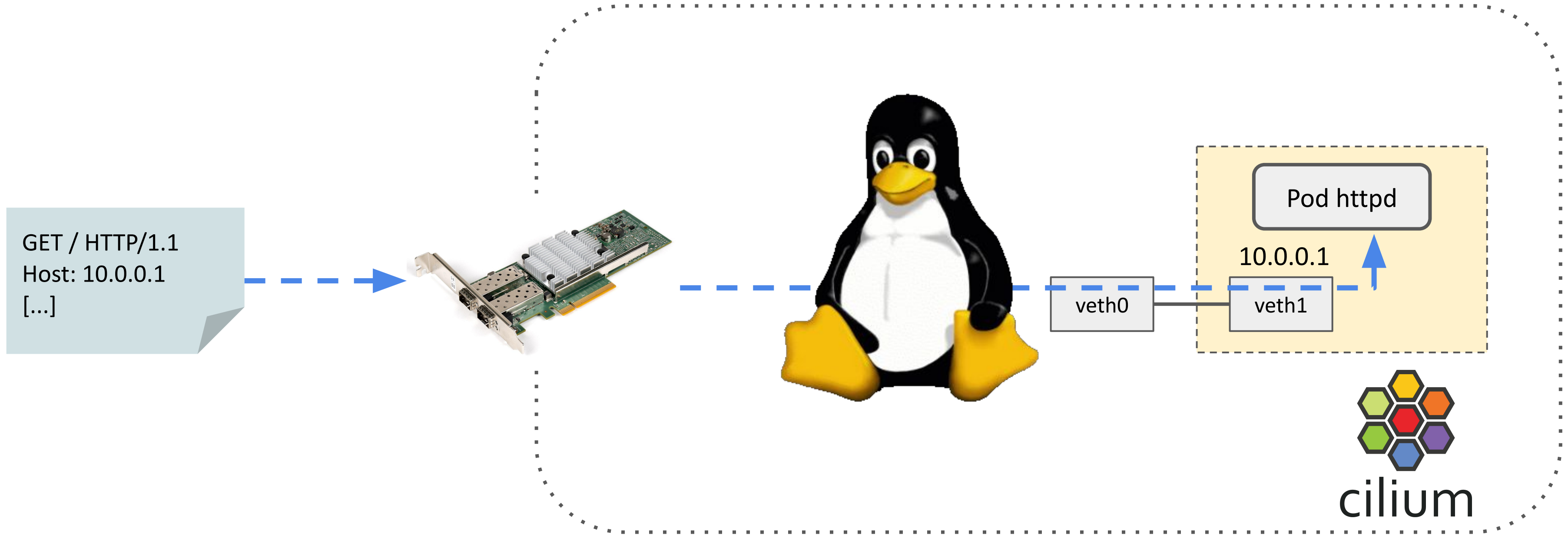
# pwru - Linux kernel and BPF-based networking debugger

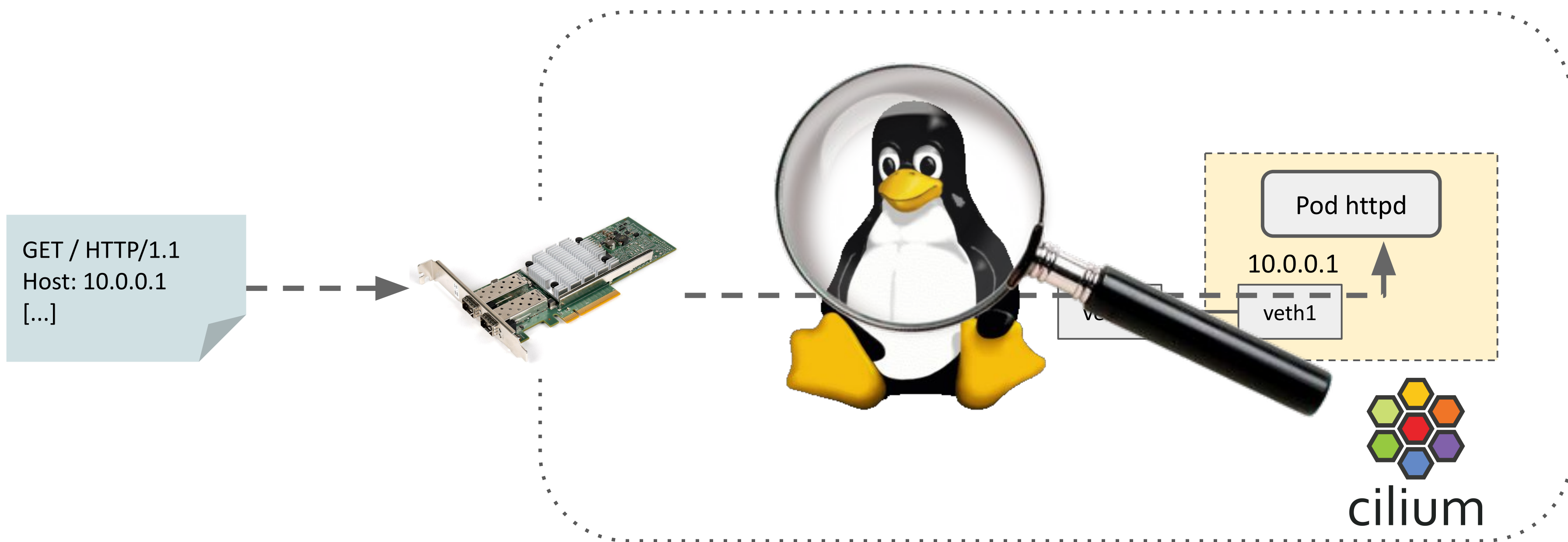
Gray Liang
Martynas Pumputis
ISOVALENT
now part of CISCO

LINUX PLUMBERS CONFERENCE Vienna, Austria / Sept. 18-20, 2024

# Problem statement



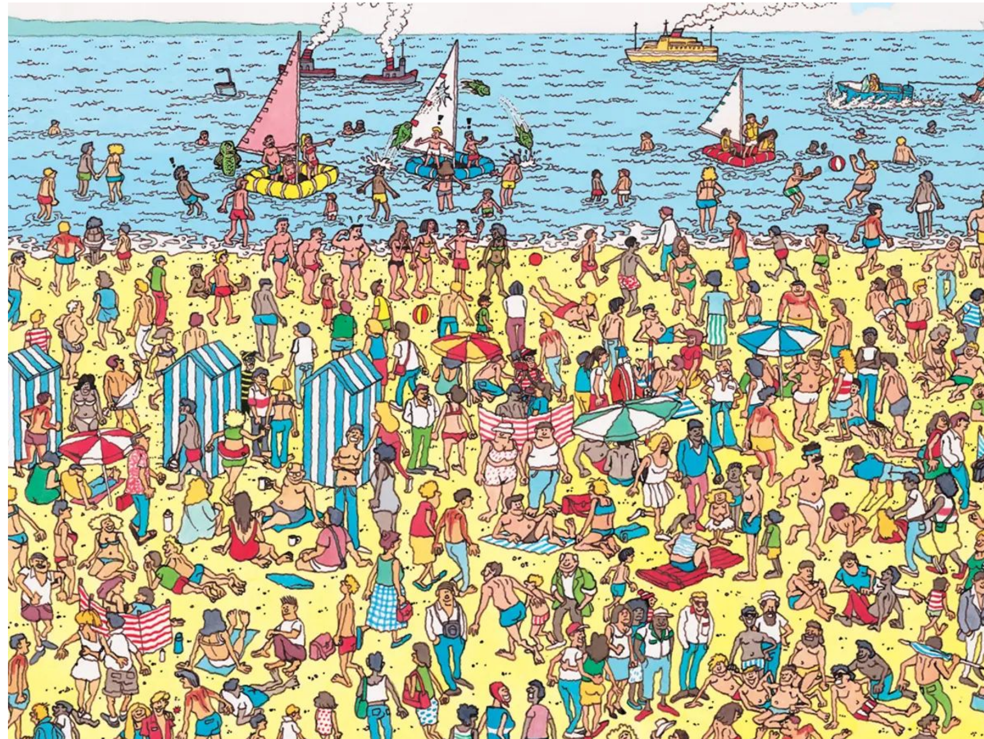
# Problem statement



# Problem statement

https://wiki.linuxfoundation.org/networking/kernel_flow

# Problem statement



Walker Books

# Problem statement

mIRC32

```
[01:12AM] <CustomerInPain> It didn't fix.
[01:30AM] <FrustradedDev> Please try a new version - foo.tar.gz
[01:31AM] <CustomerInPain> What is diff?
[01:31AM] <FrustratedDev>
diff --git a/bpf_foo.c b/bpf_foo.c
index 06cc2e9a60..c2244ff5e2 100644
--- a/bpf_foo.c
+++ b/bpf_foo.c

+              bpf_printk("debug foo\n");
```

# Existing solutions

- tcpdump
  - Too coarse-grained
- bpftrace -e 'kprobe:kfree_skb { @[kstack] = count(); }'
  - Very limited filtering
  - Requires compiler
- ipftrace2
  - Uses skb mark for filtering

github.com/cilium/pwru

(packet, where are you?)

Logo designed by Vadim Shchekoldin (Isovalent), and based on the Go gopher designed by Renee French

# Simplified idea

1. Read /sys/kernel/btf/*
2. Find all functions which accept sk_buff
3. Attach to them BPF filtering progs via kprobes/fentry
4. Print events from user space

**All in a statically linked <10MB binary without any external dependency**

Originally implemented by Yutaro Hayakawa in ipftrace2

```
# pwru --all-kmods --output-tuple --output-meta 'dst host 1.1.1.1 and tcp and port 80'

2024/09/19 11:27:54 Attaching kprobes (via kprobe)...
2508 / 2508 [-------------------------------------------------------------------------------------------] 100.00% 222 p/s
2024/09/19 11:28:06 Attached (ignored 157)
2024/09/19 11:28:06 Listening for events..
SKB                CPU PROCESS          NETNS      MARK/x  IFACE  PROTO  MTU   LEN  TUPLE FUNC
0xffff8d5ddda44ce8 4   ~/bin/curl:60597 4026531840 0         0    0x0000 1500  60   10.136.3.101:45706->1.1.1.1:80(tcp) ip_local_out
0xffff8d5ddda44ce8 4   ~/bin/curl:60597 4026531840 0         0    0x0000 1500  60   10.136.3.101:45706->1.1.1.1:80(tcp) __ip_local_out
0xffff8d5ddda44ce8 4   ~/bin/curl:60597 4026531840 0         0    0x0800 1500  60   10.136.3.101:45706->1.1.1.1:80(tcp) nf_hook_slow
0xffff8d5ddda44ce8 4   ~/bin/curl:60597 4026531840 0         0    0x0800 1500  60   10.136.3.101:45706->1.1.1.1:80(tcp) ipv4_conntrack_defrag[nf_defrag_ipv4]
0xffff8d5ddda44ce8 4   ~/bin/curl:60597 4026531840 0         0    0x0800 1500  60   10.136.3.101:45706->1.1.1.1:80(tcp) ipv4_conntrack_local[nf_conntrack]
0xffff8d5ddda44ce8 4   ~/bin/curl:60597 4026531840 0         0    0x0800 1500  60   10.136.3.101:45706->1.1.1.1:80(tcp) nf_conntrack_in[nf_conntrack]
0xffff8d5ddda44ce8 4   ~/bin/curl:60597 4026531840 0         0    0x0800 1500  60   10.136.3.101:45706->1.1.1.1:80(tcp) nf_conntrack_tcp_packet[nf_conntrack]
0xffff8d5ddda44ce8 4   ~/bin/curl:60597 4026531840 0         0    0x0800 1500  60   10.136.3.101:45706->1.1.1.1:80(tcp) nf_nat_ipv4_local_fn[nf_nat]
0xffff8d5ddda44ce8 4   ~/bin/curl:60597 4026531840 0         0    0x0800 1500  60   10.136.3.101:45706->1.1.1.1:80(tcp) nf_nat_inet_fn[nf_nat]
0xffff8d5ddda44ce8 4   ~/bin/curl:60597 4026531840 0         0    0x0800 1500  60   10.136.3.101:45706->1.1.1.1:80(tcp) ipt_do_table[ip_tables]
0xffff8d5ddda44ce8 4   ~/bin/curl:60597 4026531840 0         0    0x0800 1500  60   10.136.3.101:45706->1.1.1.1:80(tcp) ipt_do_table[ip_tables]
0xffff8d5ddda44ce8 4   ~/bin/curl:60597 4026531840 0         0    0x0800 1500  60   10.136.3.101:45706->1.1.1.1:80(tcp) kfree_skb_reason(SKB_DROP_REASON_NETFILTER_DROP)
0xffff8d5ddda44ce8 4   ~/bin/curl:60597 4026531840 0         0    0x0800 1500  60   10.136.3.101:45706->1.1.1.1:80(tcp) skb_release_head_state
0xffff8d5ddda44ce8 4   ~/bin/curl:60597 4026531840 0         0    0x0800 0     60   10.136.3.101:45706->1.1.1.1:80(tcp) tcp_wfree
0xffff8d5ddda44ce8 4   ~/bin/curl:60597 4026531840 0         0    0x0800 0     60   10.136.3.101:45706->1.1.1.1:80(tcp) skb_release_data
0xffff8d5ddda44ce8 4   ~/bin/curl:60597 4026531840 0         0    0x0800 0     60   10.136.3.101:45706->1.1.1.1:80(tcp) kfree_skbmem
```

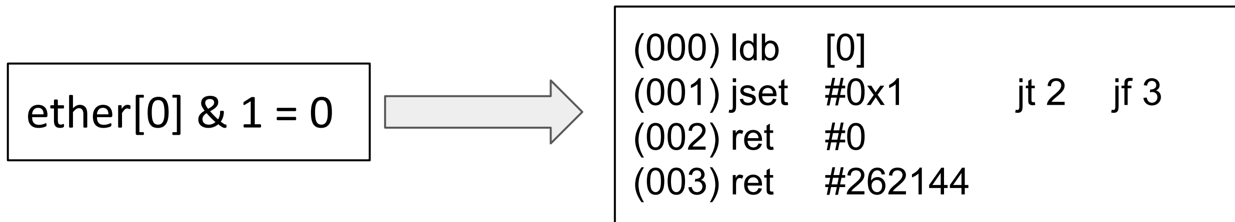
# Running

1. Standalone binary
2. docker run cilium/pwru … –output-meta "host 1.1.1.1"
3. Kubernetes (e.g., on all nodes)
4. Github action

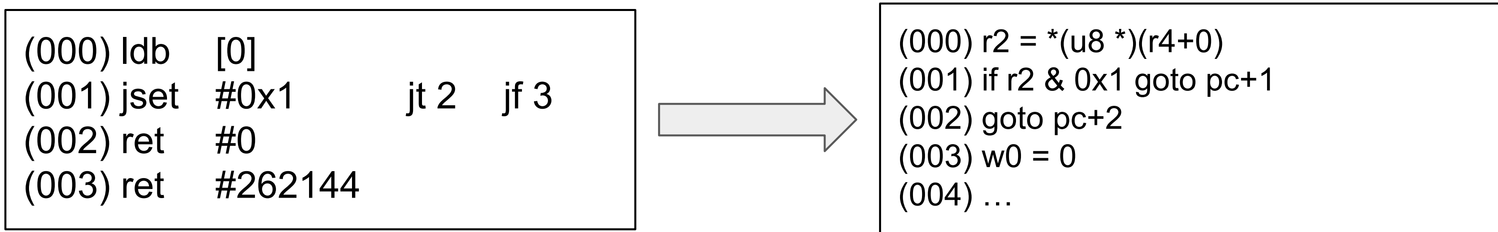
Originally implemented by Yutaro Hayakawa in ipftrace2

# tcpdump-like filters

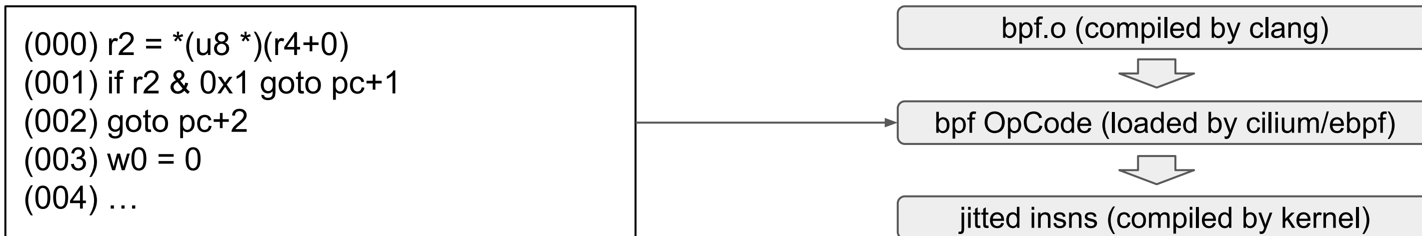
1. Compile filter expr into cbpf using libpcap:pcap_compile()




2. Convert cbpf to ebpf using cloudflare/cbpfc




3. Inject generated ebpf bytecode into PWRU



# 1. Compile filter expr into cbpf using libpcap:pcap_compile()

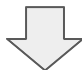
```
/*
#cgo CFLAGS: -I${SRCDIR}/../../libpcap
#cgo LDFLAGS: -L${SRCDIR}/../../libpcap -lpcap -static
#include <stdlib.h>
#include <pcap.h>
*/
import "C"

{
        cexpr := C.CString(expr)
        bpfProg := C.struct_bpf_program{}
        C.pcap_compile(pcap, &bpfProg, cexpr, 1, C.PCAP_NETMASK_UNKNOWN)

}
```

```
cd libpcap-libpcap-1.10.4/
./configure --enable-dbus=no
make
```

## 2. Convert cbpf to ebpf using cloudflare/cbpfc

```
func ToEBPF(filter []bpf.Instruction, opts EBPFOpts) (asm.Instructions, error)
```

ether[0] & 1 = 0



```
(000) ldb    [0]
(001) jset   #0x1        jt 2    jf 3
(002) ret    #0
(003) ret    #262144
```

r0 = *(u8 *)(r4 +0) // r4 is skb->data

**R4 invalid mem access 'scalar'**

```
- r1 = r10        // r10 is stack top
- r1 += -8        // r1 = r10-8
- r2 = 1          // r2 = sizeof(u8)
- r3 = r4         // r4 is start of packet data, aka L2 header
- r3 += 0         // r3 = r4+0
- call bpf_probe_read_kernel  // *(r10-8) = *(u8 *)(r4+0)
- r0 = *(u8 *)(r10 -8)  // r0 = *(r10-8)
```

```
- *(u64 *)(r10 -96) = r1     // store r1-r3 to stack
- *(u64 *)(r10 -104) = r2
- *(u64 *)(r10 -112) = r3
[...]
- r1 = *(u64 *)(r10 -96)     // restore r1-r3 from stack
- r2 = *(u64 *)(r10 -104)
- r3 = *(u64 *)(r10 -112)
```

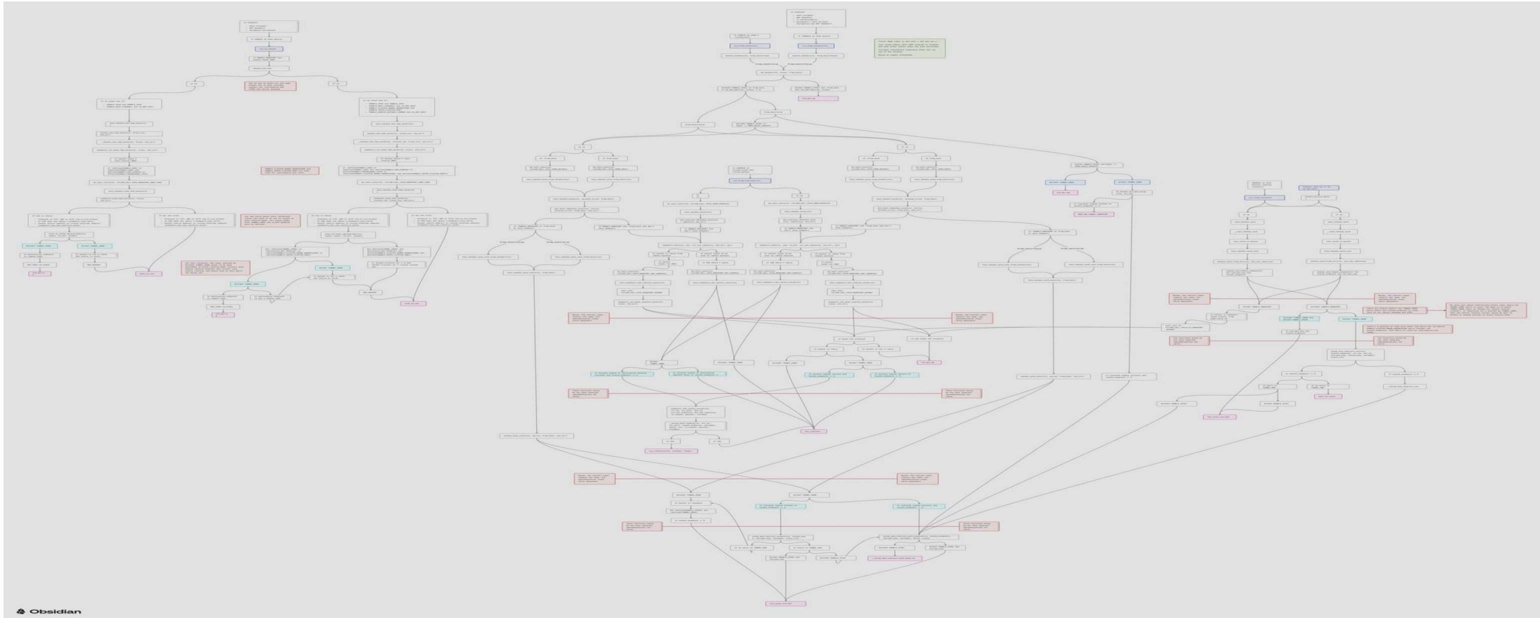
# 3. Inject generated ebpf bytecode into PWRU

```
static __noinline bool
filter_pcap_ebpf_l3(void *_skb, void *__skb, void *___skb, void *data, void* data_end)
{
        return data != data_end && _skb == __skb && __skb == ___skb;
}

static __always_inline bool
filter_pcap_l3(struct sk_buff *skb)
{
        void *skb_head = BPF_CORE_READ(skb, head);
        void *data = skb_head + BPF_CORE_READ(skb, network_header);
        void *data_end = skb_head + BPF_CORE_READ(skb, tail);
        return filter_pcap_ebpf_l3((void *)skb, (void *)skb, (void *)skb, data, data_end);
}
```

1. Generated ebpf opcode must know skb->data and skb->data_end are at r4 and r5
2. At least 4 registers are required to be available: r0, r1, r2, r3
3. Scan the PWRU bpf opcode, find the symbol **fitler_pcap_ebpf_l3**, replace it with generated bpf opcode

# BPF programs tracing

```
0xffff9f1388b7cb00 ~baa3136fe29c:13 10.244.3.106:8080->10.244.2.135:19233(tcp) __netif_rx
0xffff9f1388b7cb00 ~baa3136fe29c:13 10.244.3.106:8080->10.244.2.135:19233(tcp) tcf_classify
0xffff9f1388b7cb00 ~baa3136fe29c:13 172.21.0.2:4000->10.244.2.135:19233(tcp)   __bpf_redirect
0xffff9f1388b7cb00     eth0:87      172.21.0.2:4000->10.244.2.135:19233(tcp)   __dev_queue_xmit
0xffff9f1388b7cb00     eth0:87      172.21.0.2:4000->10.244.2.135:19233(tcp)   tcf_classify
0xffff9f1388b7cb00     eth0:87      172.21.0.2:4000->10.244.2.135:19233(tcp)   __bpf_redirect
0xffff9f1388b7cb00   cilium_wg0:2   172.21.0.2:4000->10.244.2.135:19233(tcp)   __dev_queue_xmit
```



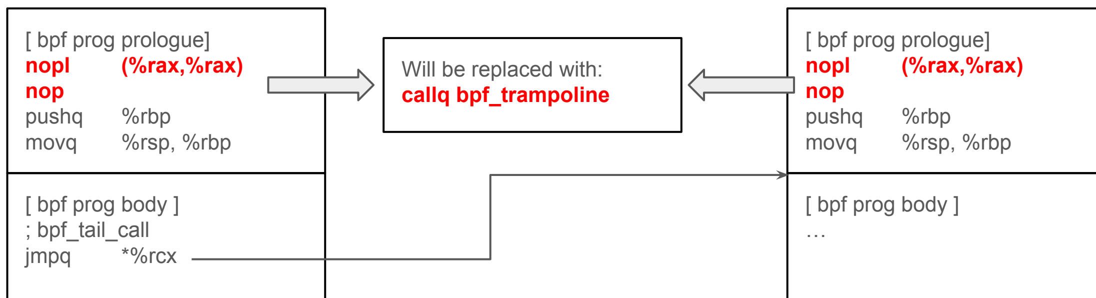

## Which part of code did SNAT for this traffic?

Graph created by Ryan Drew

## 1. Kprobe on bpf prog ❌

```
// check_kprobe_address_safe
	/* Ensure it is not in reserved area nor out of text */
	if (!(core_kernel_text((unsigned long) p->addr) ||
	    is_module_text_address((unsigned long) p->addr)) ||
	    in_gate_area_no_mm((unsigned long) p->addr) ||
	    within_kprobe_blacklist((unsigned long) p->addr) ||
	    jump_label_text_reserved(p->addr, p->addr) ||
	    static_call_text_reserved(p->addr, p->addr) ||
	    find_bug((unsigned long)p->addr) ||
	    is_cfi_preamble_symbol((unsigned long)p->addr)) {
		ret = -EINVAL;
		goto out;
	}
```

## 2. Fentry on tailcall prog ❌



```
[ bpf prog prologue]
nopl    (%rax,%rax)
nop
pushq   %rbp
movq    %rsp, %rbp
```

```
[ bpf prog body ]
…
; tmp =
map_lookup_elem(&IPV4_FRAG_DATAGRAMS_MAP,
frag_id);
callq   0xffffffffcc9e7bc8 (bpf_map_lookup_elem)

; return ctx_load_bytes(ctx, off, ports, 2 * sizeof(__be16));
callq   0xffffffffcd50aa98 (bpf_skb_load_bytes)

; ret = fib_lookup(ctx, &fib_params->l, sizeof(fib_params->l),
0);
callq   0xffffffffcd510c98 (bpf_skb_fib_lookup)
…
```

kprobe

[0] PWRU scans /proc/kcore to collect all existing bpf helper functions (with suffix “[bpf]”)

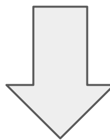
[1] Functions on the same calling chain must have the same stackid.

[2] Skbs are stored by stackid before entering bpf prog.

[3] Caller PC can be converted to symbol name as per /proc/kallsyms; caller symbols give bpf prog names, even for tailcall progs.

```
u64 stackid = unwind_get_stackid();[1]
struct sk_buff *skb = bpf_map_lookup_elem(&stackid_skb, &stackid);[2]
struct event e;
e.caller_pc = *(u64 *)PT_REGS_SP(ctx);[3]
collect_and_output_event(skb, e);
```

```
0xffff9f1388b7cb00 ~baa3136fe29c:13 10.244.3.106:8080->10.244.2.135:19233(tcp) __netif_rx
0xffff9f1388b7cb00 ~baa3136fe29c:13 10.244.3.106:8080->10.244.2.135:19233(tcp) tcf_classify
0xffff9f1388b7cb00 ~baa3136fe29c:13 172.21.0.2:4000->10.244.2.135:19233(tcp)   __bpf_redirect
0xffff9f1388b7cb00       eth0:87     172.21.0.2:4000->10.244.2.135:19233(tcp)   __dev_queue_xmit
0xffff9f1388b7cb00       eth0:87     172.21.0.2:4000->10.244.2.135:19233(tcp)   tcf_classify
0xffff9f1388b7cb00       eth0:87     172.21.0.2:4000->10.244.2.135:19233(tcp)   __bpf_redirect
0xffff9f1388b7cb00    cilium_wg0:2   172.21.0.2:4000->10.244.2.135:19233(tcp)   __dev_queue_xmit
```



```
0xffff9f1388b7cb00 10.244.3.106:8080->10.244.2.135:19233(tcp) __netif_rx          veth_xmit[veth]
0xffff9f1388b7cb00 10.244.3.106:8080->10.244.2.135:19233(tcp) tcf_classify        sch_handle_ingress.constprop.0
0xffff9f1388b7cb00 10.244.3.106:8080->10.244.2.135:19233(tcp) bpf_skb_event_output bpf_prog_fa4b302e8d3ee3ba_cil_from_container[bpf]
[...]
0xffff9f1388b7cb00 10.244.3.106:8080->10.244.2.135:19233(tcp) bpf_skb_pull_data   bpf_prog_856b34372087aa48_tail_handle_ipv4[bpf]
[...]
0xffff9f1388b7cb00 10.244.3.106:8080->10.244.2.135:19233(tcp) trie_lookup_elem    bpf_prog_9968639175839371_tail_handle_ipv4_cont[bpf]
[...]
0xffff9f1388b7cb00 10.244.3.106:8080->10.244.2.135:19233(tcp) bpf_skb_load_bytes  bpf_prog_8716dfad2d583c4e_tail_nodeport_rev_dnat_ingress_ipv4[bpf]
[...]
0xffff9f1388b7cb00 172.21.0.2:8080->10.244.2.135:19233(tcp)   bpf_skb_store_bytes bpf_prog_8716dfad2d583c4e_tail_nodeport_rev_dnat_ingress_ipv4[bpf]
0xffff9f1388b7cb00 172.21.0.2:4000->10.244.2.135:19233(tcp)   bpf_l4_csum_replace bpf_prog_8716dfad2d583c4e_tail_nodeport_rev_dnat_ingress_ipv4[bpf]
[...]
0xffff9f1388b7cb00 172.21.0.2:4000->10.244.2.135:19233(tcp)   bpf_redirect        bpf_prog_8716dfad2d583c4e_tail_nodeport_rev_dnat_ingress_ipv4[bpf]
[...]
```

# Future

1. Arguments collecting and parsing, especially for map functions.

```
bpf_snprintf_btf() + bpf_map.btf_key_type_id + bpf_map.btf_value_type_id
```

2. Source code output.

```
int tail_handle_nat_fwd_ipv4(struct __sk_buff * ctx):
bpf_prog_6b7389d73009eda0_tail_handle_nat_fwd_ipv4:
; int tail_handle_nat_fwd_ipv4(struct __ctx_buff *ctx)

; tmp = map_lookup_elem(&IPV4_FRAG_DATAGRAMS_MAP, frag_id);
 126:   movabsq $-107340655100928, %rdi
 130:   movq    -208(%rbp), %rax
 137:   callq   0xffffffffcc9e7bc8

; return ctx_load_bytes(ctx, off, ports, 2 * sizeof(__be16));
 17d:   movq    %r8, %rdi
 180:   movl    %ebx, %esi
 182:   movq    %r14, %rdx
 185:   movl    $4, %ecx
 18a:   movq    -208(%rbp), %rax
 191:   callq   0xffffffffcd50aa98

; ret = fib_lookup(ctx, &fib_params->l, sizeof(fib_params->l), 0);
 3f7:   movq    -144(%rbp), %rdi
 3fe:   movl    $64, %edx
 403:   xorl    %ecx, %ecx
 405:   movq    -208(%rbp), %rax
 40c:   callq   0xffffffffcd510c98
```

# XDP tracing & SKB tracing

# XDP: --filter-trace-xdp

```
pwru --filter-trace-xdp --filter-trace-tc --output-meta --output-tuple icmp
2024/03/04 14:26:47 Listening for events..
               SKB         FUNC
0xffffbb8cc062cc28    dummy(xdp) iface=2(ens33) 192.168.241.1:0->192.168.241.128:0(icmp)
0xffff95ed4596aa00     dummy(tc) iface=2(ens33) 192.168.241.1:0->192.168.241.128:0(icmp)
0xffffbb8cc062cc28    dummy(xdp) iface=2(ens33) 192.168.241.1:0->192.168.241.128:0(icmp)
0xffff95ed4596ba00     dummy(tc) iface=2(ens33) 192.168.241.1:0->192.168.241.128:0(icmp)
0xffffbb8cc062cc28    dummy(xdp) iface=2(ens33) 192.168.241.1:0->192.168.241.128:0(icmp)
0xffff95ed4596a700     dummy(tc) iface=2(ens33) 192.168.241.1:0->192.168.241.128:0(icmp)
0xffffbb8cc062cc28    dummy(xdp) iface=2(ens33) 192.168.241.1:0->192.168.241.128:0(icmp)
0xffff95ed4596b500     dummy(tc) iface=2(ens33) 192.168.241.1:0->192.168.241.128:0(icmp)
0xffffbb8cc062cc28    dummy(xdp) iface=2(ens33) 192.168.241.1:0->192.168.241.128:0(icmp)
0xffff95ed4596b400     dummy(tc) iface=2(ens33) 192.168.241.1:0->192.168.241.128:0(icmp)
2024/03/04 14:26:52 Printed 10 events, exiting program..
```

SKB:

1. `--filter-track-skb`

Track skbs by pointer addresses, useful when NAT / encapsulation / encryption happens.

2. `--filter-non-skb-funcs $FUNCS`

Track skbs by stackid, so non-skb kernel functions can also be probed.

An example to trace xfrm state lookup functions:

```
pwru --filter-non-skb-funcs
xfrm_state_look_at,xfrm_state_lookup,xfrm_state_lookup_byaddr,xfrm_state_lookup_bysp
i
```

3. Some cases where SKBs can be rebuilt

```
// drivers/net/veth.c
static int veth_convert_skb_to_xdp_buff(struct veth_rq *rq,
                                        struct xdp_buff *xdp,
                                        struct sk_buff **pskb)
{
        struct sk_buff *skb = *pskb;
[...]
                nskb = build_skb(page_address(page), PAGE_SIZE);
[...]
                skb_copy_header(nskb, skb);
[...]
                consume_skb(skb);
                skb = nskb;
[...]
}
```

# Q/A

Thank you!

**https://github.com/cilium/pwru**