eBPF 工作原理浅析

狄卫华

2022-06-18

个人介绍

狄卫华： 资深云原生架构师，《Linux内核观测技术BPF》译者之一，eBPF 技术的爱好者。

网站 ebpf.top 和公众号 [深入浅出BPF] 的维护者。

目录

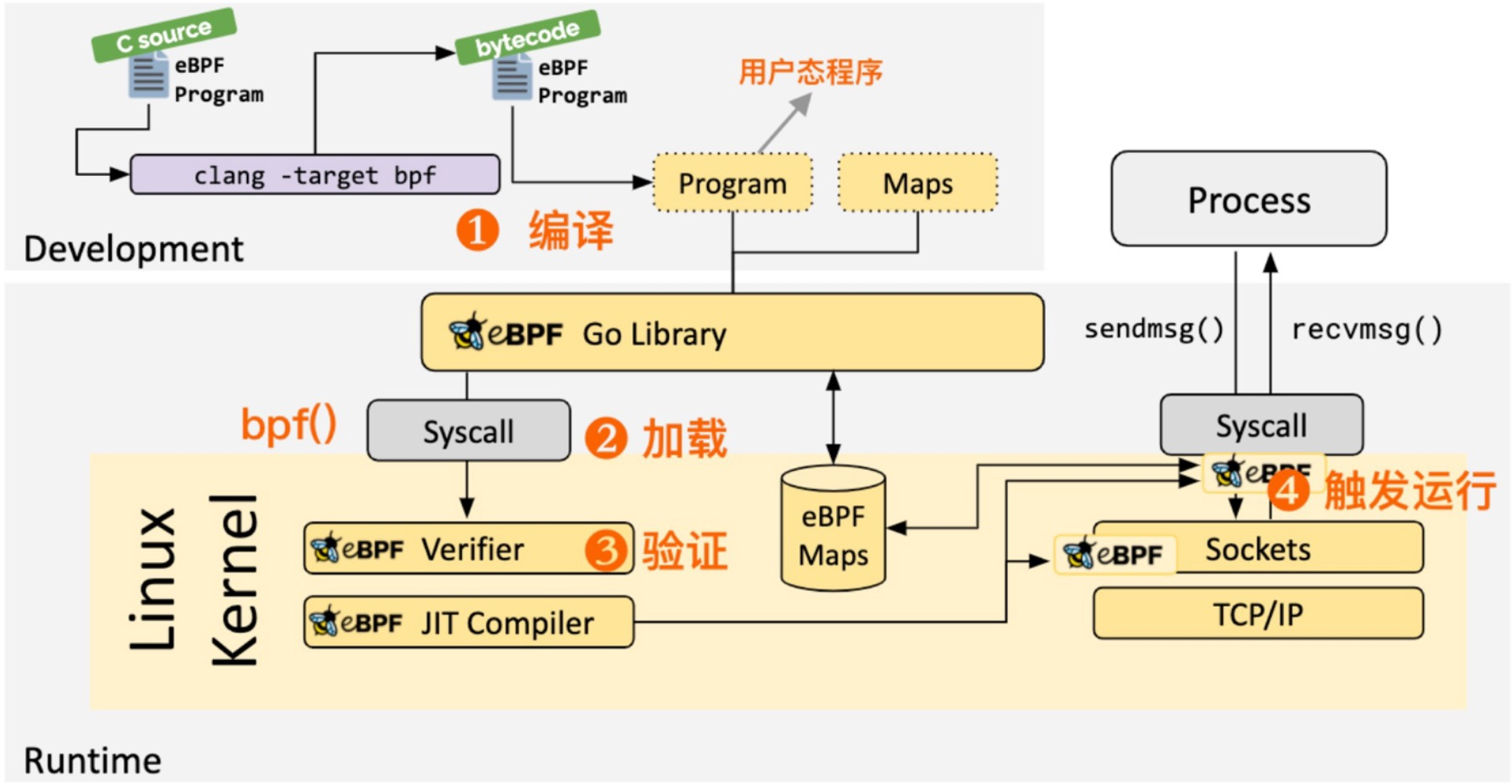
1. 整体工作流程

背景知识：eBPF 程序类型：

1. 程序类型具有不同的事件触发场景（BPF_PROG_TYPE_SOCKET_FILTER/BPF_PROG_TYPE_TRACEPOINT ...）
2. 程序类型决定可访问的 eBPF 辅助函数集合
3. 程序类型决定备用 eBPF 函数入参类型 (struct __sk_buff *skb/struct pt_regs *ctx)
4. 程序类型决定是否可以访问网络数据包

“ 本文不涉及到 BTF 和 CO-RE 内容。 ”

1. 整体工作流程


C source
eBPF Program
bytecode
eBPF Program
用户态程序
clang -target bpf
Program
Maps
Process
Development
1 编译
eBPF Go Library
sendmsg()
recvmsg()
bpf()
Syscall
2 加载
Syscall
4 触发运行
Linux Kernel
eBPF Verifier
3 验证
eBPF Maps
eBPF Sockets
eBPF JIT Compiler
TCP/IP
Runtime

2. 编写 eBPF 程序

hello_kern.c 新的进程执行时，触发 sys_enter_execve 事件，运行 bpf_prog 程序。

```c
#include <linux/bpf.h>
#define SEC(NAME) __attribute__((section(NAME), used))  // ①

static int (*bpf_trace_printk)(const char *fmt, int fmt_size,
                               ...) = (void *)BPF_FUNC_trace_printk;  // ② 为什么需要如此声明?

SEC("tracepoint/syscalls/sys_enter_execve")  // ③ 这个名字有啥讲究没?
int bpf_prog(void *ctx) {   // ④ 参数到底应该是啥类型?
  char msg[] = "Hello, BPF World!";  // ⑤ 大小?  是否可以使用 const ?
  bpf_trace_printk(msg, sizeof(msg));
  return 0;
}

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

2. 编写 eBPF 程序

```
#define SEC(NAME) __attribute__((section(NAME), used))  // ①
```

`__attribute__((section(NAME), used))`，其作用是将作用的函数或数据放入指定名为 "section_name" 对应的段中，这个在编译阶段我们可以比较明显看到。

2. 编写 eBPF 程序

```
static int (*bpf_trace_printk)(const char *fmt, int fmt_size,
                               ...) = (void *)BPF_FUNC_trace_printk; // ②
```

```
$ clang -E hello_kern.c -o hello_kern_precompile.c

$ vim hello_precompile.c
# 5064 "/usr/include/linux/bpf.h" 3 4
enum bpf_func_id {
 // ...
 BPF_FUNC_ktime_get_ns,
 BPF_FUNC_trace_printk, // 这里定义为 6
 // ...
 }
```

2. 编写 eBPF 程序

```
static int (*bpf_trace_printk)(const char *fmt, int fmt_size,
                               ...) = (void *)BPF_FUNC_trace_printk; // ②
```

=>

```
static int (*bpf_trace_printk)(const char *fmt, int fmt_size,...) = (void *)6
```

2. 编写 eBPF 程序

```
SEC("tracepoint/syscalls/sys_enter_execve")  // ③
```

```
static const struct bpf_sec_def *find_sec_def(const char *sec_name){
        // ...

        n = ARRAY_SIZE(section_defs);
        for (i = 0; i < n; i++) {
                sec_def = &section_defs[i];
                allow_sloppy = (sec_def->cookie & SEC_SLOPPY_PFX) && !strict;
                if (sec_def_matches(sec_def, sec_name, allow_sloppy))
                        return sec_def;
        }

        //...
}
```

2. 编写 eBPF 程序

```
SEC("tracepoint/syscalls/sys_enter_execve")  // ③
```

格式：“tracepoint/<category>/<name> 或 tp/<category>/<name>"

[libbpf/libbf.c#9215](#) 有删减

```
static const struct bpf_sec_def section_defs[] = {
        SEC_DEF("socket",               SOCKET_FILTER, 0, SEC_NONE | SEC_SLOPPY_PFX),
        SEC_DEF("kprobe+",              KPROBE, 0, SEC_NONE, attach_kprobe),
        SEC_DEF("uprobe+",              KPROBE, 0, SEC_NONE, attach_uprobe),
        SEC_DEF("kretprobe+",           KPROBE, 0, SEC_NONE, attach_kprobe),
        SEC_DEF("uretprobe+",           KPROBE, 0, SEC_NONE, attach_uprobe),
        SEC_DEF("tracepoint+",          TRACEPOINT, 0, SEC_NONE, attach_tp),
  // ...
```

2. 编写 eBPF 程序

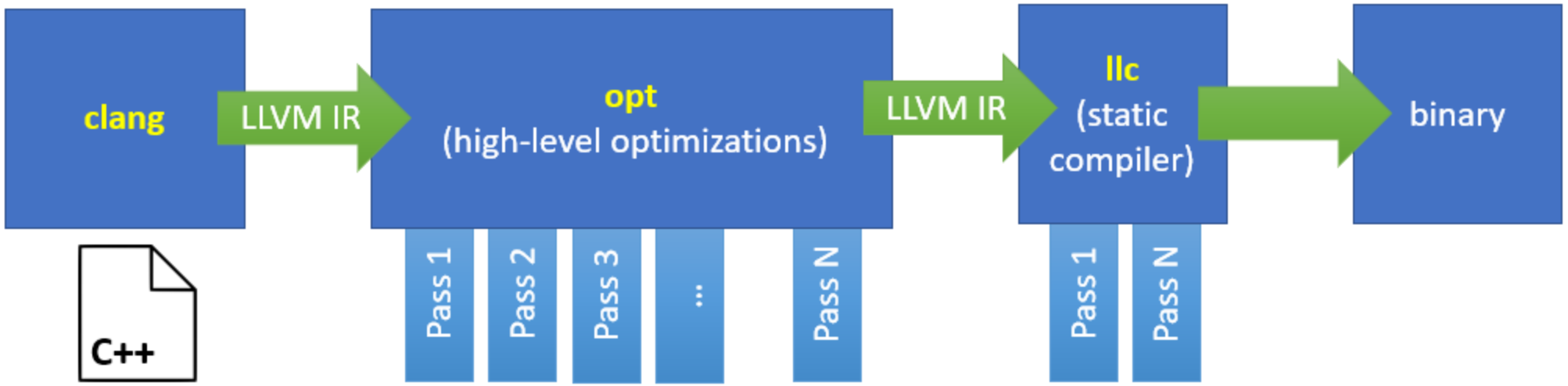
```
SEC_DEF("tracepoint+",  TRACEPOINT, 0, SEC_NONE, attach_tp),
```

[libbpf.c#L11591]

```
static int attach_tp(const struct bpf_program *prog, long cookie, struct bpf_link **link){
  // ...
}
```

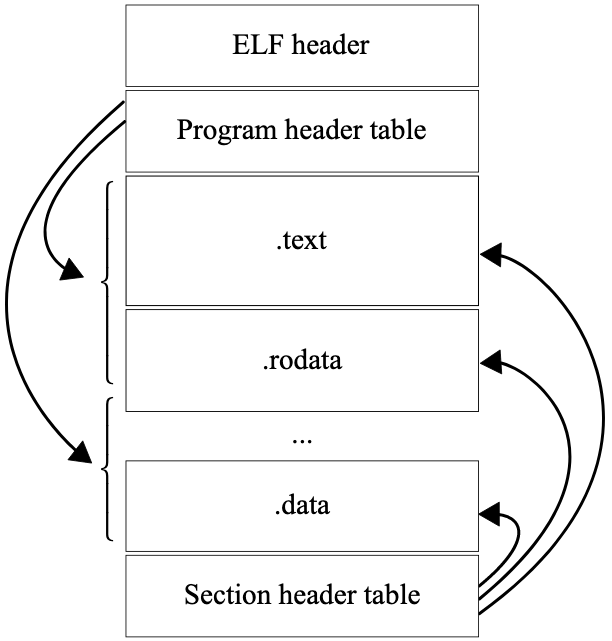
3. 编译

通过 Clang/LLVM 项目进行编译成字节码 (ELF 格式)



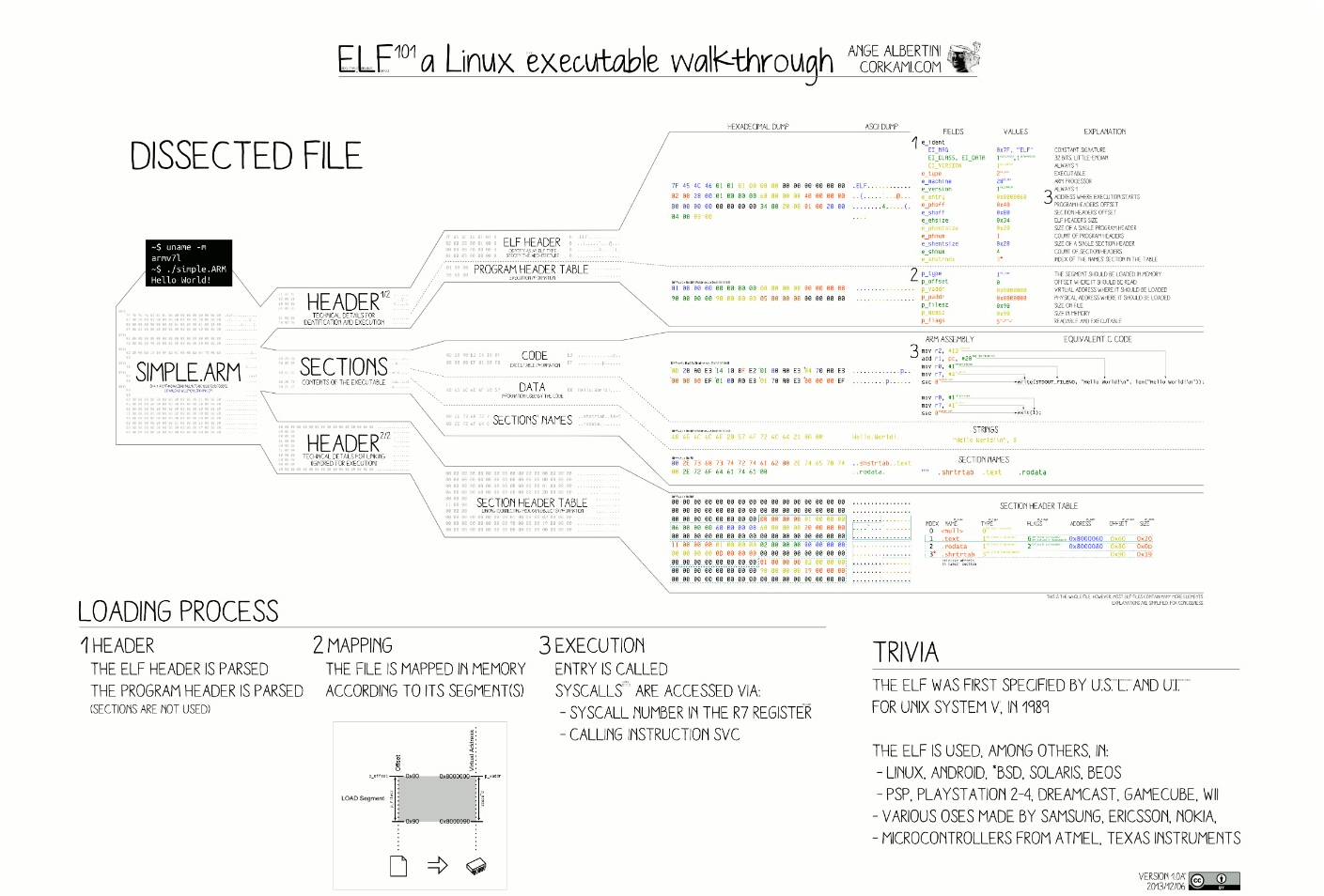

```
$ clang -target bpf -Wall -O2 -o hello_kern.o -c hello_kern.c
```

3. 编译 - ELF



来自： https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

3. 编译 - ELF


ELF 101 a Linux executable walk-through
ANGE ALBERTINI
CORKAMI.COM
DISSECTED FILE
HEXADECIMAL DUMP
ASCII DUMP
FIELDS
VALUES
EXPLANATION
~$ uname -m
armv7l
~$ ./simple.ARM
Hello World!
SIMPLE.ARM
HEADER 1/2
TECHNICAL DETAILS FOR IDENTIFICATION AND EXECUTION
ELF HEADER
PROGRAM HEADER TABLE
SECTIONS
CONTENTS OF THE EXECUTABLE
CODE
DATA
SECTIONS' NAMES
SECTION HEADER TABLE
HEADER 2/2
TECHNICAL DETAILS FOR LINKING
IGNORED FOR EXECUTION
ARM ASSEMBLY
EQUIVALENT C CODE
STRINGS
SECTION NAMES
SECTION HEADER TABLE
LOADING PROCESS
1 HEADER
THE ELF HEADER IS PARSED
THE PROGRAM HEADER IS PARSED
(SECTIONS ARE NOT USED)
2 MAPPING
THE FILE IS MAPPED IN MEMORY
ACCORDING TO ITS SEGMENT(S)
LOAD Segment
Offset
Virtual Address
0x00
0x80000000
0x90
0x80000090
3 EXECUTION
ENTRY IS CALLED
SYSCALLS ARE ACCESSED VIA:
- SYSCALL NUMBER IN THE R7 REGISTER
- CALLING INSTRUCTION SVC
TRIVIA
THE ELF WAS FIRST SPECIFIED BY U.S.L. AND U.I.
FOR UNIX SYSTEM V, IN 1989
THE ELF IS USED, AMONG OTHERS, IN:
- LINUX, ANDROID, *BSD, SOLARIS, BEOS
- PSP, PLAYSTATION 2-4, DREAMCAST, GAMECUBE, WII
- VARIOUS OSES MADE BY SAMSUNG, ERICSSON, NOKIA,
- MICROCONTROLLERS FROM ATMEL, TEXAS INSTRUMENTS
VERSION 1.04
2013/12/06

3. 编译 - Section

```
dwh0403@ubuntu:~/Desktop/hello_bpf/hello_basic$ llvm-readelf -S hello_kern.o
There are 8 section headers, starting at offset 0x1a8:

Section Headers:
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            0000000000000000 000000 000000 00      0   0  0
  [ 1] .strtab           STRTAB          0000000000000000 00012a 000078 00      0   0  1
  [ 2] .text             PROGBITS        0000000000000000 000040 000000 00  AX  0   0  4
  [ 3] tracepoint/syscalls/sys_enter_execve PROGBITS 0000000000000000 000040 000070 00  AX  0   0  8
  [ 4] .rodata.str1.1    PROGBITS        0000000000000000 0000b0 000012 01 AMS  0   0  1
  [ 5] license           PROGBITS        0000000000000000 0000c2 000004 00  WA  0   0  1
  [ 6] .llvm_addrsig     LLVM_ADDRSIG    0000000000000000 000128 000002 00   E  7   0  1
  [ 7] .symtab           SYMTAB          0000000000000000 0000c8 000060 18      1   2  8
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  R (retain), p (processor specific)
```

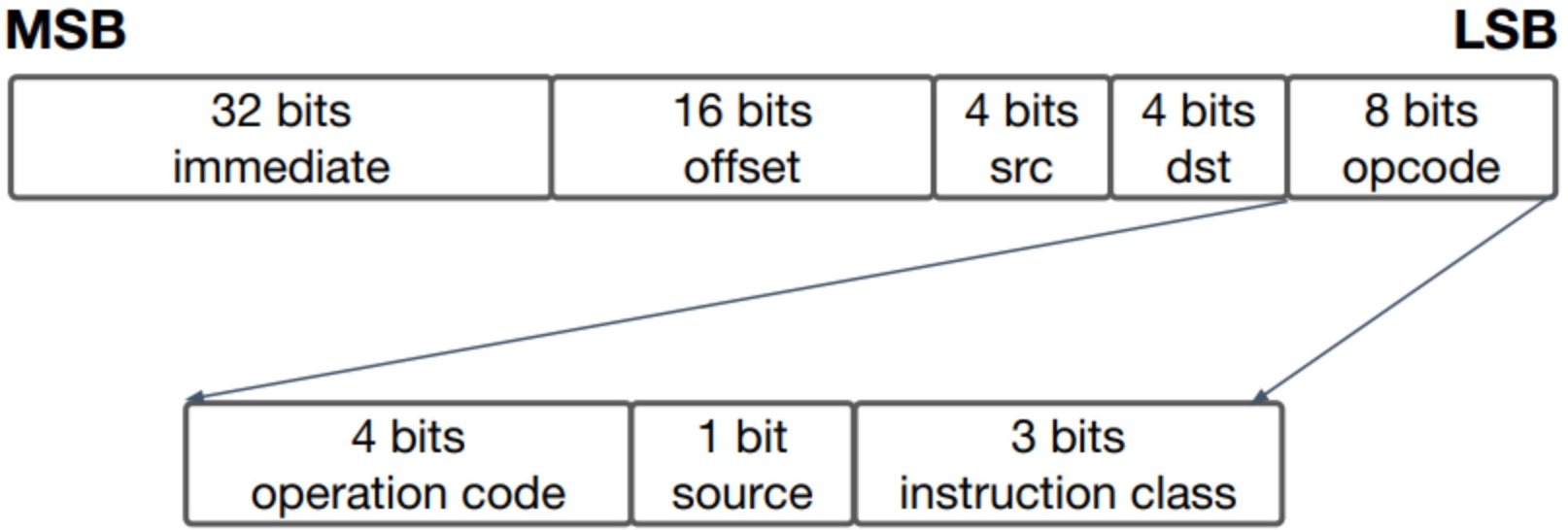
3. 编译 - Symtab

```
dwh0403@ubuntu:~/Desktop/hello_bpf/hello_basic$ llvm-readelf -S -s hello_kern.o
There are 8 section headers, starting at offset 0x1a8:

Section Headers:
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            0000000000000000 000000 000000 00      0   0  0
  [ 1] .strtab           STRTAB          0000000000000000 00012a 000078 00      0   0  1
  [ 2] .text             PROGBITS        0000000000000000 000040 000000 00  AX  0   0  4
  [ 3] tracepoint/syscalls/sys_enter_execve PROGBITS 0000000000000000 000040 000070 00  AX  0   0  8
  [ 4] .rodata.str1.1    PROGBITS        0000000000000000 0000b0 000012 01 AMS  0   0  1
  [ 5] license           PROGBITS        0000000000000000 0000c2 000004 00  WA  0   0  1
  [ 6] .llvm_addrsig     LLVM_ADDRSIG    0000000000000000 000128 000002 00   E  7   0  1
  [ 7] .symtab           SYMTAB          0000000000000000 0000c8 000060 18      1   2  8
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  R (retain), p (processor specific)

Symbol table '.symtab' contains 4 entries:
   Num:    Value          Size Type    Bind   Vis       Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT   UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT   ABS hello_kern.c
     2: 0000000000000000   112 FUNC    GLOBAL DEFAULT     3 bpf_prog
     3: 0000000000000000     4 OBJECT  GLOBAL DEFAULT     5 _license
```

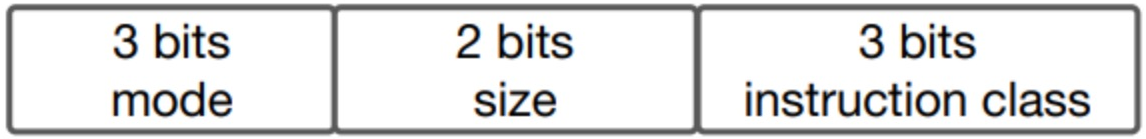
3. 编译 - Section Disassebly

```
dwh0403@ubuntu:~/Desktop/hello_bpf/hello_basic$ llvm-objdump -d hello_kern.o

hello_kern.o:   file format elf64-bpf

Disassembly of section tracepoint/syscalls/sys_enter_execve:

0000000000000000 <bpf_prog>:
       0:       b7 01 00 00 21 00 00 00 r1 = 33
       1:       6b 1a f8 ff 00 00 00 00 *(u16 *)(r10 - 8) = r1
       2:       18 01 00 00 50 46 20 57 00 00 00 00 6f 72 6c 64 r1 = 7236284523806213712 ll
       4:       7b 1a f0 ff 00 00 00 00 *(u64 *)(r10 - 16) = r1
       5:       18 01 00 00 48 65 6c 6c 00 00 00 00 6f 2c 20 42 r1 = 4764857262830019912 ll
       7:       7b 1a e8 ff 00 00 00 00 *(u64 *)(r10 - 24) = r1
       8:       bf a1 00 00 00 00 00 00 r1 = r10
       9:       07 01 00 00 e8 ff ff ff r1 += -24
      10:       b7 02 00 00 12 00 00 00 r2 = 18
      11:       85 00 00 00 06 00 00 00 call 6
      12:       b7 00 00 00 00 00 00 00 r0 = 0
      13:       95 00 00 00 00 00 00 00 exit
```

3. 编译 - eBPF 指令



Opcode for arithmetic and jump instructions



Opcode for memory instructions

3. 编译 - eBPF 寄存器

- R0：函数调用的返回值，以及 eBPF 程序的退出值
- R1 - R5：函数调用的参数
- R6 - R9：被调用者保存的函数调用将保留的寄存器
- R10：访问堆栈的只读帧指针

> 在执行 `call` 之后，寄存器 **R1-R5 包含垃圾值，不能读取**

4. 加载

v5.10 bpf_load.c

早期版本的一个加载器原型，可以先阅读学习。

整体代码大概 700 行，功能有限，但是熟悉整体脉络相对容易。

```
int load_bpf_file(char *path)
{
        return do_load_bpf_file(path, NULL);
}
```

4. 加载 (libbpf 库)

```
int main() {
    struct bpf_link *link = NULL;
    struct bpf_program *prog;
    struct bpf_object *obj;

    obj = bpf_object__open_file("hello_kern.o", NULL); // ① 打开

    prog = bpf_object__find_program_by_name(obj, "bpf_prog"); // ② 获取 prog 对象

    bpf_object__load(obj) // ③ 加载对象

    link = bpf_program__attach(prog); // ④ 附着到对应的事件


cleanup:
    bpf_link__destroy(link);
    bpf_object__close(obj);
    return 0;
}
```

4. 加载 - Open 阶段

ELF => 提取出需要的信息 （eBPF 程序 && License）

```
static struct bpf_object *bpf_object_open(const char *path, const void *obj_buf, size_t obj_buf_sz,
                                          const struct bpf_object_open_opts *opts){ ------->
  err = bpf_object__elf_init(obj);
        err = err ? : bpf_object__check_endianness(obj);
        err = err ? : bpf_object__elf_collect(obj); // 通过不同的 section 及对应的属性进行处理，比如将 ebpf 程序初始化到 progs 变量中
        err = err ? : bpf_object__collect_externs(obj);
        err = err ? : bpf_object__finalize_btf(obj);
        err = err ? : bpf_object__init_maps(obj, opts);
        err = err ? : bpf_object_init_progs(obj, opts); // 初始化，通过 SEC name 查询到对应的程序类型和附着对应的函数
        err = err ? : bpf_object__collect_relos(obj);
}
```

4. 加载 - Load 阶段

将 eBPF 程序加载到内核空间

```
static int bpf_object_load(struct bpf_object *obj, int extra_log_level, const char *target_btf_path){
        err = bpf_object__probe_loading(obj);
        err = err ? : bpf_object__load_vmlinux_btf(obj, false);
        err = err ? : bpf_object__resolve_externs(obj, obj->kconfig);
        err = err ? : bpf_object__sanitize_and_load_btf(obj);
        err = err ? : bpf_object__sanitize_maps(obj);
        err = err ? : bpf_object__init_kern_struct_ops_maps(obj);
        err = err ? : bpf_object__create_maps(obj);
        err = err ? : bpf_object__relocate(obj, obj->btf_custom_path ? : target_btf_path);
        err = err ? : bpf_object__load_progs(obj, extra_log_level); // ====
        err = err ? : bpf_object_init_prog_arrays(obj);
}
```

4. 加载 - Load 阶段

```
static int
bpf_object__load_progs(struct bpf_object *obj, int log_level)
{
        for (i = 0; i < obj->nr_programs; i++) {
                err = bpf_object_load_prog(obj, prog, obj->license, obj->kern_version);
        }
}
```

底层通过 bpf() 将 BPF 程序加载到内核中。

4. 加载 - Attach 阶段

将已经加载到内核的 eBPF 与对应的事件建立关系

```
struct bpf_link *bpf_program__attach(const struct bpf_program *prog)
{
        struct bpf_link *link = NULL;
        // ...
        err = prog->sec_def->prog_attach_fn(prog, prog->sec_def->cookie, &link);
        // ...
        return link;
}
```

```
SEC_DEF("tracepoint+",  TRACEPOINT, 0, SEC_NONE, attach_tp),
```

4. 加载 - Attach 阶段

```
static int attach_tp(const struct bpf_program *prog, long cookie, struct bpf_link **link)
{

        /* extract "tp/<category>/<name>" or "tracepoint/<category>/<name>" */
        if (str_has_pfx(prog->sec_name, "tp/"))
                tp_cat = sec_name + sizeof("tp/") - 1;
        else
                tp_cat = sec_name + sizeof("tracepoint/") - 1;
        tp_name = strchr(tp_cat, '/');

        *tp_name = '\0';
        tp_name++;

        *link = bpf_program__attach_tracepoint(prog, tp_cat, tp_name);
        return libbpf_get_error(*link);
}
```

4. 加载 - Attach 阶段

`bpf_program__attach_tracepoint`

=> `bpf_program__attach_tracepoint_opts`

```
struct bpf_link *bpf_program__attach_tracepoint_opts(const struct bpf_program *prog,
                                                     const char *tp_category,
                                                     const char *tp_name,
                                                     const struct bpf_tracepoint_opts *opts)
{
        pfd = perf_event_open_tracepoint(tp_category, tp_name);
        link = bpf_program__attach_perf_event_opts(prog, pfd, &pe_opts);
}
```

4. 加载 - Attach 阶段

```
struct bpf_link *bpf_program__attach_perf_event_opts(const struct bpf_program *prog, int pfd,
                                                     const struct bpf_perf_event_opts *opts)
{
        prog_fd = bpf_program__fd(prog);
        link = calloc(1, sizeof(*link));

        link->perf_event_fd = pfd;

        if (kernel_supports(prog->obj, FEAT_PERF_LINK)) {
                link_fd = bpf_link_create(prog_fd, pfd, BPF_PERF_EVENT, &link_opts);
                link->link.fd = link_fd;
        } else {
                ioctl(pfd, PERF_EVENT_IOC_SET_BPF, prog_fd)
                link->link.fd = pfd;
        }

        ioctl(pfd, PERF_EVENT_IOC_ENABLE, 0);
        return &link->link;
}
```

5. 验证

```
static int bpf_prog_load(union bpf_attr *attr, bpfptr_t uattr)
{
        /* GPL 兼容检测 */
        /* 指令是否超出限制，非特权 4096 条，特权 100w 条指令 */
        /* 操作网络类型需要  CAP_NET_ADMIN & CAP_SYS_ADMIN 权限 */
        /* 跟踪类型 KPROBE/TRACEPOINT  CAP_PERFMON) || CAP_SYS_ADMIN */

        /* run eBPF verifier */
        err = bpf_check(&prog, attr, uattr);

        /* 底层运行环境  JIT 还是 Interpreter */
        /* bpf_prog_run(prog, ctx) */
        prog = bpf_prog_select_runtime(prog, &err);
}
```

5. 验证 -- bpf_check()

```
int bpf_check(struct bpf_prog **prog, union bpf_attr *attr, bpfptr_t uattr)
{
        env->ops = bpf_verifier_ops[env->prog->type]; // ① 基于程序类型获取检查函数

        // ld_imm64 instructions: if it accesses map FD, replace it with actual map pointer.
        ret = resolve_pseudo_ldimm64(env);

        // 第 1 遍检测 是否包含非预期的循环指令，是否有不可达指令，是否越界
        ret = check_cfg(env);

        // 第 2 遍检测 模拟整个运行过程，特别是对于指针访问的每个细节进行检查
        ret = do_check_subprogs(env);
        ret = ret ?: do_check_main(env);

        if (ret == 0)
                ret = fixup_call_args(env); // 将调用指令 call 6 调整为真实函数地址

        // ...
}
```

5. 验证

```
const struct bpf_verifier_ops tracepoint_verifier_ops = {
        .get_func_proto  = tp_prog_func_proto,
        .is_valid_access = tp_prog_is_valid_access,
};
```

5. 验证

```
static const struct bpf_func_proto *
tp_prog_func_proto(enum bpf_func_id func_id, const struct bpf_prog *prog)
{
        switch (func_id) {
        case BPF_FUNC_perf_event_output:
                return &bpf_perf_event_output_proto_tp;
        case BPF_FUNC_get_stackid:
                return &bpf_get_stackid_proto_tp;
        case BPF_FUNC_get_stack:
                return &bpf_get_stack_proto_tp;
        case BPF_FUNC_get_attach_cookie:
                return &bpf_get_attach_cookie_proto_trace;
        default:
                return bpf_tracing_func_proto(func_id, prog);
        }
}
```

5. 验证

```
libbpf: -- BEGIN DUMP LOG ---
libbpf:
R1 type=ctx expected=fp
; int bpf_prog1(struct pt_regs *ctx)
0: (bf) r3 = r1
1: (b7) r1 = 112
2: (0f) r3 += r1
last_idx 2 first_idx 0
regs=2 stack=0 before 1: (b7) r1 = 112
3: (bf) r1 = r10
;
4: (07) r1 += -8
; struct sk_buff *skb = (struct sk_buff *)PT_REGS_PARM1_CORE(ctx);
5: (b7) r2 = 8
6: (85) call bpf_probe_read_kernel#113
last_idx 6 first_idx 0
regs=4 stack=0 before 5: (b7) r2 = 8
7: (b7) r1 = 32
; struct sk_buff *skb = (struct sk_buff *)PT_REGS_PARM1_CORE(ctx);
8: (79) r6 = *(u64 *)(r10 -8)
9: (bf) r3 = r6
10: (0f) r3 += r1
; bpf_probe_read_kernel(&flow->time, sizeof(flow->time), &skb->tstamp);
11: (b7) r2 = 8
12: (85) call bpf_probe_read_kernel#113
R1 type=inv expected=fp, pkt, pkt_meta, map_key, map_value, mem, rdonly_buf, rdwr_buf
processed 13 insns (limit 1000000) max_states_per_insn 0 total_states 0 peak_states 0 mark_read 0

libbpf: -- END LOG --
libbpf: failed to load program 'bpf_prog1'
libbpf: failed to load object 'execskel'
libbpf: failed to load BPF skeleton 'execskel': -4007
libbpf: Can't get the 0th fd from program bpf_prog1: only -1 instances
libbpf: prog 'bpf_prog1': can't attach BPF program w/o FD (did you load it?)
libbpf: prog 'bpf_prog1': failed to attach to kprobe 'kfree_skb': Invalid argument
libbpf: failed to auto-attach program 'bpf_prog1': -22
^C
l0calh0st@l0calh0st:~/kprobe2$  g g
```

5. 验证

```
18 SEC("kprobe/kfree_skb")
17 int bpf_prog1(struct pt_regs *ctx)
16 {
15     struct flow_value_record *flow ;
14     struct net_device *dev = NULL;
13     unsigned char *data;
12     int users;
11     char pkg_type;
10
 9     struct sk_buff *skb = (struct sk_buff *)PT_REGS_PARM1_CORE(ctx);
 8     bpf_probe_read_kernel(&flow->time, sizeof(flow->time), &skb->tstamp);
 7
 6     // this can be replace with bpf_probe_read_kernel() / or bpf_core_read()
 5     bpf_probe_read_kernel(&users, sizeof(users), &skb->users.refs.counter);
 4
 3     // read device
 2     // bpf_printk("debug devname %s", dev_name);
```

6. 触发运行

eBPF 支持 Tracepoint 4.7 内核版本，[98b5c2c65c29 Github]

```
diff --git a/include/trace/perf.h b/include/trace/perf.h
index 77cd9043b7e47..a182306eefd7a 100644
--- a/include/trace/perf.h
+++ b/include/trace/perf.h
@@ -34,6 +34,7 @@ perf_trace_##call(void *__data, proto)
        struct trace_event_call *event_call = __data;                   \
        struct trace_event_data_offsets_##call __maybe_unused __data_offsets;\
        struct trace_event_raw_##call *entry;                           \
+       struct bpf_prog *prog = event_call->prog;                       \
        struct pt_regs *__regs;                                         \
        u64 __count = 1;                                                \
        struct task_struct *__task = NULL;                              \
@@ -45,7 +46,7 @@ perf_trace_##call(void *__data, proto)
        __data_size = trace_event_get_offsets_##call(&__data_offsets, args); \
                                                                        \
        head = this_cpu_ptr(event_call->perf_events);                   \
-       if (__builtin_constant_p(!__task) && !__task &&                 \
+       if (!prog && __builtin_constant_p(!__task) && !__task &&        \
                                hlist_empty(head))                      \
                return;                                                 \
                                                                        \
@@ -63,6 +64,13 @@ perf_trace_##call(void *__data, proto)
                                                                        \
        { assign; }                                                     \
                                                                        \
+       if (prog) {                                                     \
+               *(struct pt_regs **)entry = __regs;                     \
+               if (!trace_call_bpf(prog, entry) || hlist_empty(head)) { \
+                       perf_swevent_put_recursion_context(rctx);       \
+                       return;                                         \
+               }                                                       \
+       }                                                               \
        perf_trace_buf_submit(entry, __entry_size, rctx,                \
```

6. 触发运行

+---------+
| 8 bytes | hidden 'struct pt_regs *' (inaccessible to bpf program)
+---------+
| N bytes | static tracepoint fields defined in tracepoint/format (bpf readonly)
+---------+
| dynamic | (inaccessible to bpf yet)
+---------+

```
$ sudo cd /sys/kernel/debug/tracing/events
$ sudo cat /syscalls/sys_enter_execve/format
name: sys_enter_execve
ID: 716

format:
        field:unsigned short common_type;       offset:0;       size:2; signed:0;
        field:unsigned char common_flags;       offset:2;       size:1; signed:0;
        field:unsigned char common_preempt_count;       offset:3;       size:1; signed:0;
        field:int common_pid;   offset:4;       size:4; signed:1;

        field:int __syscall_nr; offset:8;       size:4; signed:1;
        field:const char * filename;    offset:16;      size:8; signed:0;
        field:const char *const * argv; offset:24;      size:8; signed:0;
        field:const char *const * envp; offset:32;      size:8; signed:0;


print fmt: "filename: 0x%08lx, argv: 0x%08lx, envp: 0x%08lx",
                                        ((unsigned long)(REC->filename)),
                                        ((unsigned long)(REC->argv)),
                                        ((unsigned long)(REC->envp))
```

```
struct tracepoint__syscalls__sys_enter_execv {
        u64 __do_not_use__; // for ctx
        int nr;
        const char * filename;
        const char * argv;
        const char * envp;
};


SEC("tracepoint/syscalls/sys_enter_execve")
int bpf_prog(struct tracepoint__syscalls__sys_enter_execv *ctx) {
  char msg[] = "Hello, BPF World!";
  bpf_trace_printk(msg, sizeof(msg));

  return 0;
}
```

7. map 变身记

```
struct bpf_map_def {
        unsigned int type;
        unsigned int key_size;
        unsigned int value_size;
        unsigned int max_entries;
        unsigned int map_flags;
};

struct bpf_map_def SEC("maps") my_map = {
    .type = BPF_MAP_TYPE_ARRAY,
    .key_size = sizeof(__u32),
    .value_size = sizeof(long),
    .max_entries = 256,
};
```

7. map 变身记

```c
SEC("tracepoint/syscalls/sys_enter_execve")
int bpf_prog(struct tracepoint__syscalls__sys_enter_execv *ctx) {
  __u32 key = 0;
  long value = 1;

  bpf_map_update_elem(&my_map, &key, &value, BPF_ANY); // 本次新添加

  return 0;
}
```

```
$ llvm-readelf -S -r -s -x maps hello_kern.o
There are 9 section headers, starting at offset 0x1d0:

Section Headers:
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  ...
  [ 3] tracepoint/syscalls/sys_enter_execve PROGBITS 0000000000000000 000040 000070 00  AX  0   0  8
  [ 4] .reltracepoint/syscalls/sys_enter_execve REL 0000000000000000 000140 000010 10   I  8   3  8
  [ 5] maps              PROGBITS        0000000000000000 0000b0 000014 00  WA  0   0  4
  [ 6] license           PROGBITS        0000000000000000 0000c4 000004 00  WA  0   0  1
  ...

# 符号表
Symbol table '.symtab' contains 5 entries:
   Num:    Value          Size Type    Bind   Vis       Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT   UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT   ABS hello_kern.c
     2: 0000000000000000   112 FUNC    GLOBAL DEFAULT     3 bpf_prog
     3: 0000000000000000    20 OBJECT  GLOBAL DEFAULT     5 my_map       # 符号表, UND 5 Section Header maps
     4: 0000000000000000     4 OBJECT  GLOBAL DEFAULT     6 _license

# maps 数据区
Hex dump of section 'maps':
0x00000000 02000000 04000000 08000000 00010000 ................
0x00000010 00000000

Relocation section '.reltracepoint/syscalls/sys_enter_execve' at offset 0x140 contains 1 entries:
    Offset             Info             Type               Symbol's Value  Symbol's Name
0000000000000040  00000003 00000001 R_BPF_64_64            0000000000000000 my_map
  代码偏移量        symtab idx                                 偏移量
```

```
$ llvm-objdump -r -S hello_kern_debug.o

hello_kern_debug.o:     file format elf64-bpf

Disassembly of section tracepoint/syscalls/sys_enter_execve:

0000000000000000 <bpf_prog>:
; int bpf_prog(void *ctx) {
       0:       b7 01 00 00 00 00 00 00 r1 = 0
;   __u32 key = 0;
       1:       63 1a fc ff 00 00 00 00 *(u32 *)(r10 - 4) = r1
       2:       b7 01 00 00 01 00 00 00 r1 = 1
;   long value = 1;
       3:       7b 1a f0 ff 00 00 00 00 *(u64 *)(r10 - 16) = r1
       4:       bf a2 00 00 00 00 00 00 r2 = r10
       5:       07 02 00 00 fc ff ff ff r2 += -4
       6:       bf a3 00 00 00 00 00 00 r3 = r10
       7:       07 03 00 00 f0 ff ff ff r3 += -16
;   bpf_map_update_elem(&my_map, &key, &value, BPF_ANY);
       8:       18 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 r1 = 0 ll
                0000000000000040:  R_BPF_64_64  my_map  # 重定位符号表
      10:       b7 04 00 00 00 00 00 00 r4 = 0
      11:       85 00 00 00 02 00 00 00 call 2
;   return 0;
      12:       b7 00 00 00 00 00 00 00 r0 = 0
      13:       95 00 00 00 00 00 00 00 exit
```

7. map 变身记

```c
// 用户空间 map 查询函数
int bpf_map_lookup_elem(int fd, const void *key, void *value)


// 内核中 BPF 辅助函数 map 查询函数
void *bpf_map_lookup_elem(struct bpf_map *map, const void *key)
```

详情可参考： https://www.ebpf.top/post/map_internal/

7. map 变身记

第一次变身： 加载器 libbpf.c

```
static int bpf_object_load(struct bpf_object *obj, int extra_log_level, const char *target_btf_path){
        // ...
        err = err ? : bpf_object__create_maps(obj);

        err = err ? : bpf_object__relocate(obj, obj->btf_custom_path ? : target_btf_path);
                    //  => bpf_object__relocate_data()

        err = err ? : bpf_object__load_progs(obj, extra_log_level);
}
```

第一次变身： 加载器 libbpf.c

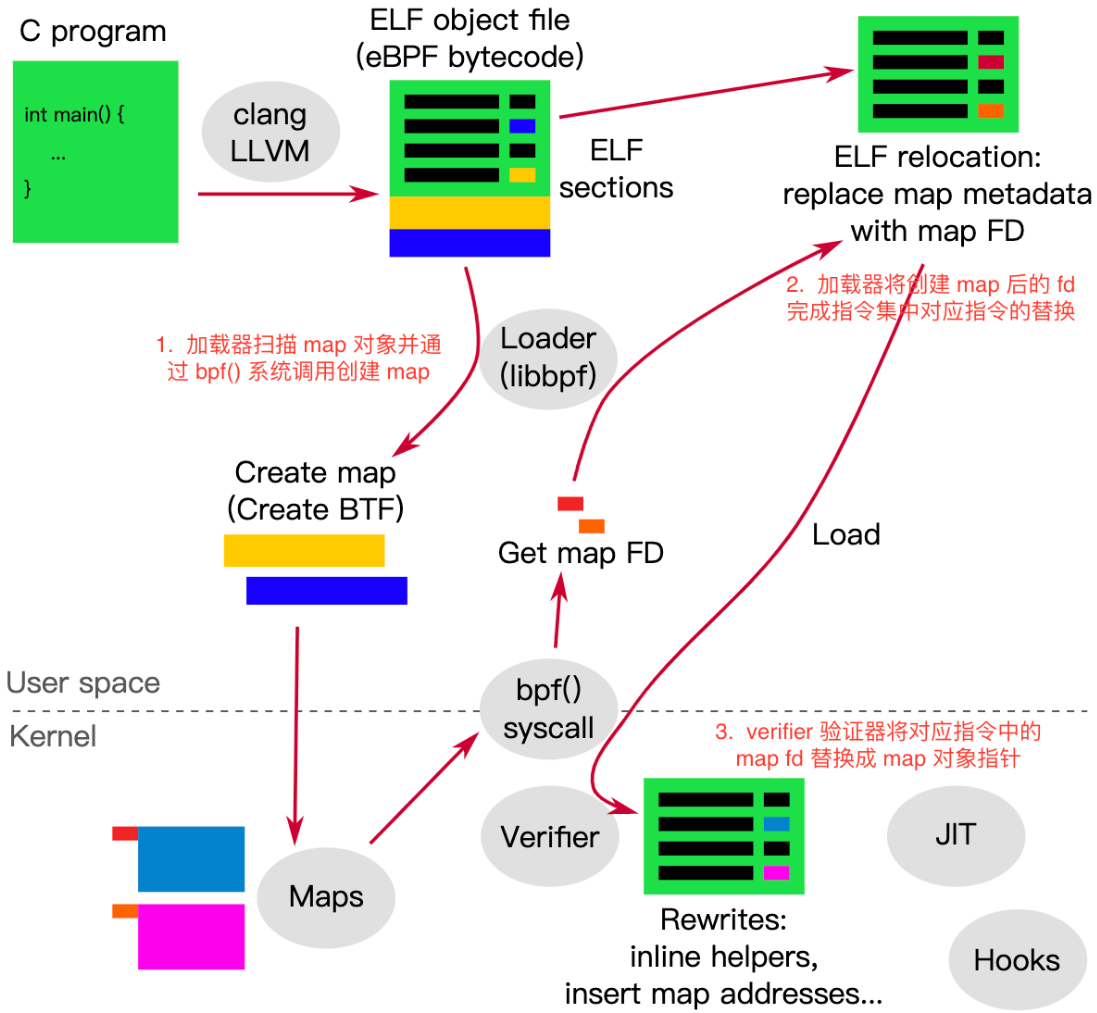
```
static int
bpf_object__relocate_data(struct bpf_object *obj, struct bpf_program *prog)
{
        for (i = 0; i < prog->nr_reloc; i++) {
                // ...
                switch (relo->type) {
                case RELO_LD64:
                        map = &obj->maps[relo->map_idx];
                        if (map->autocreate) {
                                insn[0].src_reg = BPF_PSEUDO_MAP_FD;
                                insn[0].imm = map->fd;
                        }
                }
        }
}
```

第二次变身：内核验证器 verifier.c

```
int bpf_check(struct bpf_prog **prog, union bpf_attr *attr, bpfptr_t uattr)
{
        env->prog = *prog;

        // ld_imm64 instructions:
        // if it accesses map FD, replace it with actual map pointer.
        ret = resolve_pseudo_ldimm64(env);

        // ...
}
```

```
static int resolve_pseudo_ldimm64(struct bpf_verifier_env *env)
{
    for (i = 0; i < insn_cnt; i++, insn++)
    {
      //...
      if (insn[0].src_reg == BPF_PSEUDO_MAP_FD ||
                insn[0].src_reg == BPF_PSEUDO_MAP_IDX) {
                addr = (unsigned long)map;

                // ...
                insn[0].imm = (u32)addr;
                insn[1].imm = addr >> 32;

                // ...
        }
    }
    // ...
}
```

7. map 变身记


C program
int main() {
...
}
clang
LLVM
ELF object file
(eBPF bytecode)
ELF
sections
ELF relocation:
replace map metadata
with map FD
1. 加载器扫描 map 对象并通
过 bpf() 系统调用创建 map
Loader
(libbpf)
2. 加载器将创建 map 后的 fd
完成指令集中对应指令的替换
Create map
(Create BTF)
Get map FD
Load
User space
Kernel
bpf()
syscall
3. verifier 验证器将对应指令中的
map fd 替换成 map 对象指针
Verifier
JIT
Maps
Rewrites:
inline helpers,
insert map addresses...
Hooks

总结

工作原理 4 步骤

- 编译
- 加载
- 验证
- 运行

总结

加载器 3 过程

- Open ELF -> Mem
- Load Mem -> Kernel
- Attach Event <--> Kernel

总结

思考题：

```
  char msg[] = "Hello, BPF World!";  // ⑤ 大小?  是否可以使用 const char *msg = ""
```

感谢聆听！！