eBPF Sviluppo di tools

Riccardo Colavita

Indice

Capitolo 1

Introduzione

Lo scopo di questa relazione è quello di andare ad analizzare le performance di un sistema Linux attraverso alcuni strumenti di monitoring.
Ci soffermeremo in particolar modo su strumenti quali **eBPF** (Extended Berkeley Packet Filter) che vengono aggiunti nel kernel del sistema operativo Linux 4.x, permettendo a BPF di fare molto di più del semplice filtraggio di pacchetti. Questi miglioramenti consentono l'esecuzione di programmi personalizzati per l'analisi delle performance di un sistema Linux attraverso dynamic tracing, static tracing e profiling events.

1.1 Cos'è eBPF

eBPF è un linguaggio simile all'assembly che consente di scrivere programmi in spazio utente e eseguirli nel kernel di Linux. È basato su una versione precedente (BPF o cBPF) ed è stato sottoposto a un forte sviluppo negli ultimi due anni.
Inizialmente consisteva nell'iniettare un semplice bytecode dallo spazio utente nel kernel, dove veniva controllato da un verifier, per evitare crash del kernel o problemi di sicurezza, e mandato in esecuzione ogni volta che la socket a cui era collegato riceveva un pacchetto. La semplicità del linguaggio e l'esistenza di macchine di compilazione (JIT) presenti nel kernel, sono stati per BPF, degli ottimi fattori per quel che riguarda le prestazioni di questo strumento.
Negli ultimi anni sono apparse nuove funzionalità come l'introduzione di mappe e tail calls. Le macchine di compilazione sono state riscritte e il nuovo linguaggio e ancor più vicino al linguaggio macchina nativo rispetto a cBPF. Inoltre sono stati creati nuovi punti di attacco nel kernel.

La tracciabilità di Linux è di conseguenza migliorata. Strumenti come ftrace e perfevents fanno ora parte del kernel di Linux, e permettono di analizzare le prestazioni di Linux in dettaglio e di risolvere problemi legati al processore, al kernel e alle applicazioni.
Grazie a queste nuove funzionalità, i programmi eBPF possono essere designati per vari casi d'uso, che si dividono in due campi di applicazione. Uno di questi è il dominio del *tracing kernel* e del monitoraggio degli eventi. I programmi BPF possono essere associati a sonde e possono essere confrontati con altri metodi di tracing.
L'altro dominio di applicazione è riservato alla programmazione di rete. Oltre ai filtri sulle socket, i programmi eBPF possono essere allacciati alle interfacce di ingresso o uscita di **tc** (Linux Traffic Control tool), ed eseguiti su varie attività di elaborazione di pacchetti, in modo efficiente.

Capitolo 2

Eventi Linux e Supporto BPF

In questo capitolo parleremo di come eBPF migliora l'analisi e il monitoring su sistemi Linux, consentendo l'esecuzione di mini programmi su eventi di tracing.

2.1 Architettura eBPF

BPF non viene definito solo dal set di istruzioni che mette a disposizione per Linux, ma offre anche un ulteriore infrastruttura attorno ad esso.

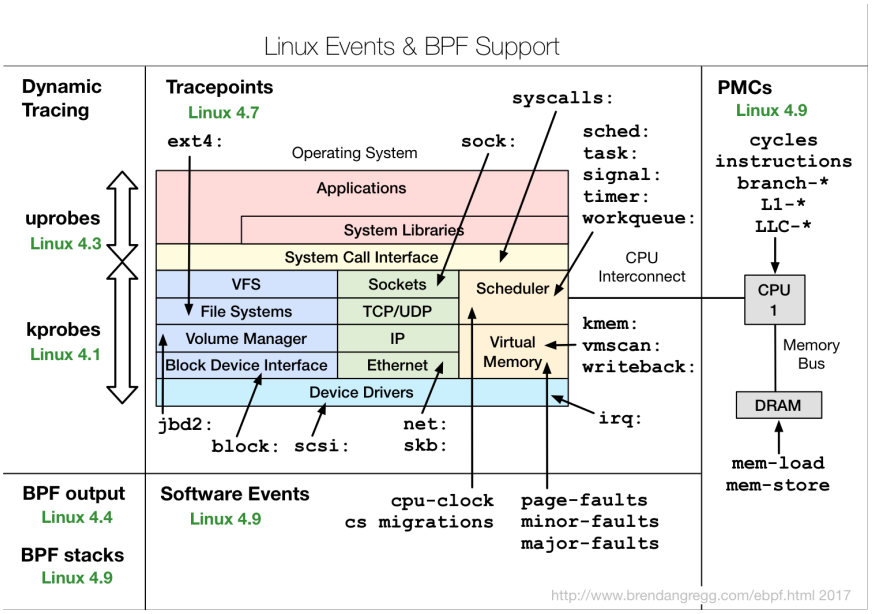

Figura 2.1

La figura 2.1 mostra alcuni degli eventi che possono presentarsi in Linux e i tool offerti da BPF per monitorare questi eventi[1]. Le seguenti sottosezioni forniscono ulteriori dettagli sui singoli aspetti dell'architettura BPF.

2.1.1 Instruction Set

BPF è un set di istruzioni RISC originariamente progettato con lo scopo di scrivere programmi in un sottoinsieme di C che possono essere compilati in istruzioni BPF attraverso un back-end di compilatore (es. LLVM), in modo che il kernel possa successivamente mapparli, attraverso un compilatore JIT che si trova nel kernel, in opcode nativi, per migliorare le prestazioni di esecuzione all'interno del kernel.
I vantaggi di avere a disposizione queste istruzioni nel kernel sono:

- Rendere il kernel programmabile senza dover attraversare i limiti dello spazio kernel/utente.
- Utilizza un approccio flessibile: è possibile creare un programma BPF per gestire solo determinate funzionalità al fine di risparmiare risorse.
- In caso di monitoring di rete, i programmi BPF possono essere aggiornati senza dover riavviare il kernel, system service e senza interrompere il traffico.
- I programmi BPF lavorano insieme al kernel, fanno uso dell'infrastruttura esistente del kernel e dei sui tool così come le garanzie di sicurezza da esso fornite.

Osservazione 2.1. L'esecuzione di un programma BPF all'interno del kernel è sempre guidata dagli eventi!

2.1.2 Funzioni di supporto

Le funzioni di supporto consentono ai programmi BPF di consultare un set di chiamate di funzioni definite nel nucleo per recuperare e inviare dati da/verso il kernel. Le funzioni di supporto disponibili possono differire per ogni tipo di programma BPF, ad esempio, i programmi BPF collegati alle sockets possono solo chiamare un sottoinsieme di helper rispetto ai programmi BPF collegati al livello tc[2].

[1] Anche i sottosistemi del kernel che fanno uso di BPF fanno parte dell'infrastruttura di BPF. e.g. tc e XDP

[2] Vedere la lista dei simboli dei file oggetto (bash: nm)

2.1.3 Maps

Le mappe sono strutture dati efficienti di chiavi/valore che risiedono nello spazio del kernel. È possibile accedervi da un programma BPF e vengono utilizzate per mantenere persistente lo stato del programma BPF. Possono anche essere accessibili tramite descrittori di file dallo spazio utente e possono essere condivise arbitrariamente con altri programmi BPF o altre applicazione definite in spazio utente.
Inoltre non è necessario che i programmi BPF che condividono le stesse mappe siano dello stesso tipo, ad esempio i programmi di tracing possono condividere mappe con i programmi per monitorare la rete.

Osservazione 2.2. Un singolo programma BPF può attualmente accedere direttamente a 64 diverse mappe.

2.1.4 Tail calls

Un altro concetto che può essere utilizzato in BPF si chiama Tail calls. Le chiamate di coda possono essere viste come un meccanismo che consente a un programma BPF di chiamarne un altro, senza tornare al vecchio programma riutilizzando lo stack frame di quello precedente.

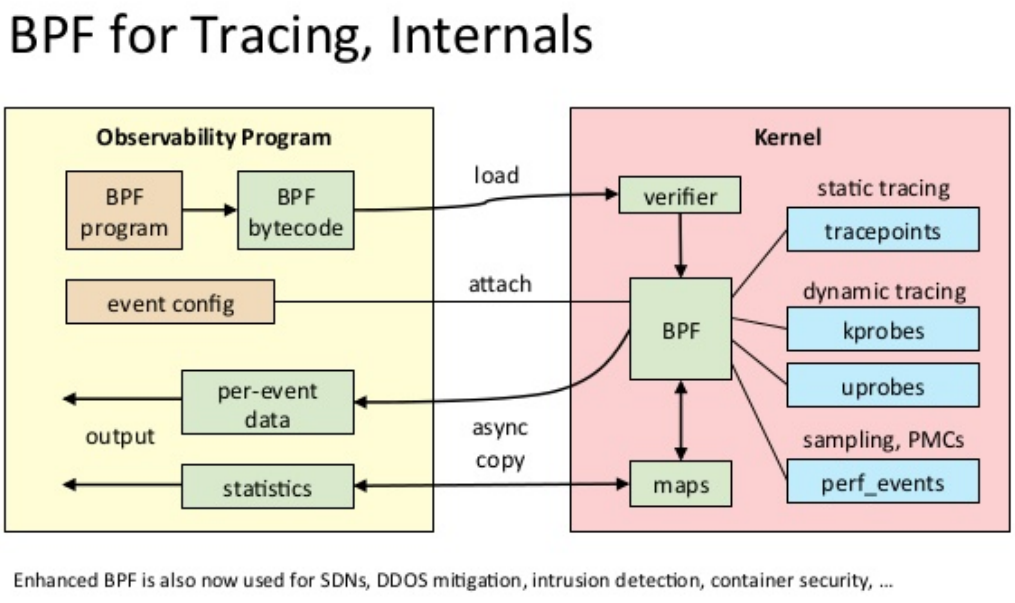

Figura 2.2

La figura 2.2 mostra il workflow relativo all'utilizzo di un programma BPF,

utilizzato per il monitoring. Il codice viene compilato in un BPF bytecode e inviato al kernel, dove il verifier può rifiutarlo se lo ritiene pericoloso[3]. Se il bytecode BPF viene accettato può essere collegato a diverse fonti di eventi:

- **kprobes**: kernel dynamic tracing.
- **uprobes**: user level dynamic tracing.
- **tracepoints**: kernel static tracing.
- **perf events**: timed sampling and PMCs.

Infine il programma BPF ha due modi per trasferire i dati misurati nello spazio dell'utente: fornendo i dettagli per ogni evento o tramite una mappa BPF come avevamo descritto nella sezione 2.1.3.

[3]Il limite massimo di istruzioni per programma è limitato alle 4096 istruzioni BPF, ciò significa che qualsiasi programma terminerà rapidamente. Sebbene il set di istruzioni contenga anche i salti avanti e indietro, il verificatore BPF nel kernel impedirà i loop in modo che la terminazione sia sempre garantita.

Capitolo 3

Sviluppo di Tools

In questo capitolo studieremo lo sviluppo di alcuni strumenti di tracing, in particolare ci soffermeremo sullo sviluppo di due strumenti di monitoring, `cdsnoop` che monitora tutti i processi bash che si spostano tra le cartelle e `pagefaultcount` che conta il numero di page-fault causati dai processi in esecuzione.

3.1 BPF Compiler Collection (bcc)

Mentre eBPF offre grandi potenzialità, c'è un problema: è difficile da usare tramite il suo assembly o attraverso un interfaccia C. Nasce allora un progetto chiamato BPF Compiler Collection (bcc).
Bcc è un toolkit che mette a disposizione una libreria utilizzata per la manipolazione del kernel e include diversi strumenti ed esempi utili. Fa uso di extended BPF (Berkeley Packet Filters), formalmente conosciuto come eBPF. La maggior parte degli strumenti che usa bcc richiede Linux 4.1 e/o versioni successive.
Bcc semplifica la scrittura di programmi BPF, grazie alla strumentazione del kernel programmabile in C (include un wrapper C attorno a LLVM) e l'utilizzo di interfacce front-end in Python e lua. È adatto a molte attività, tra cui l'analisi delle prestazioni e il controllo del traffico di rete.
Grazie a questa interfaccia possiamo creare strumenti ad hoc in grado di monitorare le performance di sistemi Linux. Nelle sezioni che seguiranno parleremo di come vengono creati alcuni di questi tool utilizzando come interfaccia front-end Python.

3.2 Primo caso di studio: `cdsnoop`

Il primo tool che andremo ad analizzare è `cdsnoop` il cui utilizzo permette di monitorare tutti i processi bash che fanno uso del comando `cd` per spostarsi tra le directory.
Per prima cosa dobbiamo andare a definire il programma eBPF.

```
[ ... ]

bpf_text = """
  #include <uapi/linux/ptrace.h>

  int printret(struct pt_regs *ctx) {
    if (!ctx->ax )
      return 0;
    char str[30] = {};
    bpf_probe_read(&str, sizeof(str), (void *)ctx->ax);
    bpf_trace_printk("%s\\n", &str);
    return 0;
    }

  int printret1(struct pt_regs *ctx) {
    if (!ctx->ax )
      return 0;
    char str[30] = {};
    bpf_probe_read(&str, sizeof(str), (void *)ctx->ax);
    bpf_trace_printk("%s\\n", &str);
    return 0;
    }

  int printerr(struct pt_regs *ctx) {
    if (!ctx->ax )
      return 0;
    char str[30] = {};
    bpf_probe_read(&str, sizeof(str), (void *)ctx->ax);
    bpf_trace_printk("%s\\n", &str);
    return 0;
    };
"""
# load BPF program
b = BPF(text=bpf_text)
b.attach_uretprobe(name="/bin/bash", sym="get_name_for_error",
    fn_name="printerr") #cd error
b.attach_uretprobe(name="/bin/bash", sym="readline", fn_name="
    printret") # bash readline
b.attach_uretprobe(name="/bin/bash", sym="get_string_value",
    fn_name="printret1") # current_dir cd
```

Il programma eBPF viene dichiarato come una variabile stringa per poi essere caricato all'interno del costruttore della classe BPF(riga 34).
Una volta caricato il programma esso viene collegato a tre diverse fonti di eventi:

- riga35: crea una uprobes per il controllo degli errori causati da altri processi bash;
- riga36: crea una uprobes per monitorare i comandi di input eseguiti da altri processi bash;
- riga37: crea una uprobes per controllare il valore della directory corrente del processo bash che ha eseguito un comando.

Gli eventi sopra elencati vengono gestiti rispettivamente dalle funzioni `printerr`, `printret` e `printret1` che leggono dal registro `ctx` il valore di ritorno della funzione a cui sono allacciati. Vediamo ora come questi eventi vengono rilevati dal programma utente.

```
1  # header
2  print("%-12s %-10s %-15s %-10s %s" % ("TIME", "PID", "PREV_DIR"
     , "CURR_DIR" ,"COMM"))
3  directory = ""
4  while 1:
5    try:
6      (task, pid, cpu, flags, ts, msg) = b.trace_fields() #return
     readline probe
7    except ValueError:
8      continue
9    if msg != '':
10     command = ""
11     try:
12       command,directory = msg.split(" ")
13     except ValueError:
14       directory = "~"
15     if command == 'cd' :
16       curr_pid = pid
17       try:
18         (task, pid, cpu, flags, ts, msg) = b.trace_fields() #
   return current_workdirectory probe
19       except ValueError:
20         continue
21       if msg == 'bash':
22         print("%-12s %-5d %-1s %-1s %-1s %s " % (strftime("%H:
             %M:%S"), pid, msg,": cd :",directory,"File o
             directory non esistente"))
23       else:
24         prec_path = msg
```

```
25          try:
26            (task, pid, cpu, flags, ts, msg) = b.trace_fields() #
    return error probe
27          except ValueError:
28            continue
29          if msg == 'bash':
30            print("%-12s %-5d %-1s %-1s %-1s %s " % (strftime("%H
              :%M:%S"), pid, msg,": cd :",directory,"File o
             directory non esistente"))
31          else:
32            if directory == "~":
33              directory = "home"
34            p , c = print_event(directory,prec_path)
35            print("%-12s %-10d %-15s %-10s %s" % (strftime("%H:%M
              :%S"),curr_pid, p,c,command+" "+directory))
```

La riga 2 corrisponde all'intestazione del programma;
Dalla riga 4 alla 35 il processo utente è in attesa dei risultati[1] delle sonde menzionate in precedenza, in particolare viene prima letto l'input dei processi bash per controllare che il commando eseguito sia `cd` (righe 6-15) dopo di che viene recuperato il path della working-directory della bash che ha eseguito `cd` (righe 16-24) e in fine viene controllato se si è verificato un errore nell'esecuzione del comando(righe 25-30); se l'esecuzione del comando è andata a buon fine l'evento viene stampato a video[2](righe 32-35).

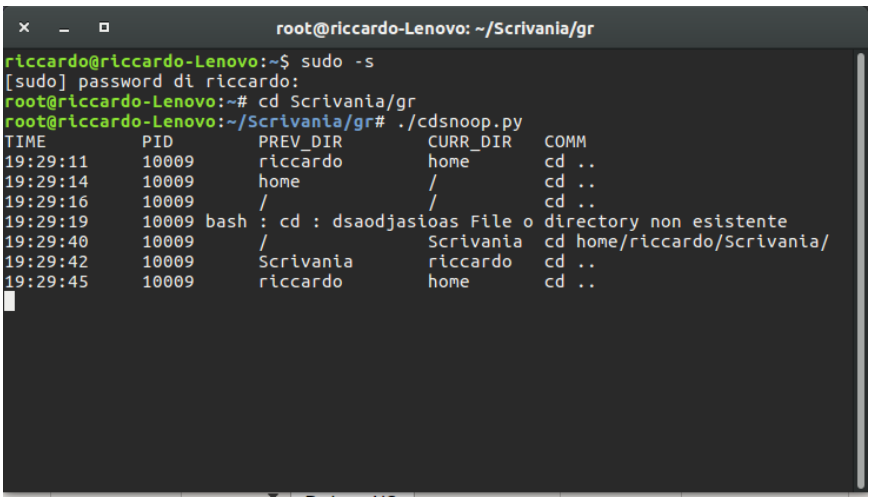

Figura 3.1: Esecuzione del programma cdsnoop.py

[1]Trattandosi di eventi asincroni non si può stabilire l'ordine di arrivo dei risultati, tranne per il comando `readline` in quanto è il primo ad essere eseguito.

[2]Nella **print-event** viene formattato l'output(vedere codice).

3.3 Secondo caso di studio: `pagefaultcount`

In quest'altra sezione ci occuperemo dello sviluppo di un tool che permette di monitorare il numero dei page-fault causati dai processi in esecuzione. Vediamo per prima cosa l'implementazione del programma eBPF.

```
1  #include <linux/ptrace.h>
2  #include <linux/sched.h>
3  #include <uapi/linux/bpf_perf_event.h>
4
5  struct data_t {
6  u32 pid;
7  u64 cpu;
8  char comm[TASK_COMM_LEN];
9  };
10
11 BPF_HASH(min_flt_table, struct data_t);
12 BPF_HASH(maj_flt_table, struct data_t);
13
14 static inline __attribute__((always_inline)) void get_key(
     struct data_t* data) {
15   data->cpu = bpf_get_smp_processor_id();
16   data->pid = bpf_get_current_pid_tgid();
17   bpf_get_current_comm(&(data->comm), sizeof(data->comm));
18   }
19
20 int page_min_flt(struct bpf_perf_event_data *ctx) {
21   struct data_t data = {};
22   get_key(&data);
23   u64 zero = 0, *val;
24   val = min_flt_table.lookup_or_init(&data, &zero);
25   (*val) += ctx->sample_period;
26   return 0;
27   }
28
29 int page_maj_flt(struct bpf_perf_event_data *ctx) {
30   struct data_t data = {};
31   get_key(&data);
32   u64 zero = 0, *val;
33   val = maj_flt_table.lookup_or_init(&data, &zero);
34   (*val) += ctx->sample_period;
35   return 0;
36   }
```

Le righe da 5 a 9 definiscono una struttura dati che contiene il pid, il nome simbolico del processo e l'id della cpu;
Nelle righe 11 e 12 vengono create 2 tabelle hash BPF per mantenere in memoria lo stato dei processi e i page-fault da loro causati;

Mentre le due funzioni non fanno altro che incrementare il numero di page-fault nelle tabelle hash.

Osservazione 3.1. Nel programma BPF del tool precedente, utilizzavamo l'interfaccia `bpf trace printk()` per spedire i risultati dal kernel allo spazio utente, mentre per quest'altro tool li preleveremo direttamente dalle due tabelle hash.

```
#!/usr/bin/env python
from __future__ import print_function
from bcc import BPF, PerfType, PerfSWConfig
import ctypes as ct
import signal
from time import sleep

def signal_ignore(signum,frame):
    print()
# load BPF program
b = BPF(src_file="source.c")
b.attach_perf_event(
    ev_type=PerfType.SOFTWARE, ev_config=PerfSWConfig.
    PAGE_FAULTS_MAJ,
    fn_name="page_maj_flt",sample_period=0,sample_freq=49)
b.attach_perf_event(
    ev_type=PerfType.SOFTWARE, ev_config=PerfSWConfig.
    PAGE_FAULTS_MIN,
    fn_name="page_min_flt",sample_period=0,sample_freq=49)

print("Running for {} seconds or hit Ctrl-C to end.".format(100))

try:
    sleep(float(100))
except KeyboardInterrupt:
    signal.signal(signal.SIGINT, signal_ignore)
    print(" ")

TASK_COMM_LEN = 16    # linux/sched.h
min_flt_count = {}
for (k, v) in b.get_table('min_flt_table').items():
    min_flt_count[(k.pid, k.cpu, k.comm)] = v.value

maj_flt_count = {}
for (k,v) in b.get_table('maj_flt_table').items():
    maj_flt_count[(k.pid , k.cpu , k.comm)] = v.value

print('{:<8s} {:<20s} {:<12s} {:<12s} {:<12s}'.format("PID","NAME
    ","CPU","MIN_FLT","MAX_FLT"))

```

```
38 for (k, v) in sorted(b.get_table('min_flt_table').items(), key=
       lambda (k,v): v.value,reverse=True):
39     try:
40         maj_miss = maj_flt_count[(k.pid, k.cpu, k.comm)]
41     except KeyError:
42         maj_miss = 0
43     print('{:<8d} {:<20s} {:<12d} {:<12d} {:<12d}'.format(
44         k.pid, k.comm.decode(), k.cpu, v.value, maj_miss))
45
46 for (k, v) in sorted(b.get_table('maj_flt_table').items(), key=
       lambda (k,v): v.value,reverse=True):
47     repeat = 1
48     try:
49         min_miss = min_flt_count[(k.pid, k.cpu, k.comm)]
50     except KeyError:
51         min_miss = 0
52         repeat = 0
53     if not repeat:
54         print('{:<8d} {:<20s} {:<12d} {:<12d} {:<12d}'.format(
55             k.pid, k.comm.decode(), k.cpu, min_miss, v.value))
```

Dove nelle righe 11-17 il programma viene allacciato a un perf-event che in questo caso è un evento SW; I valori delle tabelle hash vengono prelevati e inseriti in due dizionari (righe 28-30 e 32-34); L'output finale viene ordinato e stampato a video(righe 35-55).

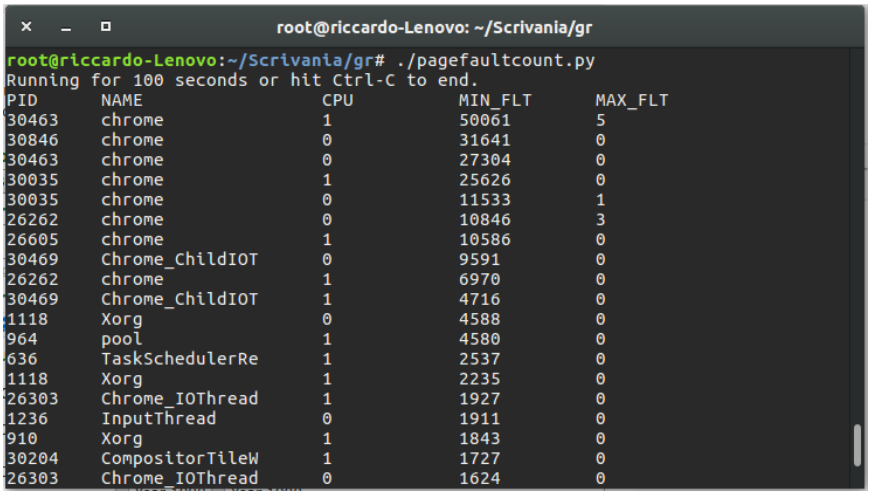

Figura 3.2: Esecuzione del programma pagefaultcount.py