eBPF入门之旅

王聪
xiyou.wangcong@gmail.com
https://wangcong.org

什么是 eBPF？

- **eBPF = 扩展的伯克利数据包过滤器**（起源于 Linux 内核网络子系统）
- 它是 Linux 内核的一部分，可以安全地执行用户编写的 eBPF 小程序
- 它可以观察和修改内核数据结构，实现高级功能
- 它足够轻量，足够通用，可以用于各种场景，尤其是内核观测和性能分析

> "eBPF是现代Linux内核中最令人兴奋的技术之一！" ——Brendan Gregg（性能工程师）

为什么需要 eBPF?

- 内核需要扩展性，需要高度可定制化
- 传统的内核模块**不够安全**，容易导致系统崩溃
- 它**不够轻量**，一般用来实现一个子系统（比如IPv6模块）
- 它**不够通用**，缺少通用的用户接口

eBPF 的优势

- **安全保障**：验证器确保代码不会崩溃系统
- **高性能**：JIT编译确保接近原生速度
- **轻量级**： 很多时候只要编写一个函数
- **可移植性**：虽然没有真正做到一次编译到处运行，但可以跨平台

eBPF 的主要应用场景

- **网络**：分析网络流量，实现过滤规则
- **安全**：检测和阻止恶意行为
- **性能**：监控和优化系统性能
- **可观测性**：提供系统洞察和调试信息
- **更多领域**： eBPF 调度器，eBPF Qdisc 等

eBPF 的工作原理

1. 在**用户空间**编写 eBPF 小程序
2. 通过 `bpf()` 系统调用将程序**加载到内核**
3. 经过**安全验证器**的严格检查
4. 在内核中**即时编译**和执行
5. 通过**map**与用户空间交互

eBPF 主要组成部分

- **eBPF 指令集**：精简设计，专为内核可编程性优化（含编译器支持）
- **安全验证器**： 验证和即时编译 eBPF 程序
- **eBPF 存储**：以 eBPF map 为代表的存储，主要用于与用户空间交互
- `bpf()` **系统调用**：主要系统调用接口，用于控制 eBPF 程序相关操作
- **libbpf**：eBPF库，用于简化eBPF程序的加载和管理
- **helper & kfunc**：内核提供的函数，辅助实现eBPF程序的功能

eBPF 指令集

指令集	指令数量	特点
eBPF	~150 条	精简设计，专为内核可编程性优化
x86 (32位)	~1000+ 条	CISC架构，复杂多样的指令
x86_64 (64位)	~1500+ 条	x86扩展，增加64位支持和更多指令
ARM (32位)	~300-400 条	RISC架构，指令相对精简
ARM64 (AArch64)	~600+ 条	ARM扩展，增加64位支持

eBPF 安全验证器

- 由于 eBPF 只有约 150 条指令，比 x86_64 的 1500+ 条少得多
- eBPF 程序执行的上下文简单而且清晰，循环有边界，没有复杂的跳转
- 验证器可以静态分析所有可能的执行路径
- 检查内存访问安全性、无限循环、死锁等
- 确保 eBPF 程序永远安全地终止，不影响系统稳定性
- 类型安全检查：helper & kfunc 参数类型检查

eBPF 程序基本结构

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

// 程序入口点，SEC 宏指定附加点
SEC("kprobe/do_sys_openat2")
int hello_world(void *ctx)
{
    // 调用 helper，打印到 trace_pipe
    bpf_printk("Hello from eBPF!");
    return 0;  // 返回0表示成功
}

// 必须的许可证声明
char LICENSE[] SEC("license") = "GPL";
```

bpftool 查看 eBPF 二进制指令

```
# 加载程序并显示其ID
$ sudo bpftool prog load hello.bpf.o /sys/fs/bpf/hello
$ sudo bpftool prog show
463: kprobe  name hello_world  tag 88b1aaa25a53cd95  gpl
        loaded_at 2025-04-15T20:10:04-0400  uid 0
        xlated 48B  jited 40B  memlock 4096B  map_ids 181
        btf_id 386

# 查看程序的指令
$ sudo bpftool prog dump xlated id 463

# 查看JIT编译后的 x86 指令
$ sudo bpftool prog dump jited id 463
```

eBPF 汇编代码详解

```
int hello_world(void * ctx):
; bpf_printk("Hello from eBPF!");
   0: (18) r1 = map[id:181][0]+0    // 加载字符串常量，使用内部的只读映射存储字符串
   2: (b7) r2 = 17                  // 将字符串长度(17字节)设置到r2寄存器
   3: (85) call bpf_trace_printk#-113730  // 调用helper函数bpf_trace_printk，r1=格式化字符串，r2=长度
; return 0;
   4: (b7) r0 = 0                   // 将返回值(r0寄存器)设置为0
   5: (95) exit                     // 结束程序并返回
```

字符串常量说明

- `map[id:181]`：引用了一个ID为181的 map，这是由eBPF加载器*自动*创建的内部 array，用于存储字符串常量
- `[0]`：表示访问这个 map 中的*第一个元素*，这里存储着目标字符串
- `+0`：表示从字符串开始处的偏移量 0

```
# bpftool map dump id 181
[{
        "value": {
            ".rodata": [{
                    "hello_world.____fmt": "Hello from eBPF!"
                }
            ]
        }
```

eBPF 指令说明

- **r0-r10**: eBPF的寄存器，其中r0用于返回值，r1-r5用于函数参数和临时值，r10是堆栈指针
- **(18)**: 内存加载指令码，用于加载一个64位的即时数到寄存器。这里从映射中加载字符串
- **(b7)**: 用于设置寄存器值的指令码，将一个32位即时数设置到寄存器
- **(85)**: 函数调用指令码，用于调用helper函数
- **(95)**: eBPF程序的结束指令，从程序中返回

eBPF Map

- eBPF 程序的传统存储方式，主要用于与用户空间交互
- 多数基于数组和哈希表， 以 key/value 的形式存储
- 可以在 eBPF 程序中直接访问，也可以在用户空间通过 fd 访问
- 有很多特殊的 map，可以存放各种数据类型，包括内核里的数据结构

Map 定义方式

```c
// 旧的定义方式
struct bpf_map_def SEC("maps") my_map = {
    .type = BPF_MAP_TYPE_HASH,
    .key_size = sizeof(int),
    .value_size = sizeof(int),
    .max_entries = 1024,
};

// 新的 BTF 定义方式
struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __type(key, int);
    __type(value, int);
    __uint(max_entries, 1024);
} my_map SEC(".maps");
```

Map 存储与加载

1. **maps部分**被放在**特殊的ELF段**中
   - 旧式: `maps` 段
   - 新式: `.maps` 段加上 BTF 信息
2. **加载过程**
   - eBPF 加载器识别特殊的 maps 段
   - 解析 map 定义并创建eBPF映射
   - 分配内核中的映射ID
   - 将程序中对 map 的引用替换为 FD 或 map ID

eBPF 数据结构

链表

- eBPF 链表是通过 `struct bpf_list_head` `struct bpf_list_node` 实现的
- 你可以在自己的数据结构中嵌入 `bpf_list_node`
- BPF 辅助函数允许安全地操作（插入、删除、遍历）链表元素

```
int bpf_list_push_front(struct bpf_list_head *head, struct bpf_list_node *node);
int bpf_list_push_back(struct bpf_list_head *head, struct bpf_list_node *node);
int bpf_list_pop_front(struct bpf_list_head *head);
int bpf_list_pop_back(struct bpf_list_head *head);
```

eBPF 数据结构

红黑树

- eBPF rbtree 是通过 `struct bpf_rbtree_root` `struct bpf_rbtree_node` 实现的
- 你可以在自己的数据结构中嵌入 `bpf_rbtree_node`
- 专为eBPF中的有序数据插入/搜索而设计，在某些情况下允许无锁和并发使用

```
int bpf_rbtree_add(struct bpf_rb_root *root, struct bpf_rb_node *node, bool (*less)(...));
struct bpf_rb_node *bpf_rbtree_first(struct bpf_rb_root *root);
struct bpf_rb_node *bpf_rbtree_remove(struct bpf_rb_root *root, struct bpf_rb_node *node);
```

eBPF 动态内存分配

- `bpf_obj_new()`：分配新对象
- `bpf_obj_drop()`：释放对象
- 验证器确保分配和释放的正确性

动态分配示例

```
struct node {
    int key;
    struct bpf_rb_node rb_node;
};

struct bpf_rb_root tree;

struct node *n = bpf_obj_new(typeof(struct node));
if (!n)
    return 0;

n->key = some_value;
bpf_rbtree_add(&tree, &n->rb_node, cmp_fn);

// later, when removing:
bpf_obj_drop(n);
return 0;
```

`bpf()` 系统调用

- `bpf()` 系统调用接口，用于 eBPF 控制面相关操作
- 定义非常通用，容易扩展
- 可以用于加载、查询、更新、删除 eBPF 程序
- 可以用于创建、查询、更新、删除 maps
- 可以用于获取 eBPF 程序的统计信息

libbpf

- **libbpf**：Linux 内核官方的 eBPF 库，简化 eBPF 开发体验
- **程序加载与重定位**：自动处理 CO-RE (Compile Once Run Everywhere) 相关的数据重定位
- **map 管理**：创建、访问、更新和删除 map，并处理不同类型 map 的特性
- **BTF (BPF Type Format) 支持**：提供类型信息支持，允许跨内核版本兼容
- **详细错误报告**：自动处理验证器消息，提供更有用的错误信息

helper 函数

- 内核中预定义的函数集合，提供 eBPF 程序访问内核功能的能力
- 具有严格限制的参数类型和返回值
- 通过特殊的 BPF_CALL 指令调用
- 因为其灵活性较低，*逐渐被 kfunc 替代*

kfunc 内核函数

- 更新的扩展机制，允许调用现有内核函数
- 优点：
  - 可以在内核模块中声明和使用，无须重新编译内核
  - 基于白名单，有点类似于 `EXPORT_SYMBOL()`
  - 直接调用内核函数，提供更高的性能
- 支持基于 BTF 的类型检查，提供更强的安全保障
- 允许 eBPF 程序更深入地与内核子系统交互
- 从 Linux 5.13 开始支持，持续扩展中

bpftrace 简介

- **bpftrace** 是一个基于 eBPF 的高级跟踪语言
- 受到 DTrace 和 awk 的启发，可以用一行命令实现复杂功能
- 基本语法结构：

```
probe_definition[,probe_definition...] { action }
```

bpftrace 语法详解

- 常见探针类型
  - `kprobe/kretprobe`：内核函数进入/返回点
  - `uprobe/uretprobe`：用户空间函数进入/返回点
  - `tracepoint`：内核静态跟踪点
  - `interval`：定时采样
- 内置变量
  - `pid`, `tid`, `uid`, `comm`：进程信息
  - `args`：探针参数
  - `retval`：kretprobe/uretprobe 返回值

bpftrace 语法详解

- 数据存储
  - `@name = expression`：创建映射
  - `@name[key] = expression`：创建索引映射
- 内置函数
  - `printf()`, `exit()`, `str()`, `count()`, `hist()`, `time()` 等

bpftrace 示例

```
# 统计系统调用频率排行
$ sudo bpftrace -e 'tracepoint:raw_syscalls:sys_enter { @[comm] = count(); }'
Attaching 1 probe...
@[systemd-userwor]: 10
@[sudo]: 16
@[irqbalance]: 38
@[gmain]: 60
@[bpftrace]: 238
@[in:imjournal]: 2639
```

```
# 监控文件打开操作
$ sudo bpftrace -e 'tracepoint:syscalls:sys_enter_openat { printf("%s opened %s\n", comm, str(args->filename)); }'
Attaching 1 probe...
bash opened /dev/null
irqbalance opened /proc/interrupts
irqbalance opened /proc/stat
```

bpftrace 示例

```
# 查看进程创建分布情况
$ sudo bpftrace -e 'tracepoint:sched:sched_process_fork { @[comm] = count(); }'
Attaching 1 probe...
@[systemd-userdbd]: 3
@[systemd]: 3
@[grepconf.sh]: 3
@[kthreadd]: 4
@[fprintd]: 5
@[bash]: 21
```

bpftrace 示例

```
# TCP connect延迟测量
$ sudo bpftrace -e '
kprobe:tcp_connect { @start[tid] = nsecs; }
kretprobe:tcp_connect {
    $duration_ms = (nsecs - @start[tid]) / 1000000;
    printf("%s (PID: %d) TCP connect latency: %d ms\n",
           comm, pid, $duration_ms);
    delete(@start[tid]);
}
'
Attaching 2 probes...
ssh (PID: 2038) TCP connect latency: 8 ms
```

bpftrace示例: execsnoop

```
$ sudo bpftrace -e '
tracepoint:syscalls:sys_enter_execve
{
    printf("%-6d %-16s exec: %s\n", pid, comm, str(args->filename));
}
'
Attaching 1 probe...
1163   (sd-worker)      exec: /usr/lib/systemd/systemd-userwork
1169   bash             exec: /usr/bin/bash
1170   bash             exec: /usr/sbin/ip
1173   bash             exec: /usr/sbin/tc
1174   bash             exec: /usr/sbin/tc
1176   bash             exec: /usr/bin/socat
1179   bash             exec: /usr/sbin/tc
1180   bash             exec: /usr/sbin/tc
1182   bash             exec: /usr/bin/socat
1184   bash             exec: /usr/sbin/tc
1185   bash             exec: /usr/sbin/tc
1187   bash             exec: /usr/bin/socat
1188   (sd-worker)      exec: /usr/lib/systemd/systemd-userwork
```

bpftrace示例: I/O延迟分析

```
$ sudo bpftrace -e '
tracepoint:block:block_rq_complete
{
    $latency = (nsecs - @start[args->dev, args->sector])/1000;
    if ($latency > 0) {
        @us = hist($latency);
        if ($latency > 1000) {
            printf("HIGH I/O latency: %d us, dev: %d, sector: %d, rwbs: %s\n",
                   $latency, args->dev, args->sector, args->rwbs);
            @[kstack] = count();
        }
        delete(@start[args->dev, args->sector]);
    }
}

tracepoint:block:block_rq_issue
{
    @start[args->dev, args->sector] = nsecs;
}
'
```

bpftrace示例: 输出

```
Attaching 2 probes...

HIGH I/O latency: 1081 us, dev: 266338304, sector: 12720664, rwbs: RA
HIGH I/O latency: 5040 us, dev: 266338304, sector: 12720592, rwbs: RA
HIGH I/O latency: 1014 us, dev: 266338304, sector: 4668272, rwbs: RA
HIGH I/O latency: 1737 us, dev: 266338304, sector: 12046480, rwbs: RA
HIGH I/O latency: 1233 us, dev: 266338304, sector: 4371768, rwbs: RA

@[
    trace_block_rq_complete+148
    trace_block_rq_complete+148
    blk_update_request+86
    blk_mq_end_request+41
    virtblk_done+268
    vring_interrupt+153
    __handle_irq_event_percpu+418
    handle_irq_event_percpu+16
    handle_irq_event+98
    handle_edge_irq+309
    __common_interrupt+144
    common_interrupt+102
    asm_common_interrupt+34
    default_idle+11
    default_idle_call+113
    cpuidle_idle_call+161
    do_idle+190
    cpu_startup_entry+47
    __pfx_common_cpu_up+0
    common_startup_64+300
]: 2

@us:
[512, 1K)              3 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[1K, 2K)               3 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[2K, 4K)               0 |                                                    |
[4K, 8K)               1 |@@@@@@@@@@@@@@@@@                                   |
```

eBPF Hello World 示例

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

SEC("kprobe/do_sys_openat2")
int hello_world(void *ctx) {
    char msg[] = "Hello World from eBPF!";
    bpf_trace_printk(msg, sizeof(msg));
    return 0;
}

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

eBPF 程序编译与加载

```
$ sudo apt install clang llvm libelf-dev linux-headers-$(uname -r)
$ clang -g -O2 -target bpf -D__TARGET_ARCH_x86_64 -c hello.c -o hello.o
```

```
# load and auto-attach
sudo bpftool prog load hello.o /sys/fs/bpf/hello type kprobe autoattach
```

```
$ sudo cat /sys/kernel/debug/tracing/trace_pipe
 systemd-userwor-1745    [001] ...21 10018.356493: bpf_trace_printk: Hello World from eBPF!
 systemd-userwor-1745    [000] ...21 10018.362589: bpf_trace_printk: Hello World from eBPF!
 systemd-userwor-1745    [000] ...21 10018.372010: bpf_trace_printk: Hello World from eBPF!
 systemd-userwor-1745    [000] ...21 10018.372627: bpf_trace_printk: Hello World from eBPF!
 systemd-userwor-1745    [000] ...21 10018.372960: bpf_trace_printk: Hello World from eBPF!
 systemd-userwor-1745    [000] ...21 10018.373236: bpf_trace_printk: Hello World from eBPF!
```

总结

- eBPF 是 Linux 内核的一部分，可以安全地执行用户编写的 eBPF 小程序
- 它可以观察和修改内核数据结构，实现高级功能
- 它足够轻量，足够通用，可以用于各种场景，尤其是内核观测和性能分析

参考资料

- eBPF 官方文档
- eBPF 白皮书
- eBPF 用户空间库
- eBPF 内核模块
- bpftrace 用户手册
- eBPF 指令集定义