eBPF 技术和开发框架简介

https://github.com/eunomia-bpf

于桐 2023.1

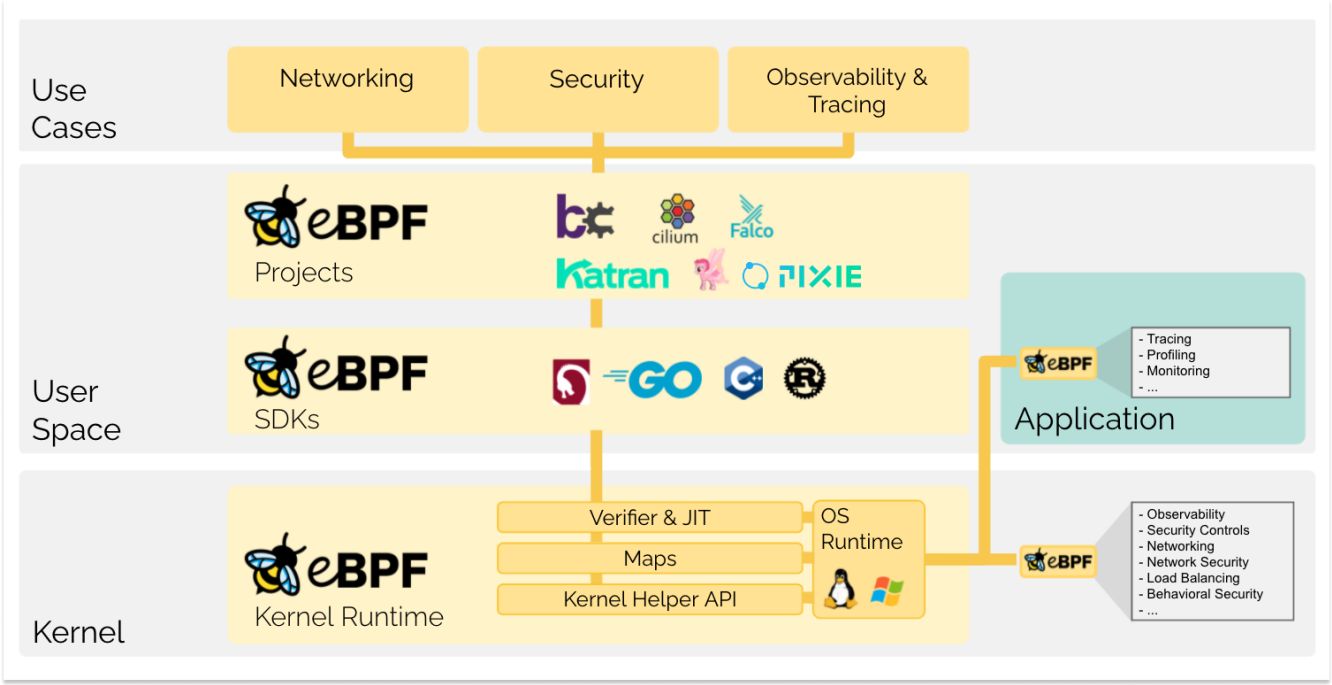
目录

- eBPF是什么
- eBPF程序和内核模块的区别
- eBPF程序能做什么
- eBPF程序如何跟用户态程序通信（eBPF Maps）
- eBPF常见开发方式

eBPF是什么

- eBPF: Extended Berkeley Packet Filter
- 用比较平白的话来说的话：
- 在Linux内核里有一个能跑字节码的虚拟机
- 我们可以在操作系统跑着的时候给这个虚拟机塞一个编译好的程序
- 然后这个程序可以做诸如监听或者修改某些内核函数的入参和返回值，监听系统调用，监听网卡输入的数据包之类的事情，并且可以通过一些手段和用户态程序通信。
- 内核有个验证器会在eBPF程序运行之前先verify一下要跑的程序，这个验证器会保证eBPF程序能在有限的指令数内终止（也因此eBPF程序不是图灵完全的）、不会访问不该访问的内存等

eBPF的结构



- eBPF的程序大致分为内核程序和用户态程序两部分
- 内核程序会经过内核提供的验证器和JIT处理，可以访问eBPF Map（下文会说），可以访问一些限定的内核函数，同时内核程序可以进行对内核中的各种东西进行观测统计之类的任务，而后把得到的原始数据通过eBPF Map传递给用户态程序
- 用户态程序可以使用Go/C++/Rust等各种语言进行开发。然后它负责从eBPF Map中读取内核程序传递的原始数据而后进行处理以便进行跟踪、性能分析、监控等操作。

和内核模块有点类似？

- 内核模块也可以在内核运行的时候加载和卸载，并且同样可以修改内核之类的，但
- 内核模块是native程序（所以显然一个内核模块编译好了后不跨平台），并且在被加载进内核后可以操作内核里所有的东西（比如调用任意函数修改任意内存），所以在写烂了的情况下会出问题（比如kernel panic）
- 而eBPF程序是的目标格式是一种平台无关的指令集，且其二进制格式同样是跨平台的。
- eBPF程序所能做的事较内核模块而言比较受限（只能监听事件和调用一些特定的内核函数(见 man BPF(2))等），相对于内核模块比较安全。
- https://www.ebpf.top/what-is-ebpf/content/2.changing-the-kernel-is-hard.html

eBPF程序能做什么

- 一般而言，eBPF程序会在一些事件触发的时候被执行。常见的事件包括：
- 进入或退出某个内核函数（kprobs/kretprobs/fentry/fexit）
- 执行到某个tracepoint（内核定义的跟踪点，可能在某个内核函数的某一行代码上）
- 监听网络接口XDP（网卡收到数据包后执行，eBPF程序可以监听或修改数据包）
- ...

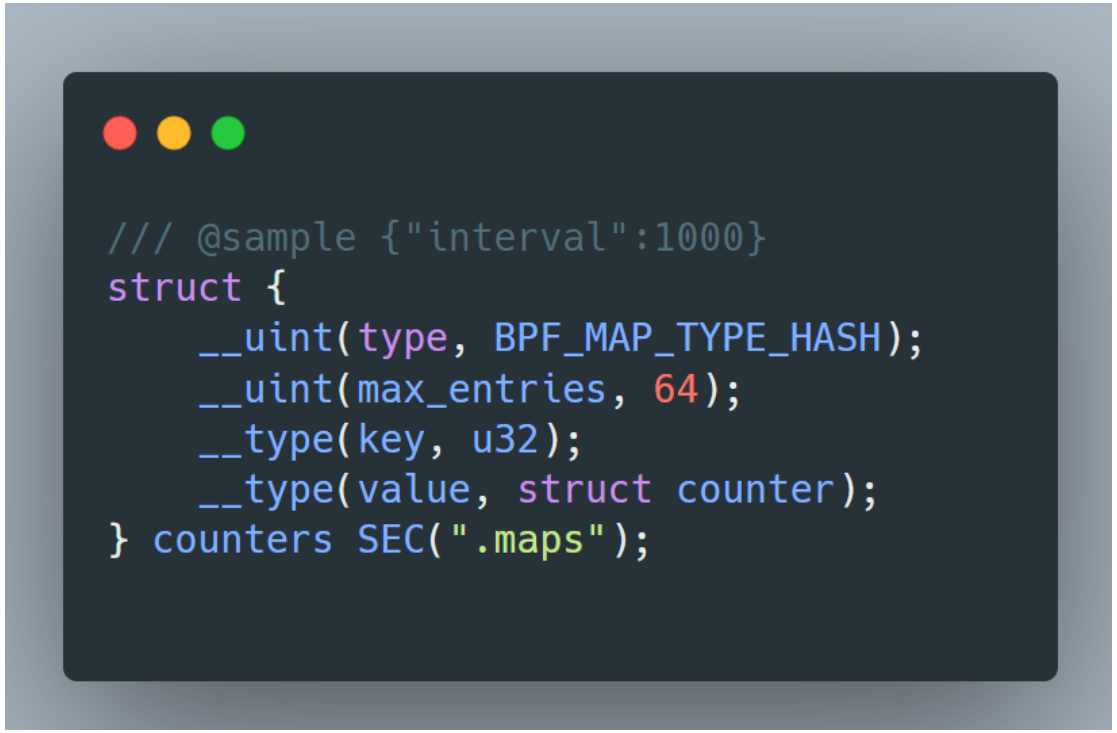
和用户程序通信？

- 我们已经知道eBPF程序是跑在内核里的了。但是考虑到eBPF程序所能执行的功能很受限（只能访问特定的数据、调用特定的函数），所以存在一些比较方便的让eBPF程序和用户态程序进行通信的方法。

eBPF Map

- eBPF Map可以被用来实现eBPF程序和用户程序的数据交换。
- 从名字可以看出来，他是个映射（键值对容器，类似于C++的std::map，Java的TreeMap/HashMap，Python的dict，Rust的BTreeMap/HashMap等）。
- 但eBPF Map相比于上一句所说的其他语言的键值对容器有些不同：
- 元素数有上限（保证eBPF程序的安全性，防止插爆内存）
- 键和值不区分具体类型，而都是（在创建这个Map的时候）就定好大小的字节串（反正写这么底层的东西了也没必要考虑比较高级的类型）。
- 然后，虽然叫Map，但底层实现可以是哈希表/数组等（在创建的时候指定）
- 用户态程序通过 bpf(2) 系统调用来创建删除和更新eBPF Map.
- eBPF程序通过一些特定的内核函数来操作eBPF Map

eBPF Map


/// @sample {"interval":1000}
struct {
__uint(type, BPF_MAP_TYPE_HASH);
__uint(max_entries, 64);
__type(key, u32);
__type(value, struct counter);
} counters SEC(".maps");

关于eBPF Map的更多信息

- 一些常见的内核支持的Map类型：
- BPF_MAP_TYPE_HASH：实现是哈希表，键和值可以是任意长度。只能在用户态程序创建。
- BPF_MAP_TYPE_ARRAY：实现是个数组。键必须是4个字节，键同时也作为数组下标。不支持删除元素。
- BPF_MAP_TYPE_PROG_ARRAY：也是个数组，键也必须是4字节。但是用途是拿来做eBPF程序跳转（从一个eBPF程序跳到另一个去执行，但保留栈）。由于eBPF程序的标识符（一个文件描述符）是四个字节，所以程序数组的值大小必须是四个字节。
- 还有其他好多，具体见 https://github.com/iovisor/bcc/blob/v0.20.0/docs/kernel-versions.md#tables-aka-maps
- man pages bpf(2)关于map type的内容有些过时了（似乎man的内容是4.2时代的）

怎么写eBPF程序

- 在用户态，Linux给了我们bpf(2)用来做几乎所有的和eBPF有关的操作，比如：
- 往内核里加载一个eBPF程序（提供程序的字节码和一些其他信息，返回一个文件描述符）
- 创建或销毁一个eBPF Map，或者往一个eBPF Map里添加删除更新元素
- 以及其他的很多东西，具体见 https://github.com/libbpf/libbpf/blob/3423d5e7cdab356d115aef7f987b4a1098ede448/include/uapi/linux/bpf.h#L131

- 所以最本真的写eBPF程序的方式是：
- 手写字节码（其实也没那么本真，还是有些helper macro的）
- 用bpf(2)把写的字节码塞进内核

- 高级一点的eBPF开发方式都是这种方式的封装。

使用bpf(2)加载eBPF程序

使用bpf(2)加载eBPF程序

- man bpf(2)中给了我们一个例子。
- 不过这个例子有点问题（有些函数名字过时了），没问题的可以看看内核例子 https://elixir.bootlin.com/linux/v5.4/source/samples/bpf/sock_example.c
- 这个例子需要libbpf才能运行。
- （libbpf提供了很多在用户态操作bpf的工具函数）
- （所以其实里面一堆bpf_xxxx的函数都是bpf(2)的封装）

使用bpf(2)加载eBPF程序

- 上文提供的例子是一个用户态程序（不过得用root跑）大概做了以下事情：
- 创建一个BPF_MAP_TYPE_ARRAY
- 构造一个装有BPF指令的数组（即所谓的手写BPF指令），并且把上面map的fd传进去
- 使用bpf_load_program加载这个程序到内核并且获取到对应的fd
- （由于上文写的BPF程序的作用是统计网卡收到的tcp、udp、icmp包数，所以）使用setsockopt(2)把加载的eBPF程序绑到lo上
- 然后开始疯狂ping 127.0.0.1，就可以从Map里读到eBPF程序统计的包个数。

使用bpf(2)加载的字节码

```
struct bpf_insn prog[] = {
        BPF_MOV64_REG(BPF_REG_6, BPF_REG_1),
        BPF_LD_ABS(BPF_B, ETH_HLEN + offsetof(struct iphdr, protocol) /* R0 = ip->proto */),
        BPF_STX_MEM(BPF_W, BPF_REG_10, BPF_REG_0, -4), /* *(u32 *)(fp - 4) = r0 */
        BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),
        BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -4), /* r2 = fp - 4 */
        BPF_LD_MAP_FD(BPF_REG_1, map_fd),
        BPF_RAW_INSN(BPF_JMP | BPF_CALL, 0, 0, 0, BPF_FUNC_map_lookup_elem),
        BPF_JMP_IMM(BPF_JEQ, BPF_REG_0, 0, 2),
        BPF_MOV64_IMM(BPF_REG_1, 1), /* r1 = 1 */
        BPF_RAW_INSN(BPF_STX | BPF_XADD | BPF_DW, BPF_REG_0, BPF_REG_1, 0, 0), /* xadd r0 += r1 */
        BPF_MOV64_IMM(BPF_REG_0, 0), /* r0 = 0 */
        BPF_EXIT_INSN(),
};
```

更好的开发方式

- 使用字节码开发eBPF程序很让人不舒服，所以大家想要用高级一点的语言（比如C，虽然没那么高级）来写eBPF内核程序。
- 接下来我们会介绍两种两种可以用C来写eBPF内核程序的开发方式:
- bcc(BPF Compilers Collection)
- libbpf与libbpf-bootstrap脚手架
- 以及一个对libbpf更高级的封装（eunomia-bpf）

libbpf

https://github.com/libbpf/libbpf

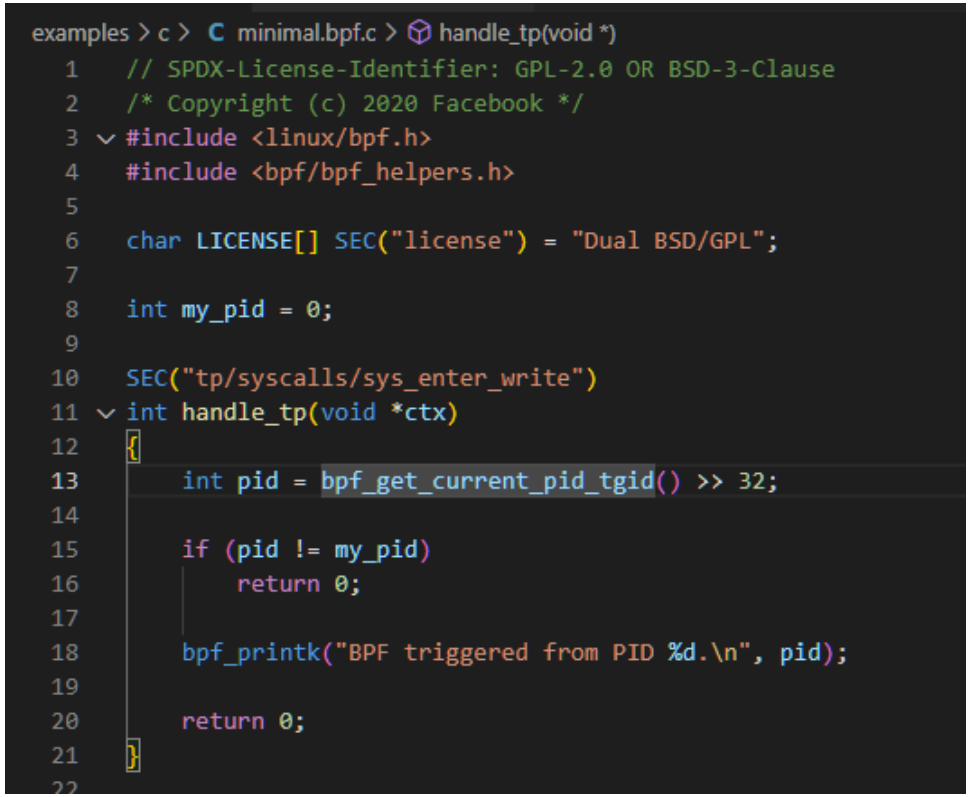
https://github.com/libbpf/libbpf-bootstrap

使用libbpf-bootstrap

- 我不喜欢直接写字节码，大家也不喜欢，所以大家想用C/C++来写eBPF内核程序。
- 然后clang支持了到eBPF字节码的target，然后有人把这些东西拼在了一起，做了libbpf-bootstrap
- 我们可以通过libbpf-bootstrap，直接用C来写eBPF内核代码，还可以通过很方便的方式来定义Map和附加事件。同时生成关于定义的Map之类的header方便用户态程序操作。
- 经过一系列处理生成一个单一用户态可执行文件。这个可执行文件在运行的时候会加载我们写的eBPF程序。

libbpf-bootstrap

- 结合libbpf，提供了一些很方便的把一段eBPF代码贴到某个事件上的方式（比如SEC("tp/syscalls/sys_enter_write")，会在后续处理过程中生成把这个函数编译出来的ebpf程序贴到write(2)上的代码）
- 注意到右边还有个全局变量（my_pid）和一个颜色不一样的函数（bpf_get_current_pid_tgid），接下来会讨论。
- https://github.com/libbpf/libbpf-bootstrap/blob/db4f7ad2a13c525601c3fbf314e9a87ff4dfdbb1/examples/c/minimal.bpf.c#L1



libbpf-bootstrap里的skel

- 一些工具会帮助我们从写的eBPF代码里生成一些被称为skel的header
- 里面包括了写的eBPF程序编译出来的ELF程序的字节数据，以及一些用来操作这个程序的工具函数
- 以及写的eBPF程序的bss段（全局变量）和rodata的Map映射。
- 右边可以看到bss和rodata的map，以及my_pid这个全局变量的拷贝。

```
struct minimal_bpf {
    struct bpf_object_skeleton *skeleton;
    struct bpf_object *obj;
    struct {
        struct bpf_map *bss;
        struct bpf_map *rodata;
    } maps;
    struct {
        struct bpf_program *handle_tp;
    } progs;
    struct {
        struct bpf_link *handle_tp;
    } links;
    struct minimal_bpf__bss {
        int my_pid;
    } *bss;
    struct minimal_bpf__rodata {
    } *rodata;
```

libbpf-bootstrap

- libbpf-bootstrap是要有用户态程序的，那么用户态程序塞什么呢？
- 代码具体可见https://github.com/libbpf/libbpf-bootstrap/blob/db4f7ad2a13c525601c3fbf314e9a87ff4dfdbb1/examples/c/minimal.c#L14
- 创建一个前一张幻灯片所说的并初始化skel
- 往里面填写全局变量的初始值
- 调用封装好的工具函数把从ELF里拆出来的eBPF程序加载进内核
- 开始跑用户态的东西

libbpf-bootstrap的SEC

- 还记得函数前面写的SEC(“tp/syscalls/sys_enter_write”)么？去看一下SEC这个宏，实际上比较重要的内容就是__attribute__((section(name), used))，就是把这个函数塞到某个段里。
- 这样在libbpf的加载函数加载这个ELF程序的时候，看一看段名就知道这个函数对应的ebpf程序是要干什么的了，然后就可以把他往正确的地方加载了。

```
SEC("tp/syscalls/sys_enter_write")
int handle_tp(void *ctx)
{
    int pid = bpf_get_current_pid_tgid() >> 32;

    if (pid != my_pid)
        return 0;
```

libbpf里颜色奇怪的函数

```
int pid = bpf_get_current_pid_tgid() >> 32;
```

- 还记得这个颜色奇怪的bpf_get_current_pid_tgid()么？
- 点进去看的话会发现他是个函数指针，并且值特别奇怪（是int 14）
- 进一步会发现那个header里一大堆bpf程序可以用的函数都是函数指针，并且值比较奇怪
- 这些函数其实是eBPF内核实现预定义的eBPF程序可以使用的内核函数，完整支持的函数及文档可以看 https://github.com/iovisor/bcc/blob/v0.20.0/docs/kernel-versions.md#helpers或者 https://github.com/libbpf/libbpf/blob/master/src/bpf_helper_defs.h
- 至于指针值如此奇怪的原因，我猜可能因为反正这些代码target到的是eBPF虚拟机，所以内置函数从1开始编号就好了

libbpf的CO-RE问题

- CO-RE： Compile Once, Run Everywhere（一次编译，四处运行）
- 使用libbpf开发的程序在编译好之后可以做到在不同的内核版本之间使用。
- 通常而言BPF程序需要访问一些内核数据结构（虽然有程序不需要访问（比如上文那个网卡抓包的））
- 内核中的结构体的成员的偏移可能改变。比如我们读的struct task_struct的某个field的偏移可能随着内核版本的改变而从16变成了24（在这个field前面又加了个新的field）
- 某个结构体成员可能被删除、重命名、挪动位置（比如thread_struct::fs在某些内核版本里叫fsbase；再比如由于内核编译参数的变化，某些结构体成员可能被compiled out）
- libbpf的加载器使用内核提供的BTF（BPF Type Format）信息来在加载eBPF程序时对结构体访问进行重定向以适配不同的内核版本。

BTF是什么

- 内核给我们提供了BTF（可以认为是一种比较轻量的，专门为BPF CO-RE优化的调试信息格式）。在编译内核的时候通过CONFIG_DEBUG_INFO_BTF=y参数启用生成BTF，而后在使用该内核的时候通过/sys/kernel/btf/vmlinux访问BTF信息。
- 如果比较熟悉内核开发的话，从vmlinux这个名字也可以看出来，BTF信息可以拿来生成所有的内核数据结构。(vmlinux.h)

BTF与libbpf

- 既然有了内核所有的结构体的布局，那么libbpf就可以在加载我们写的eBPF内核程序的时候给那些对内核结构体的访问做重定位，然后把修改好的eBPF程序再给内核加载，以此来实现同一个编译好的程序支持多个内核版本。
- 所以其实可以看出来，要使用libbpf这种基于BTF的CO-RE特性是需要编译器支持的，即编译器在编译内核程序的时候，把“st->pid”这种访问，不编译成基址+偏移量的访问，而是在目标文件中留下“访问位于内存xxxx处的task_struct结构体的pid”成员，通过这种信息让libbpf在加载eBPF程序时做重定向。
- clang支持这种编译行为

BCC

BPF Compiler Collection

https://github.com/iovisor/bcc

bcc

- BCC是一个历史很悠久（相比于libbpf）的BPF开发框架。
- 与libbpf的基于BTF的CO-RE方式不同，BCC需要与一个JIT编译器配合工作
- 开发者以源代码的形式分发eBPF程序。用户在使用的时候，BCC调用其内置的编译器（一般是llvm），使用本地预装的kernel-headers对eBPF程序进行编译，而后对其进行加载。
- 所以bcc不需要内核开启BTF支持，但要求用户在本地安装好自己内核对应的headers
- 也提供了很多高级的抽象和 API，帮助提高开发效率

BCC

```
from bcc import BPF

# define BPF program
prog = """
int hello(void *ctx) {
    bpf_trace_printk("Hello, World2!\\n");
    return 0;
}
"""

# load BPF program
b = BPF(text=prog)
b.attach_kprobe(event=b.get_syscall_fnname("clone"), fn_name="hello")

# header
print("%-18s %-16s %-6s %s" % ("TIME(s)", "COMM", "PID", "MESSAGE"))

# format output
while 1:
    try:
        (task, pid, cpu, flags, ts, msg) = b.trace_fields()
    except ValueError:
        continue
    print("%-18.9f %-16s %-6d %s" % (ts, task, pid, msg))
```

bcc

- bcc相比于libbpf有很多缺点，比如：
- 只能以源代码的方式分发eBPF程序
- 用户需要安装bcc的JIT编译器和kernel-header
- 程序的startup时间要比libbpf慢，因为JIT相比于重定位要慢

- 也存在一些优点，比如：
- 不需要内核支持BTF，对一些上古内核比较友好
- 历史悠久，有大量现有程序均使用BCC编写，相比于libbpf生态要好一些

- 所以eunomia-bpf的一个目标是，支持从BCC程序到libbpf的transpile

eunomia-bpf

https://github.com/eunomia-bpf/eunomia-bpf/

eunomia-bpf

- https://github.com/eunomia-bpf/eunomia-bpf/
- 这是我们小队做的东西，大概包含这些功能：
  - 在编写 eBPF 程序或工具时只编写内核态代码，就能自动获取内核态导出信息，自动生成命令行参数、直方图输出等（简化一些简单应用的用户态重复代码，降低上手难度和增加开发效率）
  - 使用 WASM 进行用户态交互程序的开发，在 WASM 虚拟机内部控制整个 eBPF 程序的加载和执行，以及处理 eBPF 上报的数据（和 WebAssembly 生态相结合，多种语言开发用户态程序，类似轻量级容器的隔离和快速启动机制，以及让 eBPF 复用 Wasm 的镜像管理和分发机制）
  - 可以将预编译的 eBPF 程序打包为通用的 JSON 或 WASM 模块，跨架构和内核版本进行分发，无需重新编译即可动态加载运行（增强 libbpf 应用的分发机制，让 BCC 代码实现 AOT 编译运行）
  - Libbpf 内核态代码到 BCC 内核态代码的转换

之后会有关于 eunomia-bpf 更详细的分享和讨论啦

参考资料们

- https://nakryiko.com/posts/libbpf-bootstrap/
- https://github.com/libbpf/libbpf
- https://github.com/libbpf/libbpf-bootstrap
- https://www.ebpf.top/what-is-ebpf/
- https://github.com/eunomia-bpf/eunomia-bpf/

- https://devops.com/libbpf-vs-bcc-for-bpf-development/
- https://facebookmicrosites.github.io/bpf/blog/2020/02/19/bpf-portability-and-co-re.html