pyLCC原理和开发手册

1.什么是LCC?

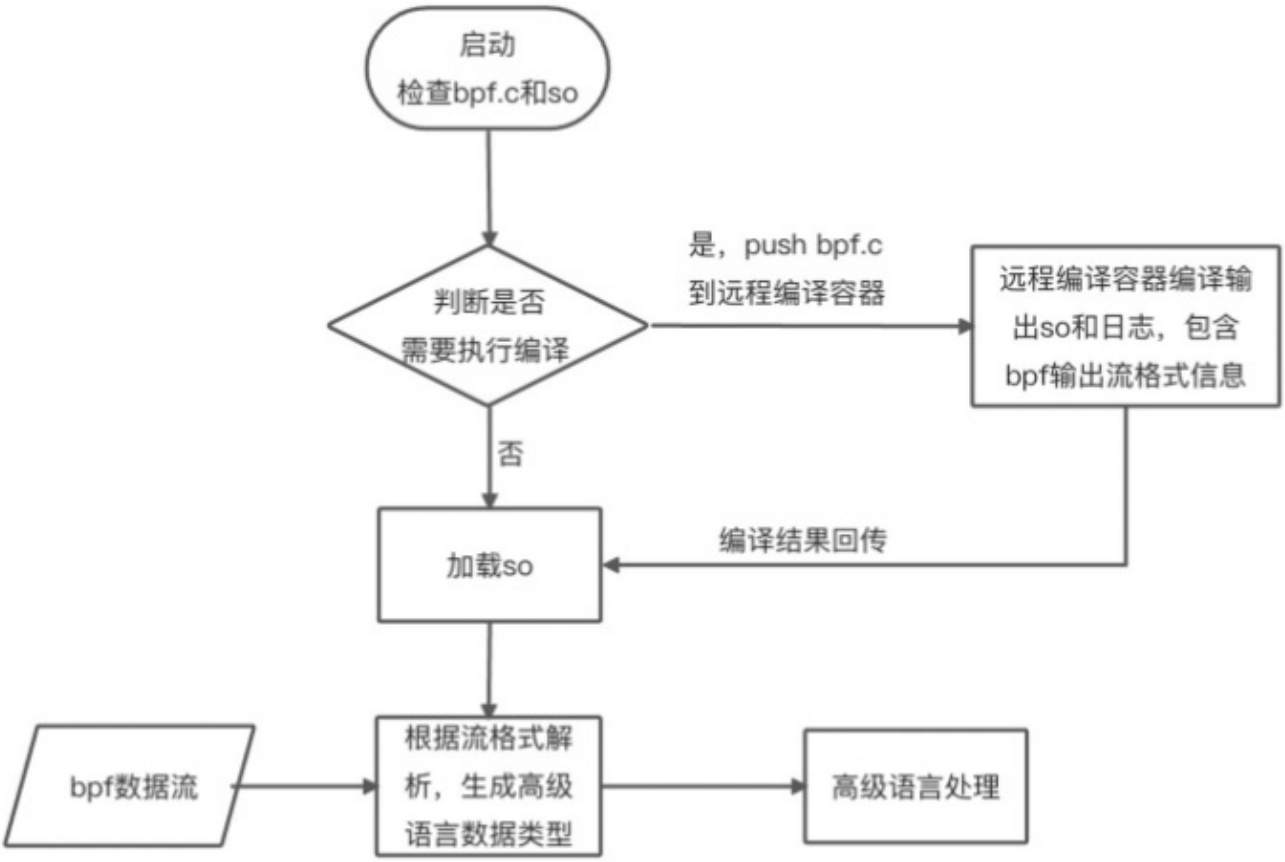
LCC（LibbpfCompilerCollection）是基于CO-RE的理念，通过libbpf把一个复杂的编译和编写工程进行极简设计，免去环境搭建，简化代码编写，优秀的格式化的数据处理能力。目前支持两种语言的开发方式（c和python），希望解放生产力，让开发者更多专注于核心的功能开发。

pyLCC：通过将复杂编译过程交由远程容器执行，支持高级语言的极简代码编写和优秀的数据处理能力，一次编译，到处运行，节省资源损耗，使得初学者能快速入门。
gLCC：基于C语言的自动化代码框架生成，支持本地容器编译，真正libbpfCO-RE支持，免去复杂环境搭建，专注功能开发。
本手册，我们先介绍pyLCC的高级语言编译开发能力。

2.pyLCC处理流程

pylcc在libbpf基础上进行封装，其核心是将复杂的编译工程交由容器执行，它具有如下优势：

- pyLCC在libbpf基础上进行封装，将复杂的编译工程交由容器执行，可选本地编译和远程编译
- 本地编译也会在docker容器里运行，无需搭建复杂的环境（安装clang等库）
- 对生产环境没有类似BCCpy脚本运行瞬时资源冲高的现象，没有资源损耗情况
- 封装一些基础的执行函数，类似BCC的代码易用度



3.准备工作

基本要求

- 能力要求：熟悉c，libpf开发特性，python

- python2.7 或者python3都可以运行，无需安装任何第三方库。
- 环境要求：可以访问pylcc.openanolis.cn。后面编译容器发布了以后，可以自行搭建编译服务执行

4. 实战

- 第一步需要执行pip install pylcc安装，把基础库安装好。
  参考：git clone git@github.com:aliyun/surftrace.git
  示例代码 在目录 tool/pylcc/guide下.

4.1 从hello world 开始

hello.py 代码

Python | 复制代码

```python
import time
from pylcc.lbcBase import ClbcBase

bpfPog = r"""
#include "lbc.h"

SEC("kprobe/wake_up_new_task")
int j_wake_up_new_task(struct pt_regs *ctx)
{
    struct task_struct* parent = (struct task_struct
*)PT_REGS_PARM1(ctx);

    bpf_printk("hello lcc, parent: %d\n", _(parent->tgid));
    return 0;
}

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

class Chello(ClbcBase):
    def __init__(self):
        super(Chello, self).__init__("hello", bpf_str=bpfPog)
        while True:
            time.sleep(1)

if __name__ == "__main__":
    hello = Chello()
    pass
```

4.1.1 bpf代码说明：

- bpf代码需要包含 lbc.h 头文件，该头文件会包含以下头文件，并且会加上我们常见的宏定义和数据类型，详情参考后面的附录，

```c
#include "vmlinux.h"
#include <linux/types.h>
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_core_read.h>
#include <bpf/bpf_tracing.h>
```

- SEC的定义和函数内部实现与libbpf应用方法保持一致；
- 访问结构体成员使用了_宏，该方法访问方式相对固定，下一节会提供core的获取方法；
- 末尾不要遗忘 _license声明

4.1.2 python代码实现部分说明:

python 部分代码从ClbcBase 类继承，__init__函数中，第一入参必须要指定，用于指定生成so的文件名。在执行完__init__函数后，bfp模块就已经注入到内核当中去执行了。

4.1.3 执行效果：

执行 python2 hello.py 运行，并查看编译结果：

```
#cat /sys/kernel/debug/tracing/trace_pipe
           <...>-1091294 [005] d... 17658161.425644: : hello lcc, parent: 106880
           <...>-4142485 [003] d... 17658161.428568: : hello lcc, parent: 4142485
           <...>-4142486 [002] d... 17658161.430972: : hello lcc, parent: 4142486
           <...>-4142486 [002] d... 17658161.431228: : hello lcc, parent: 4142486
           <...>-4142486 [002] d... 17658161.431557: : hello lcc, parent: 4142486
           <...>-4142485 [003] d... 17658161.435385: : hello lcc, parent: 4142485
           <...>-4142490 [000] d... 17658161.437562: : hello lcc, parent: 4142490
```

此时可以看到目录下新增了hello.so 文件，如果文件时间戳有更新，只要bpfProg部分内容不发生改变，就不会触发重编动作。如果bpfProg 发生变换，就会触发重新编译动作，生成新的so

4.2 往用户态传递信息

代码参考 eventOut.py

Python | 复制代码

```python
import ctypes as ct
from pylcc.lbcBase import ClbcBase

bpfPog = r"""
#include "lbc.h"
#define TASK_COMM_LEN 16
struct data_t {
    u32 c_pid;
    u32 p_pid;
    char c_comm[TASK_COMM_LEN];
    char p_comm[TASK_COMM_LEN];
};

LBC_PERF_OUTPUT(e_out, struct data_t, 128);
SEC("kprobe/wake_up_new_task")
int j_wake_up_new_task(struct pt_regs *ctx)
{
    struct task_struct* parent = (struct task_struct *)PT_REGS_PARM1(ctx);
    struct data_t data = {};

    data.c_pid = bpf_get_current_pid_tgid() >> 32;
    bpf_get_current_comm(&data.c_comm, TASK_COMM_LEN);
    data.p_pid = BPF_CORE_READ(parent, pid);
    bpf_core_read(&data.p_comm[0], TASK_COMM_LEN, &parent->comm[0]);

    bpf_perf_event_output(ctx, &e_out, BPF_F_CURRENT_CPU, &data, sizeof(data));
    return 0;
}

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

class CeventOut(ClbcBase):
    def __init__(self):
        super(CeventOut, self).__init__("eventOut", bpf_str=bpfPog)

    def _cb(self, cpu, data, size):
        stream = ct.string_at(data, size)
        e = self.maps['e_out'].event(stream)
        print("current pid:%d, comm:%s. wake_up_new_task pid: %d, comm: %s" % (
            e.c_pid, e.c_comm, e.p_pid, e.p_comm
        ))
```

```
43
44          def loop(self):
45              self.maps['e_out'].open_perf_buffer(self._cb)
46              try:
47                  self.maps['e_out'].perf_buffer_poll()
48              except KeyboardInterrupt:
49                  print("key interrupt.")
50                  exit()
51
52      if __name__ == "__main__":
53          e = CeventOut()
54          e.loop()
```

4.2.1 bpf部分代码说明：

- LBC_PERF_OUTPUT宏不能用原有的bpf_map_def ..... BPF_MAP_TYPE_PERF_EVENT_ARRAY..... 替代，虽然是同样申明一个 perf maps，但如果用原始的声明方式，python在加载的时候将无法识别出对应的内核数据类型。
- 可以使用 bpf_get_current_pid_tgid 等libbpf helper函数；
- 可以使用 bpf_core_read 等方法；
- 不可使用 bcc 独有的方法，如直接指针访问变量等；

4.2.2 python部分代码说明

以loop函数为入口：

- self.maps['e_out'].open_perf_buffer(self._cb)函数是为 e_out事件注册回调钩子函数，其中e_out命名与bpfProg中LBC_PERF_OUTPUT(e_out, struct data_t, 128) 对应；
- self.maps['e_out'].perf_buffer_poll() 即poll 对应的event事件，与bpfProg中bpf_perf_event_output(ctx, &e_out.....对应；

接下来看_cb 回调函数：

- stream = ct.string_at(data, size) 在入参中解析出数据流；
- e = self.maps['e_out'].event(stream) 将数据流生成对应的数据对象；
- 生成了数据对象后，就可以通过成员的方式来访问数据对象，该对象成员与bpfProg中struct data_t 定义保持一致

4.2.3 执行结果

```
python2 eventOut.py
current pid:241808, comm:python. wake_up_new_task parent pid: 241871, comm: python
current pid:1, comm:systemd. wake_up_new_task parent pid: 1, comm: systemd
……
```

4.3 动态修改bpfProg代码

在3.2的基础上，参考dynamicVar.py，如果只想动态过滤parent进程id为 241871，可以借鉴bcc的思路进行替换，大部分代码与eventOut.py一致，首先在bpfProg代码添加了过滤动作：

```c
    ……
    u32 pid = BPF_CORE_READ(parent, pid);
    if (pid != FILTER_PID) {
        return 0;
    }
    ……
```

然后在main入口处进行替换:

```c
if __name__ == "__main__":
    bpfPog = bpfPog.replace("FILTER_PID", sys.argv[1])
    e = CdynamicVar()
    e.loop
```

将要过滤的参数传入，执行效果

```
python2 dynamicVar.py 241871
current pid:241808, comm:python. wake_up_new_task pid: 241871, comm: python
current pid:241808, comm:python. wake_up_new_task pid: 241871, comm: python
current pid:241808, comm:python. wake_up_new_task pid: 241871, comm: python
```

4.4 hash map应用

代码参考 hashMap.py，大部分代码与eventOut.py一致。

4.4.1 bpf 部分代码

定义hashmap

```
LBC_HASH(pid_cnt, u32, u32, 1024);
```

使用方法和libbfp一致

```
    u32 *pcnt, cnt;

    pcnt =  bpf_map_lookup_elem(&pid_cnt, &pid);
    cnt  = pcnt ? *pcnt + 1 : 1;
    bpf_map_update_elem(&pid_cnt, &pid, &cnt, BPF_ANY);
```

4.4.2 python部分代码

查询maps的位置在exit退出之前打印所有信息

```python
            ……
            dMap = self.maps['pid_cnt']
            print(dMap.get())
            exit()
```

哈希表对象可以直接由 self.maps['pid_cnt'] 方法获取到，可以调用get函数，获取到dict对象。除了BPF_MAP_TYPE_HASH，lcc当前还支持BPF_MAP_TYPE_LRU_HASH、BPF_MAP_TYPE_PERCPU_HASH、 BPF_MAP_TYPE_LRU_PERCPU_HASH等类型，更多类型支持在完善中，敬请期待。

4.5、call stack获取

获取内核调用栈是bpf一项非常重要的调试功能，参考 callStack.py，大部分代码与eventOut.py一致。

4.5.1、bpf部分代码说明

外传的数据结构体中增加stack_id成员，接下来定义一个call stack成员

```c
struct data_t {
    u32 c_pid;
    u32 p_pid;
    char c_comm[TASK_COMM_LEN];
    char p_comm[TASK_COMM_LEN];
    u32 stack_id;
};

LBC_PERF_OUTPUT(e_out, struct data_t, 128);
LBC_STACK(call_stack,32);
```

在处理函数中记录call stack

```c
data.stack_id = bpf_get_stackid(ctx, &call_stack, KERN_STACKID_FLAGS);
```

4.5.2、python部分代码

通过getStacks传入stack_id，即可获取调用栈符号数组，然后列出来即可:

```c
        stacks = self.maps['call_stack'].getStacks(e.stack_id)
        print("call trace:")
        for s in stacks:
            print(s)
```

4.5.3、执行结果

```python
python callStack.py
remote server compile success.
current pid:1, comm:systemd. wake_up_new_task pid: 1, common: systemd
call trace:
startup_64
do_syscall_64
entry_SYSCALL_64_after_swapgs
```

4.6、py与bpf.c文件分离

参考 codeSeparate.py 和 independ.bpf.c，它的功能实现和eventOut.py 完全一致，不一样的是将python和bpf.c的功能拆分到了两个文件中去实现。 我们只需要关注下__init__函数

```python
    def __init__(self):
        super(codeSeparate, self).__init__("independ")
```

它没有了 bpf_str 入参，此时lcc会尝试从当前目录上下，去找independ.bpf.c并提请编译加载。

5 附录、

5.1、lbc.h头文件已定义的信息

C 复制代码

```c
#ifndef LBC_LBC_H
#define LBC_LBC_H

#define _LINUX_POSIX_TYPES_H
#define __ASM_GENERIC_POSIX_TYPES_H

#define PERF_MAX_STACK_DEPTH 127
#define BPF_F_FAST_STACK_CMP    (1ULL << 9)

#define KERN_STACKID_FLAGS (0 | BPF_F_FAST_STACK_CMP)
#define USER_STACKID_FLAGS (0 | BPF_F_FAST_STACK_CMP | BPF_F_USER_STACK)

typedef unsigned long long u64;
typedef signed long long s64;
typedef unsigned int u32;
typedef signed int s32;
typedef unsigned short u16;
typedef signed short s16;
typedef unsigned char u8;
typedef signed char s8;

enum {
    BPF_ANY         = 0, /* create new element or update existing */
    BPF_NOEXIST     = 1, /* create new element if it didn't exist */
    BPF_EXIST       = 2, /* update existing element */
    BPF_F_LOCK      = 4, /* spin_lock-ed map_lookup/map_update */
};

#define LBC_PERF_OUTPUT(MAPS, CELL, ENTRIES) \
    struct bpf_map_def SEC("maps") MAPS = { \
        .type = BPF_MAP_TYPE_PERF_EVENT_ARRAY, \
        .key_size = sizeof(int), \
        .value_size = sizeof(s32), \
        .max_entries = ENTRIES, \
    }

#define LBC_HASH(MAPS, KEY_T, VALUE_T, ENTRIES) \
    struct bpf_map_def SEC("maps") MAPS = { \
        .type = BPF_MAP_TYPE_HASH, \
        .key_size = sizeof(KEY_T), \
        .value_size = sizeof(VALUE_T), \
        .max_entries = ENTRIES, \
    }

#define LBC_LRU_HASH(MAPS, KEY_T, VALUE_T, ENTRIES) \
```

```
46      struct bpf_map_def SEC("maps") MAPS = { \
47          .type = BPF_MAP_TYPE_LRU_HASH, \
48          .key_size = sizeof(KEY_T), \
49          .value_size = sizeof(VALUE_T), \
50          .max_entries = ENTRIES, \
51      }
52
53  #define LBC_PERCPU_HASH(MAPS, KEY_T, VALUE_T, ENTRIES) \
54      struct bpf_map_def SEC("maps") MAPS = { \
55          .type = BPF_MAP_TYPE_PERCPU_HASH, \
56          .key_size = sizeof(KEY_T), \
57          .value_size = sizeof(VALUE_T), \
58          .max_entries = ENTRIES, \
59      }
60
61  #define LBC_LRU_PERCPU_HASH(MAPS, KEY_T, VALUE_T, ENTRIES) \
62      struct bpf_map_def SEC("maps") MAPS = { \
63          .type = BPF_MAP_TYPE_LRU_PERCPU_HASH, \
64          .key_size = sizeof(KEY_T), \
65          .value_size = sizeof(VALUE_T), \
66          .max_entries = ENTRIES, \
67      }
68
69  #define LBC_STACK(MAPS, ENTRIES) \
70      struct bpf_map_def SEC("maps") MAPS = { \
71          .type = BPF_MAP_TYPE_STACK_TRACE, \
72          .key_size = sizeof(u32), \
73          .value_size = PERF_MAX_STACK_DEPTH * sizeof(u64), \
74          .max_entries = ENTRIES, \
75      }
76
77  #define _(P) ({typeof(P) val = 0; bpf_probe_read((void*)&val,
    sizeof(val), (const void*)&P); val;})
78
79  #include "vmlinux.h"
80  #include <linux/types.h>
81  #include <bpf/bpf_helpers.h>
82  #include <bpf/bpf_core_read.h>
83  #include <bpf/bpf_tracing.h>
84
85  #ifndef NULL
86  #define NULL ((void*)0)
87  #endif
88  #ifndef ntohs
89  #define ntohs(x) (0xff00 & x << 8) \
90                  |(0x00ff & x >> 8)
91  #endif
92  #ifndef ntohl
```

```
93   #define ntohl(x) (0xff000000 & x << 24) \
94                    |(0x00ff0000 & x <<  8) \
95                    |(0x0000ff00 & x >>  8) \
96                    |(0x000000ff & x >> 24)
97   #endif
98   #ifndef ntohll
99   #define ntohll(x) ((((long long)ntohl(x))<<32) + (ntohl((x)>>32)))
100  #endif
101  #define BPF_F_CURRENT_CPU 0xffffffffULL
102
103  #endif //LBC_LBC_H
```

--- 阿里云龙蜥社区系统运维SIG