Data-centric Tracing with BPF

Alan Maguire, Linux Kernel Networking team, Oracle

(alan.maguire@oracle.com)

https://blogs.oracle.com/linux/post/bpf-application-development-and-libbpf

Control flow tracing

The Linux kernel has some great tools for tracking control flow.

See Documentation/trace/ftrace.rst for details.

```
# tracer: function_graph
#
# CPU  TASK/PID        DURATION                  FUNCTION CALLS
# |    |    |           |   |                     |   |   |   |
0)    sh-4802     |               |                  d_free() {
0)    sh-4802     |               |                    call_rcu() {
0)    sh-4802     |               |                      __call_rcu() {
0)    sh-4802     |   0.616 us    |                        rcu_process_gp_end();
0)    sh-4802     |   0.586 us    |                        check_for_new_grace_period();
0)    sh-4802     |   2.899 us    |                      }
0)    sh-4802     |   4.040 us    |                    }
0)    sh-4802     |   5.151 us    |                  }
0)    sh-4802     | + 49.370 us   |                }
```

+ data-centric debugging

...while debuggers such as gdb have excellent support for examining data.

```
(gdb) print test_cases

$1 = {root = {rb_node = 0xffff8800d1d16800}, size = 1,

  keycompare = 0xffffffffa0bdbe10 <ktf_map_name_compare>}
```

= Data-centric Tracing with BPF

Here we show how you can bring some of the power of debuggers in examining data in depth to your BPF-based tools.

BPF Type Format (BTF)

The key to this is the BPF Type Format – it ships with many distros and provides descriptions of

- data types
- functions
- variables

Is your kernel built with CONFIG_DEBUG_INFO_BTF?

Check for */sys/kernel/btf*

libbpf support for dumping data representations

libbpf provides btf_dump*() functions to dump BTF-based type descriptions

Used for header generation simplifying BPF program writing

```
# bpftool btf dump file /sys/kernel/btf/vmlinux format c > vmlinux.h
```

New interface recently added – btf_dump__dump_type_data() - allows us to

dump a representation of provided data

...using BTF information associated with its type

Default output (tabs for indentation, newlines)

```
(struct sk_buff){
	(union){
		(struct){
			.next = (struct sk_buff *)0xffffffffffffffff,
			.prev = (struct sk_buff *)0xffffffffffffffff,
		(union){
			.dev = (struct net_device *)0xffffffffffffffff,
			.dev_scratch = (long unsigned int)18446744073709551615,
		},
	},
 ...
```

Function Signature for dumping typed data

```
int btf_dump__dump_type_data(
    struct btf_dump *d,   // dump (BTF + dump function/opts)
    __u32 id,             // type id of target type
    const void *data,     // pointer to data we wish to display
    size_t data_sz,       // amount of data we have
    const struct btf_dump_type_data_opts *opts// options for display
);
```

Options for dumping typed data

```
struct btf_dump_type_data_opts {
        /* size of this struct, for forward/backward compatibility */
        size_t sz;
        const char *indent_str;
        int indent_level;
        /* below match "show" flags for bpf_show_snprintf() */
        bool compact;           /* no newlines/indentation */
        bool skip_names;        /* skip member/type names */
        bool emit_zeroes;       /* show 0-valued fields */
        size_t :0;
};
```

Options examples:

Compact form:

```
(struct sk_buff){(union){ (struct){ .next = (struct sk_buff
*)0xffffffffffffffff, .prev = (struct sk_buff *)0xffffffffffffffff,
(union){.dev = (struct net_device *)0xffffffffffffffff, .dev_scratch =
(long unsigned int)18446744073709551615,},}, ...
```

Compact + skip_names form:

```
{ { 0xffffffffffffffff, 0xffffffffffffffff, { 0xffffffffffffffff,
18446744073709551615,},}, ...
```

How to use the API (Userspace)

1. retrieve BTF you want to operate on:

```
struct btf *btf = libbpf_find_kernel_btf();
```

2. create a printf-like callback function called every time data is dumped:

```
static void btf_dump_printfn(void *ctx, const char *fmt, va_list args)
{
        vprintf(fmt, args); // just printf() dumped data
}
```

3. create a btf_dump using the BTF, specified callback, and options

```
struct btf_dump *d = btf_dump__new(b, NULL, &opts, btf_dump_printfn);
```

4. Get the BTF ID of the type you want to display

```
type_id = btf__find_by_name(btf, "sk_buff");
```

5. Retrieve your data from kernel (via perf event, BPF ring buffer, map - see next slide) and dump it!

```
btf_dump__dump_type_data(d, type_id, skb, skbsize, &opts)
```

How to use the API (BPF program)

1. Use a structure that contains the size of the data you're tracing, and the data. Size is important, as for a >8k task_struct you likely won't capture the whole structure, so we need to know how much valid data you're retrieved!

```
struct tracedata {
        size_t  data_size;
        char data[MAX_DATA_SIZE];
};
```

The bpf_probe_read[_kernel]() helper can be used to populate the data.

2. Use bpf_perf_event_output() or BPF ringbuf interfaces to pass dump data to userspace libbpf.

Example: ksnoop

https://lore.kernel.org/bpf/1628025796-29533-1-git-send-email-alan.maguire@oracle.com/

ksnoop utilizes kernel/module BTF to

- look up the BTF descriptions of specified functions to retrieve info about arguments, return values
- places this info in maps that can be retrieved via instruction pointer, specifying first argument is a pointer to a 256-byte area, etc
- BPF program is attached to function entry/return, and on firing we can bpf_probe_read() relevant data and send perf events to userspace

ksnoop: get function info, trace function

```
# ksnoop info ip_send_skb
int  ip_send_skb(struct net  * net, struct sk_buff  * skb);
#  ksnoop trace ip_send_skb
              TIME  CPU      PID FUNCTION/ARGS
     78101668506811    1     2813 ip_send_skb(
                                      net = *(0xffffffffb5959840)
                                       (struct net){
                                        .passive = (refcount_t){
                                         .refs = (atomic_t){
                                          .counter = (int)0x2,
                                         },
                                        },
```

ksnoop: trace specific argument

```
# ksnoop "ip_send_skb(skb)"
                TIME        PID FUNCTION/ARGS
      22996961085545       2813 ip_send_skb(
                                 skb = *(0xffff9897947a8c00)
                                  (struct sk_buff){
                                   (union){
                                    .sk = (struct sock *)0xffff9895491ebf00,
                                    .ip_defrag_offset = (int)0x491ebf00,
                                   },
                                   (union){
                                    (struct){
                                     ._skb_refdst = (long unsigned
```

ksnoop: trace specific argument field

```
# ksnoop "ip_send_skb(skb->sk)"
                TIME        PID FUNCTION/ARGS
      23610619446101       2813 ip_send_skb(
                                 skb->sk = *(0xffff9895491ebf00)
                                  (struct sock){
                                   .__sk_common = (struct sock_common){
                                    (union){
                                     .skc_addrpair = (__addrpair)0x1701a8c015d38f8d,
                                     (struct){
                                      .skc_daddr = (__be32)0x15d38f8d,
                                      .skc_rcv_saddr = (__be32)0x1701a8c0,
```

ksnoop: trace return value

```
# ksnoop "ip_send_skb(return)"
                TIME        PID FUNCTION/ARGS
      23048667523207       2813 ip_send_skb(
                                 return =
                                   (int)0x0
                                );
```

ksnoop: trace functions (if called in loose order)

```
# ksnoop -s tcp_sendmsg __tcp_transmit_skb  ip_output
            TIME  CPU      PID FUNCTION/ARGS
  95115660547342    0     2824 tcp_sendmsg(
                                 sk = *(0xffff985149b76900)
                                  (struct sock){
                                   .__sk_common = (struct sock_common){
                                    (union){
                                     .skc_addrpair = (__addrpair)0x100007f0100007f,
                                     (struct){
                                      .skc_daddr = (__be32)0x100007f,
                                      .skc_rcv_saddr = (__be32)0x100007f,
                                     },
```

ksnoop: module tracing (with module BTF)

```
# ksnoop iwl_trans_send_cmd
               TIME        PID FUNCTION/ARGS
      23093006913614       1673 iwl_trans_send_cmd(
                                 trans = *(0xffff989564d20028)
                                  (struct iwl_trans){
                                   .ops = (struct iwl_trans_ops *)0xffffffffc0e02fa0,
                                   .op_mode = (struct iwl_op_mode
*)0xffff989566849fc0,
                                   .trans_cfg = (struct iwl_cfg_trans_params
*)0xffffffffc0e05280,
                                   .cfg = (struct iwl_cfg *)0xffffffffc0e05280,
```

References

BTF: https://www.kernel.org/doc/Documentation/bpf/btf.rst

libbpf typed dump capabilities (bpf-next patchset)

https://lore.kernel.org/bpf/1626362126-27775-1-git-send-email-alan.maguire@oracle.com/

Ksnoop (bpf-next patchset)

https://lore.kernel.org/bpf/1628025796-29533-1-git-send-email-alan.maguire@oracle.com/