eBPF: Programando o Kernel

Gustavo Neves Piedade Louzada
Instituto de Informática
Universidade Federal de Goiás
Goiânia, Goiás
gustavo.neves@discente.ufg.br

Igor Rodrigues Castilho
Instituto de Informática
Universidade Federal de Goiás
Goiânia, Goiás
igor.castilho@discente.ufg.br

Hafy Mourad Jacoub de Cuba Kouzak
Instituto de Informática
Universidade Federal de Goiás
Goiânia, Goiás
hafy@discente.ufg.br

João Victor de Paiva Albuquerque
Instituto de Informática
Universidade Federal de Goiás
Goiânia, Goiás
joao_albuquerque@discente.ufg.br

Maria Eduarda de Campos Ramos
Instituto de Informática
Universidade Federal de Goiás
Goiânia, Goiás
maria_campos@discente.ufg.br

***Abstract*—O eBPF representa uma inovação significativa na execução de programas em ambientes privilegiados, como o Kernel do sistema operacional, possibilitando aos usuários carregar e executar programas diretamente no Kernel. Desse modo, é possível anexá-los a eventos específicos do sistema operacional e, a partir deles, executar um código personalizado. Essa abordagem flexível pode ser aplicada em diferentes situações, como monitoramento de rede e *sockets*, observabilidade, segurança e *tracing*.**

palavras-chave — eBPF, Kernel, Linux, Sistemas Operacionais, observabilidade.

I. Introdução e revisão bibliográfica

Originária de uma evolução do BPF (*Berkeley Packet Filter)*, o eBPF (acrônimo para *Extended Berkeley Packet Filter*) é uma tecnologia que permite a execução de programas no contexto privilegiado do kernel do sistema operacional de maneira segura e eficiente, podendo estender as capacidades do kernel, sem a necessidade de modificá-lo ou carregar módulos.

Devido à habilidade privilegiada do kernel para supervisionar e controlar todo o sistema, o sistema operacional é o local ideal para a implementação de funcionalidades de observabilidade, segurança e rede. No entanto, devido ao papel central do kernel e seus altos requisitos de segurança e estabilidade, alterar e evoluir seu comportamento torna-se uma tarefa complexa e demorada, atrasando a taxa de inovação em comparação com as funcionalidades fora do escopo do sistema operacional.

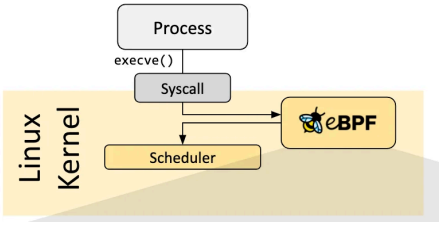
O eBPF altera essa lógica, de modo que permite a execução de programas isolados dentro do sistema operacional, permitindo a adição de capacidades adicionais em tempo de execução, sendo a segurança e a eficiência desta execução garantidas pelo próprio sistema operacional com o auxílio de um compilador *Just-In-Time* (JIT) e um mecanismo de verificação (eBPF Documentation).

Suas aplicações atuais abrangem áreas como monitoramento de rede, instrumentação do kernel, segurança, otimização de desempenho, análise de sistema, observabilidade, ferramentas de desenvolvimento e extensões de sistema de arquivos. Sua versatilidade tornou-o uma ferramenta poderosa para melhorar a observabilidade, segurança e desempenho em ambientes de produção, sendo amplamente adotado em uma variedade de cenários, desde análise de pacotes de rede até o desenvolvimento de ferramentas avançadas de observabilidade e segurança em sistemas Linux.

II. Fundamentos teóricos

Os programas eBPF constituem uma tecnologia baseada em eventos, operando quando o kernel ou um aplicativo atinge um ponto de ancoragem específico, conhecido como *hook* (Figura 1). Esses pontos de ancoragem abrangem diversas categorias, como chamadas do sistema, funções de entrada e saída, rastreamento do kernel, eventos de rede, entre outros.

Figura 1 - Hook em Kernel



Fonte: ebpf.io, 2020[1].

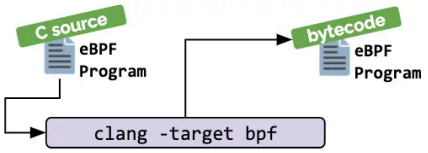
Na ausência de um gancho predefinido para uma necessidade específica, é possível criar um *kernel probe* (kprobe) ou *user probe* (uprobe). Isso permite a vinculação de

[1] Disponível em: <https://ebpf.io/what-is-ebpf/>. Acesso em: 14 jan. 2024.

programas eBPF virtualmente em qualquer parte do kernel ou em aplicações de usuário.

Em muitos cenários, a utilização direta de eBPF não é necessária. Projetos como Cilium, bcc e bpftrace oferecem abstrações sobre o eBPF, permitindo a definição de intenções sem a necessidade de escrever programas diretamente (Figura 2). Essas abstrações simplificam o desenvolvimento, proporcionando uma interface mais amigável.

Figura 2 - Exemplo de implementação de abstração



Fonte: ebpf.io, 2020[1].

Caso não haja uma abstração disponível, os programas eBPF precisam ser escritos diretamente. O kernel Linux espera que esses programas sejam carregados na forma de bytecode. Embora seja possível escrever bytecode manualmente, a prática comum envolve o uso de suítes de compiladores, como o LLVM, para converter código pseudo-C em bytecode eBPF.

O carregamento efetivo do programa eBPF no kernel Linux é realizado por meio da chamada de sistema bpf. Normalmente, bibliotecas específicas para eBPF facilitam esse processo, simplificando a interação do desenvolvedor com o kernel.

Antes de serem anexados ao gancho desejado, os programas eBPF passam por duas etapas críticas: Verificação e Compilação JIT. Essas etapas críticas garantem a segurança e a eficiência dos programas eBPF durante seu ciclo de vida no kernel Linux.

A etapa de verificação assegura a segurança do programa eBPF antes da execução. Diversas condições são validadas, incluindo a posse de privilégios necessários pelo processo que carrega o programa eBPF. Esta etapa garante que o programa não cause falhas no sistema e seja executado até o final, evitando loops infinitos que prejudiquem o processamento adicional.

A etapa JIT traduz o bytecode genérico do programa para um conjunto de instruções específico da máquina. Isso otimiza a velocidade de execução do programa eBPF, garantindo eficiência comparável ao código nativo do kernel ou a módulos de kernel carregados.

Uma característica fundamental dos programas eBPF é sua capacidade de compartilhar informações coletadas e manter o estado ao longo do tempo. Para este fim, os programas eBPF utilizam o conceito de *mapas eBPF*, permitindo o armazenamento e a recuperação eficiente de dados em diversas estruturas de dados. Esses mapas podem ser acessados não apenas por outros programas eBPF, mas também por aplicativos no espaço do usuário, possibilitando uma comunicação efetiva por meio de chamadas de sistema. Para exemplificar, os tipos de mapas suportados podem ser: tabelas de hash, matrizes, buffer circular, rastreamento de pilha, entre outros.

É crucial notar que os programas eBPF não têm a capacidade de chamar funções do kernel arbitrariamente. Tal permissão vincularia esses programas a versões específicas do kernel, complicando a compatibilidade e a manutenção dos programas. Em vez disso, os programas eBPF podem realizar chamadas de função para as chamadas de assistência, constituindo uma API estável e bem conhecida fornecida pelo kernel. Como exemplos de chamadas de assistência disponíveis, pode-se: gerar números aleatórios, obter data e hora atuais, obter contexto de processo, manipular pacotes de rede e lógica de encaminhamento.

Os programas eBPF são projetados com um foco na composabilidade, introduzindo os conceitos de chamadas de cauda (*tail calls*) e funções. As chamadas de função permitem a definição e execução de funções dentro de um programa eBPF, facilitando a modularização do código. Por outro lado, as chamadas de cauda possibilitam a execução de outro programa eBPF, substituindo o contexto de execução. Esse comportamento é análogo à chamada de sistema `execve()` para processos regulares, proporcionando uma flexibilidade notável na execução e na estruturação dos programas eBPF.

Essas características expandem significativamente o escopo e a utilidade dos programas eBPF, tornando-os não apenas poderosos em termos de rastreamento e filtragem, mas também altamente flexíveis e adaptáveis a uma variedade de cenários de uso. A seção subsequente discute uma das aplicações práticas dessas capacidades, destacando um caso de uso comum e implementação exemplar.

III. METODOLOGIA

A fim de expandir o comportamento do kernel de maneira segura, eficiente e rápida, e demonstrar a instrumentalização deste através do eBPF, utilizaremos as funcionalidades do *toolkit open source* BPF Compiler Collection (BCC).

Dentre as funcionalidades do BCC, encontramos um *wrapper* da linguagem C, que permite a declaração do módulo BPF, que será executado diretamente no kernel. Com isto, conseguimos, em um único arquivo, codificar os programas que serão executados no *kernel space* e no *user space*:

```
1  program = """
2  #include <linux/sched.h>
3  struct data_t {
4    u32 uid;
5    u32 pid;
6    u64 ts;
7    char comm[TASK_COMM_LEN];
8  };
9   BPF_PERF_OUTPUT(events);
10
11 int register_event(void *ctx){
12    struct data_t data = {};
13    data.uid = bpf_get_current_uid_gid() >> 32;
14    data.pid = bpf_get_current_pid_tgid();
15    data.ts = bpf_ktime_get_ns();
```

[1] Disponível em: <https://ebpf.io/what-is-ebpf/>. Acesso em: 14 jan. 2024.

```
16   bpf_get_current_comm(&data.comm,
sizeof(data.comm));
17   events.perf_submit(ctx, &data,
sizeof(data));
18   return 0;
19 }
20 """
21
22 b = BPF(text=program)
```

Algoritmo 1: Módulo BPF executado no kernel.

No algoritmo 1, codificamos o módulo BPF que será executado no kernel, abrangendo as linhas 1 a 22. Nele, definimos um tipo `data_t` que será utilizado para armazenar as informações de um processo, como seu nome, PID, o timestamp de sua criação e o ID do usuário que o criou. Na linha 9, utilizamos a função `BPF_PERF_OUTPUT`, que inicializa um buffer circular que servirá como meio de comunicação e compartilhamento de dados entre o programa eBPF e o programa rodando no *user space*.

A função `register_event`, definida das linhas 11 a 20, será responsável por pegar os dados dos processos, atribuí-los a uma estrutura do tipo definido anteriormente e registrá-los no buffer `events`. Na linha 22, é criado um objeto BPF, responsável por definir o programa BPF (no caso, o programa em C previamente codificado) e interagir com sua saída (bcc Reference Guide).

Após a definição do programa BPF, é necessário instrumentalizar uma função do kernel e anexar a função em C para ser executada toda vez que a chamada de sistema escolhida ocorrer, assim como mostra o algoritmo 2:

```
23 clone = b.get_syscall_fnname("clone")
24 b.attach_kprobe(event=clone,
fn_name="register_event")
```

Algoritmo 2: Instrumentalização de eventos do sistema.

Dessa maneira, toda vez que a chamada de sistema "clone", cujo nome da função é "sys_clone" e é responsável pela criação de um novo processo, é acionada, a função `register_event` será executada e armazenará os dados do novo processo que foi iniciado no buffer `events`.

No algoritmo 3, é definida a função `print_event`, que será responsável por imprimir, de maneira formatada, os eventos submetidos ao buffer. Na linha 36, abrimos para o programa do *user space* o acesso ao buffer e anexamos a função como *callback*, de modo que ela seja executada para cada evento submetido, mas apenas após a chamada da função `perf_buffer_poll`, que capta as entradas de todos os perf buffers abertos:

```
25 print("%-18s %-16s %-6s %-6s %s" %
("TIME(s)", "COMM","PID", "UID", "MESSAGE"))
26
27 start = 0
28 def print_event(cpu, data, size):
29     global start
30     event = b["events"].event(data)
31     if start == 0:
32             start = event.ts
33     time_s = (float(event.ts - start)) /
1000000000
34     printb(b"%-18.9f %-16s %-6d %-6d %s" %
(time_s, event.comm, event.pid, event.uid,
b"Hello, perf_output!"))
35
36 b["events"].open_perf_buffer(print_event)
37
38 while True:
39     try:
40         b.perf_buffer_poll()
41     except KeyboardInterrupt:
42         exit()
```

Algoritmo 3: Impressão de eventos do buffer.

Para mais detalhes sobre a implementação, o código completo está disponibilizado no Apêndice A.

IV. Resultados e Conclusões

Podemos observar na Figura 3, a saída do programa descrito, onde é possível rastrear os processos que foram iniciados, bem como seus respectivos identificadores e os usuários responsáveis por sua criação, associados a um instante no tempo.

Figura 3 - Resultados da execução do código

TIME(s)	COMM	PID	UID	MESSAGE
0.000000000	bash	4894	1000	Hello, perf_output!
10.688456526	bash	4894	1000	Hello, perf_output!
15.294329819	bash	4894	1000	Hello, perf_output!
15.295065380	bash	4894	1000	Hello, perf_output!
20.389007997	bash	4894	1000	Hello, perf_output!
20.389788545	bash	4894	1000	Hello, perf_output!
20.983838070	bash	4894	1000	Hello, perf_output!
25.508908262	ThreadPoolForeg	4401	1000	Hello, perf_output!
31.056092747	bash	4894	1000	Hello, perf_output!
31.056822878	bash	4894	1000	Hello, perf_output!
31.847809670	bash	4894	1000	Hello, perf_output!
36.937844954	bash	4894	1000	Hello, perf_output!
36.961403650	systemd	1	0	Hello, perf_output!
52.508264189	sudo	5059	0	Hello, perf_output!
53.979400571	ThreadPoolForeg	4349	1000	Hello, perf_output!
53.979402022	ThreadPoolForeg	4750	1000	Hello, perf_output!
54.072119844	ThreadPoolForeg	4485	1000	Hello, perf_output!

Fonte: Gustavo Neves Piedade Louzada, 2024.

Tal comportamento, não implementado no kernel por padrão, poderia ter muita utilidade, por exemplo, em um cenário de um servidor acessado por inúmeros usuários, já que permite, com eficiência, a rastreabilidade dos processos executados no servidor.

Portanto, conclui-se que, a partir da metodologia utilizada, foi possível verificar a utilidade do eBPF na prática. O código implementou uma funcionalidade nova no kernel, que não existe por padrão, de forma rápida, eficiente e segura. Tal atualização de comportamento, seguindo meios mais tradicionais, poderia se tornar uma tarefa extremamente custosa e complexa.

Referências

BCC Reference Guide. GitHub: IO Visor Project, 26 jul. 2016. Disponível em: https://github.com/iovisor/bcc/blob/master/docs/reference_guide.md. Acesso em: 21 jan. 2024.

EBPF Documentation. [*S. l.*], 2020. Disponível em: https://ebpf.io/what-is-ebpf/. Acesso em: 14 jan. 2024.

EBPF: Unlocking the Kernel [OFFICIAL DOCUMENTARY]. Produção: Speakeasy Productions. [S. l.: s. n.], 2023. Disponível em: https://www.youtube.com/watch?v=Wb_vD3XZYOA. Acesso em: 14 jan. 2024.

VIZARD, MIKE. Foundation Proposes Advancing eBPF Adoption Across Multiple OSes. Flórida, Estados Unidos, 12 ago. 2021. Disponível em: https://devops.com/foundation-proposes-advancing-ebpf-adoption-across-multiple-oses/. Acesso em: 14 jan. 2024.

Apêndice A: Código Completo

```
#!/usr/bin/python
from bcc import BPF
from bcc.utils import printb

# Utilização da BPF_PERF_OUTPUT
# Cria uma tabela BPF para enviar dados customizados de eventos para o user space
# via perf ring buffer
program = """
#include <linux/sched.h>
struct data_t {
  u32 uid;
  u32 pid;
  u64 ts;
  char comm[TASK_COMM_LEN];
};

 BPF_PERF_OUTPUT(events);

int register_event(void *ctx){
  struct data_t data = {};
  data.uid = bpf_get_current_uid_gid() >> 32;
  data.pid = bpf_get_current_pid_tgid();
  data.ts = bpf_ktime_get_ns();
 bpf_get_current_comm(&data.comm, sizeof(data.comm));

  events.perf_submit(ctx,&data,
  sizeof(data));
  return 0;
}
"""

b = BPF(text=program)
clone = b.get_syscall_fnname("clone")
b.attach_kprobe(event=clone, fn_name="register_event")
# header da tabela
print("%-18s %-16s %-6s %-6s %s" % ("TIME(s)", "COMM","PID", "UID", "MESSAGE"))

# Processamento dos eventos
start = 0
def print_event(cpu, data, size):
   global start
   event = b["events"].event(data)
   if start == 0:
           start = event.ts
   time_s = (float(event.ts - start)) / 1000000000
   printb(b"%-18.9f %-16s %-6d %-6d %s" % (time_s, event.comm, event.pid, event.uid,
       b"Hello, perf_output!"))

# loop com callback para print_event
b["events"].open_perf_buffer(print_event)

while True:
  try:
       b.perf_buffer_poll()
   except KeyboardInterrupt:
       exit()
```