UNIVERSIDADE FEDERAL DE OURO PRETO
DEPARTAMENTO DE COMPUTAÇÃO

Título: Semântica formal e validação de filtros de pacotes linux expressos na linguagem eBPF

Aluno: Rafael Diniz de Oliveira
Orientador: Dr. Rodrigo Geraldo Ribeiro

Relatório Final, referente ao período 08/2020 a 07/2021, apresentado à Universidade Federal de Ouro Preto, como parte das exigências do programa de iniciação científica do edital EDITAL

Ouro Preto - Minas Gerais - Brasil
27 de julho de 2021

Resumo:

Semântica formal e validação de filtros de pacotes linux expressos na linguagem eBPF

O Extended Berkeley Packet Filter (eBPF) é um subsistema Linux que permite executar extensões não confiáveis definidas pelo usuário dentro do kernel. Para proteger o kernel contra código malicioso, o eBPF utiliza técnicas de análise estática simples. Porém, a medida que a linguagem eBPF cresce em popularidade, o seu ecossistema evolui para oferecer suporte a extensões mais complexas e diverisificadas. Porém, as limitações presentes em seu verificador atual, como a sua alta taxa de falsos positivos e falta de suporte a comandos de repetição, tornaram-se um grande empecilho para sua ampla adoção por desenvolvedores de aplicações de rede. Neste sentido, o presente projeto pretende aplicar técnicas de semântica formal e sistemas de tipos de maneira a aumentar a suporte do verificador eBPF, permitindo assim que programadores desenvolvam aplicações de maneira simples e garantindo a segurança do kernel estendido.

https://github.com/lives-group/racket-ebpf

Bolsista: Rafael Diniz de Oliveira

Orientador: Prof. Dr. Rodrigo Geraldo Ribeiro

1 Introdução

Os sistemas operacionais modernos implementam a maior parte de sua funcionalidade por meio de extensões carregadas dinamicamente que proveem suporte para dispositivos de E / S, sistemas de arquivos, redes, etc. Essas extensões são executadas no modo privilegiado da CPU e, portanto, devem ser verificadas para assegurar a ausência de código malicioso. Tradicionalmente a validação destas extensões é estabelecida através do uso de testes para eliminar bugs. Além disso, as ferramentas de verificação formal são usadas, em alguns casos, para obter maior segurança [2, 7]. Apesar de tais ferramentas serem eficazes para encontrar bugs, estas não possuem fortes garantias de correção.

As extensões do kernel são um tipo especial de aplicação que se originam de fontes não confiáveis e, portanto, não podem ser consideradas seguras. Essas extensões permitem que aplicativos personalizem o kernel com algoritmos específicos para processamento de pacotes, políticas de segurança, profiling e até modifiquem como os principais subsistemas do kernel [1].

No passado, os sistemas operacionais utilizavam técnicas como sandboxing para a execução de extensões não confiáveis dentro do kernel. Além disso, algumas abordagens se valiam de linguagens de domínio específico de domínio [3, 5] e interpretadores de bytecode [8]. Porém, essas abordagens se mostraram restritivas para alguns casos em que o desempenho é um fator crítico.

Como uma forma de amenizar essas falhas, o Linux adotou uma abordagem baseada no uso de um verificador de código, chamado de Exteded Berkeley Package Filters (eBPF). Programas eBPF consistem de bytecode simples que é compilado em instruções nativas da CPU quando carregado pelo kernel. Ao contrário dos bytecodes da Java Virtual Machine, o compilador e o run-time de eBPF não impõe nenhum restrição de tipo. A validade de programas eBPF é imposta por um verificador estático que impede que programas possam acessar estruturas de dados arbitrárias do kernel.

Atualmente, programadores eBPF enfrentam quatro grandes problemas. O primeiro deles é que o verificador de eBPF apresenta uma taxa elevada de falsos positivos, isto é, rejeita como inseguros programas que não oferecem risco algum ao kernel. Essa limitação obriga desenvolvedores a reescrever seu código de forma a este ser aceito pelo verificador. Muitas vezes, essas alterações envolvem a inserção de acessos e verificações redundantes. O segundo problema é que o algoritmo utilizado pelo verificado é sensível ao número de caminhos presente no programa analisado. Terceiro, atualmente o verificador impõe a séria restrição de que programas não devem possuir loops. Finalmente, o verificador atual não é especificado formalmente e isso dificulta a predição de quando um código será ou não por ele recusado.

Diante do apresentado, o presente projeto pretende especificar formalmente um verificador de programas eBPF de forma a solucionar as limitações apresentadas. Para isso, utilizaremos técnicas conhecidas do projeto de linguagens de programação, a saber: semântica formal e sistemas de tipos [9, 6, 4].

2 Revisão da Literatura

2.1 A linguagem eBPF

Aqui você deve apresentar uma introdução à linguagem eBPF. Explicando o que é a linguagem e o porquê de sua criação.

Adicionalmente, se possível, coloque exemplos de programas eBPF simples. Como programas eBPF são escritos usando bibliotecas C ou Python, você pode procurar esses exemplos e explicá-los nessa seção.

2.2 A linguagem Racket

Aqui você deve apresentar uma breve introdução à linguagem racket. Minha idéia é você apresentar sobre a definição de funções, tipos de dados e macros. Na sequência você pode explicar a implementação daquela máquina virtual simples que fiz.

3 Desenvolvimento

Nesta seção, você deve descrever sua implementação do interpretador de ebPF.

4 Conclusão e Trabalhos Futuros

Essa seção vai depender do que você escrever na seção de desenvolvimento. Deixaremos para escrever por último.

Referências

[1] Nadav Amit, Michael Wei, and Cheng-Chun Tu. Hypercallbacks. *SIGOPS Oper. Syst. Rev.*, 51(1):54–59, September 2017.

[2] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. *SIGOPS Oper. Syst. Rev.*, 40(4):73–85, April 2006.

[3] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the spin operating system. *SIGOPS Oper. Syst. Rev.*, 29(5):267–283, December 1995.

[4] Stephen Chang, Alex Knauth, and Ben Greenman. Type systems as macros. *SIGPLAN Not.*, 52(1):694–705, January 2017.

[5] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in singularity os. *SIGOPS Oper. Syst. Rev.*, 40(4):177–190, April 2006.

[6] Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. Run your research: On the effectiveness of lightweight mechanization. *SIGPLAN Not.*, 47(1):285–296, January 2012.

[7] Akash Lal and Shaz Qadeer. Powering the static driver verifier using corral. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 202–212, New York, NY, USA, 2014. Association for Computing Machinery.

[8] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, USENIX'93, page 2, USA, 1993. USENIX Association.

[9] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.