Linux 主机 ebpf 监控项目

系统测试报告

开 发 组：项目 4——2 组

审 核 组：项目 4——1 组

拟 制 人：张书豪 刘雅璇

[2023 年 5 月 22 日]

报告撰写记录

版本号	日期	修改人	摘　　要
V0.1	5.21	刘雅璇	创建报告模板，分配测试任务，撰写测试用例
V0.2	5.22	张书豪	记录测试关键步骤，定位问题，提供截图和测试脚本

目录

1 概述

1.1 项目背景

eBPF（Extended Berkeley Packet Filter）是一种 Linux 内核技术，可以在内核中执行自定义的代码，并可以访问内核中的数据结构。通过 eBPF，可以实现在不需要重新编译内核或加载内核模块的情况下，实时监控和收集 Linux 系统的各种指标。

本项目应实现以下目标：

① 内置实现若干常规监控指标，包括：进程 tcp 建连耗时，网络重传次数，tcprtt，biolatency 等；

② 实现对这些监控指标的前端展示，具备选择展示的监控指标和观察的时间段；

③ 监控指标可配置化；

④ 监控数据支持定制化落地到指定的数据库。

1.2 测试目标

测试内容是针对 Linux 主机 ebpf 监控项目进行的系统测试，目的是判定该系统是否满足项目规定的功能需求和性能指标。主要包括：

① 是否实现要求的功能；

② 是否部署配置简单，一个可执行文件和一个配置文件；

③ 前后端是否可以稳定运行；

④ 是否存在其他性能或代码可靠性等问题。

1.3 测试范围及方法

序号	测试范围	测试方法	测试工具
1	部署测试		
2	功能测试	黑盒测试	
3	单元测试	白盒测试	
4	性能测试		

1.4 测试环境

表 1-1 运行环境总体说明

	约束
操作系统	Linux
网络环境	无
服务器	无
第三方软件	无

1.5 测试人员信息

表 1-3 测试人员信息

测试人员	测试内容
张书豪	部署测试、单元测试
刘雅璇	部署测试
白泽若	部署测试

1.6 测试中止条件

各类型测试中止条件为：

1. 功能实现与系统描述不符，可以中止此测试用例；

2. 测试环境与要求不符，可以中止此测试用例。

1.7 测试结束准则

部署测试范围覆盖部署文档的全部内容；功能测试、性能测试范围覆盖待测功能的全部页面；单元测试范围覆盖核心代码模块，每个模块测试全部代码分支。

2 测试过程

2.1 总体概况

测试内容	是否通过	总体情况说明
部署测试	否	部署测试用例总量 4，暂未通过
功能测试	/	/
单元测试	/	针对项目内核模块分析并撰写测试脚本
性能测试	/	/

2.2 部署测试

2.2.1 测试信息

信息	描述
测试轮次	每人进行 2 轮，共 6 轮测试，覆盖部署文档中提到的各部署方式
测试时间	第 1 轮：2023-5-20 第 2 轮：2023-5-22
测试平台	Linux 虚拟机运行
测试人员	全员参与部署测试

2.2.2 测试说明

（1）测试用例一：docker 部署 1

步骤	说明

步骤 1	预期结果（命令行见截图）
	Use docker
	Run the following code to run the eBPF code from the cloud to your local machine in one line:
	`$ sudo docker run --rm -it --privileged ghcr.io/eunomia-bpf/eunomia-template:latest`
	`TIME EVENT COMM PID PPID FILENAME/EXIT CODE`
	`09:25:14 EXEC sh 28142 1788 /bin/sh`
	`09:25:14 EXEC playerctl 28142 1788 /nix/store/vf3rsb7j3p7zzyjpb0a3axl8yq4z1sq5-playerctl-2.4.1/b`
	`09:25:14 EXIT playerctl 28142 1788 [1] (6ms)`
	`09:25:15 EXEC sh 28145 1788 /bin/sh`
	`09:25:15 EXEC playerctl 28145 1788 /nix/store/vf3rsb7j3p7zzyjpb0a3axl8yq4z1sq5-playerctl-2.4.1/b`
	`09:25:15 EXIT playerctl 28145 1788 [1] (6ms)`

步骤 2	部署中 [redacted]:~/Documents/STGroup2$ sudo docker run --rm -it --privileged ghcr.io/eunomia-bpf/eunomia-template:latest [sudo] password for [redacted]: Unable to find image 'ghcr.io/eunomia-bpf/eunomia-template:latest' locally latest: Pulling from eunomia-bpf/eunomia-template 2ab09b027e7f: Pull complete 4f4fb700ef54: Pull complete 3b961b4622dc: Pull complete a9a5b9687315: Downloading [=====================>] 144.6MB/323.4MB a1ab1e99039a: Download complete 3fd48ae5bea6: Download complete b463256af0c4: Downloading [==================>] 99.98MB/256.5MB 8aa9b7c2e75b: Downloading [===>] 49.19MB/59.56MB 595427afd2be: Waiting dd4675b599ee: Waiting 63d5d41540a8: Waiting 4db1f0bfc36f: Waiting
步骤 3	部署失败 [redacted]:~/Documents/STGroup2$ sudo docker run --rm -it --privileged ghcr.io/eunomia-bpf/eunomia-template:latest [sudo] password for [redacted]: Unable to find image 'ghcr.io/eunomia-bpf/eunomia-template:latest' locally latest: Pulling from eunomia-bpf/eunomia-template 2ab09b027e7f: Pull complete 4f4fb700ef54: Pull complete 3b961b4622dc: Pull complete a9a5b9687315: Pull complete a1ab1e99039a: Pull complete 3fd48ae5bea6: Pull complete b463256af0c4: Pull complete 8aa9b7c2e75b: Pull complete 595427afd2be: Pull complete dd4675b599ee: Pull complete 63d5d41540a8: Pull complete 4db1f0bfc36f: Pull complete Digest: sha256:8894ab9b6025f8e4a49f2ce6c542ae9c166fe3d03a7af7e9d5cce3c1b65daaf8 Status: Downloaded newer image for ghcr.io/eunomia-bpf/eunomia-template:latest arg: src/package.json libbpf: failed to determine tracepoint 'sched/sched_process_exec' perf event ID: No such file or directory libbpf: prog 'handle_exec': failed to create tracepoint 'sched/sched_process_exec' perf event: No such file or dir ectory libbpf: prog 'handle_exec': failed to auto-attach: -2 failed to attach skeleton Error: BpfError("load and attach ebpf program failed")

（2）测试用例二：docker 部署 2

步骤	说明
步骤 1	部署失败（命令行见截图） [redacted]:~/Documents/STGroup2/st_eunomia$ sudo docker run -it -v `pwd`/:/src/ yunwei37/ebpm:latest [sudo] password for kali: ls: cannot access '/src/*.bpf.c': No such file or directory export PATH=ATH:~/.eunomia/bin Compiling bpf object... $ clang -g -O2 -target bpf -D__TARGET_ARCH_x86 -idirafter /usr/lib/llvm-14/lib/clang/14.0.0/include -idirafter /us r/local/include -idirafter /usr/include/x86_64-linux-gnu -idirafter /usr/include -I/root/.eunomia/include -I/root /.eunomia/include/vmlinux/x86 -I/src -c /src/vmlinux.h -o /src/vmlinux.bpf.o /src/vmlinux.h:1:1: error: unknown type name 'Do' Do you want to install it? (N/y) ^ /src/vmlinux.h:1:7: error: expected ';' after top level declarator Do you want to install it? (N/y) ^ ; 2 errors generated. Error: failed to compile bpf object make: *** [Makefile:92: build] Error 1

（3）测试用例三：本机部署

步骤	说明

步骤 1	`# 2. Deploy the project on local host.`
	`# Get started.`
	`sudo apt update && sudo apt install -y direnv`
	`curl -sfL https://direnv.net/install.sh | bash  # sh <(curl -L https://nixos.org/nix/install) --daemon`
	`direnv allow`
	`# Clone repository.`
	`git clone --depth 1 https://github.com/Asher459/st_eunomia.git`
	`# Install dependencies.`
	`sudo apt update && \`
	`sudo apt install -y --no-install-recommends libelf1 libelf-dev zlib1g-dev make clang llvm`
	`# Build the project.`
	`make build`
	`# Run the project.`
	`ecli run src/package.json`

（4）测试用例四：在 GitHub codespace 部署

步骤	说明
步骤 1	预期结果 **4. Build the project** To build the project, run the following command: `make build` This will compile your code and create the necessary binaries. You can you the `Github Code space` or `Github Action` to build the project as well.

步骤 2	部署失败
	`root@codespaces-0b91d3:/workspaces/ST_lab# make build`
	`make -C src`
	`make[1]: Entering directory '/workspaces/ST_lab/src'`
	`  LIB      libbpf.a`
	`make[2]: Entering directory '/workspaces/ST_lab/src'`
	`make[2]: *** /workspaces/ST_lab/libbpf/src: No such file or directory.  Stop.`
	`make[2]: Leaving directory '/workspaces/ST_lab/src'`
	`make[1]: *** [Makefile:73: /workspaces/ST_lab/src/.output/libbpf.a] Error 2`
	`make[1]: Leaving directory '/workspaces/ST_lab/src'`
	`make: *** [Makefile:2: build] Error 2`

2.2.3 测试用例执行率

测试用例数量				测试用例执行数量	测试用例执行率
优先级	高	中	低		
Docker 部署	2			2	100%
本机部署	1			1	100%
codespace 部署		1		1	100%
总计	3	1			

2.2.4 缺陷说明

缺陷列表	测试平台	缺陷等级	具体说明
部署未成功	Linux	高	见上述测试情况

2.2.5 测试小结

部署出现问题，目前定位到了 docker 的配置文件问题，且 libbpf 运行不顺。

除部分单元测试脚本代码外，其他工作暂时无法开展，已与助教反映此情况。

2.3 功能测试

部署未成功，暂时无法进行

2.4 单元测试

部署未成功，暂时无法进行。目前针对项目内核模块分析并撰写测试脚本如下。

2.4.1 代码分析 1

分析 template.h 和 template.c 文件功能如下：

这两段代码是使用 eBPF 技术编写的用于跟踪进程创建和退出事件的内核模块。模块可以捕获跟踪事件和进程元数据，并将其提交到环形缓冲区以供用户空间进程分析。以下是两个函数的简要说明：

1. handle_exec：此函数用于处理 sched_process_exec 事件，该事件在进程执行新程序时发生。函数在处理事件时获取进程元数据，例如 PID、PPID、进程命令和执行文件名，并将其打包成 event 结构，然后将其提交到环形缓冲区以供后续分析。

2. handle_exit：此函数用于处理 sched_process_exit 事件，该事件在进程退出时发生。函数在处理事件时获取进程元数据，例如 PID、PPID、退出代码和持续时间，并将其打包成 event 结构，然后将其提交到环形缓冲区以供后续分析。

这些函数使用 BPF 映射来存储正在跟踪的进程的元数据。它们还使用 BPF 环形缓冲区来存储和提交事件数据。这些函数利用了 eBPF 技术的优点，即在内核中运行，以最小化跨空间的开销，并使用安全的 BPF 虚拟机保护内核免受恶意代码的攻击。

测试脚本如下：

```
/* SPDX-License-Identifier: (LGPL-2.1 OR BSD-2-Clause) */
/* Copyright (c) 2020 Facebook */
#ifndef __BOOTSTRAP_H
#define __BOOTSTRAP_H

#define TASK_COMM_LEN 16
#define MAX_FILENAME_LEN 127

struct event {
    int pid;
    int ppid;
    unsigned exit_code;
    unsigned long long duration_ns;
    char comm[TASK_COMM_LEN];
    char filename[MAX_FILENAME_LEN];
    bool exit_event;
};

#endif /* __BOOTSTRAP_H */
```

```
// SPDX-License-Identifier: GPL-2.0 OR BSD-3-Clause
/* Copyright (c) 2020 Facebook */
#include "vmlinux.h"
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_tracing.h>
#include <bpf/bpf_core_read.h>
#include "template.h"

```

```
9. char LICENSE[] SEC("license") = "Dual BSD/GPL";
10.
11. struct {
12.     __uint(type, BPF_MAP_TYPE_HASH);
13.     __uint(max_entries, 8192);
14.     __type(key, pid_t);
15.     __type(value, u64);
16. } exec_start SEC(".maps");
17.
18. struct {
19.     __uint(type, BPF_MAP_TYPE_RINGBUF);
20.     __uint(max_entries, 256 * 1024);
21. } rb SEC(".maps");
22.
23. const volatile unsigned long long min_duration_ns = 0;
24.
25. SEC("tp/sched/sched_process_exec")
26. int handle_exec(struct trace_event_raw_sched_process_exec *ctx)
27. {
28.     struct task_struct *task;
29.     unsigned fname_off;
30.     struct event *e;
31.     pid_t pid;
32.     u64 ts;
33.
34.     /* remember time exec() was executed for this PID */
35.     pid = bpf_get_current_pid_tgid() >> 32;
36.     ts = bpf_ktime_get_ns();
37.     bpf_map_update_elem(&exec_start, &pid, &ts, BPF_ANY);
38.
39.     /* don't emit exec events when minimum duration is specified */
40.     if (min_duration_ns)
41.         return 0;
42.
43.     /* reserve sample from BPF ringbuf */
44.     e = bpf_ringbuf_reserve(&rb, sizeof(*e), 0);
45.     if (!e)
46.         return 0;
47.
48.     /* fill out the sample with data */
49.     task = (struct task_struct *)bpf_get_current_task();
50.
51.     e->exit_event = false;
52.     e->pid = pid;
53.     e->ppid = BPF_CORE_READ(task, real_parent, tgid);
```

```
54.     bpf_get_current_comm(&e->comm, sizeof(e->comm));
55.
56.     fname_off = ctx->__data_loc_filename & 0xFFFF;
57.     bpf_probe_read_str(&e->filename, sizeof(e->filename), (void *)ctx + fname_off);
58.
59.     /* successfully submit it to user-space for post-processing */
60.     bpf_ringbuf_submit(e, 0);
61.     return 0;
62. }
63.
64. SEC("tp/sched/sched_process_exit")
65. int handle_exit(struct trace_event_raw_sched_process_template* ctx)
66. {
67.     struct task_struct *task;
68.     struct event *e;
69.     pid_t pid, tid;
70.     u64 id, ts, *start_ts, duration_ns = 0;
71.
72.     /* get PID and TID of exiting thread/process */
73.     id = bpf_get_current_pid_tgid();
74.     pid = id >> 32;
75.     tid = (u32)id;
76.
77.     /* ignore thread exits */
78.     if (pid != tid)
79.         return 0;
80.
81.     /* if we recorded start of the process, calculate lifetime duration */
82.     start_ts = bpf_map_lookup_elem(&exec_start, &pid);
83.     if (start_ts)
84.         duration_ns = bpf_ktime_get_ns() - *start_ts;
85.     else if (min_duration_ns)
86.         return 0;
87.     bpf_map_delete_elem(&exec_start, &pid);
88.
89.     /* if process didn't live long enough, return early */
90.     if (min_duration_ns && duration_ns < min_duration_ns)
91.         return 0;
92.
93.     /* reserve sample from BPF ringbuf */
94.     e = bpf_ringbuf_reserve(&rb, sizeof(*e), 0);
95.     if (!e)
96.         return 0;
97.
98.     /* fill out the sample with data */
```

```
    task = (struct task_struct *)bpf_get_current_task();

    e->exit_event = true;
    e->duration_ns = duration_ns;
    e->pid = pid;
    e->ppid = BPF_CORE_READ(task, real_parent, tgid);
    e->exit_code = (BPF_CORE_READ(task, exit_code) >> 8) & 0xff;
    bpf_get_current_comm(&e->comm, sizeof(e->comm));

    /* send data to user-space for post-processing */
    bpf_ringbuf_submit(e, 0);
    return 0;
}
```

2.4.1 代码分析 2

分析 tcp.h 和 tcp.c 文件功能如下：

这两段代码使用 eBPF 技术实现了跟踪 TCP 连接事件的内核模块。tcp.h 和 tcp.c 中定义了用于存储事件数据的结构体 tcp_conn_event。该结构体包含了事件的时间戳、源 IP 地址、目的 IP 地址、源端口、目的端口和连接类型。tcp.c 中的 trace_tcp_connect 函数使用 kprobe 跟踪 tcp_v4_connect 函数的调用，并从 TCP 套接字中提取 IP 和 TCP 头以填充 tcp_conn_event 结构。该函数使用 bpf_perf_event_output 函数将事件数据发送到用户空间。

tcp.h 和 tcp.c 的实现可以参考了以下函数：

trace_tcp_connect：此函数使用 kprobe 跟踪 tcp_v4_connect 函数的调用，并提取 IP 和 TCP 头以填充 tcp_conn_event 结构。该函数使用 bpf_perf_event_output 函数将事件数据发送到用户空间。

测试脚本如下：

```
#ifndef _TCP_H
#define _TCP_H

#include <linux/types.h>


struct tcp_conn_event {
    __u64 timestamp_ns;
    __be32 saddr;
     __be32 daddr;
     __be16 sport;
```

```
    __be16 dport;
    __u32 conn_type;
};

#endif /* _TCP_H */
```

```
#include "tcp.h"
#include <linux/bpf.h>
#include <linux/tcp.h>
#include <linux/ptrace.h>
#include <linux/ip.h>

#include <bpf/bpf_helpers.h>
#include <bpf/bpf_tracing.h>
#include <bpf/bpf_core_read.h>
#include <bpf/libbpf.h>


struct {
    __uint(type, BPF_MAP_TYPE_PERF_EVENT_ARRAY);
    __uint(key_size, sizeof(int));
    __uint(value_size, sizeof(int));
} tcp_conn_events SEC(".maps");

SEC("kprobe/tcp_v4_connect")

int trace_tcp_connect(struct pt_regs *ctx) {
    struct tcp_conn_event event = {};
    struct sock *skp = NULL;
    struct tcphdr *tcph = NULL;
    struct iphdr *iph = NULL;

    // Get the socket from the function argument
    skp = (struct sock *)PT_REGS_PARM1(ctx);

    // Extract IP and TCP headers
    iph = (struct iphdr *)((__u32 *)skp + 1);
    tcph = (struct tcphdr *)(iph + 1);

    // Fill in the event data
    event.timestamp_ns = bpf_ktime_get_ns();
    event.saddr = iph->saddr;
    event.daddr = iph->daddr;
    event.sport = tcph->source;
```

```
39.     event.dport = tcph->dest;
40.     event.conn_type = 0; // 0 for connection setup
41.
42.     // Send the event to userspace
43.     bpf_perf_event_output(ctx, &tcp_conn_events, BPF_F_CURRENT_CPU, &event, sizeof(event))
    ;
44.
45.     return 0;
46. }
47.
48. char _license[] SEC("license") = "GPL";
```

2.5 性能测试

部署未成功，暂时无法进行

3 测试报告补充说明

测试工作仓库地址：https://github.com/zhsh9/STforGroup2