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.

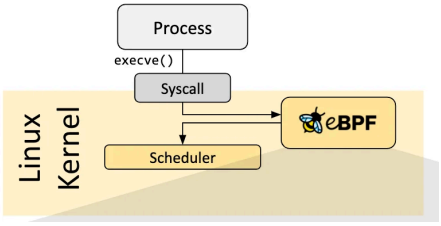
O eBPF altera essa lógica, de modo que permite a execução de programas isolados dentro do sistema operacional, permitindo a adição de capacidades adicionais em tempo de execução, sendo a segurança e a eficiência desta execução garantidas pelo próprio sistema operacional com o auxílio de um compilador *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].

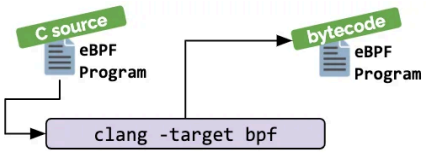
Na ausência de um gancho predefinido para uma necessidade específica, é possível criar um *kernel probe* (kprobe) ou *user probe* (uprobe). Isso permite a vinculação de

[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

IV. Resultados e Conclusões

Referências

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.

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