Combining kTLS and BPF for Introspection and Policy Enforcement

Daniel Borkmann, John Fastabend
Cilium.io

Linux Plumbers 2018, Vancouver, Nov 14, 2018

Distributed Microservices and APIs

- Shift from monolithic legacy applications to distributed microservices
  - Microservice: service that does one thing well, communicates over network, built and managed independently
- Key motivation for enterprises: speed, scale, agility
  - Competitive advantage to react faster to market
- Lowest common denominator to communicate: API
  - Typically: REST API via HTTP
  - Outsourcing: API economy around microservices[1]

---

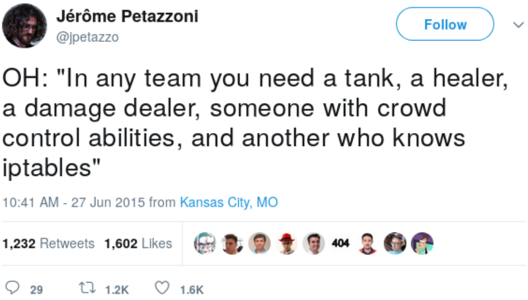
[1] REST API examples:
`https://stripe.com/docs/api/`,
`https://www.twilio.com/docs/usage/api/`,
`https://www.zuora.com/developer/api-reference/`

Kubernetes and Networking

- Microservice *itself* becomes easier to develop, debug, deploy
  - But: higher operational complexity of overall architecture
- Kubernetes $\rightarrow$ platform for automating deployment, scaling, and operations of application containers across clusters of hosts
  - At the heart of all this, obviously: Linux kernel
  - Pods as plumbing around cgroups and namespaces holding one or more containers (e.g. Docker) that share common policy
  - TCP/IP stack and socket API $\rightarrow$ communication bus for microservices

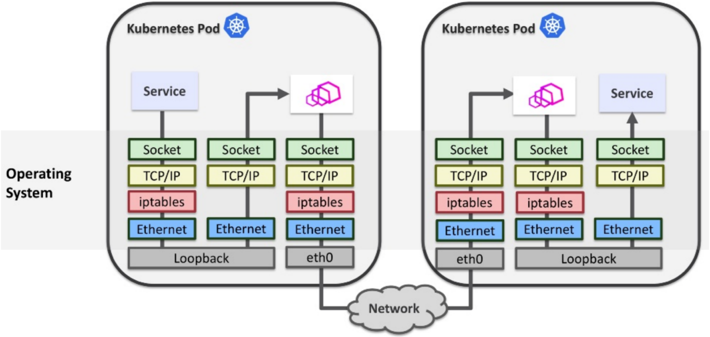
Kubernetes and Networking

- Default policy enforcement in terms of networking: iptables
  - Available also on old kernels, more or less well understood



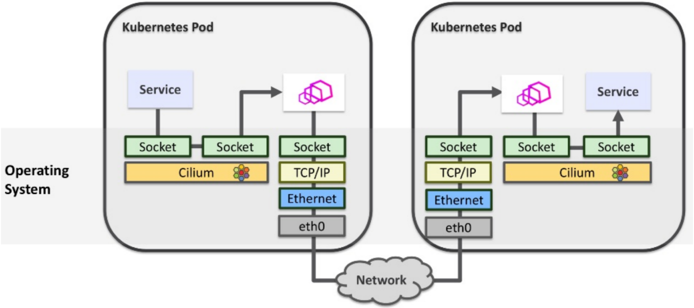
Kubernetes and Networking

- Problem: ports become meaningless in microservices API world
- Consequence: shift to L7 proxies to manage API communication
  - Injected as transparent sidecar into every Pod
  - Packet cost in times of KPTI and Retpoline mitigations even worse



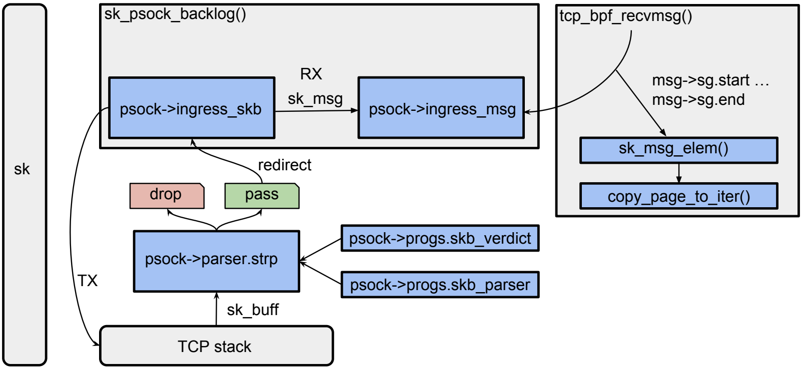
Kubernetes and Networking

- Sidecar proxies like Envoy provide many additional L7 features
  - Health checks, service discovery, load balancing, mutual TLS, etc
- Envoy can be augmented with BPF support to improve fast-path
  - Policy enforcement, introspection and redirection based on BPF



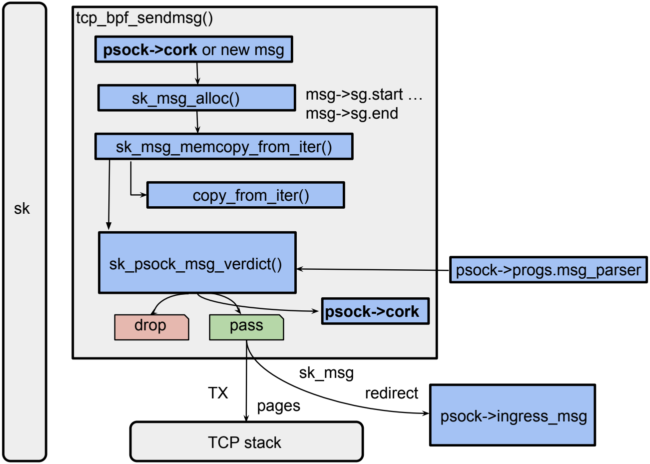
Enter: BPF at Socket Layer

- Implementation through special BPF map called sock_map
- Attached sockets get socket callbacks replaced and psock attached
- Ingress data path:



Enter: BPF at Socket Layer

- Egress data path:



kTLS and ULP Basics

- Handshake in user space, remaining work transferred into kernel
    - Zero-copy, avoiding bounce buffer in user space
- Modes: sw-based RX/TX via crypto layer, hw-based RX/TX via NIC
- TLS 1.2, AES, 128 bit key size
- Transparent to applications via ssl library integration
- Soon: TLS 1.3, support != 128 bit key sizes

kTLS and ULP Basics

- ULP (upper layer protocol) provides generic selector for TLS or others
- User space API:

```
struct tls12_crypto_info_aes_gcm_128 tls_tx = {
  .info = {
    .version     = TLS_1_2_VERSION,
    .cipher_type = TLS_CIPHER_AES_GCM_128,
  },
  .key = [...], [...]
}, tls_rx = {
  [...]
};
setsockopt(fd, SOL_TCP, TCP_ULP, "tls", sizeof("tls"));
setsockopt(fd, SOL_TLS, TLS_TX, &tls_tx, sizeof(tls_tx));
setsockopt(fd, SOL_TLS, TLS_RX, &tls_rx, sizeof(tls_rx));
```

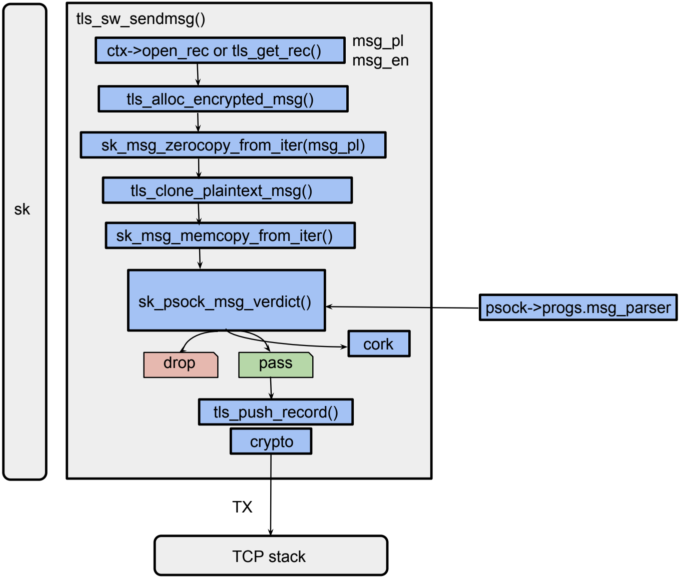
Path to Combining kTLS and BPF

- ULPs used by kTLS *and* BPF at Socket Layer → pick one
- Generic ULP stacking problematic performance, complexity wise
- Best path forward: refactoring & tearing old sock_map code apart
  - Generic sk_msg API for managing scatter/gather ring
  - psock framework on top of sk_msg with TCP as one implementation
  - Standalone BPF array/hash map where sockets are attached to

Path to Combining kTLS and BPF

- sk_msg and psock API as generic framework *across* ULPs
- Allowed for in-kernel ULP removal, keeping original TCP_ULP as-is
- Now BPF Socket Layer and kTLS *both* operate on sk_msg context
  - Allows removal of open coded TX plaintext/encrypted sg handling
  - Allows integration with BPF msg_parser program

kTLS with BPF



sk_msg Data Structure

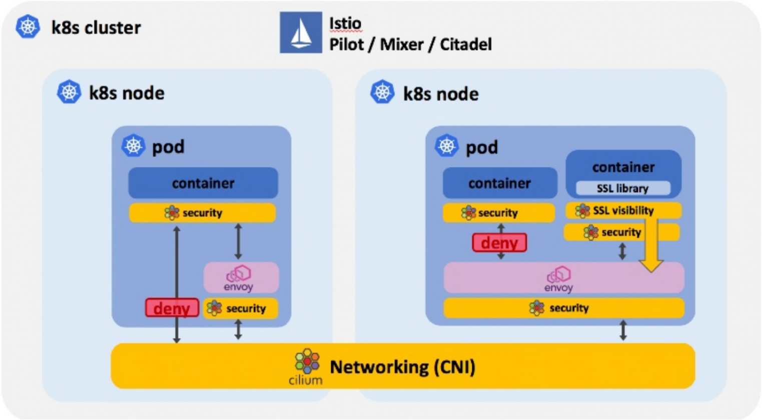
```
struct sk_msg_sg {
        u32                             start;
        u32                             curr;
        u32                             end;
        u32                             size;
        u32                             copybreak;
        bool                            copy[MAX_MSG_FRAGS];
        /* Extra element for wrap-around chaining */
        struct scatterlist              data[MAX_MSG_FRAGS + 1];
};

struct sk_msg {
        struct sk_msg_sg                sg;
        void                            *data;
        void                            *data_end;
        u32                             apply_bytes;
        u32                             cork_bytes;
        u32                             flags;
        struct sk_buff                  *skb;
        struct sock                     *sk_redir;
        struct sock                     *sk;
        struct list_head                list;
};
```

BPF Helpers for Socket Layer

- bpf_msg_apply_bytes()
- bpf_msg_cork_bytes()
- bpf_msg_redirect_map/hash()
- bpf_msg_pull_data()
- bpf_msg_push_data()
- Base BPF helpers like map lookups, etc

Orchestration

- Putting it all together: Cilium
  - API aware networking and network security for microservices
- BPF behind the scenes all the way: XDP, cls_bpf, socket layer



Summary, Next Steps

- First time kernel can enforce policy inside TLS connections!
- Next steps to work on
  - Extend currently limited set of helpers
  - Optimizations for fast-path (e.g. strparser)
  - kTLS also with AES GCM in 256 bit key size
  - Wider kTLS user space library adoption
  - Bounded loops in BPF core