西电mobisys容器云团队——基于eBPF的容器异常行为检测系统技术文档

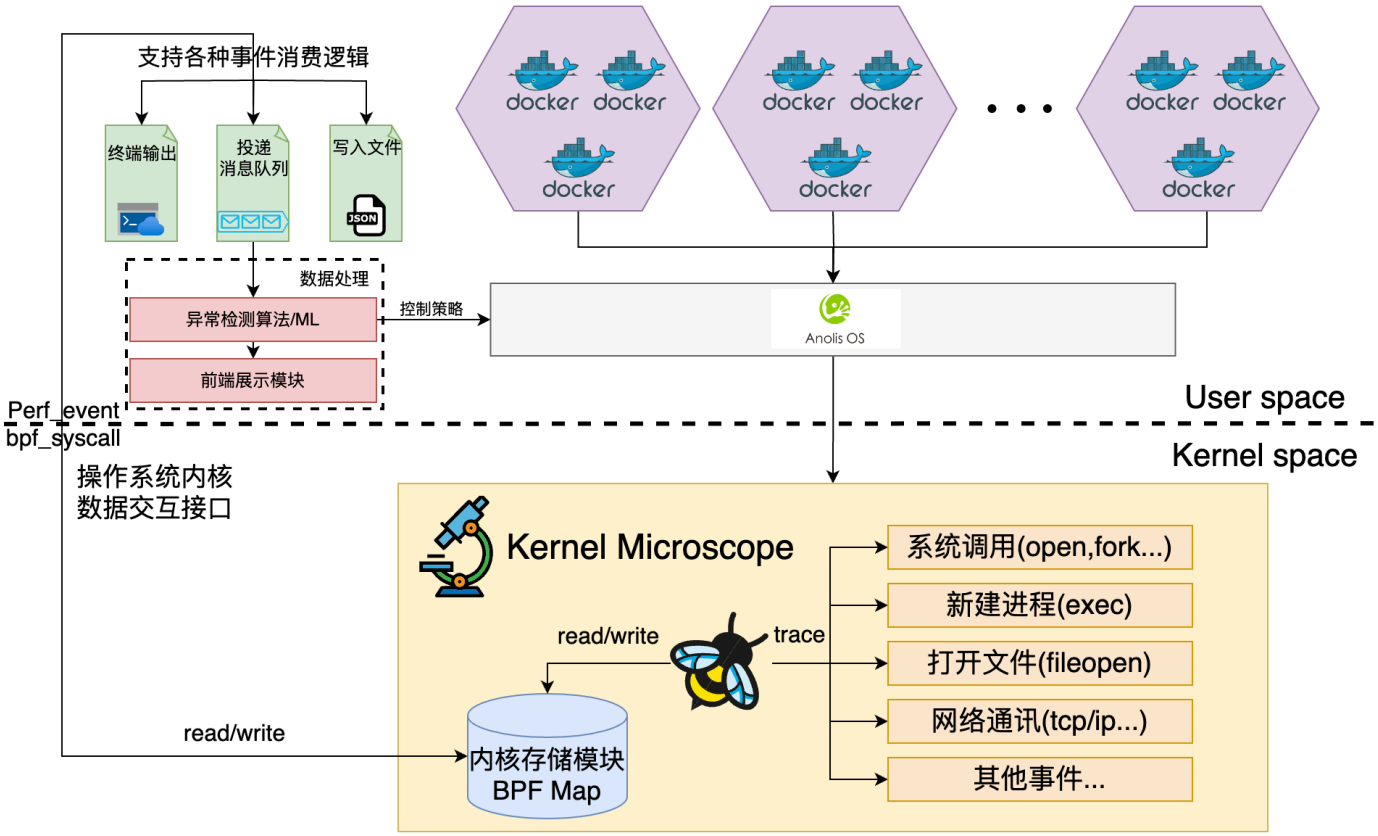
Intro

系统由两部分构成：

- 部署在龙蜥操作系统中的eBPF采集模块，采集操作系统中容器的运行时行为信息。
- 部署在任意Linux服务器中的算法处理模块，负责除eBPF以外的所有处理逻辑。

两个模块之间通过kafka消息队列进行数据交互，通过http协议进行控制交互。

系统架构图：



eBPF采集模块

使用Cilium/ebpf库

市面上有很多eBPF开发框架，较为流行的有如下几款：

- [iovisor/bcc](https://github.com/iovisor/bcc)
  - 优点：
    - 使用Python作为eBPF前端语言，功能丰富，简单易上手。
    - GitHub 20K Stars，优秀的社区生态。

        - 对其熟悉，**队伍中有BCC库的贡献者**。
    - 缺点：
        - 使用BCC实现的eBPF程序在运行时需要庞大的Python运行时依赖（包括Clang、LLVM...），不利于部署。
        - 每次运行时需要使用Clang重新编译ebpf内核态程序，性能开销大。
        - Python会将程序错误推迟到运行时再暴露，不利于系统级工具的开发。
- [libbpf/libbpf](https://github.com/libbpf/libbpf)
    - 优点：
        - 使用C/C++/Rust作为eBPF前端语言，高性能。
        - libbpf是一个C语言库，伴随内核版本分发，由内核开发人员进行维护，可靠。
        - 支持CO-RE、BTF等高级特性。
    - 缺点：
        - C/C++生态较差，无好用的第三方库（如docker库、Kafka库...）；Rust开发难度高。
- [cilium/ebpf](https://github.com/cilium/ebpf)
    - 优点：
        - 纯Go实现的ebpf框架，继承了Golang的优秀特性，编译产物无外部依赖。
        - 社区活跃。
        - 支持CO-RE、BTF等高级特性。
        - docker也是由Golang语言实现的。在“基于eBPF的容器异常行为检测”课题中，使用Golang语言能够高效的与docker后台守护进程dockerd进行通讯，获取容器信息。
    - 缺点：
        - 无。

综上所述，我们最终选择了Cilium/ebpf库作为开发框架，并**基于Cilium/ebpf库实现了一套高度抽象的eBPF开发框架，能够较为方便的添加功能**。（详见eBPF用户态程序）

为eBPF采集模块添加新的功能时，只需要实现**eBPF内核态程序**并在用户态程序中**实现userspace接口**。

eBPF内核态程序

系统资源使用情况

使用传统的采集工具

识别容器内进程

能使用eBPF采集操作系统内容器行为的基本前提是**容器虚拟化技术共用操作系统内核**。

在eBPF程序的视角中，容器内的程序与宿主机操作系统中的进程无异。因此，**如何在eBPF程序中识别出容器进程是十分重要的**。

经过研究，容器技术基于Linux操作系统中的namespace、cgroup实现。

基于此，在进程控制块`task_struct`中通过进程所属的namespace id能够判断其是否是容器进程；通过cgroups字段能够解析出其所属cgroup id。

系统调用事件

使用btf_raw_tracepoint eBPF程序类型。

Tracepoint bpf程序类型可选项：

> 参考https://mozillazg.com/2022/06/ebpf-libbpf-btf-powered-enabled-raw-tracepoint-common-questions.html
> 参考https://mozillazg.com/2022/05/ebpf-libbpf-raw-tracepoint-common-questions.html#hidraw-tracepoint-tracepoint

1. tracepoint: 最原始的tracepoint类型，基本原理是内核中预埋的静态hook点。
2. raw_tracepoint: 与tracepoint类似，但是可以自定义tracepoint的参数。

- tracepoint、raw_tracepoint的主要区别是，raw tracepoint 不会像 tracepoint 一样在传递上下文给 ebpf程序时 预先处理好事件的参数（构造好相应的参数字段）， raw tracepoint ebpf程序中访问的都是事件的原始参数。因此，raw tracepoint 相比 tracepoint 性能通常会更好一点 (数据来自https://lwn.net/Articles/750569/)

3. btf_raw_tracepoint:
   与raw_tracepoint类似，但是可以使用BTF信息来解析tracepoint的参数。

基于此，我们使用了**eBPF tailcall机制**，对于特殊的系统调用进行特殊处理，核心代码如下：

```
struct {
  __uint(type, BPF_MAP_TYPE_PROG_ARRAY);
  __type(key, uint32_t);
  __type(value, uint32_t);
  __uint(max_entries, 1024);
```

```
  __array(values, int());
} syscall_enter_tail_table SEC(".maps") = {
    .values =
        {
            [59] = &sys_enter_execve, // sys_enter_execve是函数指针，用于处理特殊系统调用
            [83] = &sys_enter_mkdir,
            ...
        },
};
SEC("tp_btf/sys_enter")
int BPF_PROG(sys_enter, struct pt_regs *regs, long syscall_id) {
  struct task_struct *curr_task = (struct task_struct *)bpf_get_current_task();
  if (get_task_level_core(curr_task) == 0) {    // 非容器进程
    // level 0 means the task is in the root pid namespace
    return 0;
  }
  struct syscallcntkey key = {
      .syscall_id = syscall_id,
      .pid = bpf_get_current_pid_tgid() >> 32,
  };
  get_cid_core((struct task_struct *)bpf_get_current_task(), &key.cid); // 获取cid
  u64 *sys_cnt = bpf_map_lookup_elem(&syscall_cnt, &key); // 获取计数器指针
  sys_cnt++;
  bpf_tail_call(ctx, &syscall_enter_tail_table, syscall_id);// 尾调用
  return 0;
}
```

文件访问事件

使用/fs/namei.c/do_filp_open()内核函数作为Hook点，理由如下：

- 函数稳定，自2.6.12版本之后（12年）再无修改。
- 无内联优化，函数符号稳定。
- 参数含义明确，其返回值是file类型的指针，指向本次打开的文件控制块。

难点：

- 需要在eBPF程序中处理文件路径，需要拼接字符串。
  - 解决方法：实现了一套简单的字符串处理工具，能够通过eBPF验证器验证。
- 文件路径是变长字符串。
  - 解决方法：对文件路径的每一层单独存储，将原始数据发送至用户态处理。

核心代码如下：

```
SEC("kretprobe/do_filp_open")
int BPF_KRETPROBE(kretprobe_do_filp_open, struct file *filp) {
  ...
```

```
  struct file *fi = filp;
  u8 *fsnamep = (u8 *)BPF_CORE_READ(fi, f_inode, i_sb, s_type, name); // 获取文件系统名称
  bpf_core_read_str(&event->fsname, FSNAME_LEN, fsnamep);

  struct dentry *cur_dentry = (struct dentry *)BPF_CORE_READ(fi, f_path.dentry);
  int offset = 0;
  for (int i = 0; i < FILE_MAXDEPTH; i++) {// 按层次拼接文件路径
    if (cur_dentry == NULL) break;
    get_dentry_name_core(cur_dentry, event->filename[i]);
    if (event->filename[i][0] == 0) break;

    cur_dentry = (struct dentry *)BPF_CORE_READ(cur_dentry, d_parent);
  }
  ...
  return 0;
}
```

网络互访事件

> 默认容器间通讯是使用基于IPv4的tcp协议

使用/net/ipv4/tcp_output.c/__tcp_transmit_skb()&&net/ipv4/tcp_ipv4.c/tcp_v4_do_rcv()内核函数作为tcp收包、发包的hook点，理由如下：

- 函数稳定，自2.6.12版本之后（12年）再无修改。
- 无内联优化，函数符号稳定，参数含义明确。
- 相较于网络通讯本身的开销，tcp协议栈中eBPF程序的开销可以忽略不计。

难点：

- TCP连接状态复杂，难以隔离出真正的通讯报文。
  - 解决方法：增加sock_alloc内核函数作为hook点，捕获到容器内进程的网络socket创建事件，后续的收发数据包事件均以此为基础。
- 将所有进程间通讯的网络事件都采集的开销太大，无法接受。
  - 解决方法：需要在内核态程序中精准识别出未经授权的容器互访，将规则进行编码并存放在bpf map中，用户态程序维护规则集，内核态读取规则集。

核心代码如下：

```
SEC("kprobe/__tcp_transmit_skb")
int BPF_KPROBE(kprobe_tcp_transmit_skb) {
  if (get_task_level_core((struct task_struct *)bpf_get_current_task()) == 0) {
    return 0;
  }
  ...
  struct cnetwork_event *event =
      bpf_ringbuf_reserve(&containernet_rb, sizeof(struct cnetwork_event), 0);
```

```
  u16 family = BPF_CORE_READ(sk, __sk_common.skc_family);
  if (family == AF_INET) { // 一定成立
    event->daddr = BPF_CORE_READ(sk, __sk_common.skc_daddr);
    event->saddr = BPF_CORE_READ(sk, __sk_common.skc_rcv_saddr);
  }
  // 网络事件过滤
  u64 tmpa = 1ll * event->daddr << 32 | event->saddr;
  u64 tmpb = 1ll * event->saddr << 32 | event->daddr;
  if ((bpf_map_lookup_elem(&cnetwork_banned, &tmpa) == NULL) &&
      (bpf_map_lookup_elem(&cnetwork_banned, &tmpb) == NULL)) { // 不在规则集中
    bpf_ringbuf_discard(event, 0);
    return 0;
  }
  ...
  event->flag = 0;  // send
  return 0;
}
```

eBPF用户态程序

整体结构

- 基于Cilium/ebpf库。
- 将eBPF事件采集模块（userspace）与事件消费模块（comsumer）分离，两者之间通过管道进行数据交互。
- eBPF事件采集模块负责将eBPF内核态程序注入内核，并读取eBPF map中的事件数据，将其进行初步整理并作为事件生产者将事件投递到事件管道中。
- 事件消费模块有多种消费逻辑，并且具有可扩展性。
- 基于此设计，实现了可扩展的ebpf数据采集框架。为eBPF采集模块添加新的功能时，只需要实现**eBPF内核态程序**并在用户态程序中**实现userspace接口**。

项目结构如下：

```
.
├── comsumer # 事件管道的消费者，可以对事件进行丢弃、投递到kafka、日志打印等处理。
│   ├── donothing.go
│   ├── kafka.go
│   ├── log.go
├── config # 配置文件
│   └── conf.yml
├── event # 对于不同类型事件的抽象
│   ├── cnetwork_event.go
│   ├── fileopen_event.go
│   ├── Ievent.go
│   └── syscall_event.go
├── kern # eBPF内核态程序
│   ├── containernet.c
│   ├── fileopen.c
│   ├── syscall.c
```

```
17  │   └── syscall_tailcall.h
18  ├── main.go
19  ├── userspace # eBPF用户态程序
20  │   ├── cnetwork
21  │   │   └── cnetwork.go
22  │   ├── csyscall
23  │   ├── fileopen
24  │   └── userspace.go
25  └── util # 工具类，用于与dockerd进行通讯，获取容器基本信息
26      ├── getcinfo.go
27      ├── getcname.go
28      └── init.go
```

异步通讯机制

- 对于eBPF采集到的不同类型事件，将其抽象成levent接口。
- 使用Golang提供的Channel特性，事件的生产&消费异步进行，大大提高了eBPF采集模块的运行效率。

算法处理模块

TODO

基于规则和算法

效果演示

见效果演示视频