bpf_trace_printk

static int (*bpf_trace_printk)(const char *fmt, int fmt_size, ...) =
(void *) BPF_FUNC_trace_printk;

https://elixir.bootlin.com/linux/v4.15/source/tools/testing/selftests/bpf/bpf_helpers.h#L23

把函数转发到另外一个函数上

#define __BPF_ENUM_FN(x) BPF_FUNC_ ## x
enum bpf_func_id {

__BPF_FUNC_MAPPER(__BPF_ENUM_FN)
__BPF_FUNC_MAX_ID,
};
#undef __BPF_ENUM_FN

https://elixir.bootlin.com/linux/v4.15/source/include/uapi/linux/bpf.h#L744

定义了转发目的函数
BPF_FUNC_trace_printk

这里是映射关系：
https://elixir.bootlin.com/linux/v4.15/source/include/uapi/linux/bpf.h#L688

trace_printk函数定义：

https://elixir.bootlin.com/linux/v4.15/source/include/linux/kernel.h#L667

```
static const struct bpf_func_proto *tracing_func_proto(enum
bpf_func_id func_id)
{
...
case BPF_FUNC_trace_printk:
        return bpf_get_trace_printk_proto();
...
}
```

https://elixir.bootlin.com/linux/v4.15/source/kernel/trace/bpf_trace.c#L506

```
const struct bpf_func_proto *bpf_get_trace_printk_proto(void)
{
    /*
     * this program might be calling bpf_trace_printk,
     * so allocate per-cpu printk buffers
     */
    trace_printk_init_buffers();

    return &bpf_trace_printk_proto;
}
```

https://elixir.bootlin.com/linux/latest/source/kernel/trace/bpf_trace.c#L451

```
static const struct bpf_func_proto bpf_trace_printk_proto = {
    .func       = bpf_trace_printk,
    .gpl_only = true,
    .ret_type = RET_INTEGER,
    .arg1_type   = ARG_PTR_TO_MEM,
    .arg2_type   = ARG_CONST_SIZE,
};
```

https://elixir.bootlin.com/linux/v4.15/source/kernel/trace/bpf_trace.c#L254

```
/*
 * Only limited trace_printk() conversion specifiers allowed:
 * %d %i %u %x %ld %li %lu %lx %lld %lli %llu %llx %p %s
 */
BPF_CALL_5(bpf_trace_printk, char *, fmt, u32, fmt_size, u64, arg1,
       u64, arg2, u64, arg3)
{
    bool str_seen = false;
    int mod[3] = {};
    int fmt_cnt = 0;
    u64 unsafe_addr;
    char buf[64];
    int i;

    /*
     * bpf_check()->check_func_arg()->check_stack_boundary()
     * guarantees that fmt points to bpf program stack,
     * fmt_size bytes of it were initialized and fmt_size > 0
     */
    if (fmt[--fmt_size] != 0)
        return -EINVAL;

    /* check format string for allowed specifiers */
    for (i = 0; i < fmt_size; i++) {
        if ((!isprint(fmt[i]) && !isspace(fmt[i])) || !isascii(fmt[i]))
            return -EINVAL;

        if (fmt[i] != '%')
            continue;

        if (fmt_cnt >= 3)
            return -EINVAL;

        /* fmt[i] != 0 && fmt[last] == 0, so we can access fmt[i + 1] */
        i++;
        if (fmt[i] == 'l') {
            mod[fmt_cnt]++;
            i++;
        } else if (fmt[i] == 'p' || fmt[i] == 's') {
            mod[fmt_cnt]++;
            i++;
            if (!isspace(fmt[i]) && !ispunct(fmt[i]) && fmt[i] != 0)
                return -EINVAL;
            fmt_cnt++;
            if (fmt[i - 1] == 's') {
                if (str_seen)
                    /* allow only one '%s' per fmt string */
                    return -EINVAL;
                str_seen = true;

                switch (fmt_cnt) {
                case 1:
                    unsafe_addr = arg1;
                    arg1 = (long) buf;
                    break;
                case 2:
                    unsafe_addr = arg2;
                    arg2 = (long) buf;
                    break;
                case 3:
                    unsafe_addr = arg3;
                    arg3 = (long) buf;
                    break;
                }
                buf[0] = 0;
                strncpy_from_unsafe(buf,
                            (void *) (long) unsafe_addr,
                            sizeof(buf));
            }
            continue;
        }

        if (fmt[i] == 'l') {
            mod[fmt_cnt]++;
            i++;
        }

        if (fmt[i] != 'i' && fmt[i] != 'd' &&
            fmt[i] != 'u' && fmt[i] != 'x')
            return -EINVAL;
        fmt_cnt++;
    }

/* Horrid workaround for getting va_list handling working with different
 * argument type combinations generically for 32 and 64 bit archs.
 */
#define __BPF_TP_EMIT()  __BPF_ARG3_TP()
#define __BPF_TP(...)
        \
    __trace_printk(1 /* Fake ip will not be printed. */,
        \
               fmt, ##__VA_ARGS__)

#define __BPF_ARG1_TP(...)
            \
    ((mod[0] == 2 || (mod[0] == 1 && __BITS_PER_LONG == 64))
    \
     ? __BPF_TP(arg1, ##__VA_ARGS__)
                \
     : ((mod[0] == 1 || (mod[0] == 0 && __BITS_PER_LONG == 32))
    \
        ? __BPF_TP((long)arg1, ##__VA_ARGS__)
            \
        : __BPF_TP((u32)arg1, ##__VA_ARGS__)))

#define __BPF_ARG2_TP(...)
            \
    ((mod[1] == 2 || (mod[1] == 1 && __BITS_PER_LONG == 64))
    \
     ? __BPF_ARG1_TP(arg2, ##__VA_ARGS__)
                \
     : ((mod[1] == 1 || (mod[1] == 0 && __BITS_PER_LONG == 32))
    \
        ? __BPF_ARG1_TP((long)arg2, ##__VA_ARGS__)
           \
        : __BPF_ARG1_TP((u32)arg2, ##__VA_ARGS__)))

#define __BPF_ARG3_TP(...)
            \
    ((mod[2] == 2 || (mod[2] == 1 && __BITS_PER_LONG == 64))
    \
     ? __BPF_ARG2_TP(arg3, ##__VA_ARGS__)
                \
     : ((mod[2] == 1 || (mod[2] == 0 && __BITS_PER_LONG == 32))
    \
        ? __BPF_ARG2_TP((long)arg3, ##__VA_ARGS__)
           \
        : __BPF_ARG2_TP((u32)arg3, ##__VA_ARGS__)))

    return __BPF_TP_EMIT();
}
```

https://elixir.bootlin.com/linux/v4.15/source/kernel/trace/bpf_trace.c#L143

开头的BPF_CALL_5是个macro，他有且仅有5个兄弟姐妹。因为只有5个registry用来寄存helper function的参数。它们的定义在这里：https://elixir.bootlin.com/linux/v4.15/source/include/linux/filter.h#L407

__trace_printk(1 /* Fake ip will not be printed. */, \
fmt,
##__VA_ARGS__)

下面是定义：

https://elixir.bootlin.com/linux/latest/source/kernel/trace/trace_printk.c#L230

下面是根定义：

https://elixir.bootlin.com/linux/latest/source/kernel/trace/trace.c#L3206