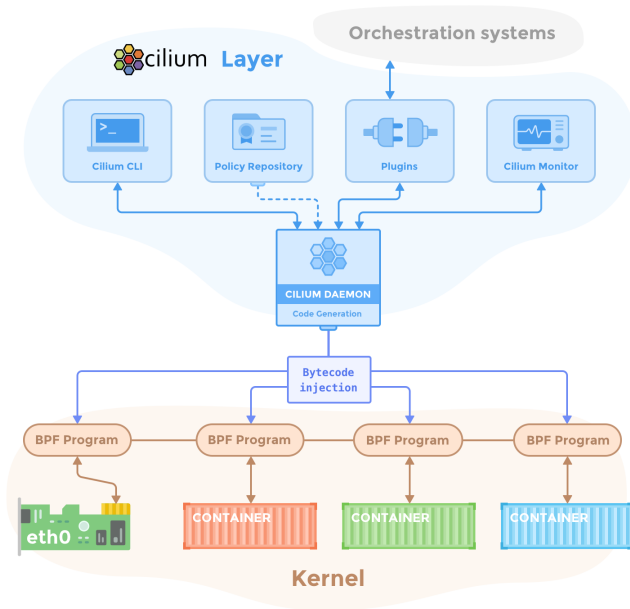


# Building socket-aware BPF programs

Joe Stringer

Cilium.io

Linux Plumbers 2018, Vancouver, BC



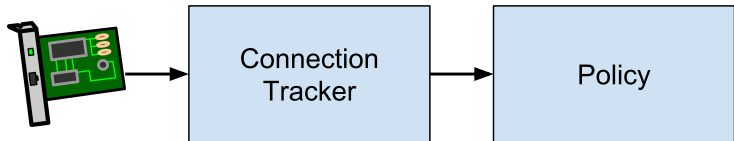
# Network Policy

"Endpoint A can talk to endpoint B"



"Endpoint B can reply to endpoint A"

# How have we built these before?



# Let's do this with BPF

- Attach BPF to packet hook ✓
- “Connection Tracking” BPF map ✓
  - Key by 5-tuple
  - Associate counters, NAT state, etc.
  - Handle tuple flipping
- “Policy” map ✓
- Deploy! ✓

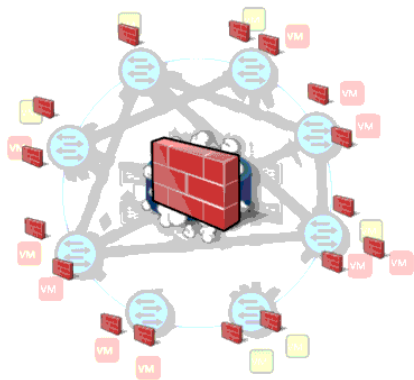
# Let's do this with BPF

- Attach BPF to packet hook ✓
- “Connection Tracking” BPF map ✓
  - Key by 5-tuple
  - Associate counters, NAT state, etc.
  - Handle tuple flipping
- “Policy” map ✓
- Deploy! ✗
  - `nf_conntrack`: table full, dropping packet
  - Hmm, how big should this map be again?
  - How do we clean this up...

# Why model it like this?

- Firewalls might not be co-located with the workload
- Firewalls should drop packets as quickly as possible
- Network stacks may be delicate flowers
- Solution? Build up state on-demand while processing packets

# Recent trends



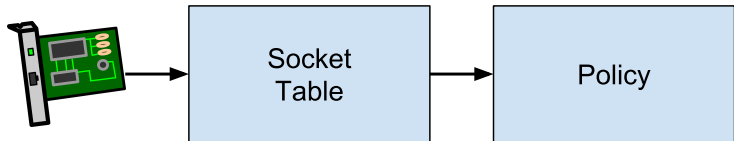


If we're co-located with the sockets . . .

. . . why build our own connection table?



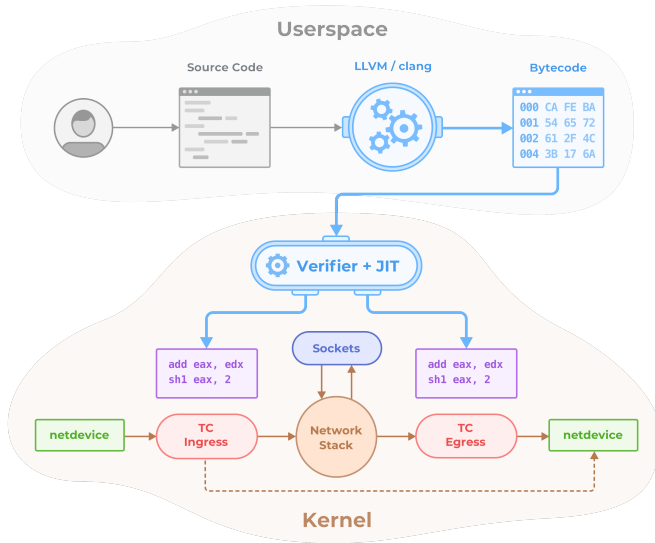
# Socket table as a connection tracker



# Socket safety

- Sockets are reference-counted internally
  - Some memory-management under RCU rules
- `BPF_PROG_TYPE_CGROUP_SOCK`
  - Access safety via reference held across BPF execution
  - Bounds safety provided via bounds access checker
- Packet hooks may execute before associated socket is known
  - Need to handle reference counting

# Extending the BPF verifier



# BPF verifier: Recap

- At load time, loop over all instructions
  - Validate pointer access
  - Ensure no loops
  - ...
- Access memory out of bounds? ✗
- Loops forever? ✗
- Everything safe? ✓

# Socket reference counting

## Implicit

```
struct bpf_sock *sk;

sk = bpf_sk_lookup(...);
if (sk) {
    ...
}
/* Kernel will free 'sk' */
```

## Explicit (mainline)

```
struct bpf_sock *sk;

sk = bpf_sk_lookup(...);
if (sk) {
    ...
    bpf_sk_release(sk);
}
```

# Reference counting in the BPF verifier

- 1 Resource acquisition
- 2 Execution paths while resource is held
- 3 Resource release



# Reference Acquisition

- Generate an identifier
- Store the identifier in the verifier state
- Associate the register with the identifier

# Reference misuse

- Mangle and release
- `bpf_tail_call()`
- `BPF_LD_ABS`, `BPF_LD_IND`

# Reference release

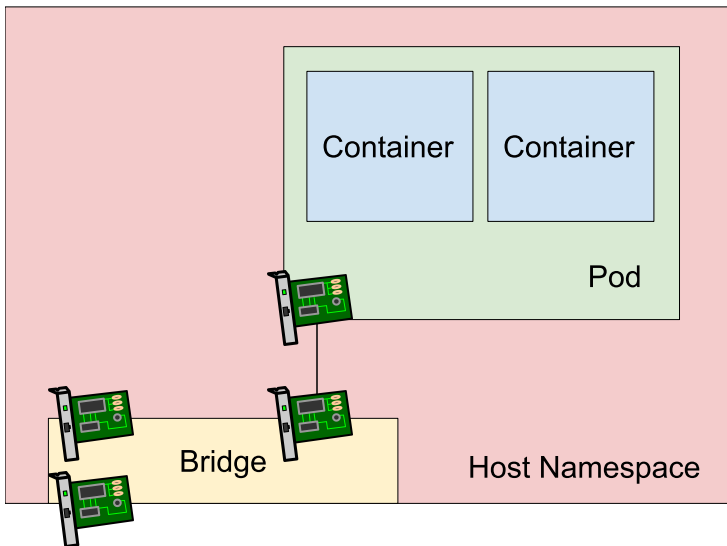
- Validation of pointers
- Remove identifier reference from state
- Unassociate register identifier associations

# Extending the BPF API

# Simplest form

- `struct bpf_sock *bpf_sk_lookup(struct sk_buff *);`
- `void bpf_sk_release(struct bpf_sock *);`

# Namespaces



# Arbitrary socket lookup

- Use any tuple for lookup
- Ease API across clsact, XDP
- Simplify packet mangle and lookup

# Extensibility

- Allow influencing lookup behaviour
  - `SO_REUSEPORT`
- Determine socket type support at load time
  - Socket type supported? Load the program
  - Not supported? Reject the program



# Optimizations

- Avoid reference counting
- Allow lookup using direct packet pointers

# Socket lookup API

```
struct bpf_sock *  
bpf_sk_lookup_tcp(void *ctx, struct bpf_sock_tuple *tuple,  
                  u32 tuple_size, u32 netns, u64 flags);  
  
struct bpf_sock *  
bpf_sk_lookup_udp(void *ctx, struct bpf_sock_tuple *tuple,  
                  u32 tuple_size, u32 netns, u64 flags);  
  
void bpf_sk_release(struct bpf_sock *sk);
```

# Socket lookup structures

```
struct bpf_sock_tuple {
    union {
        struct {
            __be32 saddr;
            __be32 daddr;
            __be16 sport;
            __be16 dport;
        } ipv4;
        struct {
            __be32 saddr[4];
            __be32 daddr[4];
            __be16 sport;
            __be16 dport;
        } ipv6;
    };
};
```

# Socket structure

```
struct bpf_sock {  
    __u32 bound_dev_if;  
    __u32 family;  
    __u32 type;  
    __u32 protocol;  
    __u32 mark;  
    __u32 priority;  
    __u32 src_ip4;           /* NBO */  
    __u32 src_ip6[4];       /* NBO */  
    __u32 src_port;         /* NBO */  
};
```

# Epilogue

## Use case: Network devices

- Socket lookup from XDP
- Management traffic? Send up the stack
- Other traffic? Forward, route, load-balance

# Future work

- More socket attribute access
- Associate metadata with sockets
- More uses for reference tracking

Thank you

Joe Stringer  
joe@cilium.io