SKB Metadata Extensions

Net-Virt Team, Red Hat

Florian Westphal
4096R/AD5FF600 fw@strlen.de
80A9 20C5 B203 E069 F586
AE9F 7091 A8D9 AD5F F600

March 2019

Agenda

1 Introduction

2 Motivation

3 Extensions

4 Alternatives

sk buff and external data – history

- `struct sk_buff` is the main networking data structure
- associates raw ("on wire") data and "data about that data"
  - pointer to packet data
  - pointer to network device, socket, route
  - pointers/offset to where network, transport header etc. are
  - list pointers
  - also: checksum, timestamp, packet hash, refcount, etc.

hard to make changes to it – almost always has performance implications

- structure size and layout (access patterns)
- initialization cost
- already large structure

sk buff and external data – history (2)

- October 2002: IPSEC transform engine and bridge-nf get added
- `struct sk_buff` gains two members:
  - pointer to sec path (ipsec transform information)
  - pointer to bridge netfilter meta data (e.g. which bridge port received packet)
- both are pretty much same:
  - Only allocated on demand
  - Refcounted
  - hook into skb clone/copy/free functions
- Unlike e.g. `skb->{sk,dev}`: referenced data gets released with the skb

Fast forward to 2018 ...

Out-of-tree multipath implementation needs to store additional meta data

- main use case: DSS map (logical 64bit mptcp sequence number to individual tcp sequence numbers)
- Natural place to store this: `skb->cb[]`
- Problem: not enough space left
  - add second control buffer towards skb end?
  - add yet another pointer to sk_buff for `"struct mptcp_skb_data"`?
  - external (percpu) storage?
- would like to upstream mptcp eventually

skb extensions (1)

- mptcp Requirements are very similar to those of transform/sec path and bridge nf
- Idea: Unify them and replace `skb->sp` and `skb->nf_bridge` with `skb->extensions`
  - Unifies secpath and nf_bridge hooks in clone/copy/free paths
  - on skb clone, increment refcount of extension structure (no copy)
  - on skb free, decrement refcount of extension structure (and free if 0)
- Assumes no extensions are needed (added) in most cases
- if MPTCP finds a better way to solve the DSS map issue: all good, otherwise use the extension framework

skb extensions (2)

- Must be able to add all extensions at the same time
- Must be able to delete an extension again
- add enum with two extension ids: `SKB_EXT_{BRIDGE_NF,SEC_PATH}`
- add 2nd new member: `u8 active_extensions`
  - flag field that lists all active extensions
  - callers can check `skb->active_extension & EXTENSION_ID`
  - Without this we get following problem:
    - extensions a and b are active
    - skb is cloned
    - extension a is to be disabled
  - with `active_extensions`: can just unset `EXTENSION_ID` bit
  - otherwise, always need to kmemdup: "disable this extension" would fail under memory pressure
- upside of 2nd field: can keep "extension" pointer in undefined state

skb ext extension structure

Container structure: reference count, allocated space, actual data

```
struct skb_ext {
   refcount_t refcnt;
   u8 offset[SKB_EXT_NUM]; /* in chunks of 8 bytes */
   u8 chunks; /* same */
   char data[0] __aligned(8);
};
```

Accessible via `skb->extensions`
Individual extensions start at `extension.offset[id]`

when first extension gets added ...

- always allocates all memory at once (fixed size allocation)
- first extension allocated comes first in memory, i.e. `extension.offset[added_id]` is 1

has several advantages:

- can use kmem cache
- memory contents undefined, we only initialize `skb_ext` part
- so no added cost from larger allocation
- no need to add krealloc support

When a new extension might make sense

1. Data is related to the skb/packet aggregate
2. Data should be freed when the skb is free'd
3. Data is not going to be relevant/needed in normal case (udp, tcp, ...)
4. There are no actions needed on clone/free, such as callbacks into kernel modules

If one of the above doesn't hold, answer is likely "Not the infrastructure you're looking for"

When a new extension makes no sense: alternatives

1. store extra data in the skb "shared info" block
   - unchanged on clone
2. add a second control buffer block at the end of `sk_buff`
   - not zeroed out on allocation
   - doesn't change position of other struct members
3. add a control block at the end of `struct sk_buff_fclones`
   - only works for outgoing skbs allocated via `alloc_skb_fclone`