Программирование в Linux

Extended BPF

Коротко о том, что было в прошлой серии (classic BPF)

- В ядре есть виртуальная машина
- Мы можем запускать в ней “произвольный” код, созданный в userspace
- Для написания этого кода есть специальный ассемблер
- Наш код можно вешать на разные подсистемы, чтобы обрабатывать события: сетевые пакеты, системные вызовы…
- Виртуальная машина простая и ограниченная
- Загрузка и подключение нашего кода выполняются одновременно

extended BPF

Виртуальная машина улучшена:

11 регистров (r0-r10)
512 байт стека
Можно использовать “неограниченную” разделяемую память (maps)

Можно вызывать некоторые функции ядра kernel helpers (пример: bpf_trace_printk)

Программы можно писать на C
(есть bpf-бэкенд llvm).

У функций может быть не более 5 аргументов

```
1  int foo(void *ctx)
2  {
3      return ctx ? 0 : 1;
4  }
```

```
TERMINAL  PROBLEMS  OUTPUT  DEBUG CONSOLE

dmis@dmis-MS-7A15:~/LinuxEgs/bpf_egs$ clang -target bpf -c foo.c -o foo
dmis@dmis-MS-7A15:~/LinuxEgs/bpf_egs$ llvm-objdump-11 -d foo

foo:    file format elf64-bpf


Disassembly of section .text:

0000000000000000 <foo>:
       0:       7b 1a f8 ff 00 00 00 00 *(u64 *)(r10 - 8) = r1
       1:       79 a1 f8 ff 00 00 00 00 r1 = *(u64 *)(r10 - 8)
       2:       b7 02 00 00 00 00 00 00 r2 = 0
       3:       b7 03 00 00 01 00 00 00 r3 = 1
       4:       7b 2a f0 ff 00 00 00 00 *(u64 *)(r10 - 16) = r2
       5:       7b 3a e8 ff 00 00 00 00 *(u64 *)(r10 - 24) = r3
       6:       15 01 02 00 00 00 00 00 if r1 == 0 goto +2 <LBB0_2>
       7:       79 a1 f0 ff 00 00 00 00 r1 = *(u64 *)(r10 - 16)
       8:       7b 1a e8 ff 00 00 00 00 *(u64 *)(r10 - 24) = r1

0000000000000048 <LBB0_2>:
       9:       79 a1 e8 ff 00 00 00 00 r1 = *(u64 *)(r10 - 24)
      10:       bf 10 00 00 00 00 00 00 r0 = r1
      11:       95 00 00 00 00 00 00 00 exit
```

В eBPF загрузка и подключение разделены.

Загружаем программу -> проходим верификацию -> получаем объект ядра (fd)

Дальше подключаем его куда хотим.

Также сами создаем объекты разделяемой памяти (maps)

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

int bpf(int cmd, union bpf_attr *attr, unsigned int size);
```

Примеры cmd:

BPF_MAP_CREATE — создать объект разделяемой памяти

BPF_MAP_LOOKUP_ELEM
BPF_MAP_UPDATE_ELEM
BPF_MAP_DELETE_ELEM
BPF_MAP_GET_NEXT_KEY

BPF_PROG_LOAD — загрузить и проверифицировать bpf-код

И еще около 30 не описанных в man команд

Lifetime BPF-объектов

BPF-объекты (prog, map) живут, пока
есть хотя бы одна ссылка на них:

- Открытый fd
- Подключенная подсистема событий
- “файл” в bpf vfs

Можно продлевать жизнь объектам,
прибивая их к файловой системе
(BPF_OBJ_PIN)

```c
__attribute__((section("xdp"), used))
int test(void *ctx)
{
        return 0;
}

char _license[] __attribute__((section("license"), used)) = "GPL";
```

```
dmis@dmis-MS-7A15:~/LinuxEgs/bpf_egs$ clang -target bpf -c test.c -o test.o
dmis@dmis-MS-7A15:~/LinuxEgs/bpf_egs$ mkdir bpf-mountpoint
dmis@dmis-MS-7A15:~/LinuxEgs/bpf_egs$ sudo mount -t bpf none bpf-mountpoint
```

```
$ sudo strace -e bpf bpftool prog load ./test.o bpf-mountpoint/test
bpf(BPF_PROG_LOAD, {prog_type=BPF_PROG_TYPE_XDP, prog_name="test", ...},
120) = 3
bpf(BPF_OBJ_PIN, {pathname="bpf-mountpoint/test", bpf_fd=3}, 120) = 0
```

```
dmis@dmis-MS-7A15:~/LinuxEgs/bpf_egs$ ls bpf-mountpoint/
test
```

libbpf + bpftool (код фильтра)

bpf_egs > C xdp-simple.bpf.c > ...

```c
#include "vmlinux.h"
#include <bpf/bpf_helpers.h>

#define MAX_CPU 8

struct bpf_map_def SEC("maps") my_map = {
    .type = BPF_MAP_TYPE_ARRAY,
    .key_size = sizeof(u32),
    .value_size = sizeof(u64),
    .max_entries = MAX_CPU

};

SEC("xdp/simple")
int simple(void *ctx)
{
    u32 key = bpf_get_smp_processor_id();
    u64* val = bpf_map_lookup_elem(&my_map, &key);
    if (!val)
        return XDP_ABORTED;

    *val += 1;
    return XDP_PASS;
}

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

Современная версия eBPF поддерживает подход compile once - run everywhere (CO-RE).
Для этого нам нужен заголовок с описанием метаинформации (BTF, binary type format) о структурах ядра (надо его сгенерить один раз)

$ bpftool btf dump file /sys/kernel/btf/vmlinux format c > vmlinux.h

Собираем
$ clang -O2 -c -target bpf xdp-simple.bpf.c -o xdp-simple.bpf.o

Сгенерим обертки для загрузки фильтра из C-кода
$ bpftool gen skeleton xdp-simple.bpf.o > xdp-simple.skel.h

libbpf + bpftool (код загрузки)

Программа цепляет фильтр к сетевому интерфейсу lo (loopback device, 127.0.0.1)

```c
#include <linux/if_link.h>
#include <err.h>
#include <unistd.h>
#include "xdp-simple.skel.h"


int main(int argc, char **argv)
{
    uint32_t flags = XDP_FLAGS_SKB_MODE;
    struct xdp_simple_bpf *obj;

    obj = xdp_simple_bpf__open_and_load();
    if (!obj)
        err(1, "failed to open and/or load BPF object\n");

    // отсоединяем ранее прикрученные фильтры от сетевого интервейса 1 (lo)
    const int iface = 1;
    bpf_set_link_xdp_fd(iface, -1, flags);
    // подключаем наш фильтр
    bpf_set_link_xdp_fd(iface, bpf_program__fd(obj->progs.simple), flags);

cleanup:
    xdp_simple_bpf__destroy(obj);
}
```

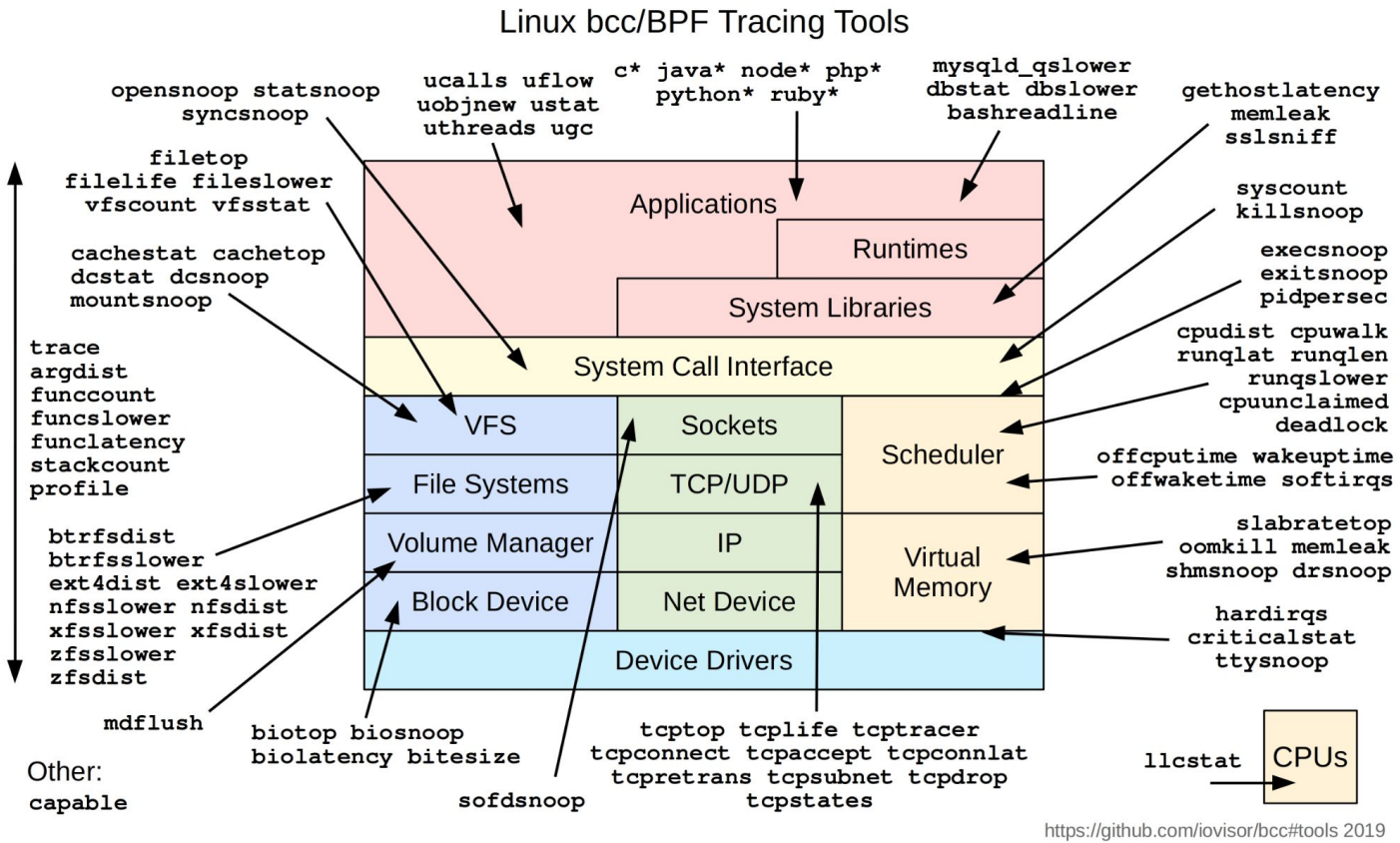
```
dmis@dmis-MS-7A15:~/LinuxEgs/bpf_egs$ sudo ./xdp-simple
[sudo] password for dmis:
dmis@dmis-MS-7A15:~/LinuxEgs/bpf_egs$ ip l show dev lo
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 xdpgeneric qdisc noqueue state UNKNOWN mode DEFAULT
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    prog/xdp id 46
dmis@dmis-MS-7A15:~/LinuxEgs/bpf_egs$ for s in `seq 234`; do sudo ping -f -c 100 127.0.0.1
dmis@dmis-MS-7A15:~/LinuxEgs/bpf_egs$ sudo bpftool map dump name my_map
key: 00 00 00 00  value: 73 24 00 00 00 00 00 00
key: 01 00 00 00  value: 54 26 00 00 00 00 00 00
key: 02 00 00 00  value: c5 2f 00 00 00 00 00 00
key: 03 00 00 00  value: fa 3f 00 00 00 00 00 00
key: 04 00 00 00  value: 00 00 00 00 00 00 00 00
key: 05 00 00 00  value: 00 00 00 00 00 00 00 00
key: 06 00 00 00  value: 00 00 00 00 00 00 00 00
key: 07 00 00 00  value: 00 00 00 00 00 00 00 00
Found 8 elements
```

```
$ clang -O2 -o xdp-simple xdp-simple.c -lelf -lbpf -lz
```

Текущее состояние дел

- 20+ типов поддерживаемых eBPF-программ (тип определяет, куда можно вешать фильтр, и что ему разрешено)
  - Для обработки сетевого трафика
  - Для трассировки и сбора статистики
- Активное развитие и недописанная документация (см. man)
- eBPF фильтры пока нельзя использовать в seccomp
- Книг с примерами мало
  - Linux Observability with BPF: Advanced Programming for Performance Analysis and Networking
- Писать на C мало кто хочет, потому написаны утилиты для более простого создания различных трассирующих ebpf-программ:
  - bcc (обертки на разных языках для userspace)
  - bpftrace (Собственный язык скриптов)

bcc


Linux bcc/BPF Tracing Tools
opensnoop statsnoop
syncsnoop
ucalls uflow
uobjnew ustat
uthreads ugc
c* java* node* php*
python* ruby*
mysqld_qslower
dbstat dbslower
bashreadline
gethostlatency
memleak
sslsniff
filetop
filelife fileslower
vfscount vfsstat
syscount
killsnoop
cachestat cachetop
dcstat dcsnoop
mountsnoop
execsnoop
exitsnoop
pidpersec
trace
argdist
funccount
funcslower
funclatency
stackcount
profile
cpudist cpuwalk
runqlat runqlen
runqslower
cpuunclaimed
deadlock
offcputime wakeuptime
offwaketime softirqs
btrfsdist
btrfsslower
ext4dist ext4slower
nfsslower nfsdist
xfsslower xfsdist
zfsslower
zfsdist
slabratetop
oomkill memleak
shmsnoop drsnoop
hardirqs
criticalstat
ttysnoop
mdflush
biotop biosnoop
biolatency bitesize
sofdsnoop
tcptop tcplife tcptracer
tcpconnect tcpaccept tcpconnlat
tcpretrans tcpsubnet tcpdrop
tcpstates
llcstat
CPUs
Other:
capable
Applications
Runtimes
System Libraries
System Call Interface
VFS
Sockets
Scheduler
File Systems
TCP/UDP
Volume Manager
IP
Virtual Memory
Block Device
Net Device
Device Drivers
https://github.com/iovisor/bcc#tools 2019

```
17  from bcc import BPF
18  from time import sleep
19
20  # load BPF program
21  b = BPF(text="""
22  #include <uapi/linux/ptrace.h>
23  #include <linux/blkdev.h>
24
25  BPF_HISTOGRAM(dist);
26  BPF_HISTOGRAM(dist_linear);
27
28  int kprobe__blk_account_io_done(struct pt_regs *ctx, struct request *req)
29  {
30          dist.increment(bpf_log2l(req->__data_len / 1024));
31          dist_linear.increment(req->__data_len / 1024);
32          return 0;
33  }
34  """)
35
36  # header
37  print("Tracing... Hit Ctrl-C to end.")
38
39  # trace until Ctrl-C
40  try:
41          sleep(99999999)
42  except KeyboardInterrupt:
43          print()
44
45  # output
46  print("log2 histogram")
47  print("~~~~~~~~~~~~~~~")
48  b["dist"].print_log2_hist("kbytes")
49
50  print("\nlinear histogram")
51  print("~~~~~~~~~~~~~~~~")
52  b["dist_linear"].print_linear_hist("kbytes")
```

Код самой ebpf-программы пишут на C (с неявно определенными макросами и функциями)
Userspace — высокоуровневые обертки

bpftrace

```
12   * Copyright 2018 Netflix, Inc.
13   * Licensed under the Apache License, Version 2.0 (the "License")
14   *
15   * 06-Sep-2018  Brendan Gregg   Created this.
16   */
17
18  BEGIN
19  {
20          printf("Tracing bash commands... Hit Ctrl-C to end.\n");
21          printf("%-9s %-6s %s\n", "TIME", "PID", "COMMAND");
22  }
23
24  uretprobe:/bin/bash:readline
25  {
26          time("%H:%M:%S  ");
27          printf("%-6d %s\n", pid, str(retval));
28  }
```

Для тех, кому совсем не хочется мучиться с C.

Специальный скриптовый язык для написания ebpf и userspace части