Observação de Ataques contra a Memória do Kernel Android: Desafios e Soluções

Cláudio Torres Júnior[1], **Jorge Correia**[1], **João Pincovscy**[2], **Marco Zanata**[1], **André Grégio**[1]

[1]Departamento de Informática – Universidade Federal do Paraná (UFPR)
SecRET (`secret.inf.ufpr.br`) – Curitiba – PR – Brasil

[2]Departamento de Engenharia Elétrica – Universidade de Brasília (UnB)
Brasília – DF – Brasil

{ctjunior, jpcorreia, mazalves, gregio}@inf.ufpr.br, pincovscy@gmail.com

Abstract. *In 2023, over 300 vulnerabilities were reported in the Linux kernel. This emphasizes the need for exploit analysis aiming at understanding and protecting affected systems (including Android) from privilege escalation, data leakage, and other attacks. In this article, we address the challenges and solutions for the security of Android kernel memory, as well as present an evaluation of available tracing and memory sanitization tools for this operating system. This research includes performance tests of these tools and the implementation of a proof of concept at the kernel level to improve memory exploit analysis, providing complete attack observation and allowing continued execution after detection, which is not achieved by the state of the art.*

Resumo. *Em 2023, foram reportadas mais de 300 vulnerabilidades no kernel Linux, corroborando a necessidade da análise de exploits para compreendê-las e proteger os sistemas afetados (inclusive Android) de escaladas de privilégio, vazamento de dados e outros ataques. Este artigo aborda os desafios e soluções para segurança da memória do kernel Android, e avalia ferramentas de tracing e sanitização de memória disponíveis para esse sistema operacional. A pesquisa inclui testes de desempenho dessas ferramentas e a implementação de uma prova de conceito em nível de kernel para melhorar a análise de exploits de memória, provendo observação completa do ataque e permitindo continuar sua execução após a detecção, o que não é alcançado pelo estado da arte.*

1. Introdução

A quantidade de *bugs* em um software é diretamente proporcional ao tamanho de seu código [McConnell 2004]. O kernel Linux, com suas quase 30 milhões de linhas de código, é alvo frequente para atacantes que podem explorar diversos de seus componentes, com potencial de escalada de privilégio, vazamento de dados sensíveis ou ataques persistentes [Hund et al. 2009].Em 2023, mais de 300 vulnerabilidades (*bugs* passíveis de exploração) no kernel Linux foram reportadas [CVEDetails 2024]. Tais preocupações se acentuam no contexto de dispositivos móveis, dado que o Android (sistema operacional mais utilizado neste tipo de dispositivo [Curry 2024]) adiciona código ao kernel Linux.

Dentre as vulnerabilidades mais comuns no kernel Android, *bugs* em memória como *use-after-free* (UAF) e *double-free* (DF) se destacam pela sua recorrên-

cia [Liang et al. 2024]. Para mitigá-las adequadamente, é necessário investigar o funcionamento dos *exploits* que tiram proveito dessas vulnerabilidades. Assim, pode-se desenvolver novas arquiteturas e ferramentas para proteção e, consequentemente, aumentar a segurança dos sistemas em geral. Contudo, mesmo entre os especialistas em segurança, os conceitos acerca dos funcionamentos dos *exploits* que exploram *bugs* relacionados à memória do kernel ainda se confundem [Zeng et al. 2022], tendo em vista as diferentes técnicas de exploração [Song et al. 2019] e a complexidade desse ambiente.

Isto ocorre em parte devido à falta de soluções integradas e especializadas em observar o fluxo de execução dos *exploits* em sistemas operacionais (inclusive móveis) de maneira completa e em cenários reais. No caso da monitoração dos *exploits* de memória, tenta-se suprir a falta dessas soluções com o uso de *tracers* e sanitizadores de memória [Lin et al. 2022, Wu et al. 2018]. Porém, esse tipo de ferramenta pode interferir no ambiente de execução, influenciar o comportamento dos *exploits* ou inserir sobrecarga a ponto de impedir a análise do ataque e, consequentemente, sua compreensão (por exemplo, evitando que uma condição de corrida ocorra ou modificando variáveis de ambiente necessárias para o sucesso do *exploit*). Além disso, analisar *exploits* de vulnerabilidades em kernel requer monitoração e análise de chamadas de funções e das corrupções que podem ocorrer neste ambiente privilegiado, impossível de ser alcançado com ferramentas em espaço de usuário. Isto é acentuado no contexto do Android, pois ferramentas importantes não estão disponíveis nesse sistema operacional.

As contribuições deste artigo incluem: (i) levantamento e a comparação de métodos que podem ser usados na análise de *exploits* de memória do kernel Android, levando em conta desempenho, funcionalidades, capacidade de observação de ataques e limitações; (ii) introdução de potenciais adaptações para soluções existentes por meio da proposição de prova de conceito que integra funcionalidades do KASAN com *tracers* personalizados usando *kprobes*, possibilitando a execução dos *exploits* após detecção da corrupção de memória. Tal abordagem inclui um mecanismo aprimorado de quarentena de memória e instrumentação que rastreia os acessos à memória, proporcionando observação completa dos *exploits* em execução e superando o estado da arte.

2. Conceitos Fundamentais

Vulnerabilidades de Memória. Ataques à memória representam uma categoria crítica de vulnerabilidades e geralmente envolvem técnicas de *Use-After-Free* e *Double-Free* [CWE 2023]. Um ataque de ***Use-After-Free (UAF)*** consiste no acesso, a partir de ponteiros que não apontam para regiões válidas (*dangling pointers*), à regiões de memórias que já foram liberadas, podendo resultar em vazamento de informações sensíveis, corrupção de dados e até execução arbitrária de código [Wu et al. 2018]. Cabe ressaltar que 88% das vulnerabilidades UAF encontradas são consideradas críticas ou de alta gravidade [Lee et al. 2015]. Já um ataque de ***Double Free (DF)*** ocorre quando uma região de memória liberada é novamente liberada (i.e., o mesmo endereço aparece duas vezes na lista de *free*), levando a um estado inconsistente do gerenciador de memória e, consequentemente a uma série de erros críticos. Isso pode permitir que atacantes manipulem a estrutura de dados do gerenciador de memória (ou até mesmo da memória) possibilitando a inserção de *payloads* maliciosos, execução de código arbitrário ou negação de serviço [Liang et al. 2024]. A análise e a mitigação de ataques à memória enfrentam várias dificuldades, tais como **espaço de busca enorme**—no qual a identificação de cami-

nhos de fluxo de dados que levam a vazamento de informação ou outros comportamentos exploráveis pode ser extremamente desafiadora devido ao espaço de busca no código do Kernel [Liang et al. 2024]—e **contexto de execução variado**—onde a forma como um *bug* se manifesta pode variar significativamente dependendo do contexto de execução.

Gerenciamento da *Heap* do kernel. O gerenciamento de memória dinâmica (*heap*) do kernel Linux e Android é realizado através do subsistema SLUB [Khan 2022], cujo objetivo é evitar fragmentação e aumentar a eficiência das alocações e desalocações frequentes. Para tanto, o SLUB mantém diversos grupos de memória denominados *Slab Caches*, que contêm blocos de memória prontos para serem alocados. Cada *Slab Cache* possui uma série de blocos de mesmo tamanho, retornados quando uma alocação de memória dentro deste tamanho é solicitada. Por exemplo, quando é feita uma alocação de 4KB, um bloco de memória da *Slab Cache* que contém blocos de 4KB é retornado, mas quando um espaço de 5KB é requisitado, o menor bloco de memória da *Slab Cache* que o consegue comportar é o de 8KB, o qual é então retornado. De forma simplificada, pode-se enxergar os blocos de uma *Slab Cache* como uma lista encadeada com política LIFO. Assim, quando um espaço de memória é liberado, o bloco referente a ele é colocado na cabeça da lista da sua *Slab Cache*. De maneira análoga, quando um espaço de memória é requisitado, o bloco da cabeça da lista é retornado. Esse tipo de gerenciamento de listas das *Slab Caches* é o que permite a exploração de vulnerabilidades *Use-After-Free* e *Double-Free*. Por exemplo, supondo a execução de duas operações subsequentes, [OP1] a desalocação de um espaço de memória de tamanho 5KB e [OP2] a alocação de um espaço de memória de 6KB.Quando [OP1] for executada, o bloco que mantinha os dados do espaço de memória liberado será colocado na cabeça da lista da *Slab Cache* de 8KB. Quando [OP2] for executada, o bloco retornado será o mesmo que foi liberado na [OP1]. Isso permite que *exploits* manipulem as listas das *Slabs Caches* de forma quase determinística [Zeng et al. 2022], aumentando consideravelmente a probabilidade de um bloco específico estar na cabeça da lista em um dado momento.

Módulos de Kernel do Linux. Um módulo de kernel é um arquivo objeto que compõe um subsistema do kernel Linux. Dois tipos de módulos são importantes no presente trabalho: Módulos Carregáveis de Kernel (LKM) e Módulos de Segurança do Linux (LSM). Os LKM são módulos que podem ser inseridos no kernel por usuários em tempo de execução, sem necessidade de recompilação, *reboot* ou modificação do código-fonte principal, e com o objetivo de fornecer funcionalidades adicionais, como *drivers* de dispositivos. No entanto, por operarem com privilégios elevados, os LKM não foram originalmente projetados para acomodar medidas de segurança suplementares e podem incorrer em risco à segurança do kernel se não forem implementados e gerenciados adequadamente [Kernel Development Community 2024]. Para superar essa limitação, são usados os LSM, que fornecem uma interface padronizada para melhorias de segurança no kernel. Um LSM é integrado de forma transparente ao kernel, com impacto mínimo no desempenho do sistema e possibilidade de controlar o acesso à diferentes recursos, como *syscalls*. Diferente dos LKM, os LSM tornam-se parte do kernel na compilação e não podem ser modificados após inicialização, proporcionando um nível mais alto de segurança.

Análise de Aplicações Android. As técnicas de análise de aplicações podem ser divididas em dois tipos: estática e dinâmica. A **análise estática** envolve o estudo do código-fonte da aplicação alvo, juntamente com as APIs utilizadas. Esse tipo de análise pode ge-

rar indícios do comportamento de um *exploit*, mas que nem sempre são suficientes para o entendimento completo do impacto de sua execução [Afonso et al. 2016, Lin et al. 2023]. Isso ocorre por que os *exploits* em geral dependem da interação entre diversas *threads* para gerar situações de condições de corrida, ou até mesmo da execução repetida de uma operação para gerar a exaustão de um subsistema. Além disso, ataques ao kernel podem passar despercebidos na análise estática, uma vez que não se observa a execução do código. Por outro lado, a **análise dinâmica** requer a execução das aplicações para monitorar seu fluxo, auxiliando na observação de comportamentos que se manifestam durante seu uso em tempo real. No entanto, essa abordagem pode exigir muitos recursos computacionais [Kang et al. 2021], além de ser limitada nos casos em que as aplicações implementem técnicas *anti-debug* e anti-emulador para detectar se estão sendo executadas em *sandbox* ou *debugger* [Marco et al. 2020, Jing et al. 2014]. A análise dinâmica é mais eficaz na identificação de ataques ao kernel, pois pode monitorar diretamente as interações da aplicação com o sistema operacional durante sua execução, detectando comportamentos anômalos ou potencialmente maliciosos.

3. Trabalhos Relacionados

Ainda que não sejam encontrados trabalhos acadêmicos que avaliam o desempenho de soluções para análise de *exploits* para memória do kernel Linux/Android, há literatura sobre ferramentas para análise de *malware*, *tracers* e sanitizadores de memória. O entendimento dessas ferramentas possibilita compreender suas principais funcionalidades e os desafios que uma ferramenta de análise de *exploits* deve superar para poder ser usada na observação de ataques em ambientes reais, especialmente os móveis.

O trabalho mais próximo deste é o de [Sutter et al. 2024], onde os autores conduzem uma revisão sistemática da literatura sobre técnicas de análise dinâmica no Android. São discutidas ferramentas e técnicas obtidas do estudo de 43 publicações, as quais incluem monitoramento de rede, rastreamento de chamadas de sistema, análise dinâmica de fluxo de dados e instrumentação binária.Os principais desafios levantados para essas abordagens estão ligados à sobrecarga de execução das aplicações, perda de desempenho do sistema como um todo devido ao alto consumo de recursos adicionais para análise dinâmica e, principalmente, interferência na experiência dos usuários.

[Or-Meir et al. 2019] sumarizam técnicas de análise de *malware* e suas ferramentas. Dentre elas, uma estratégia se destaca pelo potencial de ser utilizada no contexto de observação de *exploits*: a análise de chamadas de funções. Tendo em vista que o entendimento do funcionamento de um *exploit* está fortemente ligado ao seu fluxo de execução (e.g., sequência de chamadas de funções e seus parâmetros), ferramentas que implementam esse tipo de análise podem ser muito úteis. Entretanto, as soluções listadas no artigo limitam-se à instrumentação do processo-alvo (via injeção de código), dependem de ambientes virtuais e são restritas ao espaço de usuário. Tudo isso inviabiliza a aplicação direta delas em dispositivos Android.

[Gebai and Dagenais 2018] analisam o desempenho e as funcionalidades de diversas ferramentas de *tracing* que executam tanto em espaço de kernel como de usuário. Ainda que úteis, *tracers* não são suficientes para prover as informações necessárias para análise de *exploits* em memória. Para suprir tal necessidade, especialistas os utilizam em conjunto com sanitizadores de memória a fim de identificar as operações chaves realizadas

pelo *exploit*. [Nong et al. 2021] analisam diversos detectores de corrupção de memória, avaliando métricas como eficácia, desempenho e acurácia.

4. Soluções para Análise de *Exploits* na Memória do Kernel Android

Em geral, soluções para análise de *exploits* de memória combinam *tracers* com sanitizadores de memória [Wu et al. 2018, Lin et al. 2022]. Como o kernel Android possui peculiaridades em seu ambiente de execução, nesta seção descreve-se como tais soluções são usadas nesse contexto, enfatizando suas limitações, desempenho e usabilidade.

4.1. Ferramentas para observação de aplicações

Strace. Ferramenta em espaço de usuário para a interceptação de *syscalls* e sinais recebidos por um processo específico[1]. Permite extrair informações como argumentos e valores de retorno das chamadas de sistema. Seu funcionamento se dá pela utilização do *ptrace*[2], uma chamada de sistema que permite que um processo tenha a capacidade de monitorar outro. Quando um processo é configurado para ser rastreado, a execução deste é bloqueada a cada *syscall* ou tratamento de sinal. O *strace* é notificado para tratar tal evento e, então o processo é liberado.

Ftrace. Ferramenta em espaço de kernel que permite a análise detalhada de vários eventos, incluindo *syscalls* e funções internas do Kernel, como alocação e desalocação de memória (*__kmalloc* e *kfree*, por exemplo)[3]. Por todo o código do kernel Linux (e consequentemente o kernel Android), existem diversos *hooks* que permitem o *ftrace* manter o rastreamento de forma eficiente, sem a necessidade de trocas de contexto. Essa capacidade o torna extremamente útil para analisar *exploits* que atacam a memória, pois permite rastrear os caminhos de alocação e desalocação tomados durante a execução do processo. Além disso, por estar diretamente integrado ao kernel, o *ftrace* apresenta um *overhead* significativamente menor do que ferramentas em espaço de usuário.

Kprobe. Por mais que o sistema de *Kprobe* não seja um *tracer*, as suas funcionalidades se encaixam perfeitamente no contexto de monitoramento de funções. Ele permite a inserção de *handlers* personalizados em qualquer ponto de interesse no kernel, como *syscalls* e funções específicas. Quando uma função que recebeu um *handler* for executada, uma instrução *breakpoint* é executada com o objetivo de modificar o fluxo de execução para a função apontada pelo *handler*. Após isso, o fluxo de execução normal é retomado. O sistema de *Kbropes* se mostra mais flexível que o apresentado pelo *ftrace*, tendo em vista que este último utiliza *hooks* estáticos definidos no código-fonte do kernel. Contudo, a execução da instrução *breakpoint* implica na execução de uma exceção (interrupção de software), sendo muito mais custosa que chamadas de função comuns [Gebai and Dagenais 2018].

LSM. Embora não seja um *tracer*, um LSM é muito útil pela sua capacidade de extração de informações e desempenho, podendo ser utilizado para tal finalidade. No código-fonte das *syscalls* tem-se algumas chamadas de funções de segurança que são denominadas *hooks* de segurança, os quais realizam checagens para garantir que a *syscall* possa ser continuada. Um LSM pode criar funções auxiliares que são instaladas nesses *hooks* que, quando ativados, faz com que todas as suas funções auxiliares sejam chamadas em

[1] `https://strace.io/`
[2] `https://man7.org/linux/man-pages/man2/ptrace.2.html`
[3] `https://www.kernel.org/doc/html/v5.0/trace/ftrace.html`

sequência. Se alguma delas proibir a continuação da *syscall*, sua execução é imediatamente cancelada. Uma *syscall* pode ter entre um ou vários *hooks* declarados, fazendo com que realizar o *tracing* somente pelos nomes dos *hooks* não seja muito vantajoso. Para isso, deve-se criar funções auxiliares para os *hooks* e obter, a cada ativação, os valores dos registradores, extraindo assim o número e os argumentos das *syscalls* sendo executadas.

eBPF. Dado que o *Extended Berkeley Packet Filter* é utilizado para executar *bytecode* em um ambiente controlado e seguro dentro do Kernel [Jay Schulist 2024], este pode ser utilizado como *tracer*. Amplamente aplicado em análises de desempenho e eventos de rede, quando configurado para rastrear *syscalls*, o *eBPF* pode responder rapidamente aos eventos relacionados, executando operações definidas por usuários em espaço de Kernel.

4.2. Sanitizadores de memória em kernel

KASAN. O *Kernel Address SANitizer* é uma ferramenta capaz de identificar *bugs* em diversas áreas do espaço de endereçamento virtual do kernel Linux, inclusive da *heap*. No contexto deste trabalho, vale destacar dois mecanismos utilizados pelo KASAN: quarentena de memória e instrumentação de código. A quarentena consiste em reter espaços de memória que foram recentemente liberados, impedindo que estes sejam retornados para a fila da sua respectiva *slab cache* e, consequentemente, realocados em um intervalo de tempo próximo. Essa abordagem visa mitigar situações de UAF e DF, dado que um acesso ou uma liberação de um espaço de memória em quarentena implica em *bugs*. Juntamente com a quarentena, o KASAN deve ser capaz de identificar os acessos à memória para verificar se os endereços associados são passíveis de *bugs*. Para isso, no momento de compilação do kernel, todas as instruções *load* e *store* geradas são instrumentadas, a fim de inserir as funções de verificação do KASAN. Para códigos escritos diretamente em *assembly*, como acessos à memória atômicos ou do espaço de usuário, se faz necessária a inserção manual das funções de verificação, uma vez que nesses casos não existe a geração de código por parte do compilador. Em dispositivos ARM, o KASAN consegue utilizar o auxílio de hardware *Memory Tagging Extension* para melhorar a eficiência no rastreamento dos acesso à memória [Mitsunami 2021]. Contudo, comumente são utilizados ambientes virtuais na investigação de *exploits*. No Android, os principais ambientes virtuais da plataforma (Emulator[4] e Cuttlefish[5]) expõem uma arquitetura x86, fazendo com que o KASAN utilize a sua implementação padrão. Ainda que o KASAN funcione bem quando o objetivo seja detectar as corrupções de memória, existe uma característica no seu funcionamento que dificulta sua utilização como uma ferramenta de análise de *exploits*: no momento que uma falha de memória ocorre, o KASAN finaliza o processo responsável por gerar a corrupção. Porém, a execução do *exploit* pode não se restringir à execução de uma única corrupção de memória. De modo geral, diversas corrupções precisam ser realizadas e, além disso, uma técnica de exploração deve seguir as corrupções para alcançar objetivos como escaladas de privilégio.

BoKASAN. Baseado no KASAN, o *Binary-only Kernel Address Sanitizer* buscou superar o desafio de necessitar da recompilação do código do kernel, através da instrumentação dinâmica seletiva dos seus binários [Cho et al. 2023]. Essa abordagem reduziu significantemente a sobrecarga de desempenho enquanto que permitiu a descoberta de *bugs* em

[4]`https:////source.android.com/docs/setup/create/avd`
[5]`https://source.android.com/docs/setup/create/cuttlefish`

número comparável ao KASAN puro. Entretanto, tal sobrecarga aumenta drasticamente em cenários de acesso intensivo à memória (como é comum em situações de condições de corrida). Além disso, sua cobertura de detecção é limitada, uma vez que a ferramenta foi feita para ser utilizada em conjunto com *fuzzers* e não como detector de *bugs* em tempo real. Devido ao escopo delimitado de atuação diferente do presente trabalho, o BoKASAN não foi usado nas comparações feitas na Seção 5.

ViK. Prova de conceito com versões em software (≈20% de sobrecarga de tempo de execução) e com assistência de hardware (≈2% de sobrecarga) para analisar as interações dos objetos com a memória capaz de identificar UAF e DF. Para tanto, ViK atribui identificadores aleatórios a objetos alocados e os armazena em bits não utilizados dos ponteiros, verificando a validade deles antes da desreferenciação [Cho et al. 2022]. Como o objetivo da solução é mitigar ataques em memória, ViK interrompe a execução do *exploit* ao detectá-los, dificultando a observação completa de um ataque. A falta de código disponível impossibilitou comparações usando o ViK como sanitizador de memória.

5. Testes e Resultados

Com o objetivo de verificar a sobrecarga de desempenho dos *tracers* que podem ser utilizados em ambientes Android, bem como a interferência causada pelo KASAN (único sanitizador de memória publicamente disponível para o kernel), foram conduzidos uma série de testes de *micro-benchmark*. Seguindo as abordagens de análise de *exploits* da literatura, avaliou-se a sobrecarga de cada *tracer* sobre diferentes chamadas de sistemas, além de seu uso em conjunto com e sem o KASAN habilitado. Informações relevantes sobre os experimentos estão descritas abaixo:

- Os testes foram executados em uma máquina com Ubuntu 23.10 x86-64, equipada com um AMD Ryzen 7 5800H e 16GB de RAM. O emulador utilizado foi o *Android emulator* para Android 10 com Kernel 4.14 x86-64.
- Tendo em vista que diversos mecanismos de *tracing* são construídos usando outras ferramentas como base, neste trabalho foram selecionadas essas ferramentas fundamentais para compor a análise de desempenho. O uso do kernel 4.14 no ambiente de testes impossibilitou a avaliação do eBPF por possuir suporte limitado para essa tecnologia. No entanto, [Gebai and Dagenais 2018] indicam que realizar rastreamento de funções utilizando eBPF pode apresentar um *overhead* superior ao registrado com o *ftrace*.
- Embora não sejam capazes de realizar rastreamento de funções e *syscalls* por si só, fez-se uso de *Kprobes* e LSM neste trabalho para desenvolver novos *tracers* e permitir a comparação de suas capacidades com as demais soluções disponíveis.
- As *syscalls* usadas no *micro-benchmark* deste artigo foram `Open/Close`, `Write`, `Read`, `Stat`, `Fstat`, `Fork` e `Mmap/Munmap`.

Os testes de *micro-benchmark* foram obtidos através da chamada consecutiva de cada *syscall* (ou grupo, como *open/close* e *mmap/mummap*); o teste de cada *syscall* foi repetido 100 mil vezes em intervalos de 10 execuções, e a média aritmética dos tempos de execução foi calculada. Os resultados desses testes são apresentados na Figura 1.

5.1. Discussão

A partir dos resultados obtidos dos testes de *micro-benchmark* com o escopo supracitado, é possível extrair informações relevantes acerca do desempenho das ferramentas avaliadas

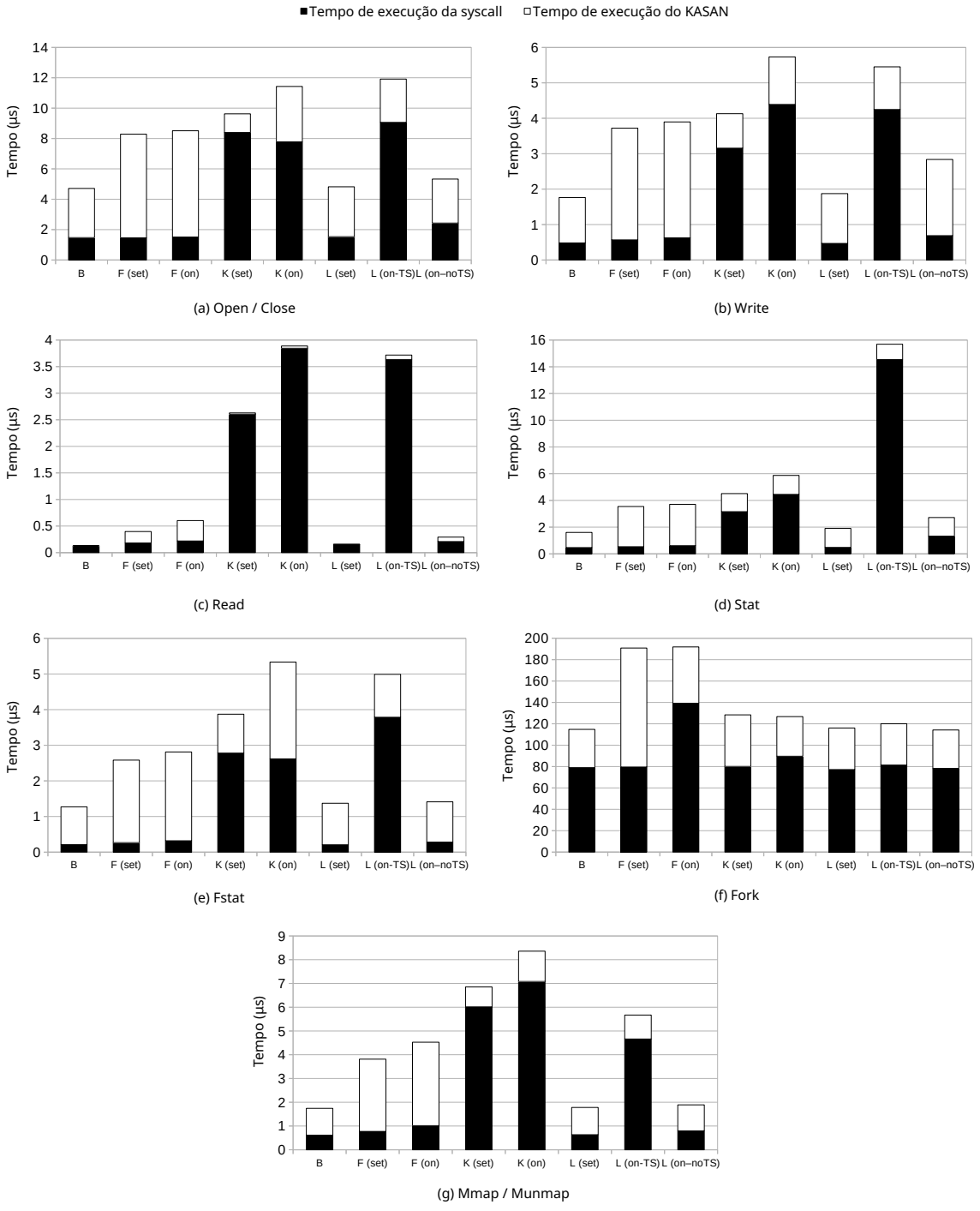

Figura 1. Tempo de execução de chamadas de sistemas e KASAN, utilizando combinações de *tracers*, onde B é o *baseline*, F é *ftrace*, K é *kprobe* e L é *LSM*; (*set*) significa que a ferramenta foi apenas instalada, (on) que o *tracing* está habilitado. Mais detalhes na Seção 5.1.

em diferentes combinações. Além disso, foi possível identificar vantagens potenciais e limitações das combinações de *tracers* com e sem o sanitizador de memória KASAN.

A seguir, descreve-se cada ferramenta mostrada no gráfico e discute-se os resultados alcançados em relação ao tempo médio em microssegundos do teste das *syscalls*.

Strace gerou a maior perda de desempenho dentre todos as ferramentas testadas, chegando a ser, por exemplo, 682x mais lento que a medição-base para a *syscall* `read`, e 962x quando utilizado em conjunto com o KASAN. Portanto, objetivando a melhor visualização das informações, os tempos de execução desta ferramenta não foram exibidos nos gráficos da Figura 1. Tendo em vista as decisões de projeto do `strace`, deve existir uma troca de contexto entre o processo que está sendo analisado e o referido *tracer*, o que é uma operação custosa. Vale ressaltar que o *strace* inclui nas informações providas as *syscalls* que foram chamadas, juntamente com seus argumentos e valores de retorno, permitindo uma visão completa de cada operação. Verificou-se também um desvio padrão anormal para o *strace*, que pode ser explicado pela variabilidade na latência introduzida pelas trocas de contexto, uma vez que este comportamento depende do funcionamento do escalonador de processos do sistema operacional. A Tabela 1 mostra o cálculo da média e desvio-padrão em microssegundos do `strace` com e sem o KASAN, para fins de comparação com os dados apresentados pelas outras ferramentas da Figura 1.

Tabela 1. Tempo em microssegundos e desvio-padrão da execução do *strace* com KASAN desabilitado e habilitado para as *syscalls* selecionadas.

Syscall	Sem KASAN	Com KASAN
`open/close`	155.911 ± 8.195	228.114 ± 23.174
`write`	87.761 ± 1.897	119.051 ± 24.875
`read`	85.254 ± 0.562	118.340 ± 13.888
`stat`	90.714 ± 2.182	140.018 ± 2.144
`fstat`	88.851 ± 0.815	118.262 ± 28.930
`fork`	322.087 ± 1.818	531.523 ± 90.882
`mmap/munmap`	171.113 ± 1.583	158.549 ± 54.768

Kernel Base (representado como "B" na Figura 1) representa os resultados de tempo calculados para cada *syscall* sem a utilização de *tracers*, servindo de *baseline* para as demais análises. Os demais *tracers* foram testados utilizando dois modos: *tracer* em execução mas configurado para não rastrear quaisquer *syscalls* (denotado por "set" na Figura 1); *tracer* em execução e configurado para rastrear a *syscall* em questão (denotado por "on" na Figura 1).

Ftrace (representado como "F" na Figura 1) realiza a captura dos traços dentro do espaço do Kernel, evitando as frequentes trocas de contexto causadas pelo *strace* e, por isso, resultando em um *overhead* muito menor quando comparado com este último. Esse método de *tracing* coleta somente as *syscalls* sendo chamadas, sem informações extras de argumentos e valores de retorno. Ao se analisar os gráficos, pode-se perceber que, quando o KASAN não está sendo utilizado, a degradação de desempenho causada pelo *Ftrace* é mínima. Dessa forma, o tempo das chamadas de sistemas sendo rastreadas é bem próximo ao tempo base. Contudo, este comportamento muda ao se habilitar o KASAN, incorrendo em uma degradação de desempenho em torno de 50%. Observa-se também

que, com o KASAN habilitado, existe uma degradação de desempenho mesmo que o *Ftrace* não esteja configurado para rastrear qualquer *syscall*. Este comportamento é um indicativo de que uma ferramenta que combine *Ftrace* e KASAN pode não ser a melhor alternativa, ainda que existam trabalhos que utilizem esta abordagem [Lin et al. 2022, Wu et al. 2018].

Kprobes (representado como "K" na Figura 1) também realizam a captura dos traços em espaço de kernel. Contudo, seu uso implica na inserção de instruções de *breakpoint* a cada operação rastreada. Dessa forma, apesar de seu desempenho ser muito melhor que o *strace*, é inferior ao *Ftrace* para a maioria das *syscalls* analisadas. A degradação de desempenho imposta pelo sistema baseado em *kprobes* existe em todos os contextos de testes, inclusive com o KASAN desabilitado e sem rastrear qualquer *syscall*, o que pode ser explicado pela mudança frequente do fluxo de execução causada pela execução das instruções de *breakpoint* para os *handlers* registrados. Tendo em vista que a verificação do processo a ser monitorado é feita pelo próprio *handler*, todos os processos sofrem com a mudança no fluxo de execução. Em contrapartida à perda de desempenho, o sistema de *kprobes* permite a coleta dos parâmetros passados para as *syscalls*, assim como o valor de retorno. Portanto, o uso de *kprobes* indica um maior potencial de análise quando comparado com o *Ftrace*.

LSM (representado como "L" na Figura 1) utiliza *hooks* estáticos no código, assim como o *Ftrace*. Contudo, é possível alterar o código que cada *hook* executa. Desta forma, inserir novas implementações de funções que serão chamadas pelos *hooks* implica em um *overhead* relativamente baixo. Durante a execução dos experimentos, ficou perceptível que a operação de coleta do *timestamp*, implementada através da chamada de função *ktime_get_ns* presente no Kernel, é bastante custosa. Para verificar tal constatação, executou-se o *tracer* LSM criado neste artigo de forma a capturar os *timestamps* (representado por "L(on-TS)") das chamadas, bem como sem capturá-los (representado por "L(on-noTS"). Ao se analisar o desempenho desta abordagem, é perceptível que a degradação de desempenho causada pelo LSM sem a captura do *timestamp* é ínfima quando comparada com o *baseline*, independente do contexto de execução ou da ativação do KASAN. A principal limitação imposta pelo LSM é que só existem *hooks* na entrada das *syscalls*. Isso impossibilita a captura do valor de retorno das *syscalls*, que pode ser importante para se analisar o traço de execução de um ataque.

De forma geral, os resultados obtidos com o KASAN ativado (barras brancas das subfiguras) mostram que, mesmo sem a utilização de *tracers*, tem-se um *overhead* significativo em relação ao kernel "normal". Essa alteração é esperada devido às verificações adicionais realizadas pelo KASAN para detectar e reportar problemas de memória. Esse *overhead* é aplicado à todos os processos sendo executados, enquanto que a instrumentação feita pelo KASAN é inserida em tempo de compilação.

Além de observar cada *tracer* de forma independente, uma característica importante deve ser levada em consideração ao se analisar os gráficos: o tempo de execução das *syscalls*. A *syscall* `fork`, por exemplo, apresenta um tempo de execução muito maior que as demais. Dessa forma, o tempo de execução dos *tracers*, que conceitualmente é independente e constante para todas as *syscalls*, acaba compondo uma porcentagem ínfima do tempo de execução total da *syscall*, o que explica o motivo dos tempos para essa chamada de sistema estarem próximos. Nas demais chamadas de sistema, o tempo de execução dos

tracers ficam mais evidentes, destacando a sobrecarga de desempenho dos *tracers* com o KASAN. Ademais, podemos salientar a baixa sobrecarga de desempenho do KASAN na chamada de sistema `read`, que pode ser explicada pela baixa quantidade de acessos à memória por parte dessa *syscall* em detrimento às outras [Cho et al. 2023].

Também é preciso analisar a forma com que cada *tracer* lida com os traços de execução gerados. Isso porque *tracers* que geram o relatório ao mesmo tempo em que rastreiam os eventos solicitados podem apresentar um *overhead* maior durante as suas execuções. O *strace*, por exemplo, sempre realiza a escrita dos eventos ao mesmo tempo do rastreio. Por outro lado, tanto os método utilizando *Kprobes* quanto LSM geram o relatório de traços somente ao final da execução, escrevendo-o em um arquivo em disco. O *ftrace*, por sua vez, armazena os eventos durante a sua execução e os escrevem em um sistema de arquivos virtual. Logo, a degradação de desempenho não é tão impactante, tendo em vista que os dados permanecem em memória. Tal análise se faz importante no contexto da observação de *exploits*, pois é preferível que os traços estejam disponíveis à longo prazo, possibilitando o acesso futuro por parte de analistas de segurança.

Após investigar os resultados apresentados, percebe-se que a sobrecarga de desempenho causada pelas ferramentas de *tracing* e pelo KASAN, na maioria das situações, é consideravelmente alta. Por outro lado, ao estudar ataques relacionados à acessos indevidos à memória, levar em consideração o tempo de execução é um fator importante. Muitos ataques à memória utilizam-se de condições de corrida para que a execução maliciosa seja bem-sucedida, o que pode ser impedido por conta da utilização de ferramentas de análise. Como exemplo, a utilização de um *tracer* durante a execução da CVE-2022-46395 [6] impede a sua execução devido a uma condição de corrida muito restrita.

6. Prova de Conceito

A partir dos testes de desempenho da combinação de ferramentas e interpretação dos resultados obtidos, foi possível identificar algumas limitações e possíveis abordagens acerca dos métodos de análises de *exploits* de memória do kernel existentes na literatura. O fator limitante principal está relacionado à falta de capacidade de observação por parte do KASAN: no momento que ocorre a corrupção de memória, o processo é interrompido imediatamente. Embora justificável em um contexto de proteção de usuários, essa característica é prejudicial ao se analisar um *exploit* para fins de sua compreensão. Nesses casos, é preferível que a execução continue mesmo após a exploração do *bug* para se ter a análise completa do seu comportamento. É preferível também que as verificações possam ser restritas a um processo específico (e aos seus processos filhos), não criando degradações de desempenho nos demais processos do sistema.

Finalmente, tendo em vista as restrições apresentadas, a forma com que o KASAN mantém o rastreamento dos espaços de memória pode ser simplificada visando ganho de desempenho. As alterações mencionadas têm grande potencial quando aplicadas ao KASAN em conjunto com *tracers*, pois, se estes puderem fornecer a pilha de chamadas de funções e os valores dos seus argumentos, a análise do *exploit* se torna muito mais eficaz. Além disso, é interessante que funções internas do kernel, como *kmalloc* e *kfree* possam ser rastreadas.

[6] `https://nvd.nist.gov/vuln/detail/CVE-2022-46395`

Para atender as necessidades descritas acima, optou-se por criar uma prova de conceito que integre funcionalidades do KASAN com um mecanismo de *tracing* baseado em *kprobes*. Tal abordagem foi escolhida por sua capacidade de permitir a implementação de *handlers* personalizados não somente em *syscalls*, mas também em funções arbitrárias do kernel, auxiliando na integração com o KASAN modificado. Além disso, os *hooks* das *kprobes* proporcionam a capacidade de capturar parâmetros de funções e *syscalls*, assim como os valores de retorno. Juntamente com a captura, é possível alterar valores de parâmetros utilizando *handlers kprobe*, que permitem a modificação do fluxo de execução para a implementação de mecanismos de quarentena. Esta flexibilidade supera as outras técnicas de *tracing* avaliadas neste artigo, permitindo controle refinado do processo sendo rastreado e detecção de corrupção de memória.

Quarentena. A implementação da funcionalidade de quarentena de memória foi feita a partir do registro de um *handler kprobe* na função *kfree*, responsável por liberar memória da *heap* do kernel. No momento em que a função é chamada, o *handler* registrado salva o endereço do ponteiro recebido pela função *kfree* e o insere em uma tabela *hash*. Ao final dessa operação, o valor do parâmetro recebido pela função *kfree* foi modificado para zero. Dessa forma, após a execução completa do *handler*, a função *kfree* retorna imediatamente sem desalocar memória alguma (devido ao parâmetro com valor zero). Esse fluxo de execução impede que o bloco de memória volte para a lista da *slab cache*, ao passo que o mantém em controle da prova de conceito criada para verificações futuras.

Detecção de UAF/DF. As verificações de acesso à memória foram realizadas modificando-se as funções de instrumentação inseridas em tempo de compilação pelo KASAN. Dessa forma, antes da execução das instruções de `load` e `store`, funções personalizadas são executadas objetivando a verificação do endereço sendo acessado. O endereço da página física que contém o endereço virtual em questão é utilizado como chave de busca em uma tabela *hash*. Caso o endereço acessado pertença a algum espaço de memória da tabela, o que indica que ele está liberado, o sistema detecta a ocorrência de UAF. O *hook* na função de liberação de memória também é utilizado para, antes de adicionar a região de memória à tabela, verificar se a entrada já está inserida. Caso esteja, o sistema detecta a ocorrência de *double-free*.

Observabilidade. Para permitir que o *exploit* continue sua execução após a corrupção de memória, foi implementado um mecanismo de espelhamento entre a memória em quarentena e a memória alocada pelo *exploit*. Com isso, uma modificação na memória em quarentena é refletida na memória alocada e vice-versa, permitindo que o *exploit* execute sem a influência do mecanismo de quarentena proposto.

Detecção de Escalada de Privilégio (EoP). Além de todas as funcionalidades citadas anteriormente, a prova de conceito criada também utiliza os argumentos das *sycalls sys_read* e *sys_write* para verificar se existe uma escrita em um endereço acima do *addr_limit*, verificando de que região da memória os *buffers* dessas *syscalls* estão lendo/escrevendo. O *addr_limit* delimita o fim do espaço de memória destinado a um processo de espaço de usuário. Com isso, após a exploração de uma vulnerabilidade, *exploits* podem utilizar a capacidade de escrita em endereços arbitrários para sobrescrever credenciais localizadas acima do *addr_limit*, alcançando assim uma escalada de privilégio. Ao manter o rastreamento de escritas acima desse espaço de memória, é possível verificar o exato momento que o *exploit* sobrescreve as estruturas privilegiadas e, consequentemente a EoP ocorre.

6.1. Estudo de Caso

A fim de validar empiricamente a prova de conceito (*PoC*) desenvolvida, esta foi executada de forma a monitorar um *exploit* disponível para a CVE-2019-2215[7]. Tal vulnerabilidade foi escolhida por permitir o disparo de um UAF no *Binder* (mecanismo de intercomunicação de processos do Android) sem a necessidade de interação com o usuário. Na Figura 2 apresenta-se, de forma visual em interface interativa feita para a prova de conceito, os traços do comportamento de execução rastreado para o referido *exploit*.

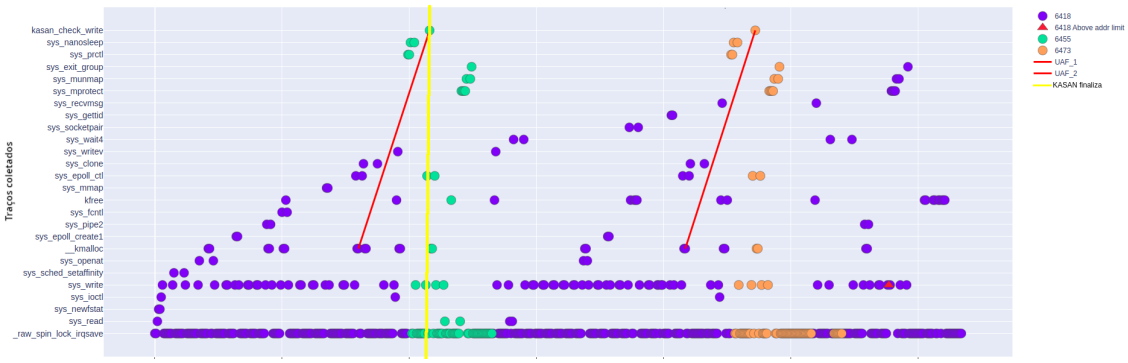

Figura 2. Visualização do traço coletado pela *PoC* em *exploit* da CVE-2019-2215. Figura criada pelo autor utilizando Plotly.

Ao se analisar os resultados, é perceptível que o *exploit* induz várias execuções da função *_raw_spin_lock_irq_save*, que é justamente a função que dispara o código vulnerável. Nota-se também que o *exploit* realiza duas operações de UAF (linhas vermelhas), o que não seria possível verificar utilizando o KASAN originalmente, pois a execução se encerraria na primeira corrupção de memória, representada pela linha vertical amarela. Nesse ponto, vale ressaltar que mesmo o KASAN identificando o primeiro UAF, ele não foi abusado pelo *exploit*, dado que a memória encontra-se em quarentena.

Observa-se também que o *exploit* funciona a partir da interação de três processos diferentes, ilustrados pelos círculos roxos, verdes e alaranjados, de modo que os acessos que levaram o UAF dependem da interação entre estes processos. Por fim, foi identificada uma execução da *syscall sys_write* que realiza um acesso acima do *addr_limit*, indicando o momento que a escalada de privilégio ocorre, marcado com um triângulo vermelho (porção inferior direita da figura). Ressalta-se que o *hook* das funções *__kmalloc* e *kfree* foi realizado tão somente para obtenção da região acessada indevidamente, deixando as outras funções sem o impacto dos *handlers* utilizados pelo *kprobe*.

A Figura 3 mostra tanto o desempenho quanto a extensão do traço coletado comparando a PoC proposta com outras abordagens disponíveis. Nela, apresenta-se a diferença entre o tempo de execução normal, a execução com a solução proposta neste artigo, e a execução do KASAN com o *ftrace*. O *exploit* utilizado possui dois eventos de *sleep* de dois segundos cada, interferindo na medida "real" do impacto do *tracing* em conjunto com a detecção de corrupção de memória do kernel Android. Por esse motivo, optou-se por mostrar a execução do *exploit* com dedução do tempo das interrupções feitas pelo *sleep* para fins de melhor visualização da sobrecarga imposta.

[7] `https://nvd.nist.gov/vuln/detail/CVE-2019-2215`

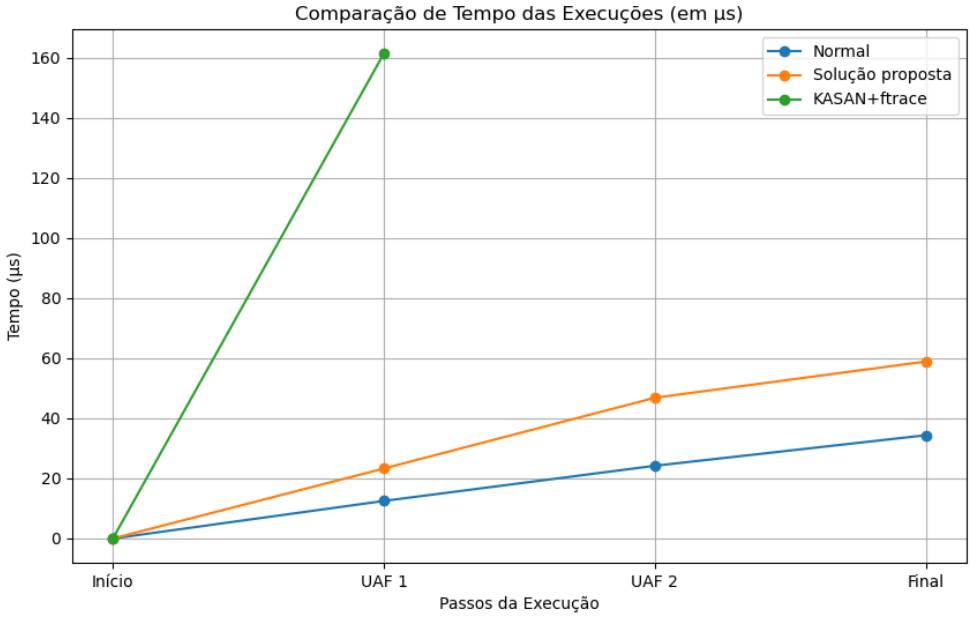

Figura 3. Comparação de Tempo das Execuções (em µs) e Abrangência de Coleta do Traço do Exploit para a CVE-2019-2215.

Percebe-se que, enquanto o modo de execução com KASAN e *ftrace* tem sobrecarga maior que as outras execuções, ele finaliza no primeiro UAF, não permitindo que haja um estudo completo do real objetivo do *exploit*. Esse comportamento sempre estará presente nos ambientes que utilizam o KASAN, devido sua política de quarentena. Por outro lado, a PoC proposta permite a observação do comprometimento da vulnerabilidade até o final da execução do *exploit*, com sobrecarga bem menor.

7. Conclusão

Este artigo abordou os desafios e soluções para a análise de *exploits* de memória no kernel Android. A avaliação de diversas ferramentas de *tracing* e sanitização de memória revelou seu desempenho, capacidades e limitações. Os testes realizados mostraram que ferramentas como *strace* e *ftrace* fornecem informações detalhadas sobre *syscalls*, mas introduzem *overhead* significativo em tempo de execução. Já o KASAN, apesar de detectar corrupções de memória de maneira efetiva, interrompe a execução dos *exploits*, limitando a sua análise. Soluções no estado-da-arte fazem uso do KASAN, apresentando as mesmas limitações de funcionalidade e desempenho, ou dependem de suporte de hardware e dispositivos específicos para atuarem, além de não estarem disponíveis publicamente para reprodução dos experimentos. Para superar essas limitações, foi proposta uma prova de conceito que integra o KASAN com *tracers* personalizados usando *kprobes*. A solução proposta permite a continuação da execução após a detecção de corrupção de memória, provendo uma análise mais abrangente dos *exploits* via observação completa do ataque. Além disso, inclui um mecanismo de quarentena de memória e instrumentação aprimorado para rastrear acessos à memória, o que melhora a precisão e profundidade da análise de *exploits*.

Agradecimentos

Este trabalho teve apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

Referências

Afonso, V. M., de Geus, P. L., Bianchi, A., Fratantonio, Y., Krügel, C., Vigna, G., Doupé, A., and Polino, M. (2016). Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy. In *Network and Distributed System Security Symposium*.

Cho, H., Park, J., Oest, A., Bao, T., Wang, R., Shoshitaishvili, Y., Doupé, A., and Ahn, G.-J. (2022). Vik: practical mitigation of temporal memory safety violations through object id inspection. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*, pages 271–284.

Cho, M., An, D., Jin, H., and Kwon, T. (2023). BoKASAN: Binary-only kernel address sanitizer for effective kernel fuzzing. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4985–5002, Anaheim, CA. USENIX Association.

Curry, D. (2024). Android statistics (2024). `https://www.businessofapps.com/data/android-statistics/`.

CVEDetails (2024). Application sandbox. `https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2023/Linux-Linux-Kernel.html`.

CWE (2023). 2023 cwe top 10 kev weaknesses. `https://cwe.mitre.org/top25/archive/2023/2023_kev_list.html`.

Gebai, M. and Dagenais, M. R. (2018). Survey and analysis of kernel and userspace tracers on linux: Design, implementation, and overhead. *ACM Computing Surveys (CSUR)*, 51(2):1–33.

Hund, R., Holz, T., and Freiling, F. C. (2009). Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *USENIX Security Symposium*.

Jay Schulist, Daniel Borkmann, A. S. (2024). Linux socket filtering aka berkeley packet filter (bpf). `https://www.kernel.org/doc/html/latest/networking/filter.html`.

Jing, Y., Zhao, Z., Ahn, G.-J., and Hu, H. (2014). Morpheus: Automatically generating heuristics to detect android emulators. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC '14, page 216–225, New York, NY, USA. Association for Computing Machinery.

Kang, H., Liu, G., Wu, Z., Tian, Y., and Zhang, L. (2021). A modified flowdroid based on chi-square test of permissions. *Entropy*, 23(2).

Kernel Development Community (2024). Kernel modules. `https://linux-kernel-labs.github.io/refs/heads/master/labs/kernel_modules.html`.

Khan, I. (2022). Linux slub allocator internals and debugging, part 1 of 4. https://blogs.oracle.com/linux/post/linux-slub-allocator-internals-and-debugging-1.

Lee, B., Song, C., Jang, Y., Wang, T., Kim, T., Lu, L., and Lee, W. (2015). Preventing use-after-free with dangling pointers nullification. In *NDSS'15*.

Liang, Z., Zou, X., Song, C., and Qian, Z. (2024). K-leak: Towards automating the generation of multi-step infoleak exploits against the linux kernel. In *31th Annual Network and Distributed System Security Symposium, NDSS*.

Lin, Y., Wong, J., and Gao, D. (2023). Fa3: Fine-grained android application analysis. In *Proceedings of the 24th International Workshop on Mobile Computing Systems and Applications*, HotMobile '23, page 74–80, New York, NY, USA. Association for Computing Machinery.

Lin, Z., Chen, Y., Wu, Y., Mu, D., Yu, C., Xing, X., and Li, K. (2022). Grebe: Unveiling exploitation potential for linux kernel bugs. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2078–2095. IEEE.

Marco, A., Cestaro, R., Conti, M., and Losiouk, E. (2020). Mascara: a novel attack leveraging android virtualization.

McConnell, S. (2004). *Code complete*. Pearson Education.

Mitsunami, K. (2021). Delivering enhanced security through memory tagging extension. https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/enhanced-security-through-mte.

Nong, Y., Cai, H., Ye, P., Li, L., and Chen, F. (2021). Evaluating and comparing memory error vulnerability detectors. *Information and Software Technology*, 137:106614.

Or-Meir, O., Nissim, N., Elovici, Y., and Rokach, L. (2019). Dynamic malware analysis in the modern era—a state of the art survey. *ACM Computing Surveys (CSUR)*, 52(5).

Song, D., Lettner, J., Rajasekaran, P., Na, Y., Volckaert, S., Larsen, P., and Franz, M. (2019). Sok: Sanitizing for security. In *IEEE Symposium on Security and Privacy*.

Sutter, T., Kehrer, T., Rennhard, M., Tellenbach, B., and Klein, J. (2024). Dynamic security analysis on android: A systematic literature review. *IEEE Access*.

Wu, W., Chen, Y., Xu, J., Xing, X., Gong, X., and Zou, W. (2018). FUZE: Towards facilitating exploit generation for kernel Use-After-Free vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 781–797. USENIX Association.

Zeng, K., Chen, Y., Cho, H., Xing, X., Doupé, A., Shoshitaishvili, Y., and Bao, T. (2022). Playing for {K (H) eaps}: Understanding and improving linux kernel exploit reliability. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 71–88.