Building socket-aware BPF programs

Joe Stringer

Cilium.io

Linux Plumbers 2018, Vancouver, BC

A (selective) history of traffic filtering

1992	The BSD Packet Filter [McCanne, Jacobson]
1994	Stateful firewalls become commercially available
...	
1999	Implementing a Distributed Firewall [Ioannidis *et al.*]
	Linux TC (upstream 2000), Netfilter (upstream 2001)
2002	BSD/OS IPFW [Lidl *et al.*]
2005	A model of stateful firewalls [Gouda, Liu]
2008	Linux Network Namespaces
...	
2014	Linux gets Extended BPF

Berkeley Packet Filter

- Initial use case: network monitoring
  - Open socket
  - Attach filter
- TCPdump becomes the de facto monitoring tool
- BPF remains stateless, and just for socket filtering

Network Policy

"Endpoint A can talk to endpoint B"

$$\Longrightarrow$$

"Endpoint B can reply to endpoint A"

Putting the “state” in “stateful firewalls”

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

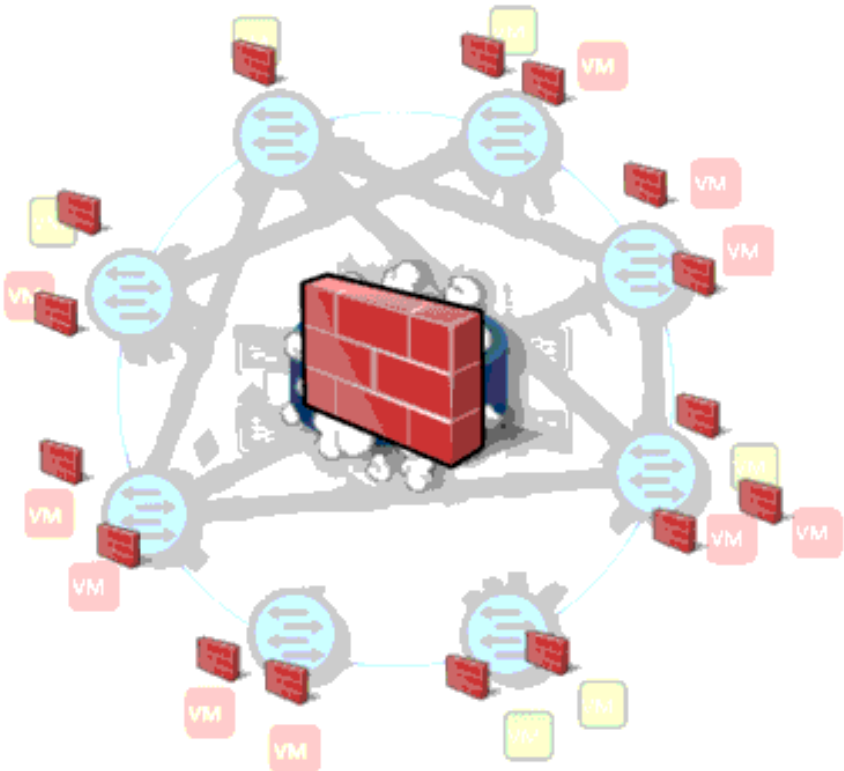
Let's do this with BPF

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

Recent trends: Distributed filtering



http://www.routetocloud.com/2015/04/nsx-distributed-firewall-deep-dive/

https://www.flickr.com/photos/lukeprice88/9703431992

Socket table as a connection tracker

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

. . . why build our own connection table?

Socket-aware BPF programs

- Locality
- Safety
- Namespacing
- Socket identity
- Metadata

BPF verifier: Recap

- At load time, loop over all instructions
  - Validate pointer access
  - Build up program states
  - ...
- Access memory out of bounds? ✗
- Ends in a "bad" state? ✗
- Everything safe? ✓

Socket safety

- Sockets are reference-counted internally
  - Some memory-management under RCU rules
- Earlier this year: `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

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

- Resource acquisition
- Execution paths while resource is held
- 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

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 */
};
```

Conclusion

- foo
  - bar