingress and egress
- packets under the form of _sk_buff

TC

```
SEC("tc_redirect")
int redirect(struct __sk_buff *skb)
{
    void *data = (void *)(unsigned long long)skb->data;

    if (bpf_ntohs(eth->h_proto) != ETH_P_IP)
        return TC_ACT_SHOT;

    key = bpf_ntohl(iph->saddr);
    nextHop = bpf_map_lookup_elem(&redirect_map_ipv4, &key);

    if (nextHop != NULL) {
        neighInfo.ipv4_nh = bpf_htonl(nextHop->nextHop);
        neighInfo.nh_family = AF_INET;
        long res = bpf_redirect_neigh(nextHop->interfaceID, &neighInfo,
sizeof(neighInfo), 0);
        return res;
    }
    return TC_ACT_OK;
}
```

TC

```
SEC("tc_redirect")
int redirect(struct __sk_buff *skb)
{
    void *data = (void *)(unsigned long long)skb->data;

    if (bpf_ntohs(eth->h_proto) != ETH_P_IP)
        return TC_ACT_SHOT;

    key = bpf_ntohl(iph->saddr);
    nextHop = bpf_map_lookup_elem(&redirect_map_ipv4, &key);

    if (nextHop != NULL) {
        neighInfo.ipv4_nh = bpf_htonl(nextHop->nextHop);
        neighInfo.nh_family = AF_INET;
        long res = bpf_redirect_neigh(nextHop->interfaceID, &neighInfo,
sizeof(neighInfo), 0);
        return res;
    }
    return TC_ACT_OK;
}
```

TC

```
SEC("tc_redirect")
int redirect(struct __sk_buff *skb)
{
    void *data = (void *)(unsigned long long)skb->data;

    if (bpf_ntohs(eth->h_proto) != ETH_P_IP)
        return TC_ACT_SHOT;

    key = bpf_ntohl(iph->saddr);
    nextHop = bpf_map_lookup_elem(&redirect_map_ipv4, &key);

    if (nextHop != NULL) {
        neighInfo.ipv4_nh = bpf_htonl(nextHop->nextHop);
        neighInfo.nh_family = AF_INET;
        long res = bpf_redirect_neigh(nextHop->interfaceID, &neighInfo,
sizeof(neighInfo), 0);
        return res;
    }
    return TC_ACT_OK;
}
```

The Go side

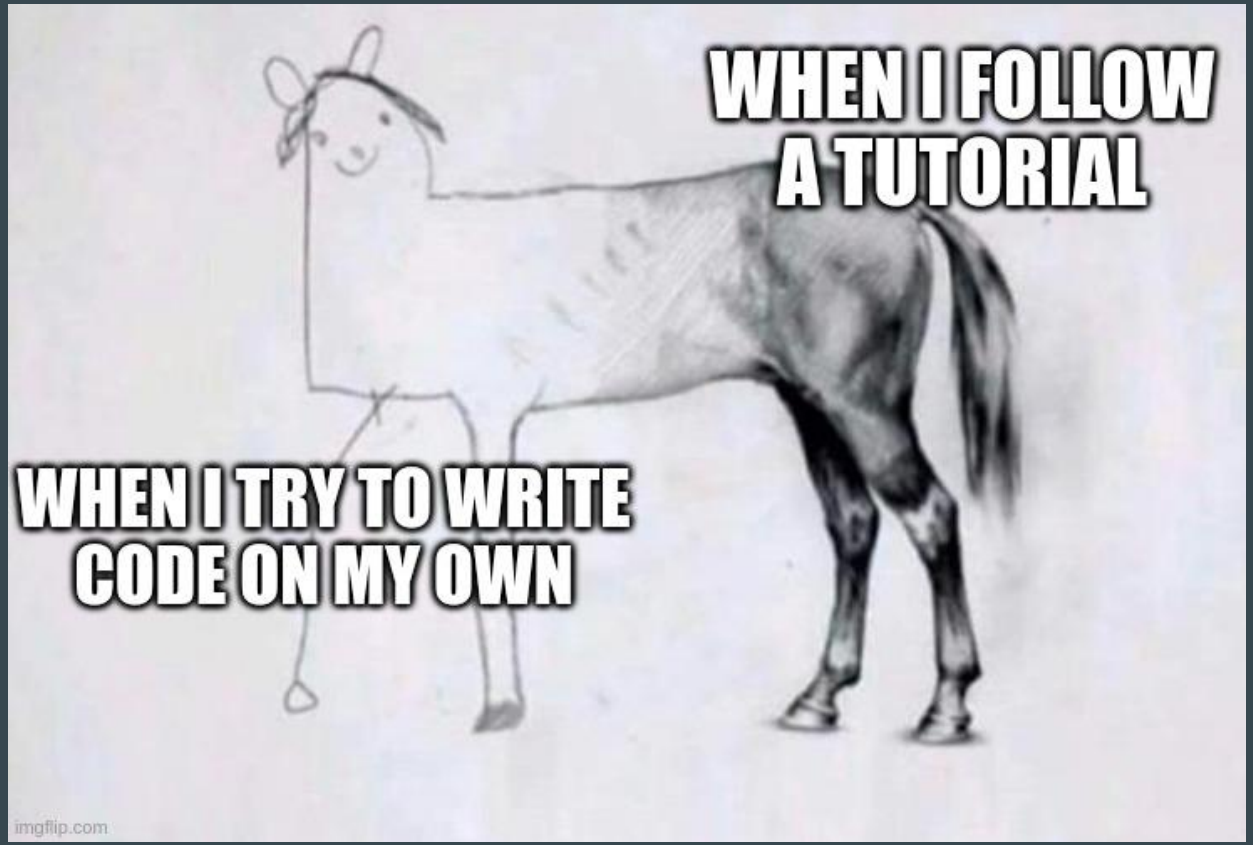
Attaching the program

```
link, _ := link.Kprobe("sys_openat", objs.KprobeOpenat, nil)

kp, _ := link.Tracepoint("syscalls", "sys_enter_openat", objs.HandleOpenat, nil)

ex, _ := link.OpenExecutable(*openSSLPath)

up, _ := ex.Uprobe("SSL_write", objs.ProbeEntrySSL_write, nil)
```

Interacting with the Maps

```
rd, _ := ringbuf.NewReader(objs.openMaps.RingBuffer)

_ := objs.ActionFileMap.Put(key, action)
iterator := objs.ActionFileMap.Iterate()
```

The Reality


WHEN I FOLLOW
A TUTORIAL
WHEN I TRY TO WRITE
CODE ON MY OWN
imgflip.com

Making the verifier happy is

an art

Missed check on XDP packet

```
invalid access to packet, off=12 size=2, R9(id=0,off=12,r=0): R9 offset is
outside of the packet (9 line(s) omitted)
```

Using memcpy instead of bpf_probe_read_str

```
load program: permission denied: 16: (71) r2 = *(u8 *)(r1 +63): R1
invalid mem access 'scalar' (24 line(s) omitted)
```

Each program type is a micro-framework

- different context argument
- different meaning of return values
- different eBPF helpers available
- different ways to load the program
- different lifecycles

we need be familiar with

the kernel

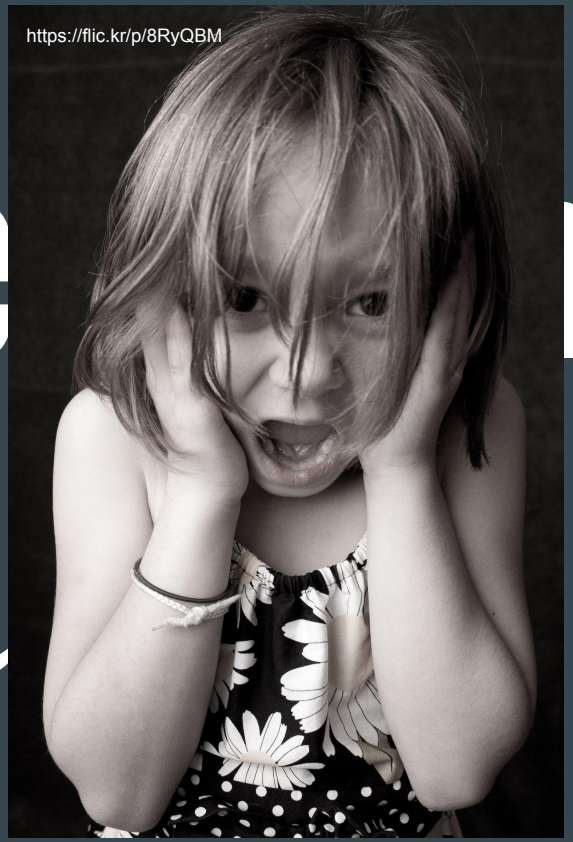
There is no debugger

- Did we attach the right program?
- Are we passing the parameters correctly?
- Are we parsing the arguments correctly?
- How about packet manipulation?

Older kernels support!

Olde nels

su !



How can we debug?

Log all the things!

```
bpf_printk("openssl process_SSL_data len :%d buf %s\n", len, buf);
```

```
sudo cat /sys/kernel/debug/tracing/trace_pipe | more
            sudo-28199   [009] ...21  5407.812081: bpf_trace_printk: got event /etc/passwd
            sudo-28199   [009] ...21  5407.812205: bpf_trace_printk: got event /etc/login.defs
            sudo-28199   [009] ...21  5407.812882: bpf_trace_printk: got event /usr/share/login.defs.d
            sudo-28199   [009] ...21  5407.812906: bpf_trace_printk: got event /etc/login.defs.d
 systemd-journal-940     [003] ...21  5407.813073: bpf_trace_printk: got event /proc/28199/comm
            sudo-28199   [009] ...21  5407.813139: bpf_trace_printk: got event /etc/security/pam_env.conf
            sudo-28199   [009] ...21  5407.813206: bpf_trace_printk: got event /etc/environment
 systemd-journal-940     [003] ...21  5407.813206: bpf_trace_printk: got event /proc/28199/cmdline
            sudo-28199   [009] ...21  5407.813250: bpf_trace_printk: got event /etc/login.defs
```

BPFTool to the rescue

```
bpftool prog show

...

393: kprobe  name probe_entry_SSL_write  tag 758445bff28b440d
gpl
    loaded_at 2023-11-06T23:11:03+0100  uid 0
    xlated 368B  jited 220B  memlock 4096B  map_ids 150,151,152
    btf_id 291
    pids uprobessl(31240)
```

BPFTool to the rescue

```
bpftool prog dump xlated name probe_entry_SSL_write
```

```
int probe_entry_SSL_write(struct pt_regs * ctx):
; int probe_entry_SSL_write(struct pt_regs *ctx)
   0: (bf) r6 = r1
; u64 current_pid_tgid = bpf_get_current_pid_tgid();
   1: (85) call bpf_get_current_pid_tgid#197680
   2: (bf) r8 = r0
; u64 current_pid_tgid = bpf_get_current_pid_tgid();
   3: (7b) *(u64 *)(r10 -8) = r8
   4: (b7) r7 = 0
```

BPFTool to the rescue

```
bpftool map dump name params_array
```

```
[{
        "key": 0,
        "value": {
            "pid": 4504
        }
    }
]
```

Check for tracepoints

```
➜  ~ sudo more
/sys/kernel/tracing/available_events | grep xdp
xdp:mem_return_failed
xdp:mem_connect
xdp:mem_disconnect
xdp:xdp_devmap_xmit
xdp:xdp_cpumap_enqueue
xdp:xdp_cpumap_kthread
xdp:xdp_redirect_map_err
xdp:xdp_redirect_map
xdp:xdp_redirect_err
xdp:xdp_redirect
xdp:xdp_bulk_tx
xdp:xdp_exception
```

Check for tracepoints

```
sudo bpftrace -e 'tracepoint:xdp:* { @cnt[probe] = count(); }'
Attaching 12 probes...
^C

@cnt[tracepoint:xdp:xdp_bulk_tx]: 10
```

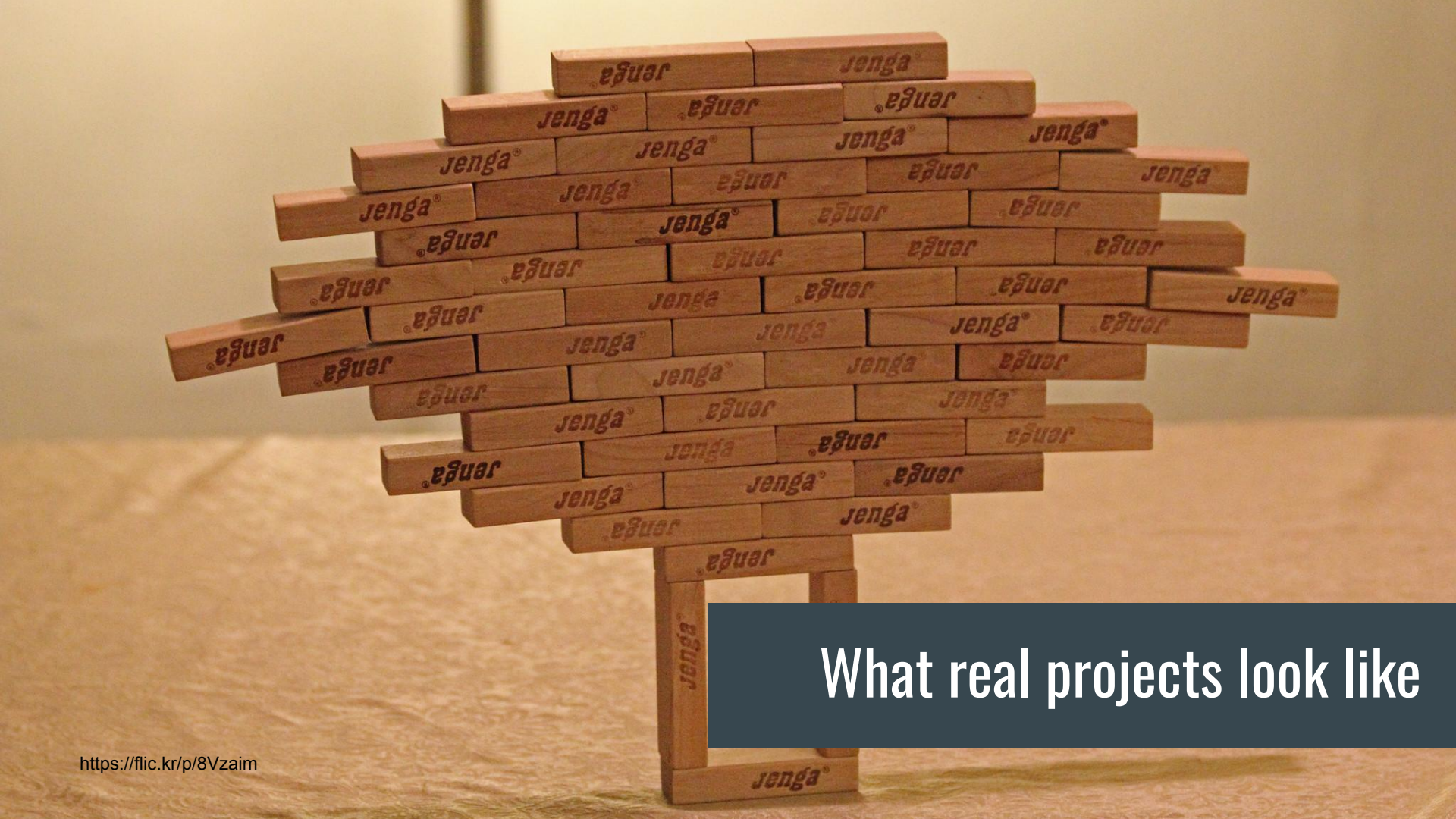
```
bpftrace -e \
 'tracepoint:xdp:xdp_bulk_tx{@redir_errno[-args->err] = count();}'

Attaching 1 probe...
^C

@redir_errno[6]: 2
```

Test in isolation

Don’t be afraid to look into the kernel

Take “inspiration”

https://ebpf.io/applications/

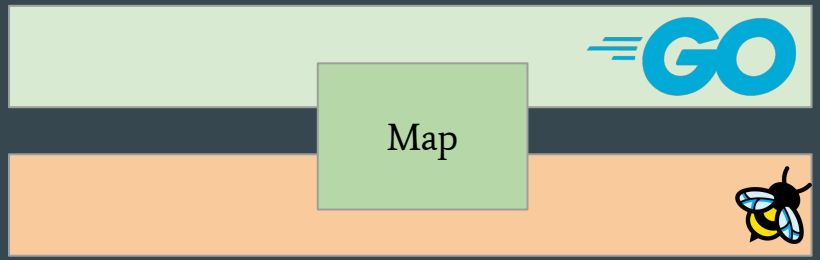


What real projects look like

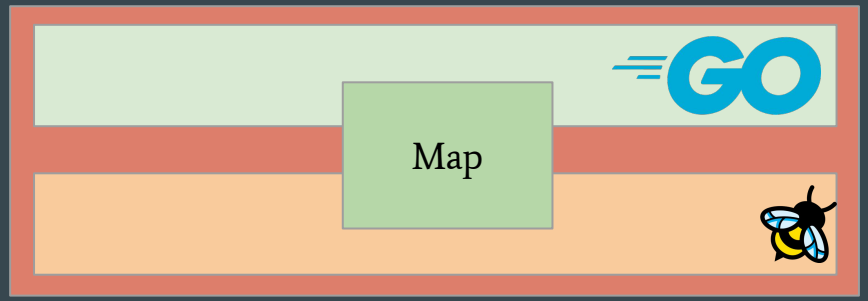
https://flic.kr/p/8Vzaim

A small eBPF layer and a large userspace application

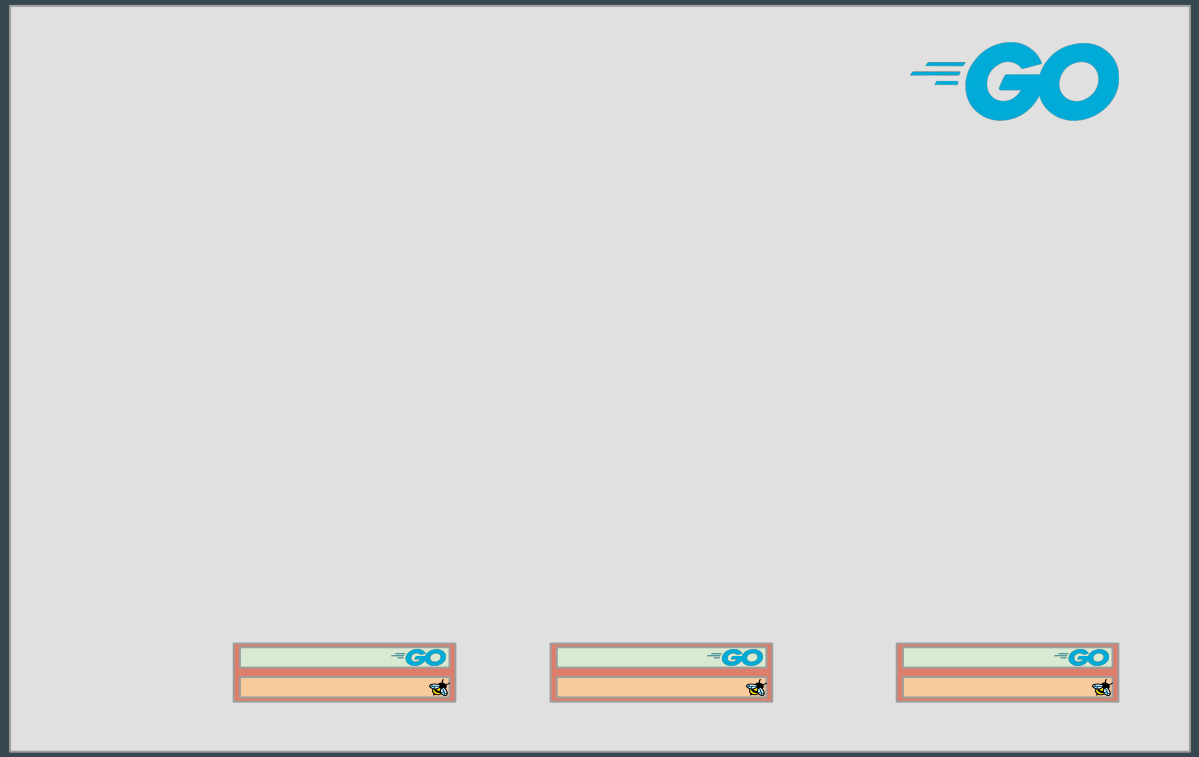
A small eBPF layer and a large userspace application



A small eBPF layer and a large userspace application



A small eBPF layer and a large userspace application



A small eBPF layer and a large userspace application




A small eBPF layer and a large userspace application

A small eBPF layer and a large userspace application

The userspace side is in charge of

- where to attach the program
- where to store the events
- present the data to the user
- what parameters to pass to the program

Wrapping up

Wrapping up

- A kernel side in C, a userspace side in Go
- You need to be familiar with the kernel!
- Maps as an API
- Every program type is different
- It’s powerful

Wrapping up

- A kernel side in C, a userspace side in Go
- You need to be familiar with the kernel!
- Maps as an API
- Every program type is different
- It’s powerful
- It’s difficult to tame

Resources

Resources

- Liz Rice's "Learning eBPF" book
- ebpf.io
- docs.kernel.org/bpf/index.html
- ebpf.io/applications
- github.com/cilium/ebpf/tree/main/examples

Thanks!

Any questions?

@fedepaol

hachyderm.io/@fedepaol

fedepaol@gmail.com

fpaoline@redhat.com

Slides at: speakerdeck.com/fedepaol