eBPF Dynencap + Reflection

Brian Vazquez <brianvv@google.com>
Agenda

- Motivation
- In-kernel dynencap (Old)
- Bpf-ying dynencap (New)
- Reflection
- Conclusion
Motivation
Motivation and Problem statement

Motivation
Traffic Engineering (TE) to forward traffic via specific routers

Possible Approach (prior LWT)
Setting up multiple tunnel devices

Problem
TE using tunnels required root permission and configuring hundreds of devices would have impacted the performance of the system
Goals

● Allow unprivileged* applications to pick the exit point, without having to create tunnel devices or mocking with the routing table
● Allow each connection to be encapsulated to a different exit point
● Allow the exit point to change in the middle of the connection

* enforcement can be implemented at per-cgroup level
In-kernel dynencap
In-kernel Dynencap

- Add **per-socket state** that is used in an IP tunnel device
- Using `setsockopt`: Modify the per-socket state to change destination and/or encapsulation headers

**Setting destination**

- `ENCAP_GW`

**Changing encapsulation headers**

- `ENCAP_UDP`
In-kernel Dynencap

- **Host Configuration (MSS clamping)**
  - Use `SO_MARK` to select between a standard routing table and a special routing table
  - `Rules/Routes` to guarantee packets fit into mtu after encapsulation

```
# ip -6 rule show
0:     from all lookup local
1000:  from all fwmark 0xF lookup 1000
32766: from all lookup main

# ip -6 route show table 1000
default via fe80:: dev dynencap6 src fdaa::1 metric 1024 mtu lock 1444 advmss 1372 pref medium
```

- `mtu = 1500 - 40 ipv6 - 16 encap = 1444`
- `mss = 1444 - 40 ipv6 - 20 tcp - 12 opts = 1372`
Challenges: cached MSS

Problem:

It is possible to change routing based on certain actions i.e. setting ip_tos/so_mark. If these actions are performed in the middle of a connection which requires a different mss, the changes aren't reflected since the MSS is cached.
Challenges: cached MSS

Fix: Patch (To be proposed):

```c
void inet_csk_refresh_route(struct sock *sk) {
    struct dst_entry *dst;
    /* Do not attempt refreshing the route on listeners and closed * sockets. */
    if ((1 << sk->sk_state) & (TCPF_CLOSE | TCPF_LISTEN))
        return;
    /* Forget the old dst and look up a new one. */
    sk_dst_reset(sk);
    inet_csk(sk)->icsk_af_ops->rebuild_header(sk);
    /* See if the new route has a different MTU we should sync. */
    dst = sk_dst_get(sk);
    if (dst) {
        u32 mtu = dst_mtu(dst);
        if (mtu != inet_csk(sk)->icsk_pmtu_cookie)
            inet_csk(sk)->icsk_sync_mss(sk, mtu);
        dst_release(dst);
    }
}
```
bpf-ying
dynencap
Comparison

- LWT attaches to routes, TC attaches to qdisc
- Both run before software segmentation (GSO)
- Both received a skb as a context but LWT is more restricted in terms of reading/writing fields, and bpf helpers i.e. LWT don't have access to sk_local_storage

Decision

TC was chosen based on available bpf helpers
Keep encap data in a `sk_local_storage` map

```c
struct bpf_map_def __section("maps") dynencap_map = {
    .type = BPF_MAP_TYPE_SK_STORAGE,
    .key_size = sizeof(int),
    .value_size = sizeof(struct bpf_dyndest),
    .map_flags = BPF_F_NO_PREALLOC | BPF_F_CLONE,
};
```
eBPF Dynencap: Control Path

- Provide/modify encap data at the sk level with `setsockopt`

```c
SEC("cgroup/setsockopt")
int _dynencap_setsockopt(struct bpf_sockopt *ctx) {
    switch (ctx->optname) {
        case ENCAP_UDP:
            return setsockopt_dyndest_encap_udp(ctx);
        case ENCAP_GW:
            return dyndest_set_dst(ctx);
        default:
            ctx->optlen = 0;
            return 1;
    }
}
```
static int dyndest_set_dst(struct bpf_sockopt *ctx) {
    
    /*Create sk_storage */
    dd = bpf_sk_storage_get(&dynencap_map, ctx->sk, 0,
                            ctx->optlen ? BPF_SK_STORAGE_GET_F_CREATE : 0);

    /* Store IPv6 in sk local */
    optval_memcpy(ctx, &dd->dst.addr6, 0, sizeof(dd->dst.addr6));

    /* Mark packets */
    bpf_setsockopt(ctx, SOL_SOCKET, SO_MARK, &mark, sizeof(ctx->sk->mark));
    return -1;
}

* variables, error and boundary checks are omitted
eBPF Dynencap: Data Path

- Read encap data at TC egress hook, and modify the packet

```c
int _dynencap(struct __sk_buff *skb) {
    ...
    /* lookup dynencap struct */
    dd = bpf_sk_storage_get(&dynencap_map, sk, 0, 0);
    /* read outer network header, to reuse most fields */
    bpf_skb_load_bytes(...);
    /* add room for encap */
    bpf_skb_adjust_room(skb, encap_len, BPF_ADJ_ROOM_NET, flags);
    /* modify outer header */
    ...
    /* Store outer and encap headers */
    bpf_skb_store_bytes(skb, offset, &outer_ip6, sizeof(outer_ip6), BPF_F_INVALIDATE_HASH);
    ...
    return TC_ACT_PIPE;
}
```

* variables, error and boundary checks are omitted
Challenges: TSO/GSO

Problem:

- Neither TSO/GSO understand custom/multiple levels of encapsulation
- Packets need to fit the mtu after encapsulation headers are added

```c
__dev_queue_xmit
  sch_handle_egress
    tcf_classify
__dev_xmit_skb
  sch_direct_xmit
    validate_xmit_skb_list
    validate_xmit_skb
    skb_gso_segment
```
Challenges: TSO/GSO

Fix: Add a tunnel device to force software segmentation to take place before packet is modified by BPF

```c
__dev_queue_xmit(dynencap6) // tunnel device
__dev_xmit_skb
    sch_direct_xmit
    validate_xmit_skb_list
    validate_xmit_skb
    skb_gso_segment       // <--- executed because TSO is off. builds segments
bond_start_xmit
__dev_queue_xmit(eth0)
    sch_handle_egress
    tcf_classify        // <--- now inserts headers on segment skbs
    sch_direct_xmit
    bond_start_xmit
```
Reflection
Encap Reflection: motivation

- In the past, different reflection features have been implemented: ToS, fwmark. Now, with eBPF, implementing encapsulation headers reflection is possible.

- As part of the TE, sometimes packets have to traverse along the same path, and may or may not need additional metadata such as a virtual network ID.

- Most of the times, this encap data is irrelevant for the server processes. They don’t need to be aware of the overlay network.
Reflection: how to extend eBPF dynencap

- Egress logic of eBPF dynencap can be reused. The only difference is how the BPF MAP is populated

- Instead of using setsockopt to specify the encapsulation headers, we want to store the data for incoming connections. The cgroup_skb_ingress hook is used to capture the data
Reflection: "cgroup_skb/ingress"

```c
SEC("cgroup_skb/ingress")
int rx_reflection_store(struct __sk_buff *skb) {
...
    bpf_skb_load_bytes_relative(skb, 
        offset, &ip6_outer, sizeof(struct ipv6hdr), 
        BPF_HDR_START_MAC);
    outer_len = bpf_ntohs(ip6_outer.payload_len);
    if (outer_len < inner_len)  
        return -1;
    populate_map(skb, dd, &ip6_outer, offset);
...
}
```

* variables, error and boundary checks are omitted
```c
static __always_inline int populate_map(struct __sk_buff *skb,
                                       struct bpf_dyndest *dd, struct ipv6hdr *ip6_outer,
                                       int offset)
{
    ...
    memcpy(&dd->dst.addr6, &ip6_outer->saddr, sizeof(ip6_outer->saddr));
    memcpy(&dd->src.addr6, &ip6_outer->daddr, sizeof(ip6_outer->daddr));

    bpf_skb_load_bytes_relative(skb, offset,
                               &dd, i,
                               BPF_HDR_START_MAC) < 0

    ...
}
```

* variables, error and boundary checks are omitted
Challenges: BPF_MAP_TYPE_SK_STORAGE

Problem:

- sk_storage isn’t available for listener sockets (req socket)

Fix:

- Have an ephemeral entry in a global bpf map isolated per cgroup with a 5-tuple as a key:

  ```c
  struct bpf_map_def __section("maps") syn_encap_map = {
      .type = BPF_MAP_TYPE_LRU_HASH,
      .key_size = sizeof(struct connection),
      .value_size = sizeof(struct bpf_dyndest),
      .max_entries = 1000,
  };
  ```
Conclusion
Conclusion

What went well?

BPF-fying dynencap solved the goals initially set, and it was easily extended for encap headers reflection without invasive changes in the kernel.

What went wrong?

Modifying packets in the middle of the connection uncovered unexpected issues (MSS cache, GSO/TSO), which led to non-trivial fixes.

Nice to have?

- sk_local_storage for listener sockets?
- Tunnel (dummy) device without headers?
- BPF_MAP_TYPE_NS_STORAGE?
Thanks to Coco Li, Mahesh Bandewar, Stanislav Fomichev and Willem de Bruijn

Thank you!
Questions?