Introduction to eBPF Socket Maps

Linux Kernel Maintainer
Cong Wang xiyou.wangcong@gmail.com
Netdev 0x19, 2025-03-10

Agenda

1. eBPF Socket Maps Overview
2. Socket Map Programs
3. Sockops
4. Use Cases
    - Socket Splicing
    - Load Balancing
    - Packet Filtering
    - Socket Monitoring
    - L7 Protocol Parsing
    - TCP Hijacking
5. History and Future Directions

eBPF Socket Maps Overview

- **Socket Maps**: A special BPF map to store active sockets.
  - Contains FDs of sockets that are explicitly added.
  - Both hash table and array indexes can be used for lookups.
- **4 Different Programs**:
  - `BPF_SK_MSG_VERDICT`: Verdict on a generic TCP socket message.
  - `BPF_SK_SKB_STREAM_PARSE`: A TCP L7 protocol parser.
  - `BPF_SK_SKB_STREAM_VERDICT`: Verdict on a TCP packet.
  - `BPF_SK_SKB_VERDICT`: A generic verdict on a `struct __sk_buff`.
- **Sockops**: A set of socket layer hooks for TCP family.

Socket Map Program Attachment

- Unlike traditional eBPF programs attached to kernel hooks:
  - These 4 programs are **attached directly to socket maps**
  - They operate on packets flowing through the sockets stored within.
  - This enables direct socket-to-socket operations like redirection.
  - Provides fine-grained control over socket data paths.

XDP Socket Map

- **XDP Socket Map**: `BPF_MAP_TYPE_XSKMAP`
  - Special map type for redirecting XDP packets directly to XSK.
  - Uses `bpf_redirect_map()` for packet redirection.
  - Different from regular socket maps (focuses on XDP layer).
  - Not in our scope of discussion today.

Message Verdict

- Decides whether to pass, drop, or redirect a TCP socket message on TX.
- Operates on `struct sk_msg_md` instead of `struct __sk_buff`.
- Only supports regular TCP and kTLS messages.
- **Use Cases**:
  - Rate limiting: Drop messages if QPS exceeds threshold.
  - Socket layer redirection: Redirect messages early on TX.
  - Content filtering: Filtering L7 messages.

SKB Verdict

- Operates on `struct __sk_buff` for RX.
- Easy to use if you are already familiar with `struct sk_buff`.
- Generic for all types of sockets, but still requires explicit support of each socket.
- Currently only supports: TCP, UDP, UDS, vsock.
- Can access full packet data.
- **Use Cases**:
  - Heterogeneous socket redirection and splicing.
  - Socket layer filtering and load balancing.

SKB Stream Verdict

- Like SKB Verdict, but only for TCP sockets due to historical reasons.
- Can be combined with SKB stream parser togther.
- **Use Cases**:
  - L7 load balancing based on request content.
  - Protocol-aware message filtering.
  - Request routing based on L7 headers.

SKB Stream Parser

- A true TCP L7 parser in the kernel.
- Teach the kernel to learn L7 message boundaries.
- SKB's are reassembled on the fly.
- Still operates on `struct __sk_buff`.
- Requires a paired verdict program to make decisions.
- **Use Cases**:
  - Parse application protocols (HTTP, Redis, etc).
  - Extract L7 metadata for load balancing.

Sockops (Socket Operations)

- Hook into TCP socket lifecycle events (`connect()`, `accept()`, etc.).
- Combined with socket map to perform per-socket actions, e.g. with `bpf_setsockopt()`.
- Cooperate with socket map via `bpf_sock_ops`, FD-free and transparent to user-space.
- Per cgroup granularity.

TCP Connection Setup Hooks

- `BPF_SOCK_OPS_ACTIVE_ESTABLISHED_CB`
  - Triggered when client successfully connects to server
- `BPF_SOCK_OPS_PASSIVE_ESTABLISHED_CB`
  - Triggered when server accepts a client connection
- `BPF_SOCK_OPS_TCP_CONNECT_CB`
  - Called during TCP `connect()`
  - Access to initial connection parameters

TCP State Change Hooks

- `BPF_SOCK_OPS_STATE_CB`
  - Called when TCP state changes
  - Useful for connection tracking and monitoring
- `BPF_SOCK_OPS_TCP_LISTEN_CB`
  - Called when socket enters listen state
  - Access to server socket configuration

TCP Timing Hooks

- `BPF_SOCK_OPS_RTT_CB`
  - Called when Round Trip Time (RTT) is measured
  - Access to latest RTT measurements
- `BPF_SOCK_OPS_RTO_CB`
  - Called on Retransmission Timeout (RTO) events
  - Can modify RTO parameters
  - Useful for custom congestion control
- `BPF_SOCK_OPS_RETRANS_CB`
  - Called on TCP packet retransmission
  - Access to retransmission statistics
  - Can implement custom retransmission policies

TCP Option Hooks

- `BPF_SOCK_OPS_WRITE_HDR_OPT_CB`
  - Called to write TCP header options
  - Access to TCP option fields
- `BPF_SOCK_OPS_PARSE_HDR_OPT_CB`
  - Called to parse TCP header options
  - Access to TCP option fields
  - Can implement custom TCP options
- `BPF_SOCK_OPS_HDR_OPT_LEN_CB`
  - Called to get TCP header option length
  - Determines space needed for options
  - Must be consistent with parse callback

Use Case: Socket Splicing

- Forward data directly between two sockets:
  - Use `BPF_SK_SKB_VERDICT` for heterogenous socket splicing.
  - User-space program needs to explicitly insert their socket FD into this sockmap.

```c
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>

struct {
    __uint(type, BPF_MAP_TYPE_SOCKMAP);
    __uint(max_entries, 2);
    __type(key, __u32);
    __type(value, __u64);
} sockmap SEC(".maps");

SEC("sk_skb")
int socket_splice(struct __sk_buff *skb)
{
    __u32 peer_idx = 1;
    if (bpf_sk_redirect_map(skb, &sockmap, peer_idx, 0) < 0) {
        bpf_printk("Splice: redirect failed\n");
        return SK_DROP;
    }
    return SK_PASS;
}

char _license[] SEC("license") = "GPL";
```

Use Case: Load Balancing

- Distribute connections to multiple backends.
- Store backend sockets in a sockmap:
  - Select backend based on a simple hash/round-robin.
  - Redirect message via `bpf_msg_redirect_map`

```c
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>

#define NUM_BACKENDS 8

struct {
    __uint(type, BPF_MAP_TYPE_SOCKMAP);
    __uint(max_entries, NUM_BACKENDS);
    __type(key, __u32);
    __type(value, __u64);
} backend_map SEC(".maps");

struct {
    __uint(type, BPF_MAP_TYPE_ARRAY);
    __uint(max_entries, 1);
    __type(key, __u32);
    __type(value, __u32);
} rr_index SEC(".maps");

SEC("sk_msg")
int load_balance(struct sk_msg_md *msg)
{
    __u32 key = 0;
    __u32 *p_idx = bpf_map_lookup_elem(&rr_index, &key);
    __u32 backend_key = 0;

    if (p_idx) {
        backend_key = *p_idx;
        __u32 new_index = (*p_idx + 1) % NUM_BACKENDS;
        bpf_map_update_elem(&rr_index, &key, &new_index, BPF_ANY);
    }

    long err = bpf_msg_redirect_map(msg, &backend_map, backend_key, 0);
    if (err) {
        bpf_printk("Load Balancer: redirect to backend %d failed\n", backend_key);
        return SK_DROP;
    }
    return SK_PASS;
}

char _license[] SEC("license") = "GPL";
```

Use Case: Packet Filtering

- Filter RX packets at socket layer.
- Can be used for firewall-like behavior:
  - Parse TCP header for destination port.
  - If (port == disallowed_port) then `SK_DROP`

```c
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_endian.h>

SEC("sk_skb")
int packet_filter(struct __sk_buff *skb)
{
    __u16 dest_port;
    if (bpf_skb_load_bytes(skb, 22, &dest_port, sizeof(dest_port)) < 0)
        return SK_PASS;
    if (bpf_ntohs(dest_port) == 8080) {
        bpf_printk("Packet Filter: dropping traffic to port 8080\n");
        return SK_DROP;
    }
    return SK_PASS;
}

char _license[] SEC("license") = "GPL";
```

Use Case: Socket Monitoring

- Collect real-time socket metrics (bytes, latency, etc.).
- Update statistics in a BPF map:
  - Lookup stats map using a socket ID.
  - Increment byte counter based on message size.

```c
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>

struct stats_t {
    __u64 bytes_count;
};

struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __uint(max_entries, 1024);
    __type(key, __u32);
    __type(value, struct stats_t);
} stats_map SEC(".maps");

SEC("sk_msg")
int socket_monitor(struct sk_msg_md *msg)
{
    __u32 key = msg->sk;
    struct stats_t *val = bpf_map_lookup_elem(&stats_map, &key);
    if (val) {
        val->bytes_count += (msg->data_end - msg->data);
    }
    bpf_printk("Socket Monitoring: updated stats\n");
    return SK_PASS;
}

char _license[] SEC("license") = "GPL";
```

Use Case: L7 Protocol Parsing

- Parse Thrift messages:
  - Reads the first 4 bytes to get the frame length.
  - Reads 4 bytes at offset 4, which in TBinaryProtocol contains the version.
  - Filters the message based on version.

```c
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_endian.h>

#define THRIFT_EXPECTED_VERSION 0x80010000

SEC("sk_skb/parser")
int thrift_parser(struct __sk_buff *skb)
{
    __u32 frame_size_net = 0;
    int ret;

    ret = bpf_skb_load_bytes(skb, 0, &frame_size_net, sizeof(frame_size_net));
    if (ret < 0) {
        bpf_printk("Thrift Parser: failed to load frame size\n");
        return 0;
    }

    return bpf_ntohl(frame_size_net);
}
```

```
SEC("sk_skb/verdict")
int thrift_verdict(struct __sk_buff *skb)
{
    __u32 version_net = 0;
    int ret;

    ret = bpf_skb_load_bytes(skb, 4, &version_net, sizeof(version_net));
    if (ret < 0) {
        bpf_printk("Thrift Verdict: failed to load version field\n");
        return SK_PASS;
    }

    __u32 version = bpf_ntohl(version_net);
    bpf_printk("Thrift Verdict: version field = 0x%x\n", version);

    if (version != THRIFT_EXPECTED_VERSION) {
        bpf_printk("Thrift Verdict: unexpected version (0x%x), dropping message\n", version);
        return SK_DROP;
    }

    bpf_printk("Thrift Verdict: version acceptable, passing message\n");
    return SK_PASS;
}

char _license[] SEC("license") = "GPL";
```

Use Case: TCP Hijacking

- Intercept and modify TCP connections in real-time:
  - Use sockops to detect new connections
  - Store socket in sockmap
  - Redirect traffic through proxy socket

```c
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>

struct {
    __uint(type, BPF_MAP_TYPE_SOCKMAP);
    __uint(max_entries, 1);
    __type(key, __u32);
    __type(value, __u64);
} proxy_map SEC(".maps");

SEC("sockops")
int intercept_conn(struct bpf_sock_ops *skops)
{
    if (skops->op != BPF_SOCK_OPS_ACTIVE_ESTABLISHED_CB)
        return 0;

    __u32 key = 0;
    __u64 proxy_fd = skops->sk;

    bpf_map_update_elem(&proxy_map, &key, &proxy_fd, BPF_ANY);
    bpf_sock_ops_cb_flags_set(skops, BPF_SOCK_OPS_STATE_CB_FLAG);
    return 0;
}

SEC("sk_msg")
int redirect_to_proxy(struct sk_msg_md *msg)
{
    __u32 key = 0;
    long err = bpf_msg_redirect_map(msg, &proxy_map, key, 0);
    if (err) {
        bpf_printk("TCP Hijack: redirect to proxy failed\n");
        return SK_DROP;
    }
    return SK_PASS;
}

char _license[] SEC("license") = "GPL";
```

Known Limitations

- **sk_msg_md vs __sk_buff**:
  - `sk_msg_md` was invented for `BPF_SK_MSG_VERDICT` where no skb is involved.
  - `sk_msg` is still heavily used on the data path in the kernel.
  - `__sk_buff` is more sophisticated than `sk_msg_md` and more familiar.
- **Transparency**:
  - FD is a pain point and managing socket in user-space is not friendly.
  - Practically, we have to use `bpf_sock_ops` for transparency.
- **sockops**:
  - Only supports TCP sockets so far.
  - Not easy to extend to other protocols.

History & Recent Advances

- **History**:
  - 2017-08-15: sockmap was introduced by John Fastabend.
  - 2017-08-28: `BPF_SK_SKB_STREAM_VERDICT` and `BPF_SK_SKB_STREAM_PARSER` were introduced by John Fastabend.
  - 2018-03-18: `BPF_SK_MSG_VERDICT` was introduced by John Fastabend.
  - 2021-03-30: `BPF_SK_SKB_VERDICT` was introduced by Cong Wang.
  - 2021-07-04: UDS support for sockmap was added by Cong Wang.
  - 2023-03-27: vsock support for sockmap was added by Bobby Eshleman.
- **Recent Advances**:
  - `sk_msg` batching for improved performance, by Zijian Zhang.

Future Directions

- Generic sockops support for all socket types.
- More socket support for sockmap.
  - Even skb-less socket, e.g. SMC-R.
    - Proxying RDMA network with TCP network.
- Zero-copy for skb redirection.
  - Decouple `sk_msg` from `sk_buff` data path?
  - Replace `->sendmsg()` with direct skb queueing.
- Unify TX and RX data paths?
  - Potentially replace KCM socket?

Thank You!

Questions?

Contact: Cong Wang xiyou.wangcong@gmail.com