LUCAS ALIONÇO PEREZ

ANÁLISE DE VIABILIDADE DE USO DE FERRAMENTAS NATIVAS DO SISTEMA OPERACIONAL ANDROID PARA DETECÇÃO DE ATAQUES

(*versão pré-defesa, compilada em 7 de julho de 2023*)

Trabalho apresentado como requisito parcial à conclusão do Curso de Bacharelado em Ciência da Computação, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: André Ricardo Abed Grégio.

CURITIBA PR

2023

RESUMO

A proposta deste trabalho consiste em uma análise de viabilidade do uso de ferramentas nativas do sistema operacional Android para detecção de ataques. O objetivo do estudo é verificar quais ferramentas existentes no Android podem ser utilizadas para proteção de dispositivos móveis por meio da realização de experimentos para identificar ameaças, correlacionando os dados entregues por cada ferramenta ao código fonte dos ataques. O trabalho é dividido em quatro capítulos. No primeiro capítulo, é apresentado um panorama geral sobre os sistemas Linux e Android, indicando os problemas de segurança no ambiente mobile e o propósito da pesquisa. O segundo capítulo aborda os conceitos fundamentais envolvendo a arquitetura do Android, categorias de ataques realizados por *malware*, definições sobre *tracing* e trabalhos relacionados. O terceiro capítulo descreve a proposta do trabalho, incluindo uma breve avaliação de cada ferramenta disponível para coleta de traços no ambiente Android. Já o quarto capítulo relata os resultados obtidos pelos experimentos realizados com as ferramentas, terminando com apontamentos sobre a oportunidade e relevância de novos estudos. Os resultados obtidos foram correlacionados com as características dos ataques identificados, permitindo uma avaliação das ferramentas utilizadas e suas limitações técnicas.

Palavras-chave: *Tracing*. Segurança de Sistemas. Ameaças móveis

ABSTRACT

The purpose of this work consists of a feasibility analysis of the use of native Android operating system tools for attack detection. The objective of the study is to verify which existing tools on Android can be used to protect mobile devices by carrying out experiments to identify threats, correlating the data delivered by each tool to the source code of the attacks. The work is divided into four chapters. In the first chapter, an overview of Linux and Android systems is presented, indicating the security problems in the mobile environment and the purpose of the research. The second chapter addresses the fundamental concepts involving the Android architecture, categories of attacks carried out by *malware*, definitions about *tracing* and related work. The third chapter describes the work proposal, including a brief evaluation of each tool available for collecting traces in the Android environment. The fourth chapter reports the results obtained by the experiments carried out with the tools, ending with notes on the opportunity and relevance of further studies. The results obtained were correlated with the characteristics of the identified attacks, allowing an evaluation of the tools used and their technical limitations.

Keywords: *Tracing*. Systems Security. Mobile Threats

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ACRÔNIMOS

ADB	*Android Debug Bridge*
AES	*Advanced Encryption Standard*
ANATEL	Agência Nacional de Telecomunicações
AOSP	*Android Open Source Project*
API	*Application Programming Interface*
C&C	*Command-and-Control*
CPU	*Central Processing Unit*
CVE	*Common Vulnerabilities and Exposures*
DSA	*Digital Signature Algorithm*
eBPF	*Extended Berkeley Packet Filter*
GCC	*GNU Compiler Collection*
GNU	*GNU's Not Unix*
GPL	*General Public License*
GPU	*Graphics Processing Unit*
HTML	*Hypertext Markup Language*
HTTP	*Hypertext Transfer Protocol*
iOS	*iPhone Operating System*
I/O	*Input/Output*
IPv6	*Internet Protocol Version 6*
JSON	*JavaScript Object Notation*
KASAN	*The Kernel Address Sanitizer*
LTS	*Long-term Support*
LTTng	*Linux Trace Toolkit: Next Generation*
MAC	*Mandatory Access Control*
MINIX	*Mini-Unix*
PID	*Process Identifier*
POC	*Proof of Concept*
RSA	Rivest-Shamir-Adleman
SDK	*Software Development Kits*
SD	*Secure Digital*
SELinux	*Security-Enhanced Linux*
SHA	*secure Hashing Algorithm*
SMS	*Short Message Service*
SSH	*Secure Socket Shell*

SSL	*Secure Sockets Layer*
SUID	*Set owner User ID*
UID	*User Identifier*
UI	*User Interface*
UNIX	*UNiplexed Information Computing System*
USB	*Universal Serial Bus*

SUMÁRIO

1 INTRODUÇÃO

Este trabalho está disposto em quatro capítulos. O primeiro, de introdução, apresenta um panorama geral sobre os sistemas Linux e Android, indicando os problemas de segurança no ambiente mobile e o propósito da pesquisa. O segundo capítulo trata dos conceitos fundamentais envolvendo a arquitetura do Android, categorias de ataques realizados por *malware*, definições sobre *tracing* e trabalhos relacionados. O terceiro capítulo descreve a proposta do trabalho, bem como uma breve avaliação de cada ferramenta disponível para realização de *tracing* no ambiente Android. O quarto capítulo relata os resultados obtidos pelos experimentos realizados com as ferramentas, concluindo apontamentos sobre a oportunidade e relevância de novos estudos.

1.1 INTRODUÇÃO AO LINUX

No contexto dos computadores com sistemas proprietários existentes na década de 60, Ken Thompson e Dennis Ritchie desenvolveram o sistema operacional UNIX, compatível com mais de um equipamento. Baseado nos princípios do software livre de Richard Stallman, seu projeto GNU e no MINIX (um sistema operacional derivado do UNIX, criado para fins didáticos por Andrew Tanenbaum), Linus Torvalds desenvolveu o primeiro *kernel* do Linux (Maziero, 2019b), sob a licença GPL (General Public License) (Farias, 2006).

Kernel é o programa que executa as funções básicas de um sistema operacional, controlando e oferecendo interfaces para o hardware, administrando o uso de memória e arquivos do sistema (Mitchell et al., 2001). Ele implementa as principais abstrações utilizadas pelos aplicativos e programas utilitários, sendo dividido em módulos para facilitar o trabalho dos programadores, já que está em constante transformação e aprimoramento (Maziero, 2019b). O *kernel* de um sistema não é útil ao usuário e deve ser utilizado dentro de um sistema maior (Wankhede, 2023).

Dado que todo usuário do sistema também pode ser seu programador, inúmeras distribuições do sistema Linux foram disponibilizadas, incluindo as mais populares como Ubuntu, Mint, Debian, FreeBSD e Red Hat, entre outras. Essas versões do Linux se diferenciam entre si pelo acréscimo de programas escolhidos por cada equipe ao *kernel* do sistema (Soares, 2015).

As principais características do Linux são seu suporte a multitarefas, existência de multiusuários, confiabilidade, estabilidade e suporte às principais linguagens de programação. Além disso, o fato de ser gratuito, atualizado com frequência e dispensar a necessidade de licença para uso o torna um sistema operacional de grande importância para a computação em geral (Farias, 2006).

O Linux está presente em 2,83% dos desktops ativos mundialmente (StatCounter, 2023a). A baixa popularidade desse sistema operacional se dá principalmente pela ausência de softwares utilizados pela indústria (como Photoshop, Microsoft Excel e CorelDRAW), falta de

compatibilidade para jogos em geral e navegabilidade pouco intuitiva para iniciantes (Arslan, 2022).

Embora não lidere a lista de sistemas operacionais mais populares para desktop, o Linux ainda é uma escolha popular para configuração de servidores e sistemas embarcados, atuando como bancos de dados, equipamentos de rede ou unidades de controle de sistemas críticos (Vurdelja et al., 2020). De fato, observa-se que 81,2% dos servidores web utilizam o Linux como sistema operacional (W3Techs, 2023), assim como ocorre com 90% do processamento em nuvem (Nolte, 2022).

É possível atribuir essa vantagem aos fatores de código aberto, baixo custo, alta performance, compatibilidade, segurança (Insights, 2023), escalabilidade, eficiência no uso de recursos e grande quantidade de profissionais capacitados no seu uso (Nolte, 2022).

Entretanto, além de servir como base para maioria dos servidores web e núcleos de processamento em nuvem, o Linux também fornece os alicerces do sistema operacional móvel mais popular: o Android.

1.2 INTRODUÇÃO AO ANDROID

O Android é um sistema operacional voltado principalmente para dispositivos móveis, lançado pela Google em 2007 (Maziero, 2019b). Atualmente, a 14ª versão do sistema se encontra em fase beta, com lançamento previsto para o final de 2023 (Google, 2023a).

Tanto as distribuições Linux quanto o Android compartilham do mesmo *kernel* base. Enquanto equipes de trabalho escolhem os softwares e fazem alterações ao *kernel* do Linux para disponibilizar novas distribuições, a Google combina versões estáveis com suporte de longo prazo (LTS) do *kernel* junto a modificações específicas para criar *Android Common Kernels*, que são utilizadas como base para distribuições Android (Google, 2023e). A Figura 1.1 representa a estrutura interna de um sistema operacional Android.

Assim como o *kernel* do Linux, o *kernel* do Android também é um projeto de código livre denominado *Android Open System Platform* (AOSP), e por meio dele é disponibilizada a estrutura básica e os componentes principais necessários para construção de uma distribuição Android (Google, 2022b). A Google dedicou os recursos de engenharia profissional necessários para garantir que o Android seja uma plataforma de software competitiva, tratando o projeto como uma operação de desenvolvimento de produtos em grande escala. Entretanto, algumas partes das versões do Android (incluindo as APIs da plataforma principal) são desenvolvidas em âmbito privado, e as alterações feitas na parte pública do projeto precisam ser aprovadas por funcionários da Google (Google, 2023g).

Além de desenvolver *kernels* para Android, a Google também oferece diversos serviços para os usuários do sistema operacional Android, incluindo a loja de aplicativos Google Play, que disponibiliza mais de um milhão de aplicativos. O Android garante suporte para personalização

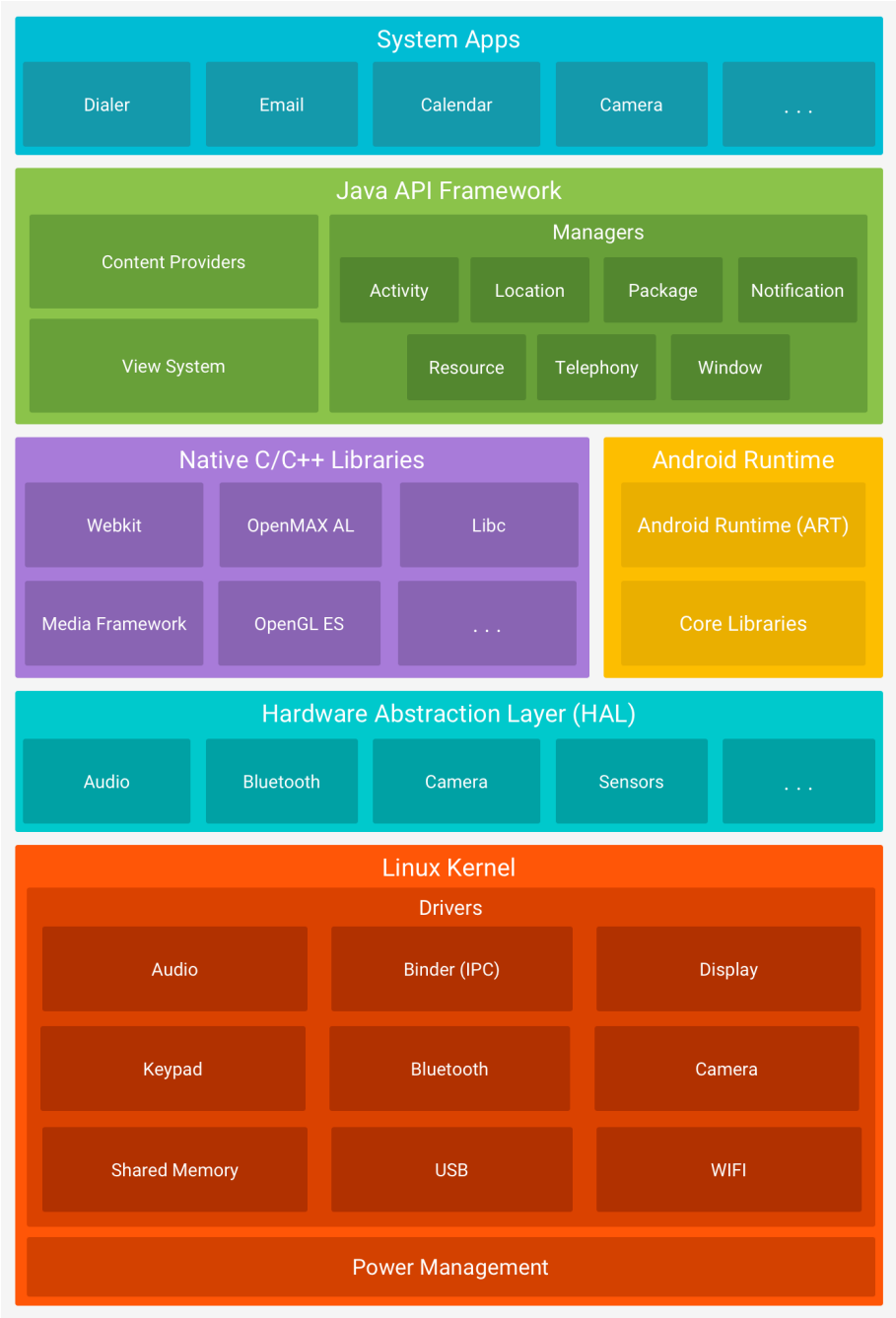

Figura 1.1: Estrutura de um sistema operacional Android (Google, 2023b).

de aparência, ícones e configurações do sistema, o que o torna o sistema operacional mais popular atualmente (Dimjašević et al., 2016).

Segundo a Agência Nacional de Telecomunicações (ANATEL), o Brasil possuía 256,4 milhões de celulares ativos em fevereiro de 2022, atingindo a marca de 104,1 aparelhos por 100 habitantes (ANATEL, 2022). Sobre os sistemas operacionais móveis, observa-se que o Android se encontra presente em 78,86% dos aparelhos em âmbito nacional, enquanto o iOS ocupa o segundo lugar, presente em 20,77% dos dispositivos (StatCounter, 2023b).

A grande popularidade do Android também o torna um sistema operacional atrativo para atividade maliciosa, incluindo o desenvolvimento e distribuição de *malware*. Essas ameaças

podem assumir várias formas, desde aplicativos infectados até links maliciosos em mensagens ou emails, o que torna a segurança móvel uma preocupação crescente.

1.3 ATAQUES NO ANDROID

Os dispositivos Android são alvo de desenvolvedores maliciosos, que buscam se infiltrar nesses equipamentos para manipular os dados do usuário. Uma forma de disseminar *malware* é pela inserção de códigos maliciosos em aplicativos e publicá-los na Google Play ou em lojas de aplicativos não oficiais. Depois de instalados, esses aplicativos podem ser capazes de se disfarçar e enviar informações de volta para o servidor do invasor através da internet (Tchakounte e Dayang, 2013). Em outros casos, o objetivo de alguns *malware* pode ser enviar SMS sem o consentimento do usuário, realizar chamadas telefônicas, roubar dados bancários ou até mesmo bloquear o dispositivo (Polisciuc et al., 2020).

Dados obtidos em 2022 informam que naquele ano foram detectados 1.661.734 instaladores maliciosos, 196.476 novos *trojans* para aplicativos bancários e 10.543 novos *trojans ransomware*. Apenas em dezembro de 2022, 2.212.114 ataques foram detectados em dispositivos móveis (Securelist, 2023). Dados mais antigos apontam que 92% dos *malware* para dispositivos móveis têm o Android como alvo principal (Dimjašević et al., 2016).

Grande parte dos ataques ocorre por download de aplicativos maliciosos a partir da Google Play. A título exemplificativo, descobriu-se que vários *malware* da familia *Harly* (*trojans* da categoria *subscribers*) foram baixados mais de 2,5 milhões vezes diretamente da loja de aplicativos oficial da Google, inscrevendo usuários em serviços pagos sem autorização. *Trojans* bancários também se destacam, pois mascaram suas atividades ilícitas atrás de aplicativos ingenuamente considerados inofensivos, como gerenciadores de arquivos (Securelist, 2023). Versões modificadas do aplicativo WhatsApp foram encontradas com código malicioso, também inscrevendo o usuário em serviços pagos sem autorização (Securelist, 2022).

Dos principais *malware* para dispositivos móveis identificados em 2022, 27,39% deles são da categoria *RiskTool* (aplicações que realizam ações sem autorização do usuário, como enviar mensagens de texto para assinar serviços), 24,05% pertencem à categoria *Adware* (cujo objetivo é infestar o dispositivo com propagandas, a fim de gerar lucro ao atacante através de cliques) e 15,56% fazem parte da família *Trojans* (que coleta dados do usuário e os envia de volta para o invasor, como senhas bancárias e capturas de tela) (Securelist, 2023). De forma geral, os usuários não estão cientes das técnicas utilizadas pelos atacantes: eles inconscientemente auxiliam o invasor a executar as ações maliciosas (Tchakounte e Dayang, 2013).

Cerca de 15 anos após o lançamento do Android, os avanços na área de segurança de sistemas impuseram aos desenvolvedores de *malware* a necessidade de estudar novas técnicas para fazer com que seus programas não fossem detectados por mecanismos usuais de proteção (Urbanski et al., 2016). Como consequência, a detecção de malware no Android é uma área de pesquisa de interesse tanto da indústria quanto da academia (Dimjašević et al., 2016).

A atividade de detecção de ataques no Android se diferencia das rotinas de segurança aplicadas a outros computadores Linux, pois é limitada por um fator de extrema importância no contexto móvel: o consumo energético.

Processador, bateria e memória são considerados os mais importantes e cruciais recursos de um smartphone. A execução local de aplicações para detecção de qualquer tipo de *malware* pode aumentar a taxa de consumo desses recursos (Gera et al., 2021). Aplicativos antivírus, como Avast, Kaspersky e McAfee, são capazes de rapidamente esgotar a energia disponível e, além disso, adotam técnicas de análise estática, o que limita o alcance de detecção de ataques (Khune e Thangakumar, 2012).

O consumo excessivo dos recursos de hardware pode ser minimizado com a adoção de uma abordagem mista, na qual a análise detalhada e computacionalmente custosa pode ser feita em servidores remotos a partir do envio de parâmetros de uso de recursos e outros dados pelo aparelho celular cliente. Entretanto, a necessidade de disponibilidade constante de acesso à internet e seus custos associados pode tornar esta abordagem inviável (Faruki et al., 2015).

1.4 PROPOSTA E CONTRIBUIÇÃO DESTE TRABALHO

Considerando que o Android (a) é um sistema baseado no *kernel* do Linux, (b) é o sistema operacional móvel mais popular, (c) é o grande foco de ataques que possuem sistemas móveis como alvo e (d) possui limitações quanto à detecção desses ataques, propõe-se verificar a viabilidade do uso de ferramentas nativas do sistema operacional para proteção dos dispositivos móveis com Android.

Deste modo, o objetivo principal do trabalho é analisar ferramentas de *tracing* existentes no Android (de preferência que funcionem nativamente) que permitam sua utilização em prol da segurança do sistema. Para tanto, foram realizados experimentos controlados com vulnerabilidades selecionadas para identificar ataques, correlacionando os dados de saída de cada ferramenta avaliada às características dos códigos de exploração presentes nos ataques e exibindo os resultados alcançados. Uma das contribuições deste trabalho é servir de base para uma lista de ferramentas que auxilie analistas de segurança, mostrando o alcance, tipos de saídas e as limitações técnicas encontradas em cada uma na obtenção de conhecimento mais profundo sobre a execução de ataques em sistemas operacionais Android.

2 REVISÃO TEÓRICA E TRABALHOS RELACIONADOS

Neste capítulo detalhes sobre a arquitetura do Android são apresentadas, acompanhadas de definições clássicas sobre *malware* e *tracing*. Após uma breve exposição sobre ferramentas de *tracing* disponíveis no contexto do Linux e do Android, indicações de estudos relacionados são apontados ao final.

2.1 PLATAFORMA ANDROID

A base da plataforma Android é o *kernel* do Linux, que tem sido amplamente utilizado por anos e está presente em milhões de ambientes sensíveis à segurança. Ao longo de sua história, ele vem sendo constantemente pesquisado, atacado e corrigido por milhares de desenvolvedores, tornando-se um *kernel* estável e seguro, confiável para muitas empresas e profissionais de segurança.

Como base para um ambiente de computação móvel, o *kernel* do Linux oferece ao Android várias características de segurança importantes, incluindo um modelo de permissões baseado em usuário, isolamento de processos, um mecanismo extensível para comunicação interprocessos segura, a capacidade de remover partes desnecessárias e potencialmente inseguras do *kernel*, entre outras. É possível destacar as seguintes características de segurança do Android (Google, 2023j):

- *Isolamento de Aplicativos*: Como um sistema operacional multiusuário, um objetivo fundamental de segurança do *kernel* do Linux é isolar os recursos entre os usuários. A plataforma Android aproveita a proteção baseada em usuários do Linux para identificar os recursos dos aplicativos, isolando-os uns dos outros com o objetivo de impedir que programas maliciosos interfiram no funcionamento deles ou do sistema operacional. Para isso, o Android atribui um identificador de usuário único (UID) a cada aplicativo Android e o executa em seu próprio processo. Nesse sistema, cada aplicativo é executado como seu próprio usuário e, a menos que o desenvolvedor compartilhe arquivos explicitamente com outros aplicativos, os arquivos criados por um aplicativo não podem ser lidos ou alterados por outro.

- *Criptografia e integridade do sistema operacional*: O Android também conta com várias partições protegidas que abrigam tanto o *kernel* do sistema quanto as bibliotecas do sistema operacional e os dados dos aplicativos instalados, com suporte à criptografia. A partir da versão 7 do sistema operacional, o esquema de inicialização verificada (*verified boot*) foi disponibilizado, impedindo que aparelhos comprometidos conseguissem funcionar corretamente. A inicialização verificada garante a integridade do software do dispositivo a partir de uma raiz confiável de hardware até a partição do sistema. Durante

a inicialização, cada estágio verifica criptograficamente a integridade e autenticidade do próximo estágio antes de executá-lo. O Android fornece um conjunto de APIs criptográficas para uso por aplicativos, incluindo a implementação de primitivas criptográficas padrão comumente usadas, como AES, RSA, DSA e SHA. Além disso, APIs também são fornecidas para protocolos de nível superior, como SSL e HTTP.

- *Controle de acesso*: O Android utiliza *SELinux* (*Security-Enhanced Linux*) para impor o controle de acesso obrigatório (*Mandatory Access Control - MAC*) em todos os processos, incluindo aqueles executados com privilégios de superusuário (Google, 2022a). SELinux é um conjunto de modificações e políticas de segurança implementadas no *kernel* do Linux para fornecer um nível aprimorado de controle, restringindo as permissões e os acessos dos processos e usuários. O SELinux opera com base no princípio de negação padrão (onde qualquer ação que não seja explicitamente permitida é negada) e o Android inclui este mecanismo de segurança no modo de imposição (no qual as negações de permissões são impostas e registradas em *log*) (Haines e Moore, 2022).

- *Comunicação interprocessos*: No Linux, a comunicação entre processos pode ocorrer através de uma grande variedade de mecanismos, como sinais, *pipes, sockets*, filas de mensagem, semáforos e memória compartilhada. O Android administra a comunicação entre processos por meio do *Binder*, um mecanismo personalizado que fornece um *framework* para comunicação entre diferentes processos e permite que métodos em objetos remotos sejam chamados como se fossem métodos de objetos locais. O *Binder* inclui um modelo cliente-servidor, uma camada de *middleware* e um *driver* de *kernel*. Ele também suporta recursos básicos de segurança, como identificação de UID e PID (*User Identifier* e *Process Identifier*, respectivamente) (Schreiber, 2011).

Os mecanismos de segurança do Android apresentam um alto grau de estabilidade, segurança e confiabilidade. Entretanto, vulnerabilidades no sistema operacional não deixam de ser encontradas por pesquisadores e agentes maliciosos, ressaltando a necessidade de aperfeiçoamento contínuo da plataforma. Ataques que têm o Android como alvo podem ser altamente complexos e abusar das estruturas que não são compartilhadas com o *kernel* comum do Linux.

2.2 ATAQUES COMPUTACIONAIS E EM DISPOSITIVOS MÓVEIS

Os primeiros ataques a sistemas computacionais datam do início da década de 1960 e se tratavam de processos para roubo de senhas e credenciais. Em 1988, registrou-se o primeiro grande ataque via internet: um *worm* que se replicava nas máquinas da rede e esgotava seus recursos (Wolf, 2022).

Atualmente, os ataques são altamente sofisticados e complexos, abusam de vulnerabilidades mínimas, são capazes de se ocultar em ambientes monitorados e ainda geram grandes danos. O vírus Melissa, por exemplo, causou 80 milhões de dólares em prejuízo à Microsoft em

1999; um ataque aos bancos de dados de usuários da Playstation Network foi responsável pelo vazamento de informações pessoais de 77 milhões de pessoas em 2014; o *ransomware* WannaCry afetou mais de 200.000 computadores em 150 países no ano de 2017, causando um impacto de mais de 6 milhões de libras (Carter, 2021).

Na última década, tornou-se perceptível a mudança do paradigma de acesso à internet. Hoje, o celular é o principal meio de acesso à internet no Brasil (Tokarnia, 2020). Dados globais estimaram que, em 2019, cerca de 50,38% de todo tráfego web teve origem em aparelhos móveis, enquanto máquinas desktop representaram 46,51% do tráfego web total. Os tablets contribuem com a menor quantidade de tráfego, estimada em 3% (Bouchrika, 2023).

Esses avanços tecnológicos e mudanças no acesso à internet têm implicações significativas no cenário de segurança cibernética. Os ataques direcionados a dispositivos móveis têm o potencial de causar grandes prejuízos em diversos aspectos, como roubo de informações pessoais, acesso não autorizado a serviços e vandalização em redes sociais, por exemplo. Adicionalmente, ataques direcionados a dispositivos móveis também podem afetar empresas e organizações, uma vez que muitos dispositivos são utilizados para acessar dados e sistemas corporativos (Powers, 2021).

2.2.1 Definições

Para auxiliar na compreensão dos tópicos expostos neste trabalho, resgatam-se algumas definições de termos comumente utilizados no contexto de ataques computacionais e em dispositivos móveis:

- *Vulnerabilidade*: Uma vulnerabilidade é um defeito ou problema presente na especificação, implementação, configuração ou operação de um software ou sistema, que possa ser explorado para violar as suas propriedades de segurança. A grande maioria das vulnerabilidades ocorre devido a erros de programação, como, por exemplo, não verificar a conformidade dos dados recebidos de um usuário ou da rede (Maziero, 2019a).

- *Ataque*: Um ataque é o ato de utilizar (ou explorar) uma vulnerabilidade para violar uma propriedade de segurança do sistema. Ataques podem ser de quatro tipos, como interrupção (impedir o fluxo normal das informações ou acessos), interceptação (obter acesso indevido a um fluxo de informações, sem necessariamente modificá-las), modificação (modificar de forma indevida informações ou partes do sistema, violando sua integridade) ou fabricação (produzir informações falsas ou introduzir módulos ou componentes maliciosos no sistema). Podem ainda ser passivos (quando visam capturar informações confidenciais) ou ativos (quando visam introduzir modificações no sistema para beneficiar o atacante ou impedir seu uso pelos usuários válidos), locais (executados por usuários válidos do sistema) ou remotos (realizados através da rede, sem fazer uso de uma conta de usuário local) (Maziero, 2019a).

- *Malware*: Denomina-se genericamente *malware* (*malicious software)* todo programa cuja intenção é realizar atividades ilícitas, como realizar ataques, roubar informações ou dissimular a presença de intrusos em um sistema. Existe uma grande diversidade de *malware*, destinados às mais diversas finalidades (Maziero, 2019a). Atuam sem o consentimento ou conhecimento do usuário (Grégio, 2012).

- *Vírus*: Um dos primeiros tipos populares de *malware*. Um vírus de computador é um código que se anexa a outro programa e se espalha quando este programa infectado é executado (Grégio, 2012).

- *Antivírus*: Um software desenvolvido para proteger computadores, seja pela varredura do sistema de arquivos na busca por programas maliciosos ou por monitoramento das atividades do sistema (Grégio, 2012).

- *Worm*: Outro tipo popular de *malware*, um *worm* é um programa capaz de se espalhar autonomamente. Os *worms* geralmente procuram serviços vulneráveis, exploram-nos e copiam eles mesmos para a máquina comprometida (Grégio, 2012).

- *Trojan*: Um programa de computador que pode apresentar recursos legítimos, mas que também possui funções ocultas cuja intenção é maliciosa (Grégio, 2012).

- *Rootkit*: Um tipo especial de *malware* que geralmente opera no nível do *kernel*. Os *rootkits* podem ocultar programas e arquivos maliciosos e diretórios e comunicações de rede. Eles também pode desabilitar os mecanismos de segurança e modificar o comportamento do sistema infectado (Grégio, 2012).

- *Exploit*: Um tipo de *malware*. É um programa escrito para explorar vulnerabilidades conhecidas, como prova de conceito ou como parte de um ataque. Os *exploits* podem estar incorporados a outros *malware* (como *worms* e *trojans*) ou constituírem ferramentas autônomas, usadas em ataques manuais (Maziero, 2019a).

- *Hooking*: Técnica utilizada para interceptar eventos, chamadas e/ou mensagens com o objetivo de alterar ou monitorar o comportamento do sistema operacional.

Apesar da classificação clássica indicada acima, os *malwares* atuais normalmente apresentam várias funcionalidades complementares. Por exemplo, um mesmo *malware* pode se propagar como *worm*, dissimular-se no sistema como *rootkit*, capturar informações locais usando *keylogger* e manter uma *backdoor* para acesso remoto.

2.2.2 Categorias de ataques mais frequentes

No contexto do Android, alguns tipos de ataques ocorrem com mais frequência que outros, como infica a Figura 2.1. Observa-se a tendência de que sejam encontradas em maior

número vulnerabilidades que permitem a execução de código arbitrário, abusam de *buffer overflows* ou evitam restrições sobre permissões, representando mais de 40% de todas as vulnerabilidades cadastradas para o Android (CVEDetails, 2023b).

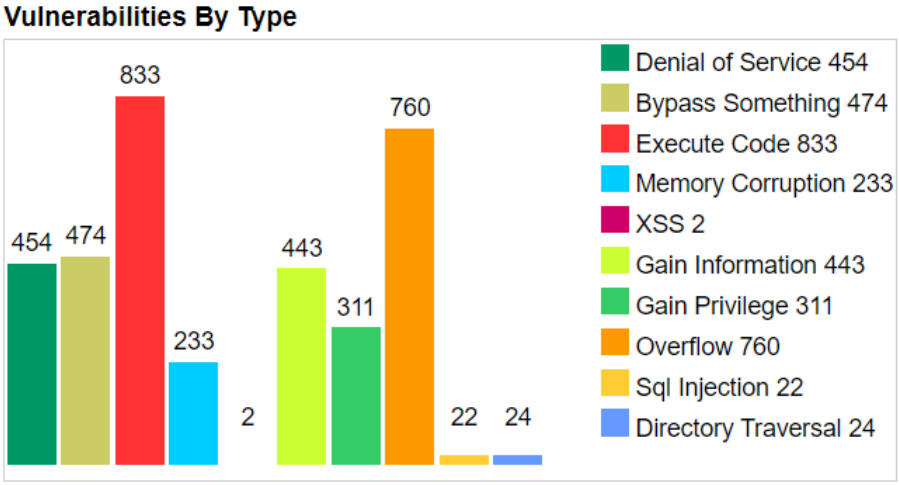

Figura 2.1: Vulnerabilidades registradas para Android, por tipo (CVEDetails, 2023b)

Execução arbitrária de código é o tipo de ataque que permite que atacantes injetem seu próprio código malicioso em um sistema-alvo vulnerável sem o consentimento dos usuários. As ramificações deste tipo de ataque são significativas e potencialmente desastrosas, tornado-o uma das ameaças mais urgentes enfrentadas pelos times de segurança atualmente (NordVPN, 2023).

Em um ataque de *buffer overflow* sobre um programa vulnerável, o invasor tenta modificar o estado da memória do programa para que ele ceda o controle da máquina ao atacante. O ataque ocorre através do fornecimento de dados de entrada cuidadosamente elaborados para extrapolar os limites do programa. A vulnerabilidade existe em programas que não verificam se a entrada excede seu tamanho de *buffer* de memória e copiam os dados em excesso para locais adjacentes ao *buffer*. Ao controlar o conteúdo, o invasor pode fazer com que o programa se desvie de sua finalidade pretendida (Ruwase e Lam, 2004).

Vulnerabilidades sobre o esquema de permissões existem quando a checagem de controle de acesso não é aplicada corretamente. Nesse cenário, invasores são capazes de acessar dados ou executar ações que não deveriam ser permitidas, levando a problemas mais sérios, o que inclui vazamento de informações privadas, negação de serviço e execução de código arbitrário (CWE, 2023).

Do mesmo modo que ocorre com a classificação de *malwares*, ataques podem ter comportamentos complementares. Dado que o maior objetivo de todas as vulnerabilidades é a obtenção dos privilégios de superusuário no sistema atacado, a existência de múltiplas vulnerabilidades e de múltiplos tipos de ataque permite que haja colaboração entre eles para que tal objetivo seja alcançado. Por exemplo, uma aplicação que não checa os limites do *buffer* pode ser utilizada como meio de driblar as permissões dentro do sistema, permitindo que um código arbitrário seja executado. De fato, vê-se que inúmeras CVEs (*Common Vulnerabilities*

and Exposures) são classificadas em mais de um tipo, como a CVE-2023-21130, que pertence aos tipos de "execução arbitrária de código"e "*buffer overflow*"simultaneamente (CVEDetails, 2023a).

2.3 TRACING

A evolução do processo de software permitiu o desenvolvimento de aplicações complexas, numerosas e interdependentes. Softwares atuais podem se valer de técnicas de paralelização e são executados em sistemas operacionais sofisticados como o Linux, o que leva à necessidade natural de análise de performance de software (LTTng, 2021). Compreender o comportamento dos programas em execução permite a redução de custos com tecnologia da informação, a construção de arquiteturas escaláveis e a solução de problemas técnicos, como localizar gargalos e discrepâncias no sistema em geral (Gregg, 2013).

Sob a ótica da segurança da informação, a análise de performance fornece o instrumental necessário para detecção de ameaças modernas. Conforme atacantes progressivamente introduzem novas técnicas para evadir sistemas de detecção por assinatura e por regras, surgem *malwares* dotados de alguma capacidade de ocultação de informações para, por exemplo, exfiltrar dados para servidores de Comando e Controle (C&C) ou para embutir funções de ataque ou arquivos de configuração em arquivos aparentemente inofensivos, como imagens (Caviglione et al., 2021).

Nessa perspectiva, o uso de tecnologias que fornecem uma visão prática sobre aplicações, infraestruturas e dispositivos é necessário. Tais tecnologias devem ser capazes de monitorar, inspecionar e rastrear processos em tempo de execução, permitindo o tratamento de ataques específicos sem sobrecarregar o sistema (Caviglione et al., 2021). Em outras palavras, a detecção de tais ataques se dá principalmente pela aplicação de técnicas de análise dinâmica, que consiste na execução de aplicações para captura de ações maliciosas (Polisciuc et al., 2020). A operacionalização das técnicas de análise dinâmica, por sua vez, pode se valer de ferramentas de análise de performance, como as de *tracing*.

2.3.1 Definições

Tracing é a técnica usada para obter várias informações sobre o estado do sistema ou do processo em execução (Vurdelja et al., 2020). Quando executado, um *tracer* passa a gravar a sessão de execução do software de forma que, quando pontos específicos de instrumentação são encontrados, eventos são gerados e salvos em um arquivo de *trace* (LTTng, 2021). Um arquivo de *trace* contém detalhes suficientes para reconstruir uma linha do tempo de eventos e usualmente apresentam dados de eventos que ocorrem em baixo nível, como dados do *scheduler* (escalonador de tarefas), dados sobre as *threads*, *syscalls* (chamadas de sistema), etc. (Perfetto, 2023e).

Tracing se diferencia de *logging* por se tratar de uma forma estruturada que permite a reconstrução da sequência dos eventos identificados (Perfetto, 2023e). *Tracers* são projetados

para capturar eventos de baixo nível e que ocorrem em frequência consideravelmente maior que as mensagens de *log* (atingindo a marca de milhares de eventos por segundo). Técnicas de *logging* são adequadas para análise de alto nível de eventos menos frequentes, como acessos de usuário, condições de exceção, transações de bancos de dados, etc (LTTng, 2021). Ademais, os *tracers* colocam grande ênfase em causar o mínimo de *overhead* (sobrecarga), de modo a não romper o fluxo natural de execução do software sob análise (Perfetto, 2023e).

Alguns conceitos fundamentais que são usualmente utilizados no contexto de *tracing* são os seguintes (Gebai e Dagenais, 2018):

- *Tracepoint*: É uma instrução colocada diretamente no código de uma aplicação que fornece um meio de invocar uma *probe*. Geralmente, um *tracepoint* pode ser ativado e desativado dinamicamente.
- *Probe*: É uma função que está ligada a um *tracepoint*, é chamada sempre que o mesmo for encontrado em tempo de execução (se ativado). Tipicamente, a *probe* é uma função pequena e rápida, para que não haja um *overhead* significativo, perturbando o menos possível o sistema.
- *Evento*: Marca o encontro de um *tracepoint* em tempo de execução. Pode representar tanto algo lógico, como uma troca de contexto ou um endereço no código, como uma entrada de função. Um evento é pontual, ou seja, não possui duração.
- *Payload*: Um evento normalmente possui um *payload*, responsável por conter informações adicionais relacionadas ao primeiro.
- *Ring buffer*: Estrutura de dados que serve como um *placeholder* dos eventos.
- *Operações Atômicas*: É uma operação que tem como característica ser indivisível, isto é, valores e estados intermediários são invisíveis para outras operações. Operações atômicas, em sua maioria, necessitam de suporte do *hardware*.

Os *tracers* se diferenciam de acordo com a forma de chamada de *probes* (*probe callback*), podendo inclusive dar suporte a várias delas e deixar a decisão de qual mecanismo utilizar por conta do usuário. As ferramentas implementam seus mecanismos de chamadas de *probes* de diversas maneiras e elas são divididas em duas categorias:

- *Estática*: Nesta categoria, os programas são instrumentados de forma que as chamadas de *probes* são incorporadas ao binário em tempo de compilação, quando a localização do *tracepoint* é conhecida previamente. Esse tipo de mecanismo requer suporte do compilador e tem cada chamada de função de um programa precedida por uma chamada para *probe*. A implementação dessa rotina resta sob responsabilidade dos desenvolvedores ou do *tracer* e pode ser utilizada para implementar o *tracing* em

si, ferramentas de perfilamento ou qualquer outro recurso de monitoramento. *Probe callbacks* estáticos incluem *static tracepoints* (inserção manual diretamente no código dos programas pelos desenvolvedores, utilizando macros disponíveis no *kernel* do Linux) e *function instrumentation* (inserção automática de chamadas para *probes* diretamente pelo compilador antes de cada chamada de função). Este mecanismo é implementado pelo LTTng, Ftrace, Perf e eBPF, por exemplo.

- *Dinâmica*: Diferentemente do que ocorre com os mecanismos estáticos, os mecanismos dinâmicos instrumentalizam os programas em tempo de execução em locais definidos pelo usuário. O caráter dinâmico desse método permite que ele seja ligado e desligado durante seu processamento. Métodos dinâmicos incluem *traps* (implementado através de *kprobes* no *kernel* do Linux) e *trampolines* (uma alternativa com menos *overhead* em comparação às *traps*, mas com maior complexidade de implementação). SystemTap, Strace e Extrae são exemplos de ferramentas que implementam o método dinâmico.

2.3.2 Ferramentas de coleta de traços disponíveis

A coleta de traços no sistema Linux pode ocorrer por meio do uso de diversas ferramentas, as quais atuam nos diferentes níveis de sistema, conforme demonstra a Figura 2.2.

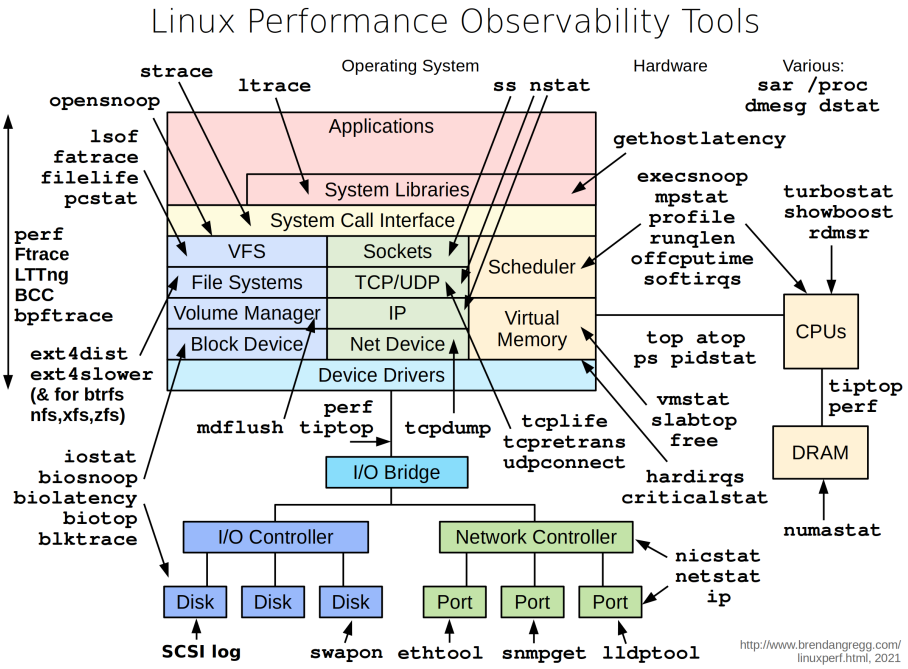

Figura 2.2: Ferramentas de análise de performance e coleta de traços do Linux (Gregg, 2021).

Algumas ferramentas merecem destaque, como as seguintes (Gebai e Dagenais, 2018):

- *Ftrace*: É um *tracer* incluso no *kernel* do Linux que funciona realizando o *tracing* de todas as entradas de suas funções. Pode ser utilizado de diversas maneiras, entre

elas *tracing* de funções, chamadas de sistema e *tracepoints*. Pode utilizar tanto a infraestrutura *TRACE_EVENT* para *tracing* estático, quanto a Kprobe para dinâmico.

- *LTTng*: Possui muitas similaridades em relação ao Ftrace, no entanto, não é nativo ao *kernel*. O LTTng foi desenvolvido e implementado objetivando minimizar o impacto na performance.

- *LTTng-kprobe*: Possui as mesmas características do LTTng, porém faz uso de kprobes ao invés de TRACE_EVENTS para realizar *probe callbacks*.

- *Perf*: É uma ferramenta de monitoramento de performance integrada ao *kernel*. Pode ser utilizada para interagir com a infraestrutura de *tracepoints*, registrando chamadas de sistema, por exemplo. O escopo do monitoramento do Perf se restringe apenas a um processo.

- *eBPF*: Evoluída do *Bekerley Package Filter*, é uma ferramenta de monitoramento nativa do Linux que permite aos usuários escreverem programas que podem ser inseridos dinamicamente em qualquer local no *kernel*, utilizando *kprobes*. Programas em eBPF são compilados para *bytecode*, que são interpretados e executados por uma espécie de simulação do kernel.

- *Strace*: Ferramenta utilizada para realizar o *tracing* de chamadas de sistema. O Strace funciona se prendendo a um processo e interceptando todas as suas chamadas, bem como seus respectivos argumentos. Devido ao fato de vários processos simultâneos serem necessários para realizar o monitoramento, o Strace causa um grande *overhead*, o que é contraindicado em ambientes de produção.

- *SysDig*: É uma ferramenta moderna cujo principal utilidade é o monitoramento de sistemas. Pode ser empregada na captura, filtragem e decodificação de chamadas de sistema e outros eventos do sistema operacional. O *SysDig* também permite aos usuários criarem *scripts* de análise customizados, chamados de *chisels*. No entanto, o *SysDig* não dá controle aos usuários sobre a parte de *tracing* do *kernel*, sendo essa funcionalidade utilizada apenas na coleta de algumas métricas sobre as aplicações monitoradas, como *reads*, *writes* e uso de CPU.

- *Ltrace*: Ferramenta semelhante ao *Strace*, que mostra as chamadas a bibliotecas compartilhadas realizadas por uma aplicação.

- *Dtrace*: Uma das ferramentas pioneiras de análise de desempenho e solução de problemas incluída por padrão em vários sistemas operacionais, incluindo Solaris, Mac OS X e FreeBSD. Em que pese um versão para Linux tenha sido anunciada, não foram encontrados indícios de que seu desenvolvimento continua ativo.

- *Blktrace*: Responsável por gerar *traces* do tráfego de I/O em dispositivos de bloco, como USB e cartões SD.

Quando se adota o Android como sistema operacional objeto de estudo, verifica-se que o conjunto de ferramentas de *tracing* disponíveis representa um pequeno subconjunto daquelas com suporte para Linux. Além das ferramentas *Strace* e *Ftrace* (descritas anteriormente), as seguintes ferramentas se encontram presentes nativamente nas distribuições Android:

- *Systrace e Atrace*: Systrace é a ferramenta primária para análise de desempenho no Android. Entretanto, seu funcionamento está intimamente ligado às chamadas à *Ftrace* que executa e é projetada para atuar através do *host*. *Atrace* é a ferramenta que controla a coleta de traços no espaço do usuário e configura o *Ftrace*. *Systrace* usa o *Atrace* para habilitar o *tracing*; em seguida, lê o buffer do *Ftrace* e encapsula tudo em um visualizador HTML autocontido (Google, 2022c).

- *Perfetto*: É uma ferramenta de instrumentação de desempenho e análise de traços que permite a coleta de informações de performance via *Android Debug Bridge* (*adb*). Utiliza outras fontes de dados para coleta de traços, como *ftrace*, *atrace* e *heapprofd* (Google, 2023f).

O escopo deste trabalho se restringirá às ferramentas disponíveis nativamente no Android. Detalhes sobre o funcionamento de cada uma delas serão apresentados no próximo capítulo.

2.4 TRABALHOS RELACIONADOS

A coleta e análise de traços de *malware* por meio do uso de ferramentas nativas do Linux é um objeto recorrente de estudos. Em (Shende, 1999), uma breve descrição das ferramentas de *tracing* e *profiling* do Linux é apresentada, com enfoque nas questões de performance. Em (Caviglione et al., 2021), utiliza-se a ferramenta *eBPF* para coletar traços de ataques de *stegomalware* que atuam via alteração do sistema de arquivos e tentativas de comunicação de rede ocultas aninhadas nos fluxos de tráfego IPv6. Este último estudo indica que até mesmo os programas *eBPF* simples são capazes de capturar informações úteis para a detecção de anomalias, com uma sobrecarga mínima.

Em (Gebai e Dagenais, 2018), há uma análise detalhada das ferramentas de *tracing* disponíveis no Linux, incluindo as que atuam no nível do usuário e as que atuam no nível do *kernel*. Entretanto, o objetivo final do estudo visava a análise de desempenho e de sobrecarga do sistema, não se propondo a analisar questões de segurança. Em (Nguyen et al., 2022), uma ferramenta de coleta de traços foi desenvolvida e testada para obtenção de informações de *ransomwares*. Os traços coletados foram comparados aos que eram obtidos pela ferramenta *strace*, concluindo-se que a ferramenta testada obtinha o mesmo conjunto de informações com uma sobrecarga menor.

Em (Vurdelja et al., 2020) é oferecida uma visão geral das soluções para detecção de *malware* através do uso de *tracers* no contexto do Linux. Uma infraestrutura *sandbox* para realização de análise dinãmica de *malware* foi criada para garantir o isolamento seguro do *host* e, para coleta dos dados, foi utilizado *strace*. O estudo indica que programas maliciosos geralmente são distribuídos como arquivos executáveis com bibliotecas estáticas e fazem mais chamadas de sistema relacionadas à rede, o que permitiria diferenciá-los de programas benignos, mais ocupados em fazer chamadas para carregamento de bibliotecas dinâmicas e chamadas de sistema relacionadas ao sistema de arquivos.

Especificamente em relação a ambientes Android, em (Canfora et al., 2013) encontra-se uma abordagem de análise de *syscalls* para detecção de *malware*. O estudo se propôs a identificar um conjunto de *syscalls* que poderiam ser utilizadas para fins maliciosos e estabelecer uma soma ponderada de um conjunto de permissões do sistema operacional, calculando a ocorrência de certas *syscalls* para indicar a existência de uma ação maliciosa. Os estudos prosseguiram em (Canfora et al., 2015), demonstrando que a técnica de análise de *syscalls* pode atingir 97% de precisão.

Restringindo o escopo da análise de ferramentas de *tracing* para o campo do Android, (Tchakounte e Dayang, 2013) utilizam *strace* sobre duas amostras de *malware*, obtendo informações de que até mesmo toques na tela fora da área dos aplicativos maliciosos podem fazer parte de um ataque.

Em (Dimjašević et al., 2016), a ferramenta `MALINE` é apresentada. Ela utiliza *strace* para coleta de traços de aplicativos maliciosos em aparelhos Android emulados e métodos de *machine learning* para classificar ataques.

Estudos sobre as ferramentas de *tracing* são frequentemente encontrados nos meios acadêmicos. Entretanto, este trabalho se diferencia dos demais considerando os seguintes pontos:

- *Escopo limitado ao ambiente Android*: Diversos pesquisadores já propuseram estudos sobre os *tracers* disponíveis para Linux. No entanto, muitos deles observam tais ferramentas sob a ótica da análise de performance, deixando a segurança em segundo plano. Este trabalho limita seu escopo às ferramentas disponíveis no Android, o que representa um subconjunto daquelas com suporte para Linux, além de se preocupar principalmente com questões de segurança dos dispositivos móveis.

- *Análise de todas as ferramentas disponíveis no Android*: Estudos que tratam da coleta de traços em sistemas Android o fazem ou pela implementação de uma ferramenta própria ou pelo uso de *strace*. Este trabalho se propõe a avaliar outras ferramentas de coleta de traço, incluindo a *Perfetto*, cuja aplicação não foi encontrada nas demais pesquisar relacionadas.

- *Correlação dos traços e dos ataques*: Outros trabalhos já analisaram os traços coletados via *strace* com o comportamento de *malware*, mas a falta de detalhes não permite que a

vantagem do uso de cada ferramenta seja apurada. Este estudo se propõe a correlacionar traços obtidos pelas diversas ferramentas para registrar as características de cada ataque, fornecendo um panorama geral do comportamento do *malware* por meio de análise estática e dinâmica.

Os próximos capítulos detalharão as ferramentas utilizadas na metodologia de experimentação e os resultados obtidos.

3 FERRAMENTAS DE *TRACING* NATIVAS DO ANDROID

Embora o Linux tenha um grande ecossistema de ferramentas de *tracing* disponíveis, o número de instrumentos com suporte ao Android é pequeno. Destacam-se, nesse conjunto de ferramentas, o *Systrace/Atrace*, o *Ftrace*, o *Strace* e o *Perfetto*.

Para fins deste estudo, foram consideradas como ferramentas nativas aquelas utilizáveis a partir de uma *shell* de um aparelho Android emulado, obtida através da *Android Debug Bridge* (*adb*). A documentação do Android referencia estas ferramentas como *command line tools* e uma lista delas pode ser obtida com o comando `adb shell ls /system/bin`.

A versão 10 do Android (Android Q, em arquitetura x86, API 29, *kernel* versão 4.14.175, sem habilitação de serviços Google Play) foi escolhida para realização dos testes das ferramentas, que são utilizadas para instrumentalização de programas simples, como `ps`. Apesar de simples, tais programas são úteis pois fornecem uma visão geral dos potenciais resultados que podem ser obtidos, permitindo que uma análise de usabilidade de cada ferramenta seja feita. Ademais, a versão do Android escolhida também será utilizada para testes em ataques reais, relatados no próximo capítulo.

3.1 *SYSTRACE* E *ATRACE*

Como mencionado no capítulo anterior, a documentação oficial do Android indica que o *systrace* é a ferramenta primária para análise de desempenho, tendo sua funcionalidade atrelada ao *ftrace* (Google, 2022c). Nesse contexto, *atrace* seria a ferramenta que controla a coleta de traços no espaço do usuário e configura o *ftrace* no aparelho celular. A operacionalização do *systrace* se daria pelo uso do *atrace* para habilitação do *tracing*, com a conseguinte leitura do *buffer* do *ftrace* e encapsulamento dos dados em um arquivo HTML (Google, 2022c).

Entretanto, ainda que a documentação oficial mencione o *systrace* como uma ferramenta para coleta de traços (Google, 2023c), ele deixou de ser distribuído pela Google em março de 2022 (Google, 2023i).

Atrace, por sua vez, ainda é acessível via `adb shell`. Contudo, a falta de documentação do uso da ferramenta dificulta a tarefa de detalhar seu funcionamento, de forma que a análise do código fonte do utilitário *systrace* foi necessária (Google, 2013).

A execução do *atrace* segue o formato `atrace [argumentos] [categorias de traço]` e deve ser ajustada por meio da indicação dos parâmetros. As categorias disponíveis são exibidas por meio de uma tabela constante no apêndice A. A maioria dos argumentos são idênticos aos disponíveis para o utilitário *systrace* (Google, 2023c), dos quais se destacam os seguintes:

- `-a <appname>`: habilita a coleta de traços no aplicativo indicado por `appname`;

- `-b <N>`: utiliza um *buffer* de `N` *kilobytes*;
- `-z`: comprime os traços coletados;
- `-t`: ajusta o intervalo da coleta de traços;
- `-o`: redireciona a saída do programa para o arquivo especificado;
- `-list_categories`: exibe uma lista com as categorias disponíveis para captura de traços.

Ao utilizar o comando `atrace -o /data/local/tmp/atrace -b 96000 sched freq idle am wm binder_driver -t 10`, o *tracer* passa a observar o sistema por 10 segundos e captura eventos acerca do escalonador de tarefas, *activity manager, window manager*, tempo ocioso de CPU e *Binder*, salvando os traços em um arquivo de texto. O programa *ps* foi executado durante o intervalo de observação. Um recorte do conteúdo do arquivo é exibido na Listagem 3.1.

Listing 3.1: Recorte de traços obtidos pela ferramenta *atrace* durante execução do programa `ps`

```
TRACE:
# tracer: nop
#
# entries-in-buffer/entries-written: 44093/44093 #P:4
#
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
#                            || / _--=> preempt-depth
#                            ||| /   delay
#         TASK-PID TGID  CPU# ||||  TIMESTAMP FUNCTION
#           | |      |     |  ||||     |       |
        atrace-3903 ( 3903) [000] .... 6777.461079: tracing_mark_write:
                                            trace_event_clock_sync:
                                            parent_ts=6777.460938
        atrace-3903 ( 3903) [000] .... 6777.461086: tracing_mark_write:
                                            trace_event_clock_sync:
                                            realtime_ts=1687803209747
        atrace-3903 ( 3903) [000] d..2 6777.461103: sched_switch:
                                            prev_comm=atrace
                                            prev_pid=3903
                                            prev_prio=120
                                            prev_state=S ==>
                                            next_comm=swapper/0
                                            next_pid=0 next_prio=120
        <idle>-0  (-----) [000] d..1 6777.461112: cpu_idle: state=1 cpu_id=0
           \\...
```

```
28            ps-3904 ( 3904) [002] d..2 6779.124823: sched_waking:
29                                                    comm=migration/2 pid=22
30                                                    prio=0 target_cpu=002
31            ps-3904 ( 3904) [002] dN.3 6779.124827: sched_wakeup:
32                                                    comm=migration/2 pid=22
33                                                    prio=0 target_cpu=002
34            ps-3904 ( 3904) [002] d..2 6779.124832: sched_switch:
35                                                    prev_comm=sh prev_pid=3904
36                                                    prev_prio=120
37                                                    prev_state=R+ ==>
38                                                    next_comm=migration/2
39                                                    next_pid=22 next_prio=0
```

O arquivo de saída do *atrace* é entregue em formato de texto simples, que pode ser lido pela maioria dos editores de texto.

3.2 *FTRACE*

O *ftrace* é uma ferramenta de depuração que permite aos desenvolvedores a análise do que ocorre no espaço do *kernel* do Linux. Embora *ftrace* seja considerado tipicamente como um *tracer* de funções, ele é melhor entendido como um *framework* com variados utilitários para coleta de traços (Google, 2022d).

Um dos usos mais comuns do ftrace é o *tracing* de eventos. Ao longo do código do *kernel* existem centenas de pontos de eventos estáticos que pode ser habilitados por meio do *tracefs* que permitem a análise do ocorre em certas partes do *kernel* em tempo de execução (Rostedt, 2022).

O *ftrace* usa o sistema de arquivos *tracefs* para manter os arquivos de controle bem como os arquivos que contém a saída com os traços coletados. Quando *tracefs* é configurado no *kernel*, o diretório `/sys/kernel/tracing` também é criado. O diretório contém arquivos úteis como `current_tracer` (configura ou exibe o *tracer* atual), `avaliable_tracers` (lista os diferentes tipos de *tracers* que foram compilados com o *kernel*) e `tracing_on` (contém informações sobre a sessão atual de coleta de traços) (Rostedt, 2022).

A documentação sobre *ftrace* é desatualizada e escassa. Nas páginas oficiais da documentação do Android, indica-se que o diretório do *tracefs* se encontra em `/sys/kernel/tracing` (Google, 2022d). Entretanto, a documentação oficial do *ftrace* indica que o diretório padrão em versões do *kernel* anteriores à 4.1 se encontra em `/sys/kernel/debug/tracing` (Rostedt, 2022). Apesar de se utilizar a versão 4.14.175 do *kernel*, verifica-se que o diretório existente é o último apontado, o que demonstra o grau de desatualização da documentação da ferramenta no ambiente Android.

Ao tentar utilizar a ferramenta para coleta de traços do programa `ps`, percebeu-se que os testes com a ferramenta *ftrace* não podem ser concretizados no sistema emulado. Nota-se que, de acordo com o conteúdo do arquivo `available_tracers`, nenhum *tracer* é habilitado no

kernel padrão disponibilizado com a imagem do Android 10, conforme exibe a Listagem 3.2. O *tracer* `nop` é utilizado para desabilitar todas as formas de *tracing* (Rostedt, 2022), e é o único existente na plataforma.

Listing 3.2: Ausência de *tracers* habilitados na versão do *kernel* disponibilizado pelo Android Studio, indicado pelo conteúdo do arquivo `available_tracers`

```
1 generic_x86_64:/sys/kernel/debug/tracing # cat available_tracers
2 nop
```

Não foi possível identificar a razão pela qual *atrace* obtém êxito durante a captura de traços, dado que ele utiliza a ferramenta *ftrace*. Desta forma, restou prejudicada a apresentação de demais funcionalidades do *ftrace*, bem como sua utilização como ferramenta de coleta de traços.

3.3 *STRACE*

Strace é um utilitário do Linux utilizado para monitorar e interceptar interações entre processos e o *kernel*, o que inclui chamadas de sistema, envio de sinais e mudanças de estado (Strace, 2023a). A base do funcionamento do *strace* se encontra no *ptrace*, uma ferramenta também nativa do Linux que oferece meios de acompanhamento de *syscalls*, dados de memória e estado de registradores de outros processos (Abdalla, 2019).

O uso de *strace* é simples, baseado na estrutura `strace <programa>`. Neste caso, o programa monitorado é executado até o seu encerramento e as chamadas de sistema, bem como seus argumentos e valores de retorno, são listados na saída de erros padrão ou em um arquivo, se especificado. A utilização ocorre de forma idêntica em sistemas Linux *desktop*.

O comportamento do *strace* pode ser ajustado com a indicação de argumentos de execução, que podem mudar o funcionamento padrão dele ou alterar o formato da saída (Strace, 2023b). O programa a ser monitorado deve ser o último argumento do *strace*. Alguns argumentos que podem ser utilizados são:

- `-p` (*attach)*: Anexa *strace* ao processo com determinado PID e realiza o *tracing*. Pode-se interromper a coleta de traços a qualquer momento por meio das teclas `CTRL + C`. Nesse ponto, o *strace* se desatará do processo, que continuará executando normalmente. É possível fornecer mais de um PID para ser analisado, separando os valores inseridos após o argumento por vírgulas, espaços, tabs ou caracteres de nova linha.

- `-f` (*follow-forks*): Realiza também o *tracing* de processos filhos gerados em tempo de execução, na medida em que os forem criados. Se utilizado juntamente com o argumento `-p`, serão coletados os traços do processo com o PID designado, bem como de todas as *threads*, se este for *multi-threaded*.

- `-t` (*absolute-timestamps*): Inclui informações de tempo de relógio nos traços coletados. Admite-se a inclusão de até três argumentos `-t`, aumentando a precisão a cada inserção

sucessiva. O uso de `-t` acrescenta informações no formato `hh:mm:ss`; `-tt` adiciona milissegundos ao formato padrão e `-ttt` substitui o tempo de relógio pela *timestamp* do UNIX;

- `-T` (*syscall-times*): Exibe o tempo gasto em cada *syscall* no traço de saída, em microssegundos. O valor é mostrado ao final de cada linha de traço.

- `-o` (*output*): Grava os dados obtidos da coleta de traços no arquivo com o nome especificado ao invés de exibir na saída de erros padrão `stderr`.

A execução do *strace* sobre o programa `ps` gera um arquivo com mais de duas mil linhas contendo *syscalls*, obtido com o comando `strace -o /data/local/tmp/saida ps` em uma *shell* do aparelho emulado. Um recorte com o início e o final do conteúdo do arquivo é exibido na Listagem 3.3.

Listing 3.3: Recorte de traços obtidos pela ferramenta *strace* durante execução do programa `ps`

```
execve("/system/bin/ps", ["ps"], 0x7ffde43c1d40 /* 23 vars */) = 0
arch_prctl(ARCH_SET_FS, 0x7ffd02adfc40) = 0
getpid()                        = 4063
mmap(NULL, 12288, PROT_READ|PROT_WRITE,
   MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7b9ce484d000
set_tid_address(0x7b9ce499aee8) = 4063
faccessat(AT_FDCWD, "/dev/urandom", R_OK) = 0
futex(0x7b9ce4997058, FUTEX_WAKE_PRIVATE, 2147483647) = 0
...
...
...
openat(3, "4063/status", O_RDONLY) = 4
read(4, "Name:\tps\nUmask:\t0000\nState:\tR (r"..., 3607) = 1084
read(4, "", 2523)               = 0
close(4)                        = 0
sysinfo({uptime=6922, loads=[52192, 80864, 40032], totalram=2089172992,
   freeram=400179200, sharedram=2306048,
   bufferram=7823360, totalswap=1566875648,
   freeswap=1539350528, procs=767, totalhigh=0, freehigh=0, mem_unit=1}) = 0
openat(3, "4063/wchan", O_RDONLY) = 4
read(4, "0", 2066)              = 1
read(4, "", 2065)               = 0
close(4)                        = 0
openat(3, "4063/cmdline", O_RDONLY) = 4
read(4, "ps\0", 2830)           = 3
read(4, "", 2827)               = 0
close(4)                        = 0
write(1, "shell   4063 4061 14596"..., 67) = 67
getdents64(3, /* 0 entries */, 4200) = 0
close(3)                        = 0
```

```
31 exit_group(0)                    = ?
32 +++ exited with 0 +++
```

O arquivo de saída do *strace* é entregue em formato de texto simples, que pode ser lido pela maioria dos editores de texto, tal como ocorre com o *atrace*.

3.4 PERFETTO

Como indicado anteriormente, Perfetto é uma ferramenta de análise de desempenho de sistemas operacionais de código aberto, desenvolvida pela equipe do Google. Trata-se de uma ferramenta de coleta de dados de baixo nível, que permite a análise detalhada do comportamento do sistema, identificação de gargalos, problemas de latência e outras questões de desempenho. Com sua interface de linha de comando e módulos de visualização gráfica, é utilizado para depuração e otimização de sistemas, incluindo o Android, o Chrome OS, o Linux e outros sistemas operacionais baseados em Unix. É uma ferramenta de código aberto que pode ser estendida com plug-ins personalizados para coleta de dados específicos de domínio e análises personalizadas. Está disponível para o Android desde o lançamento da versão 9 do sistema operacional (Android P), mas sua habilitação por padrão só ocorreu a partir da versão 11 (Android R) (Perfetto, 2023a).

No âmbito da segurança, Perfetto pode ser usado para investigar *malware* por meio do monitoramento do sistema em tempo real. Ao capturar *syscalls*, a ferramenta pode detectar chamadas incomuns e suspeitas feitas pelo *malware* ao sistema operacional. Além disso, ao monitorar o uso da CPU, consegue detectar quando um *malware* está executando processos ou operações intensivas, o que pode ser um indicativo de atividade maliciosa.

A atividade de rede também pode ser monitorada com a Perfetto para detectar conexões maliciosas ou envio de dados suspeitos para servidores remotos. Essa análise pode ajudar a identificar os servidores de comando e controle usados pelo *malware*, tornando-se útil para desativar a ameaça.

É possível ainda usar o *Perfetto* para analisar o comportamento de arquivos suspeitos, detectando quando um *malware* está criando ou manipulando documentos no sistema de arquivos. Com tais informações, os desenvolvedores conseguem identificar a extensão e a gravidade da infecção do *malware* e tomar medidas para neutralizá-lo.

A ferramenta possui uma interface gráfica disponível em `https://ui.perfetto.dev/`, demonstrada na Figura 3.1. Por meio dela é possível conectar um aparelho celular e iniciar a captura de traços. Diversas opções de captura podem ser configuradas, como tamanho de *buffer*, duração da captura, detalhes do escalonamento de tarefas pelo processador, frequência da CPU, *syscalls*, uso de memória, frequência de operação de GPU e consumo de energia, entre outras (Perfetto, 2023c). Ele também permite a configuração dos traços a serem capturados por meio de comandos em linha de comando, de modo que as opções selecionadas pela UI são convertidas em instruções e utilizadas diretamente por uma *shell* via

`adb`. A interface gráfica da Perfetto admite da mesma forma a análise de traços pelo upload de arquivos nos formatos `Perfetto protobuf trace`, `chrome://tracing JSON`, `systrace`, `Fuchsia trace` e `Ninja Build log` (Perfetto, 2023d).

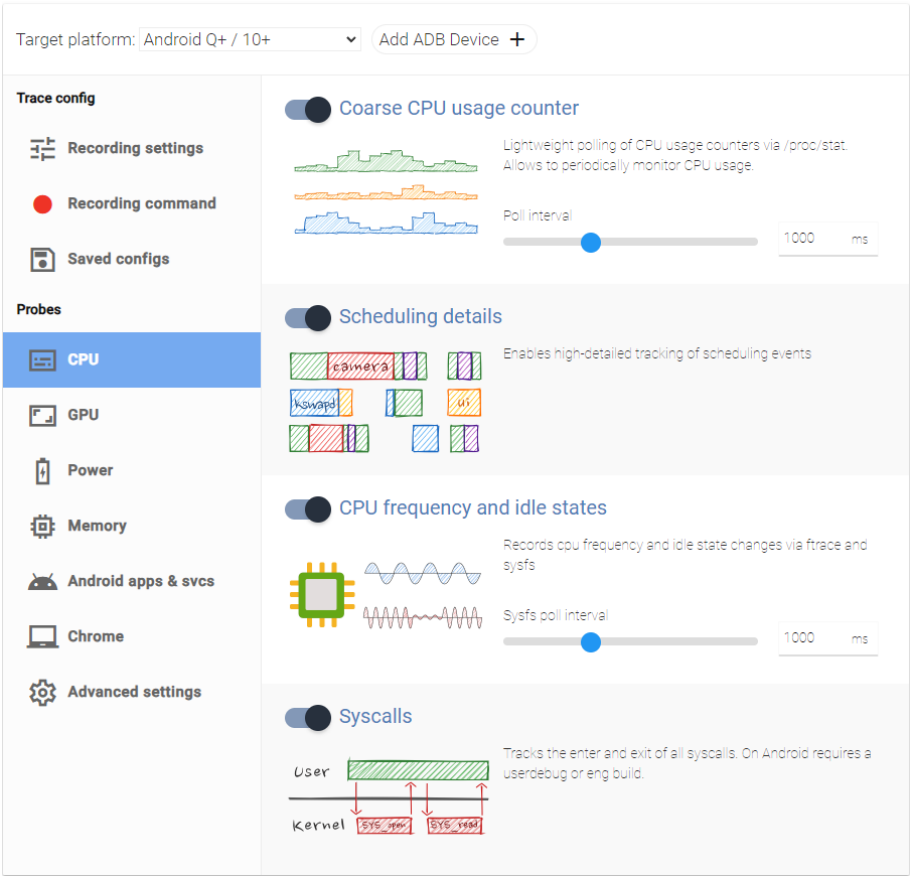

Figura 3.1: Opções de traços disponíveis para registro pela UI do Perfetto

A ferramenta também pode ser implicitamente invocada a partir do aparelho celular. Com as opções de desenvolvedor ativadas (Google, 2023d), opções de captura de traços se tornam disponíveis, como mostra a Figura 3.2. Os traços coletados são armazenados no diretório `/data/misc/perfetto-traces/` do aparelho e podem ser inseridos na interface gráfica do *Perfetto* para análise posterior. Os resultados da captura são ordenados de acordo com as *threads* dos processos e configurações de observação, organizados ao longo do tempo.

Uma característica desta ferramenta é a observação do sistema como um todo. Diferentemente do que ocorre com *strace*, todos os processos do sistema são instrumentados. A filtragem por PID deve ocorrer em uma fase de pós-processamento.

A ferramenta Perfetto possui limitações, como a impossibilidade de capturar os argumentos de syscalls. Tal limitação se dá para evitar sobrecarga no sistema em razão do tamanho do arquivo de *trace* (Perfetto, 2023b).

O uso da ferramenta em si é facilitado e devidamente documentado. Opções de inicialização pela interface web ou pelas configurações avançadas do aparelho celular estão

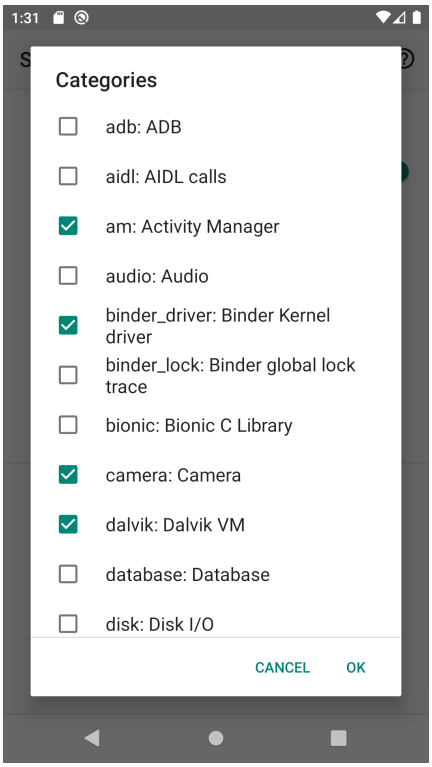

Figura 3.2: Interface para captura de traços no aparelho Android

disponíveis, bem como scripts auxiliares para captura de traços via linha de comando de `adb` (Google, 2023h).

Para fins de teste da ferramenta, foi habilitada a captura de todos os traços disponíveis, incluindo *syscalls*, memória e informações do escalonador de tarefas. O comando utilizado para iniciar tal captura pela linha via `adb` encontra-se no apêndice B. É necessário destacar que a inclusão de traços de frequência de CPU pela interface gráfica do Perfetto gerou um comando inválido para execução via `adb`. Traços sobre uso de rede e *heap* de aplicações Java também foram desconsiderados por exigirem versões do Android posteriores à 10.

O registro dos traços ocorreu pelo período de 10 segundos, ao longo do qual todo o sistema foi observado e o comando `ps` foi executado via `adb shell`. Como resultado, um arquivo de traço de aproximadamente 3,3 MB foi gerado. Na plataforma de visualização web do *Perfetto*, é possível encontrar traços relativos ao comando `ps`, bem como todas as demais informações do sistema. Um recorte dos traços obtidos é exibido na Figura 3.3.

A grande quantidade de dados fornecida pelas ferramentas nativas do Android é capaz de demonstrar o comportamento de um processo no nível do sistema. O próximo capítulo trará os experimentos de utilização de tais ferramentas em contextos de CVEs reais, a fim de verificar a utilidade delas como instrumentos de detecção de ataques em sistemas Android. Um sumário das características de cada ferramenta é apresentado na Tabela 3.1. A ferramenta `ftrace` não pôde ser avaliada por indisponibilidade no *kernel* utilizado para testes.

Ferramenta	Formato da saída	Escopo da coleta	Documentação	Observações
`atrace`	Texto simples	Traços em nível de sistema em geral	Escassa	Em processo de descontinuação em favor do `Perfetto`. A maioria das informações apresentas faz referencia ao escalonador de tarefas. Não captura *syscalls*
`strace`	Texto simples	Traços em nível do programa monitorado	Abundante	Uso facilitado, semelhante ao uso em sistemas Linux *desktop*. Não observa ações que ocorrem foram do contexto do programa monitorado, como aquelas do escalonador de tarefas.
`Perfetto`	Formato próprio para upload na interface web	Traços em nível de sistema em geral	Abundante	Oferece visualização gráfica dos traços coletados. A ausência de um formato de saída em modo textual prejudica a análise de traços. As *syscalls* são capturadas sem informações sobre os argumentos utilizados.

Tabela 3.1: Tabela de características de cada ferramenta de *tracing* disponível no Android

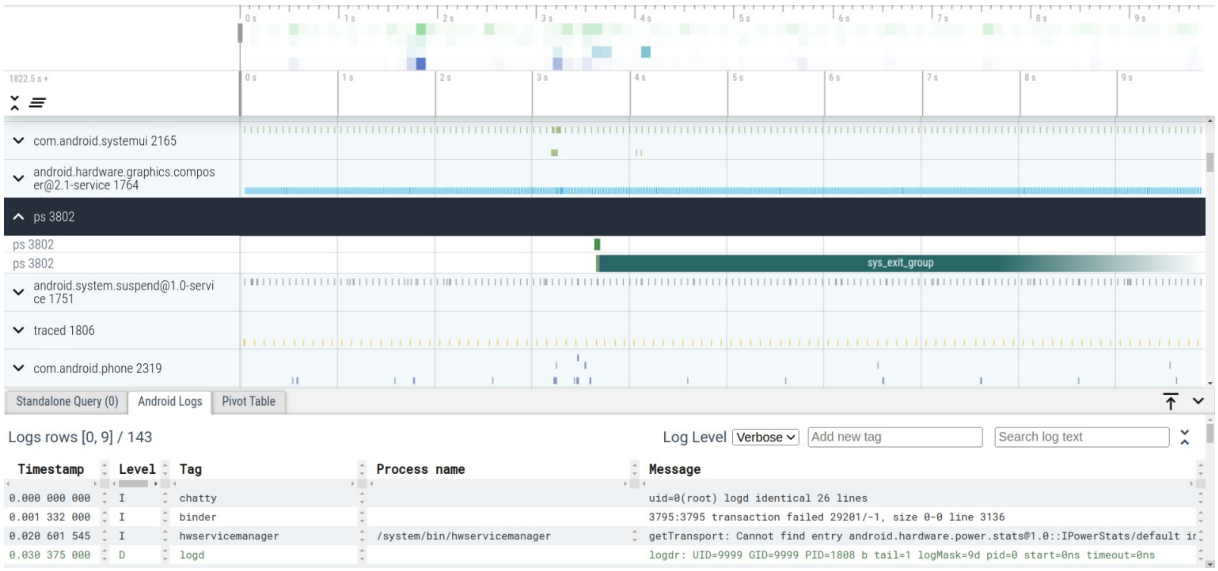

Figura 3.3: Recorte de traços obtidos no *Perfetto* para execução do programa *ps*

4 EXPERIMENTOS E RESULTADOS

Os experimentos com as ferramentas de *tracing* nativas do Android e os resultados obtidos são expostos neste capítulo. A coleta de traços a partir da reprodução de ataques reais em ambientes controlados permite que os *tracers* sejam testados como instrumentos de detecção, dado o conjunto de informações que cada um é capaz de fornecer sobre o sistema e os processos em execução.

4.1 SELEÇÃO DE CVES

As vulnerabilidades que permitem a execução de código arbitrário representam a maior parte das ameaças para o Android. Portanto, as seguintes CVEs foram selecionadas para análise:

- *CVE-2019-2215*: Trata-se de uma vulnerabilidade do tipo *Use-After-Free* (em que a memória é erroneamente usada de novo após ser liberada para o sistema) no *Binder* (mecanismo de comunicação inter-processos) do Android. O objetivo da vulnerabilidade é obter a escalada de privilégio através do vazamento do endereço de uma estrutura interna do *Binder*, tornando a memória reservada ao kernel acessível para processos sem privilégio de superusuário (Stone, 2019).
- *CVE-2019-13272*:Uma falha no *kernel* do Linux anterior à versão 5.1.17 manipula incorretamente a gravação das credenciais de um processo que deseja criar um relacionamento *ptrace*, permitindo que usuários locais obtenham privilégios de superusuário por meio do abuso de certos cenários de relacionamento de processos pai-filho (CVE, 2019).
- *CVE-2021-0920*: Um *Use-After-Free* pode ocorrer devido a uma condição de corrida, levando à escalada local de privilégios. A interação do usuário não é necessária para exploração dessa vulnerabilidade (CVE, 2021).

Tais vulnerabilidades contam com amplos estudos por pesquisadores em todo o mundo, de forma que as *provas de conceito* (*POC*, uma implementação que busca validar a existência de uma vulnerabilidade de segurança (Kaspersky, 2023)) de cada uma delas podem ser encontradas na internet. O objetivo final de todas essas vulnerabilidades é obter privilégios de superusuário, inclusive através da colaboração entre elas.

4.2 INFRAESTRUTURA E MÉTODO DE ANÁLISE

O computador utilizado para realização do testes é um desktop Intel Core i7 com 16 GB RAM, equipado com Kali Linux 2022.4. As versões 2022.1.1 do Android Studio, 31.3.15

do Android Emulator e 33.0.3 do Android SDK Platform-tools (que inclui o utilitário `adb`) também foram usadas. A versão 4.14 do *kernel* foi utilizada, compilada de acordo com cada vulnerabilidade (Ansari, 2022).

Os experimentos foram executados sobre o Android Emulator, utilitário da Google disponibilizado com o Android Studio que permite a criação de ambientes virtuais para teste e execução de aplicativos em um computador, simulando o hardware e o software de um dispositivo real. Novamente o Android 10 foi escolhido como versão de testes, pois é a versão vulnerável às CVEs apontadas.

Os traços de cada experimento foram obtidos a partir da instrumentação das *POCs* com as ferramentas *strace*, *atrace* e *Perfetto*. A ferramenta *ftrace* não pôde ser testada, de acordo com a justificativa apresentada no capítulo anterior. O seguinte fluxo, ilustrado na Figura 4.1, foi seguido para extração dos dados:

1. A *POC* é compilada na máquina *host* (`gcc`);
2. O aparelho virtual é emulado de acordo com o *kernel* especificado (`emulator`);
3. O arquivo executável é enviado ao aparelho emulado via `adb push` para o diretório `/data/local/tmp/`;
4. O dispositivo emulado é acessado via `adb shell`;
5. A *POC* é executada e permanece em um estado de espera pela inserção de um caracter qualquer a partir da entrada padrão;
6. O PID do processo é encontrado (`ps | grep`);
7. Cada intrumento para captura de traços é então utilizado:
   - Para *strace*, o comando utilizado é `strace -p <PID> -f -T -ttt -o /data/local/tmp/strace_output` dentro da `adb shell`;
   - Para *atrace*, o comando utilizado é `atrace -b 96000 -t 10 -o /data/local/tmp/atrace_output`, acompanhado de todas as categorias disponíveis;
   - Para *Perfetto*, instruções contidas em um arquivo `config.pbtx` são enviadas a partir da máquina *host* (`cat config.pbtx | adb shell perfetto`);
8. A execução da *POC* prossegue;
9. Depois de executado, a coleta de traços é interrompida e os arquivos com os resultados são recuperados do aparelho emulado via `adb pull`;
10. Os traços são analisados na máquina *host*;

Ainda que a execução de várias instâncias de uma mesma *POC* seja capaz de entregar traços diferentes, o núcleo da ação potencialmente maliciosa é verificado em todos os resultados obtidos. Dessa forma, apenas um registro de traço de execução precisa ser analisado.

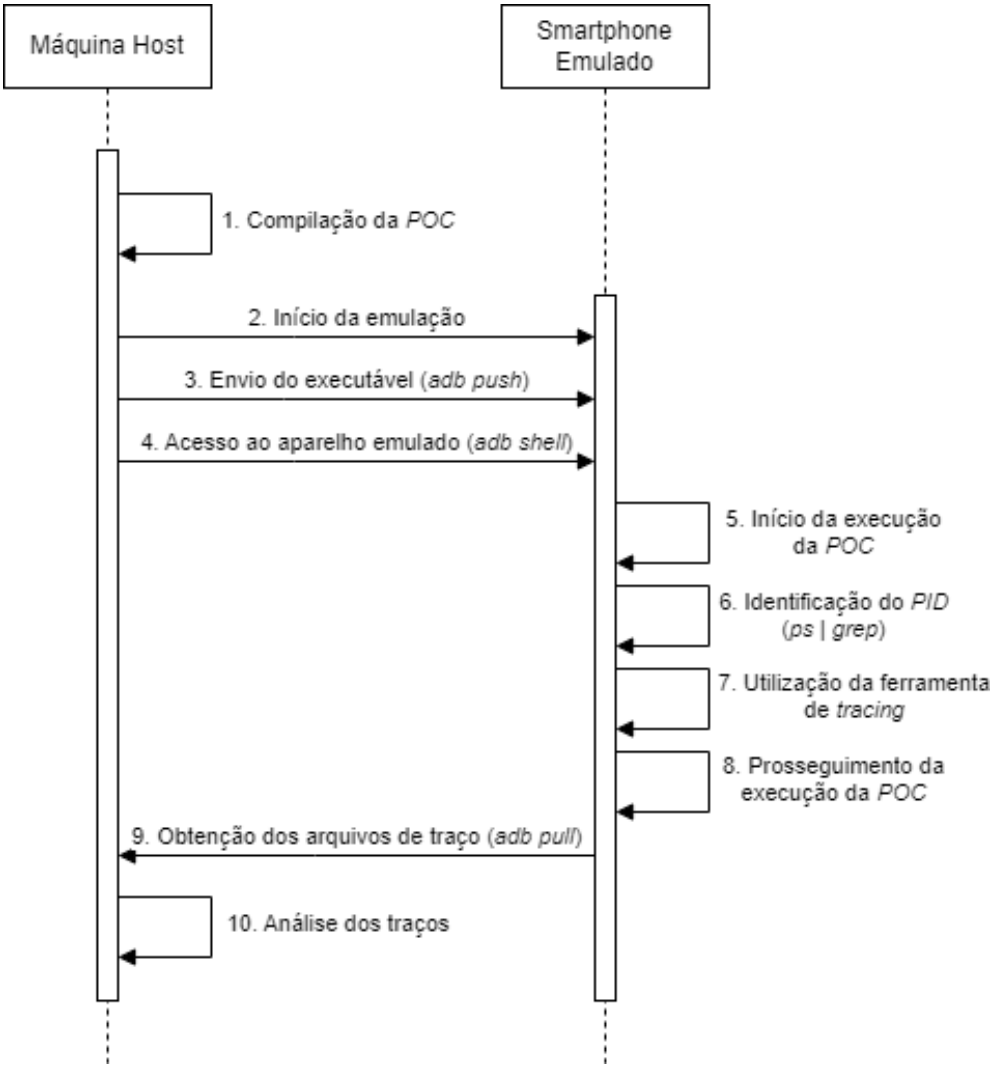

Figura 4.1: Passos para coleta de traços

A análise dos resultados se deu por meio da correlação de blocos de traços coletados e linhas de código fonte consideradas críticas nas *POCs*. Cada ferramenta produz dados que podem ser relacionados para compreensão do comportamento do programa malicioso no sistema emulado.

Dessa forma, a relevância das informações dadas pelas ferramentas no contexto dos ataques também deve ser verificada para fins de análise de viabilidade de uso de cada ferramenta como instrumento de detecção.

4.3 RESULTADOS OBTIDOS E CARACTERÍSTICAS DOS ATAQUES ENCONTRADOS NOS TRAÇOS

Cada CVE estudada apresenta um comportamento distinto e cada ferramenta de coleta de traços entrega resultados complementares e/ou concorrentes entre si. A relação entre linhas do código fonte das *POCs* e registros nos traços encontrados permite que o comportamento de aplicações maliciosas seja identificado. Ressalta-se que apenas as linhas de código consideradas críticas para a execução do ataque foram buscadas dentro dos traços. Os resultados dos experimentos estão relatados a seguir, por CVE e ferramenta.

4.3.1 CVE-2019-2215

Um *exploit* para a CVE-2019-2215, capaz de fornecer uma *root shell* no aparelho emulado sem autorização do usuário, é fornecido em (Ansari, 2022). A Listagem 4.1 indica a parte crítica do *exploit*, que se concentra nas instruções responsáveis pela liberação, corrupção e leitura de memória.

Listing 4.1: Recorte das funções críticas da *POC* da CVE-2019-2215

```
/**
 * Free the binder thread structure
 */
void BinderUaF::freeBinderThread() {
   INFO("[+] Freeing binder_thread\n");

   ioctl(m_binder_fd, BINDER_THREAD_EXIT, NULL);
}

//...

//
   // Reallocate binder_thread as iovec array
   //
   // We need to make sure this writev call blocks
   // This will only happen when the pipe is already full
   //

   ssize_t nBytesWritten = writev(pipe_fd[1], iovecStack, IOVEC_COUNT);

   //
   // If the corruption was successful, the total bytes written
   // should be equal to 0x2000. This is because there are two
   // valid iovec and the length of each is 0x1000
   //

//...
```

```
28
29 //
30       // child process
31       //
32
33       //
34       // There is a race window between the unlink and blocking
35       // in writev, so sleep for a while to ensure that we are
36       // blocking in writev before the unlink happens
37       //
38
39       sleep(2);
40
41       //
42       // Trigger the unlink operation on the reallocated chunk
43       //
44
45       unlinkEventPollWaitQueueFromBinderThreadWaitQueue();
46       //
47
48 \\...
49
50    // Now read the actual data from the corrupted iovec
51    // This is the leaked data from kernel address space
52    // and will contain the task_struct pointer
53    //
54
55    nBytesRead = read(pipe_fd[0], dataBuffer, sizeof(dataBuffer));
56
57 \\...
```

No código fonte, um processo pai libera uma estrutura previamente alocado do *Binder* em `freeBinderThread()` e inicia um longo procedimento de escrita por meio de `writev(pipe_fd[1], iovecStack, IOVEC_COUNT)`. No momento em que a tarefa de escrita é interrompida pelo sistema operacional para realização de troca de contextos, o processo filho libera a estrutura `epoll` utilizada na operação de escrita em `unlinkEventPollWaitQueueFromBinderThreadWaitQueue()`. Assim, o endereço de memória apontado por *epoll* passa a conter dados corrompidos.

A coleta de traços da execução do *exploit* por meio do *atrace* produziu um arquivo com 7,8 MB e 65.269 entradas de dados. Apenas 102 linhas mencionam o texto `"cve-2019-2215"`, das quais 69 fazem referência às tarefas do escalonador. Palavras chaves existentes no código fonte, como `"iovec"`, `"task struct"`, `"free"` e `"leak"` não foram encontradas nos registros.

Entretanto, um registro aparentou indicar o propósito do ataque executado pelo *exploit*: o último registro com o termo `"cve-2019-2215"` indica que uma operação de `rename` foi executada, alterando o nome da tarefa para `sh` (indicativo de *shell*), conforme exibido na Listagem 4.2. Em um momento posterior, uma nova operação de `task_rename` foi executada, mantendo o nome `sh`.

Listing 4.2: Anotação da operação *task_rename* pela ferramenta *atrace*, referente à CVE-2019-2215

```
...
adbd-1898 ( 1898) [001] d..2 183.836582: sched_switch: prev_comm=adbd
                         prev_pid=1898 prev_prio=120
                         prev_state=S ==> next_comm=adbd
                         next_pid=2057 next_prio=120
adbd-2057 ( 1898) [001] d..2 183.836852: sched_switch: prev_comm=adbd
                         prev_pid=2057 prev_prio=120
                         prev_state=S ==> next_comm=swapper/1
                         next_pid=0 next_prio=120
<idle>-0 (-----) [001] d..1 183.836890: cpu_idle: state=1 cpu_id=1

\\operação de rename
sh-3927 ( 3927) [000] ...1 183.836909: task_rename: pid=3927
                         oldcomm=cve-2019-2215-e newcomm=sh
                         oom_score_adj=-1000

<idle>-0  (-----) [001] d.h1 183.837552: irq_handler_entry: irq=18
                         name=goldfish_pipe
<idle>-0  (-----) [001] d.h1 183.837608: softirq_raise: vec=6
                          [action=TASKLET]
\\...
DispSync-1948 ( 1873) [001] d..2 183.863312: sched_switch:
                           prev_comm=DispSync prev_pid=1948
                           prev_prio=97 prev_state=S ==>
                           next_comm=swapper/1
                           next_pid=0 next_prio=120
<idle>-0  (-----) [001] d..1 183.863371: cpu_idle: state=1 cpu_id=1
sh-3927 ( 3927) [000] ...1 183.863723: task_rename: pid=3927
                        oldcomm=sh newcomm=sh oom_score_adj=-1000
<idle>-0  (-----) [001] d.h1 183.863908: irq_handler_entry:
                           irq=18 name=goldfish_pipe
<idle>-0  (-----) [001] d.h1 183.863974: softirq_raise: vec=6
                           [action=TASKLET]
...
```

Considerando que o objetivo final do *exploit* é obter uma *root shell*, é possível deduzir que a primeira operação de `trace_rename` foi responsável pela criação de uma nova sessão *shell* e a segunda operação responsável pela entrega de um terminal com privilégios de superusuário.

Nenhum outro registro no traço do *atrace* aparenta possuir relevância na detecção do ataque, além da informação sobre `task_rename`. Assim, vê-se que o *atrace* possui o potencial para entrega de informações relevantes, embora a falta de detalhes sobre as operações e a ausência de informações sobre *syscalls* reduzam sua utilidade.

Já o *strace* produziu um arquivo de 148,4 KiB, com 1402 linhas de registros de *syscalls*. De acordo com as impressões do *exploit* encontradas ao longo traço, a conclusão da ação maliciosa na linha número 15 da Listagem 4.3, com a invocação de uma *shell*.

Listing 4.3: Registro da invocação de uma *root shell* capturados pela ferramenta *strace*, referente à CVE-2019-2215

```
1  ...
2  3790 write(1, "[+] Verifying if rooted\n", 24) = 24 <0.000480>
3
4  \\ getuid() retorna 0, indicando que o processo possui
5  \\ privilégios de superusuário
6  3790 getuid()    = 0 <0.000559>
7
8  3790 write(1, "\t[*] uid: 0x0\n", 14) = 14 <0.000513>
9  3790 write(1, "\t[*] Rooting successful\n", 24) = 24 <0.000960>
10 3790 write(1, "[+] Spawning root shell\n", 24) = 24 <0.000475>
11 3790 rt_sigprocmask(SIG_BLOCK, [CHLD], [RTMIN], 8) = 0 <0.000361>
12 3790 rt_sigaction(SIGINT, {sa_handler=SIG_IGN, sa_mask=[],
13      sa_flags=SA_RESTORER, sa_restorer=0x252fa0},
14      {sa_handler=SIG_DFL, sa_mask=[], sa_flags=0}, 8) = 0 <0.000389>
15 3790 rt_sigaction(SIGQUIT, {sa_handler=SIG_IGN, sa_mask=[],
16      sa_flags=SA_RESTORER, sa_restorer=0x252fa0},
17      {sa_handler=SIG_DFL, sa_mask=[], sa_flags=0}, 8) = 0 <0.000273>
18 3790 gettid()    = 3790 <0.000235>
19 ...
20 3798 rt_sigprocmask(SIG_SETMASK, [RTMIN], NULL, 8) = 0 <0.000064>
21
22 \\abertura de uma nova shell
23 3798 execve("/system/bin/sh",
24      ["sh", "-c", "/bin/sh"], 0x7ffe491360c8 /* 23 vars */) = 0 <0.000659>
25 3798 arch_prctl(ARCH_SET_FS, 0x7ffce93260d0) = 0 <0.000039>
26 ...
```

Os traços coletados pelo *strace* permitem ainda identificar o momento em que a operação responsável pela corrupção de memória é executada. Como relatado anteriormente, uma troca de contexto durante o procedimento de escrita possibilita que um processo filho libere memória da estrutura `epoll`, conforme indica a Listagem 4.4.

Listing 4.4: *Syscalls* capturadas pela ferramenta *strace*, referentes à corrupção de memória executada pela *POC* da CVE-2019-2215

```
1  3790 write(1, "[+] Linking eppoll_entry->wait.e"..., 65) = 65 <0.000072>
2
3  \\ adição da struct epoll
```

```
3790 epoll_ctl(4, EPOLL_CTL_ADD, 3, {EPOLLIN,
     {u32=0, u64=0}}) = 0 <0.000055>

3790 clone(child_stack=NULL,
     flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD,
     child_tidptr=0x2e34e8) = 3796 <0.000298>
3790 write(1, "[+] Freeing binder_thread\n", 26 <unfinished ...>
3796 prctl(PR_SET_VMA, PR_SET_VMA_ANON_NAME, 0x76251cec6000, 12288,
     "stack_and_tls:main" <unfinished ...>
3790 <... write resumed> ) = 26 <0.000135>
3796 <... prctl resumed> ) = 0 <0.000059>

\\liberação da estrutura do binder
3790 ioctl(3, BINDER_THREAD_EXIT, 0) = 0 <0.000054>

\\inicio da operação de escrita
3790 writev(6, [{iov_base=NULL, iov_len=0}, {iov_base=NULL, iov_len=0},
     {iov_base=NULL, iov_len=0}, {iov_base=NULL, iov_len=0},
     {iov_base=NULL, iov_len=0}, {iov_base=NULL, iov_len=0},
     {iov_base=NULL, iov_len=0}, {iov_base=NULL, iov_len=0},
     {iov_base=NULL, iov_len=0}, {iov_base=NULL, iov_len=0},
     iov_base="\0\0\0\0\0\0\0\0\0\0\"..., iov_len=4096},
     {iov_base=0x41414141, iov_len=4096},
     {iov_base=NULL, iov_len=0}, {iov_base=NULL, iov_len=0},
     {iov_base=NULL, iov_len=0}, {iov_base=NULL, iov_len=0},
     {iov_base=NULL, iov_len=0}, {iov_base=NULL, iov_len=0},
     {iov_base=NULL, iov_len=0}, {iov_base=NULL, iov_len=0},
     {iov_base=NULL, iov_len=0}, {iov_base=NULL, iov_len=0},
     {iov_base=NULL, iov_len=0}, {iov_base=NULL, iov_len=0},
     {iov_base=NULL, iov_len=0}], 25 <unfinished ...>

\\interrupção da operação de escrita com troca de contexto
3796 nanosleep({tv_sec=2, tv_nsec=0}, 0x7ffe49135d50) = 0 <2.000540>
3796 write(1, "[+] Un-linking eppoll_entry->wai"..., 70) = 70 <0.000590>

\\liberação do ponteiro epoll pelo processo filho
3796 epoll_ctl(4, EPOLL_CTL_DEL, 3, 0x76251ca0a040) = 0 <0.000490>

\\leitura da memória corrompida
3796 read(5, "\0\0\0\0\0\0\0\0\0\0\"..., 4096) = 4096 <0.000541>

\\retomada do procedimento de escrita
3790 <... writev resumed> ) = 8192 <2.003891>
```

No ataque sob estudo, o *strace* foi capaz de registrar uma operação de escrita não usual, indicativa de ação maliciosa. Além disso, observa-se que o programa buscou a verificação das

permissões atuais do processo por meio de `getuid`, o que sugere tempo hábil para prevenção da efetivação do ataque, que ocorre somente na chamada de `execve` para abertura de nova *shell*.

Nesse sentido, nota-se que *strace* foi capaz de fornecer informações relevantes e úteis para detecção de um ataque que abusa da CVE-2019-2215.

Em relação à ferramenta *Perfetto*, os dados entregues foram armazenados em um arquivo com 3,2 MB. Após upload para o mecanismo de visualização por meio da interface gráfica, nota-se que nenhum processo com nome `"cve-2019-2215-exploit"` consta na lista dos resultados observados. Dados sobre esse processo são encontrados apenas nas informações do escalonador de tarefas, como se vê na Figura 4.2.

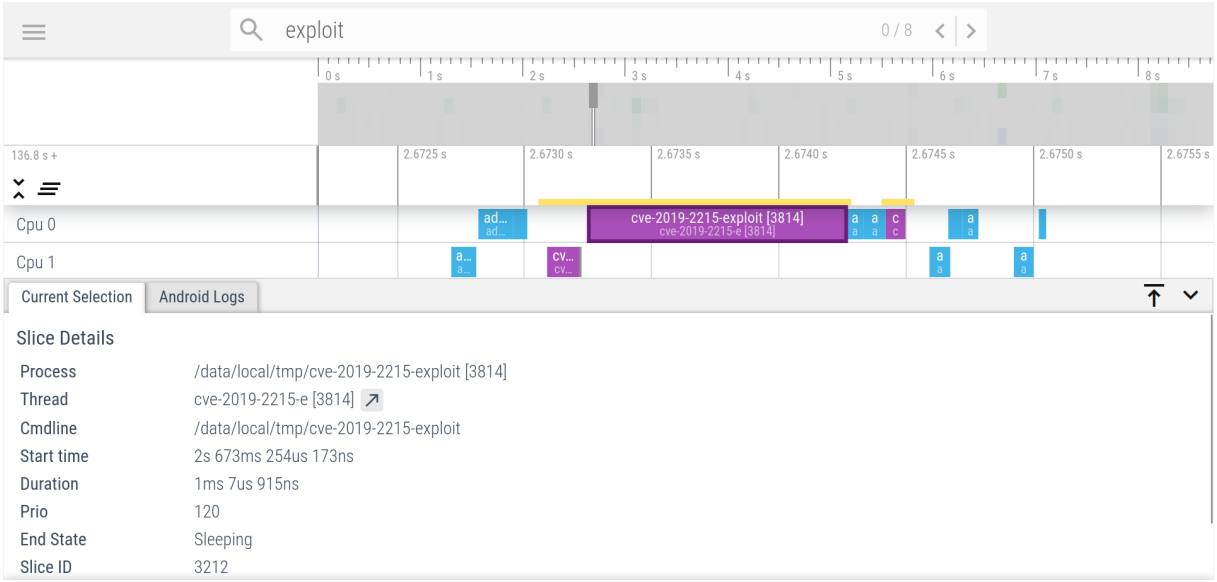

Figura 4.2: Informações sobre CVE-2019-2215 no traço *Perfetto*

Deste modo, como dados exatos sobre o processo buscado não puderam ser encontrados, resta prejudicada a análise da ferramenta *Perfetto* neste momento.

4.3.2 CVE-2019-13272

A natureza da vulnerabilidade explorada pela CVE-2019-13272 difere das demais pois depende da execução de programas com SUID (*Set owner User ID*) para elevação de seus privilégios (Horn, 2019). A *POC* escolhida para testes utiliza a função `ptrace` para atingir tal objetivo, conforme exibe a Listagem 4.5:

Listing 4.5: Código-fonte da *POC* referente à CVE-2019-13272

```
1 #define _GNU_SOURCE
2 #include <unistd.h>
3 #include <signal.h>
4 #include <sched.h>
5 #include <err.h>
6 #include <sys/prctl.h>
7 #include <sys/types.h>
```

```
8  #include <sys/ptrace.h>
9  #include <stdio.h>
10
11 int grandchild_fn(void *dummy)
12 {
13    if (ptrace(PTRACE_TRACEME, 0, NULL, NULL))
14       err(1, "traceme");
15    return 0;
16 }
17
18 int main(void)
19 {
20    int aux;
21    scanf("%d", &aux);
22
23    pid_t child = fork();
24    if (child == -1)
25       err(1, "fork");
26
27    /* child */
28    if (child == 0)
29    {
30       static char child_stack[0x100000];
31       prctl(PR_SET_PDEATHSIG, SIGKILL);
32       while (1)
33       {
34          if (clone(grandchild_fn, child_stack + sizeof(child_stack),
35          CLONE_FILES | CLONE_FS | CLONE_IO | CLONE_PARENT | CLONE_VM |
36          CLONE_SIGHAND | CLONE_SYSVSEM | CLONE_VFORK, NULL) == -1)
37             err(1, "clone failed");
38       }
39    }
40
41    /* parent */
42    uid_t uid = getuid();
43    while (1)
44    {
45       if (setresuid(uid, uid, uid))
46          err(1, "setresuid");
47    }
48 }
```

Tal condição para êxito do ataque implica em dificuldades para coleta de traços. A execução simples do programa via `adb` resulta em falha crítica do sistema, forçando a reinicialização do aparelho emulado, conforme se vê nos *logs* obtidos no terminal de emulação, indicados na Listagem 4.6.

Listing 4.6: *Kernel Panic*, referente à execução da POC para CVE-2019-13272

```
[ 302.689086] Kernel panic - not syncing: CRED: put_cred_rcu()
              sees 00000000dc90934d with usage 1
[ 302.689086]
[ 302.690712] CPU: 2 PID: 6857 Comm: poc13272 Not tainted 4.14.150+ #21
...
[ 302.733594] Kernel Offset: 0x36e00000 from 0xffffffff80200000
              (relocation range: 0xffffffff80000000-0xffffffffbfffffff)
[ 302.752762] Rebooting in 5 seconds..
```

A obtenção de traços com *atrace* e *Perfetto* (que monitoram o sistema como um todo e não apenas um processo) foi prejudicada pela reinicialização súbita do sistema, a qual impede a recuperação dos arquivos de traço que estavam sendo gravados.

A ferramenta *strace* também não teve êxito na coleta de traços. A situação em que a *POC* causa *kernel panic* não ocorre quando o programa é monitorado pela ferramenta. Entretanto, mensagens de erro com o texto `poc13272: traceme: Operation not permitted` são exibidas na saída padrão, indicando que a vulnerabilidade sequer chega a ser ativada. A vulnerabilidade deixa de ser ativada pois, ao executar a *POC* com monitoramento do *strace*, configurações de SUID passam a ser ignoradas (Spraul, 2011).

Deste modo, vê-se que as ferramentas de *tracing* nativas do Android não foram capazes de coletar traços e permitir a análise de ataques que abusam da CVE-13272.

4.3.3 CVE-2021-0920

A CVE-2021-0920 abusa de uma vulnerabilidade ativada por uma condição de corrida no *kernel* do Linux, oriunda de uma falha do tipo *Use-After-Free*. Nesse ponto, entende-se que a parte crítica do código-fonte se encontra nas linhas que caracterizam o uso depois da liberação da memória, feita pelo *garbage collector* do *kernel*. A Listagem 4.7 mostra as linhas de código críticas supracitadas.

Listing 4.7: Linhas críticas do *exploit* referente à vulnerabilidade CVE-2021-0920, ilustrando o potencial uso de memória liberada

```
__int64 t_recvmsg()
{
   // ...

   pin_cpu(2LL);
   setpriority(PRIO_PROCESS, 0, -19);
   pthread_mutex_lock(&condition_mutex_recvmsg);
   pthread_cond_wait(&condition_cond_recvmsg, &condition_mutex_recvmsg);
   pthread_mutex_unlock(&condition_mutex_recvmsg);
   usleep(0xFAu);

   v0 = recvmsg(sv[1], &v11, 0x80042);
   v14 = v0 < 0;
```

```
14      fd = received_fds_ptr[4];
15      v1 = recvmsg(fd, &v11, 0x80042); // data
16      v14 = v1 < 0;
17      g_victim_sock_fd = received_fds_ptr[4];
18
19      // ...
20
21      close(fd);
22      close(v16);
23      close(sv[1]);
24      pthread_exit(0LL);
25      return 0LL;
26  }
```

Nesta função, a função `recvmsg()` é chamada duas vezes. Primeiro, ela é chamada com `sv[1]` como descritor de *socket* e a mensagem recebida é armazenada em `v11`. Em seguida, ele é chamado novamente com `fd` (que é recuperado de `receive_fds_ptr[4]`) como o descritor de *socket* e a mensagem recebida é armazenada em `v11` novamente. Entretanto, o valor de `fd` é obtido a partir de espaço de memória potencialmente liberado (`received_fds_ptr[4]`). Se a memória apontada por `receive_fds_ptr` foi liberada ou modificada simultaneamente devido à condição de corrida, o acesso a `received_fds_ptr[4]` pode resultar no uso de memória liberada.

A ferramenta *atrace* produziu um arquivo de aproximadamente 24,5 MB, contendo 208.745 linhas de informações. Dessas, 22137 linhas contém o texto `"poc0920"`, das quais 13.882 apresentam informações sobre o escalonador de tarefas e dados sobre *wakeup, switch* e *exit* de *tasks*. Das 8255 linhas que possuem o texto `"poc0920"` mas não mencionam o escalonador de tarefas, apenas 330 linhas também não informam sobre *softirqs* (mecanismos de interrupção do sistema). Nessas últimas 330 linhas, apenas informações sobre protocolos de rede e fila de tarefas são exibidas. Algumas chamadas ao *garbage collector* são encontradas, mas sua concentração pode ser considerada insuficiente para que alguma suspeita sobre o programa em execução seja levantada.

Dessa forma, não se observa, no traço entregue pela ferramenta *atrace*, um conjunto de informações relevantes para detecção do ataque executado pela *POC*. Considerando que a grande parte das pesquisas na área de análise dinâmica pelo estudo de *syscalls* e que o *atrace* não exibe dados relevantes neste sentido, não se vislumbra a utilização dela (por si só) como uma ferramenta de detecção de ataques.

A ferramenta *strace*, por sua vez, entregou um arquivo de aproximadamente 4,8 MB, contendo 39.978 linhas de informações sobre *syscalls* executadas durante o período de observação. Diferentemente do que ocorre com *atrace*, apenas o processo da *POC* e seus filhos são observados pelo *strace*.

O traço entregue pela ferramenta exibe com certa clareza cada bloco de execução das funções da *POC*. Por exemplo, verifica-se as 20 impressões da string `"wut?"` realizadas pela função `trigger()`, que é invocada 20 vezes na função `main()`. De modo semelhante, nota-se que um conjunto de 6 linhas de *syscalls* relacionadas à criação de *sockets* se repete 100 vezes a cada chamada da função `setup_race()`, conforme demonstra a Listagem 4.8.

Listing 4.8: Correlação de linhas de código do *exploit* com *syscalls* obtidas por meio de *strace*

```
\\ código malicioso

   socketpair(1, 1, 0, &v5[2 * i + 510]);
   for (int k = 0; k <= 4; ++k)
      v5[5 * i + 10 + k] = socket(1, 1, 0);

\\ syscalls correspondentes

socketpair(AF_UNIX, SOCK_STREAM, 0, [26, 27]) = 0 <0.000863>
socket(AF_UNIX, SOCK_STREAM, 0) = 28 <0.000788>
socket(AF_UNIX, SOCK_STREAM, 0) = 29 <0.000782>
socket(AF_UNIX, SOCK_STREAM, 0) = 30 <0.000762>
socket(AF_UNIX, SOCK_STREAM, 0) = 31 <0.000760>
socket(AF_UNIX, SOCK_STREAM, 0) = 32 <0.000800>
```

Em alto nível, afirma-se que todo o conteúdo do código fonte pode ser encontrado a partir da dedução e análise da sequência de invocação de *syscalls*.

De modo geral, a estrutura do traço entregue pelo *strace* (organizado como uma linha do tempo) favorece a análise do ataque. Visto que se trata da criação de uma condição corrida para obtenção de privilégios de superusuário a partir da manipulação de *sockets*, a presença de um grande número de linhas com o texto `"socket"` (presente em 12.101 das 39.198 linhas do arquivo) denuncia a ação maliciosa explorada na CVE-2021-0920.

É notável também a presença de uma única linha indicando a chamada da *syscall* `LSEEK`. Tal comando teve sua execução malsucedida, de acordo com o código de retorno `ESPIPE - Illegal seek` indicado. A referida *syscall* foi invocada ao fim da execução do programa, momento em que a *POC* atinge seu objetivo e causa *kernel panic* (falha crítica no *kernel* do Linux).

A existência de padrões reconhecíveis do ataque, a alta proximidade semântica entre os termos usados no código fonte da *POC* e as *syscalls* registradas e a pouca quantidade de ruído no traço (em comparação ao traço entregue pelo *atrace*) permitem a utilização de *strace* como um instrumento de detecção de ataques.

Já a ferramenta *Perfetto* entrou um arquivo de 40 MB. O visualizador de traços (disponível por meio do upload de arquivos em `https://ui.perfetto.dev/`) exibe registros sobre *syscalls*, estado do escalonador de tarefas e *logs* do sistema, separados de acordo com cada programa em execução.

Destaca-se a presença de um *log* do *KASAN* (*The Kernel Address Sanitizer*, um detector dinâmico de segurança de memória) registrado no traço, indicando a ocorrência de uma violação do tipo *Use-After-Free* causa pelo *garbage collector* do Linux. Entretanto, dado que tal informação é registrada apenas após a concretização do ataque, sua utilidade é reduzida.

A respeito dos traços relativos a execução da *POC*, vê-se a grande quantidade de processos filhos que são criados ao longo do intervalo de observação. Todos os *sleeps*, conjunto de chamadas de *syscalls* e trocas de mensagens via *sockets* são expostos graficamente. As *threads* mais tardias recebem cada vez mais tempo de execução pelo escalonador de tarefas. A *thread* principal, que permanecia adormecida durante grande parte da execução do ataque, é reativada após fim da última e logo depois se verifica o registro do *log* anotado pelo *KASAN*.

A sequência de eventos (registrados pelas *syscalls*) permite que a parte crítica do código fonte seja ser encontrada, conforme exibido no recorte retratado na Figura 4.3.

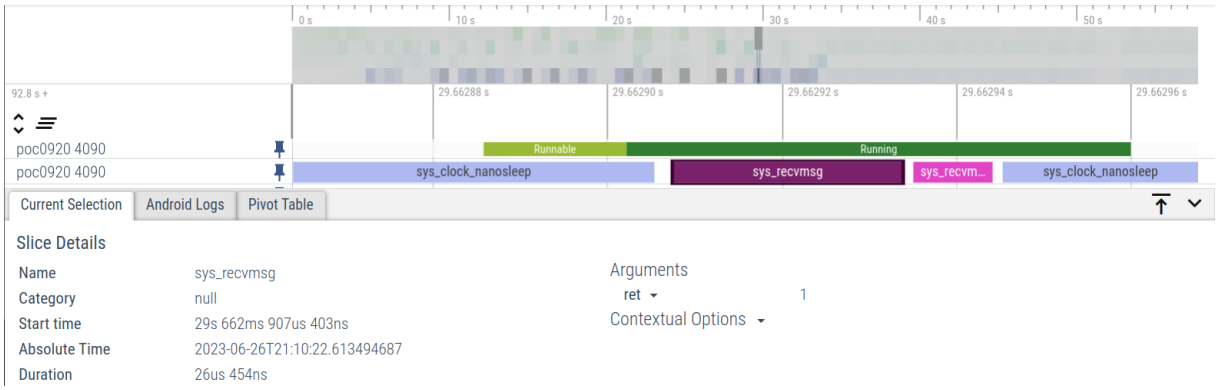

Figura 4.3: Recorte do *Perfetto* associado à parte crítica do código fonte

A Figura 4.4 exibe o recorte do *Perfetto* referente ao momento em que a *POC* em execução obtém êxito na realização do ataque.

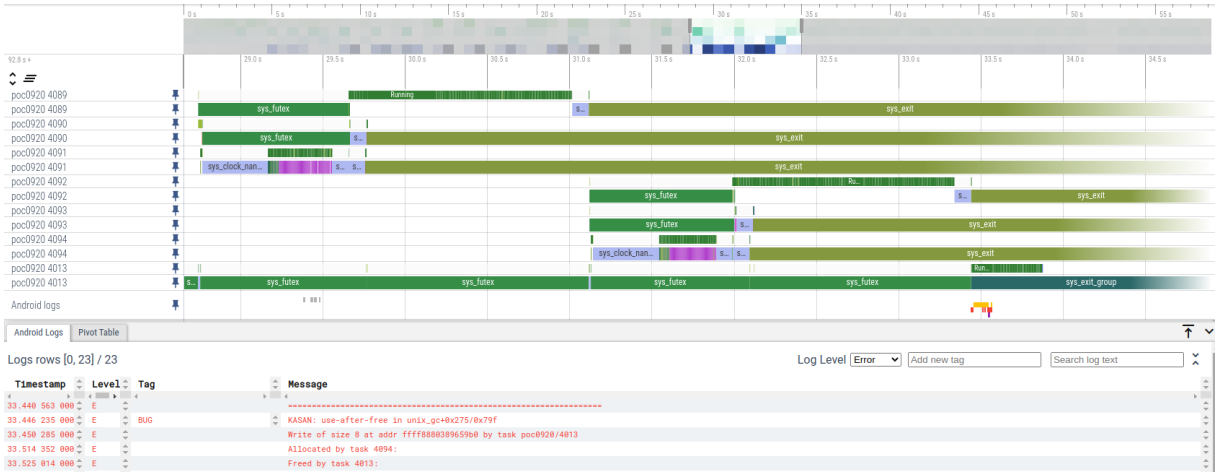

Figura 4.4: Recorte do *Perfetto* indicando *Use-After-Free*

A falta da disponibilização de um arquivo em texto simples dificulta que análises automatizadas possam ser realizadas. O arquivo convertido para `JSON` (também oferecido pelo

Perfetto) não pôde ser aberto no sistema utilizado para testes por conta de *crashes* constantes após renderização de linhas de texto muito longas (com mais de 100 MB), o que impossibilita a sua análise.

Além disso, a ausência de dados sobre os argumentos das *syscalls* limita o uso da ferramenta *Perfetto*. Um mesmo conjunto de *syscalls* pode ser encontrado em traços de aplicações inofensivas e de aplicações maliciosas; sua diferenciação poderia ser verificada pelos argumentos utilizados.

No caso da CVE sob análise, *Perfetto* oferece instrumentos necessários para análise de comportamento da *POC* em tempo de execução. Entretanto, tal análise só pode ser feita manualmente, o que inibiria a utilização da ferramenta como instrumento de detecção em larga escala.

4.4 DISCUSSÃO

A partir dos experimentos conduzidos e dos resultados obtidos, é possível estabelecer se as ferramentas nativas de *tracing* do Android são capazes de detectar ataques. O *ftrace* não pôde ser avaliado pois o *kernel* do Android oferecido pelo Android Emulator não contém o sistema de arquivos necessário para o funcionamento da ferramenta, conforme relatado no capítulo anterior.

A ferramenta *atrace*, apesar de pouco documentada e estar em processo de descontinuação pela Google, fornece informações a nível de sistema em modo textual, o que facilita a interpretação e análise dos traços coletados. Entretanto, poucas informações de relevância para os ataques executados foram encontradas nos arquivos de saída da ferramenta. Quando utilizada para obtenção de traços da *POC* para CVE-2021-0920, a maioria dos dados entregues eram relativos às tarefas do *scheduler*. Registros relativos ao *garbage collector* foram encontrados em pouca quantidade, não levantando suspeita de ação maliciosa pela aplicação em execução.

A mesma análise pode ser feita quando *atrace* foi utilizado para captura de traços da CVE-2019-2215. Porém, sobre esta vulnerabilidade, a ferramenta foi a única que trouxe informações sobre a operação *task_rename*, aqui considerada crítica para detecção de um ataque.

Verifica-se que a ferramenta *strace* foi responsável pelo fornecimento do maior número de informações relevantes sobre os ataques submetidos a teste. Quando utilizada para coleta de traços da CVE-2019-0920, uma grande quantidade de registros de operação sobre *sockets* pôde ser correlacionada a partes do código fonte com relativa facilidade. Ademais, o registro de uma operação imprópria `LSEEK` também pode caracterizar o ataque. Ao monitorar a execução da *POC* da CVE-2019-2215, *strace* forneceu dados precisos sobre o momento em que a corrupção de memória ocorreu, bem como registrou o êxito do ataque ao invocar uma nova *shell*.

A ferramenta *Perfetto* não aparenta possuir utilidade para detecção de ataques. O mecanismo de visualização de traços de forma gráfica (pela interface web) pode ser interessante para análise de incidentes que já ocorreram. Entretanto, a falta de um formato de saída em texto simples prejudica a usabilidade para detecção de ataques em curso. A ausência do registro dos

argumentos das *syscalls* também eleva demasiadamente o grau de generalização das chamadas de sistema, impossibilitando a distinção de uma aplicação maliciosa de outra inofensiva. Dessa forma, entende-se que o foco do *Perfetto* é a análise de performance do sistema em geral - e não a análise com sob ótica da segurança de aparelhos Android.

Considerando *strace* e *atrace*, é possível verificar que suas limitações são complementares: ao passo em que *atrace* é capaz de informações relevantes sobre o sistema em geral e carece de dados sobre *syscalls*, vê-se que *strace* não é capaz capturar interações que ocorrem fora do escopo do programa monitorado, como a chamada da tarefa *task_rename* programa pelo *scheduler*. Sendo assim, um programa de monitoramento poderia ser considerado ideal se abrigasse ambas fontes de informação.

Não obstante, vê-se que os modelos de *tracers* disponíveis no Android não puderam ser utilizados para captura de traços da *POC* que abusa da vulnerabilidade exposta pela CVE-2021-13272. Um mecanismo adequado para captura de traços também deveria ter a capacidade de gravar dados em disco em cenários de *kernel panic*, o que não se verificou em nenhum dos testes realizados.

Sobre a atividade de análise dos traços coletados pelos *tracers*, é possível indicar a inviabilidade da análise manual dos arquivos de saída entregues pelas ferramentas. Os arquivos de texto são extensos (alguns com mais de 200.000 linhas, como no caso do documento entregue pelo *atrace* para análise da CVE-2019-0920) e sua interpretação depende de conhecimento técnico sobre o funcionamento do *kernel* do Linux e do Android em baixo nível. O potencial de descoberta de indícios de ação maliciosa em arquivos de traço poderia ser ampliado pela aplicação de técnicas de aprendizado de máquina e inteligência artificial, por exemplo.

5 CONCLUSÃO

Este estudo se propôs a verificar a viabilidade do uso de ferramentas nativas do sistema operacional para proteção dos dispositivos móveis com Android. No primeiro capítulo, informações introdutórias sobre os sistemas Linux e Android foram apresentadas, seguidas de definições técnicas, informações sobre ferramentas de coleta de traços e estudos sobre trabalhos relacionados no capítulo 2. No capítulo 3, detalhes sobre o funcionamento das ferramentas de *tracing* disponíveis nativamente no Android foram expostos.

No quarto e último capítulo, detalhes sobre as CVE-2019-2215, CVE-2019-13272 e CVE-2021-0920 foram apresentados, bem como foi o método de experimentação em ambiente Android emulado. Os resultados obtidos pela utilização de cada ferramenta de *tracing* sobre os ataques também foram dispostos, indicando os potenciais e limitações de cada *tracer*. A análise definitiva sobre a viabilidade do uso de ferramentas nativas do Android para detecção de ataques é exposta ao final.

Estudos futuros que se propuserem a tratar sobre ferramentas de *tracing* para Android podem avaliar o desenvolvimento de ferramentas de *tracing* mais robustas e completas, capazes de fornecer dados sobre o sistema operacional e *syscalls* em formatos compatíveis com aqueles utilizados para aprendizado de máquina e inteligência artificial. Ressalta-se que medidas de *overhead* não foram consideradas neste estudo. Entretanto, tal fator também deve ser levado em consideração para propostas de trabalhos futuros.

Destarte, conclui-se que:

- É viável o uso de ferramentas de *tracing* nativas do Android para a detecção de um conjunto de ataques. A coleta de traços não obtém sucesso caso um ataque cause a indisponibilidade do sistema (como os *crashes* provocados pela CVE-2021-13272).

- Apenas as ferramentas nativas *atrace* e *strace* apresentam certo grau de utilidade na detecção de ataques. A feramenta *Perfetto* não fornece meios práticos para análise de traços pois foca em questões de desempenho e performance, colocando a segurança em segundo plano. Ainda, informações sobre a CVE-2019-2215 não foram encontradas nos traços fornecidos pela referida ferramenta.

- O uso isolado de *atrace* ou *strace* não é viável: tais ferramentas devem ser utilizadas de forma complementar, dado as limitações que cada uma possui. No caso das ferramentas de *tracing* nativas do Android, notou-se que o `atrace` capturou informações relevantes nas ações do escalonador de tarefas, algo não verificado pelo *strace*. Por sua vez, esta última ferramenta ofereceu dados cruciais para identificações de ataques a partir de *syscalls*, aspecto não observado pelo `atrace`.

- Existem ataques (tal como aqueles que exploram a CVE-2021-13272) que não podem ter seus traços coletados por ferramentas nativas pois causam a indisponibilidade do sistema. Assim, apenas o uso de *atrace* ou *strace* para coleta de traços é insuficiente para detectar todos os tipos de ataques.

REFERÊNCIAS

Abdalla, S. (2019). Say this five times fast: strace, ptrace, dtrace, dtruss. https://dev.to/captainsafia/say-this-five-times-fast-strace-ptrace-dtrace-dtruss-3e1b. Acessado em 26/06/2023.

ANATEL (2022). Infográfico setorial de telecomunicações fev/2022. https://sei.anatel.gov.br/sei/modulos/pesquisa/md_pesq_documento_consulta_externa.php?eEP-wqk1skrd8hSlk5Z3rN4EVg9uLJqrLYJw_9INcO64YqHrlAMQv6UbFt7rY_EvwmlpuyS0KTqHatd7yr-PfJhjOsmVqh30Pawxe9KZ5hVKUvSfhh8IMnGgiFAxwkjO. Acessado em 25/06/2023.

Ansari, A. (2022). Android kernel exploitation. https://cloudfuzz.github.io/android-kernel-exploitation/. Acessado em 25/06/2023.

Arslan, A. (2022). 7 reasons why linux isn't dominating the desktop os market. https://www.makeuseof.com/reasons-linux-isnt-dominating-desktop-market/. Acessado em 25/06/2023.

Bouchrika, I. (2023). Mobile vs desktop usage statistics for 2023. https://research.com/software/mobile-vs-desktop-usage. Acessado em 25/06/2023.

Canfora, G., Medvet, E., Mercaldo, F. e Visaggio, C. A. (2015). Detecting android malware using sequences of system calls. Em *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*, DeMobile 2015, página 13–20, New York, NY, USA. Association for Computing Machinery.

Canfora, G., Mercaldo, F. e Visaggio, C. A. (2013). A classifier of malicious android applications. Em *2013 International Conference on Availability, Reliability and Security*, páginas 607–614.

Carter, L. (2021). 10 biggest cyber attacks in history. https://clearinsurance.com.au/10-biggest-cyber-attacks-in-history/. Acessado em 25/06/2023.

Caviglione, L., Mazurczyk, W., Repetto, M., Schaffhauser, A. e Zuppelli, M. (2021). Kernel-level tracing for detecting stegomalware and covert channels in linux environments. *Computer Networks*, 191:108010.

CVE (2019). Cve-2019-13272. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-13272. Acessado em 25/06/2023.

CVE (2021). Cve-2021-0920. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-0920. Acessado em 25/06/2023.

CVEDetails (2023a). Vulnerability details: Cve-2023-21130. https://www.cvedetails.com/cve/CVE-2023-21130. Acessado em 25/06/2023.

CVEDetails (2023b). Vulnerability statistics. https://www.cvedetails.com/product/19997/Google-Android.html?vendor_id=1224. Acessado em 25/06/2023.

CWE (2023). Cwe-862: Missing authorization. https://cwe.mitre.org/data/definitions/862.html. Acessado em 25/06/2023.

Dimjašević, M., Atzeni, S., Ugrina, I. e Rakamaric, Z. (2016). Evaluation of android malware detection based on system calls. Em *Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics*, IWSPA '16, página 1–8, New York, NY, USA. Association for Computing Machinery.

Farias, P. C. B. (2006). *Curso Essencial de Linux*. Digerati Books.

Faruki, P., Bharmal, A., Laxmi, V., Ganmoor, V., Gaur, M. S., Conti, M. e Rajarajan, M. (2015). Android security: A survey of issues, malware penetration, and defenses. *IEEE Communications Surveys I& Tutorials*, 17(2):998–1022.

Gebai, M. e Dagenais, M. R. (2018). Survey and analysis of kernel and userspace tracers on linux: Design, implementation, and overhead. *ACM Comput. Surv.*, 51(2).

Gera, T., Singh, J., Thakur, D. e Kaur, A. (2021). A survey on andro-financial malware attacks, detection methods and current issues. Em *2021 6th International Conference on Signal Processing, Computing and Control (ISPCC)*, páginas 587–593.

Google (2013). systrace.py. https://chromium.googlesource.com/android_tools/+/d8a5bfea861dfbacd9a74275c00561f7bb27d6e3/sdk/platform-tools/systrace/systrace.py. Acessado em 26/06/2023.

Google (2022a). Security-enhanced linux in android. https://source.android.com/docs/security/features/selinux. Acessado em 25/06/2023.

Google (2022b). Set up for android development. https://source.android.com/docs/setup/about?hl=en. Acessado em 25/06/2023.

Google (2022c). Understanding systrace. https://source.android.com/docs/core/tests/debug/systrace. Acessado em 25/06/2023.

Google (2022d). Using ftrace. https://source.android.com/docs/core/tests/debug/ftrace?hl=en. Acessado em 26/06/2023.

Google (2023a). Android 14 preview. https://developer.android.com/about/versions/14/overview#timeline. Acessado em 25/06/2023.

Google (2023b). Android software stack. https://developer.android.com/guide/platform. Acessado em 25/06/2023.

Google (2023c). Capture a system trace on the command line. https://developer.android.com/topic/performance/tracing/command-line. Acessado em 26/06/2023.

Google (2023d). Configure on-device developer options. https://developer.android.com/studio/debug/dev-options. Acessado em 25/06/2023.

Google (2023e). Kernel overview. https://source.android.com/docs/core/architecture/kernel?hl=en. Acessado em 25/06/2023.

Google (2023f). Perfetto. https://developer.android.com/tools/perfetto. Acessado em 25/06/2023.

Google (2023g). Perguntas frequentes. https://source.android.com/docs/setup/about/faqs?hl=pt-br. Acessado em 25/06/2023.

Google (2023h). Record android trace. https://cs.android.com/android/platform/superproject/+/master:external/perfetto/tools/record_android_trace. Acessado em 25/06/2023.

Google (2023i). Sdk platform tools release notes. https://developer.android.com/tools/releases/platform-tools. Acessado em 26/06/2023.

Google (2023j). System and kernel security. https://source.android.com/docs/security/overview/kernel-security. Acessado em 25/06/2023.

Gregg, B. (2013). Linux performance analysis and tools. https://www.brendangregg.com/Slides/SCaLE_Linux_Performance2013.pdf. Acessado em 25/06/2023.

Gregg, B. (2021). Linux performance. https://www.brendangregg.com/linuxperf.html. Acessado em 25/06/2023.

Grégio, A. R. A. (2012). *Comportamento de programas maliciosos*. Tese de doutorado, Universidade Estadual de Campinas.

Haines, R. e Moore, P. (2022). The selinux notebook. https://github.com/SELinuxProject/selinux-notebook. Acessado em 25/06/2023.

Horn, J. (2019). Issue 1903: Linux: broken permission and object lifetime handling for ptrace_traceme. https://bugs.chromium.org/p/project-zero/issues/detail?id=1903. Acessado em 28/06/2023.

Insights, F. B. (2023). Server operating system market volune, share i& covid-19 impact analysis. https://www.fortunebusinessinsights.com/server-operating-system-market-106601. Acessado em 25/06/2023.

Kaspersky (2023). Poc (proof of concept). https://encyclopedia.kaspersky.com/glossary/poc-proof-of-concept/. Acessado em 25/06/2023.

Khune, R. S. e Thangakumar, J. (2012). A cloud-based intrusion detection system for android smartphones. Em *2012 International Conference on Radar, Communication and Computing (ICRCC)*, páginas 180–184.

LTTng (2021). The lttng documentation. https://lttng.org/docs/v2.13/#doc-what-is-tracing. Acessado em 25/06/2023.

Maziero, C. (2019a). Conceitos básicos de segurança. https://wiki.inf.ufpr.br/maziero/lib/exe/fetch.php?media=sc:seg-texto-01.pdf. Acessado em 04/07/2023.

Maziero, C. (2019b). *Sistemas operacionais: conceitos e mecanismos*. DINF - UFPR.

Mitchell, M., Oldham, J. e Samuel, A. (2001). *Advanced Linux Programming*. New Riders.

Nguyen, T., Orenbach, M. e Atamli, A. (2022). Live system call trace reconstruction on linux. *Forensic Science International: Digital Investigation*, 42:301398. Proceedings of the Twenty-Second Annual DFRWS USA.

Nolte, C. (2022). Why the linux kernel is still important. https://devops.com/why-the-linux-kernel-is-still-important/. Acessado em 25/06/2023.

NordVPN (2023). Arbitrary code execution. https://nordvpn.com/pt-br/cybersecurity/glossary/arbitrary-code-execution/. Acessado em 25/06/2023.

Perfetto (2023a). Quickstart: Record traces on android. https://perfetto.dev/docs/quickstart/android-tracing. Acessado em 25/06/2023.

Perfetto (2023b). System calls. https://perfetto.dev/docs/data-sources/syscalls. Acessado em 25/06/2023.

Perfetto (2023c). Trace configuration. https://perfetto.dev/docs/concepts/config. Acessado em 25/06/2023.

Perfetto (2023d). Trace processor. https://perfetto.dev/docs/analysis/trace-processor. Acessado em 25/06/2023.

Perfetto (2023e). Tracing 101. https://perfetto.dev/docs/tracing-101. Acessado em 25/06/2023.

Polisciuc, R., Albini, L., Grégio, A. e Bona, L. (2020). Análise de aplicativos no android utilizando traços de execução. Em *Anais do XX Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais*, páginas 133–146, Porto Alegre, RS, Brasil. SBC.

Powers, J. (2021). The ultimate guide to mobile device security in the workplace. https://www.techtarget.com/searchmobilecomputing/The-ultimate-guide-to-mobile-device-security-in-the-workplace. Acessado em 25/06/2023.

Rostedt, S. (2022). ftrace - function tracer. https://www.kernel.org/doc/html/v5.4/trace/ftrace.html. Acessado em 26/06/2023.

Ruwase, O. e Lam, M. S. (2004). A practical dynamic buffer overflow detector. Em *Network and Distributed System Security Symposium*.

Schreiber, T. (2011). Android binder: Android interprocess communication.

Securelist (2022). Malicious whatsapp mod distributed through legitimate apps. https://securelist.com/malicious-whatsapp-mod-distributed-through-legitimate-apps/107690/. Acessado em 25/06/2023.

Securelist (2023). The mobile malware threat landscape in 2022. https://securelist.com/mobile-threat-report-2022/108844/. Acessado em 25/06/2023.

Shende, S. S. (1999). Profiling and tracing in linux.

Soares, A. (2015). Dádiva e internet: os artífices dos tutoriais de software livre. Dissertação de Mestrado, Programa de Pós-Graduação em Ciências Sociais - Universidade Federal do Rio Grande do Norte, Natal - RN.

Spraul, M. (2011). Re: 2.4.16 + strace 4.4 + setuid programs. https://lkml.indiana.edu/hypermail/linux/kernel/0112.0/1528.html. Acessado em 28/06/2023.

StatCounter (2023a). Desktop operating system market share worldwide. https://gs.statcounter.com/os-market-share/desktop/worldwide. Acessado em 25/06/2023.

StatCounter (2023b). Mobile operating system market share brazil. https://gs.statcounter.com/os-market-share/mobile/brazil. Acessado em 25/06/2023.

Stone, M. (2019). Bad binder: Android in-the-wild exploit. https://googleprojectzero.blogspot.com/2019/11/bad-binder-android-in-wild-exploit.html. Acessado em 25/06/2023.

Strace (2023a). *STRACE - linux syscall tracer*. strace.io.

Strace (2023b). *strace(1) — Linux manual page*. Man7.org.

Tchakounte, F. e Dayang, P. (2013). System calls analysis of malwares on android. *Maejo international journal of science and technology*, 2.

Tokarnia, M. (2020). Celular é o principal meio de acesso à internet no país. https://agenciabrasil.ebc.com.br/economia/noticia/2020-04/celular-e-o-principal-meio-de-acesso-internet-no-pais. Acessado em 25/06/2023.

Urbanski, M., Mazurczyk, W., Lalande, J.-F. e Caviglione, L. (2016). Detecting local covert channels using process activity correlation on android smartphones. *Computer Systems Science and Engineering*, 32.

Vurdelja, I., Blažić, I., Drašković, D. e Nikolić, B. (2020). Detection of linux malware using system tracers – an overview of solutions.

W3Techs (2023). Usage statistics of operating systems for websites. https://w3techs.com/technologies/overview/operating_system. Acessado em 25/06/2023.

Wankhede, C. (2023). Is android really just linux? https://www.androidauthority.com/android-linux-784964/. Acessado em 25/06/2023.

Wolf, A. (2022). A brief history of cybercrime. https://arcticwolf.com/resources/blog/decade-of-cybercrime/. Acessado em 25/06/2023.

APÊNDICE A – TABELA DE CATEGORIAS - ATRACE

A ferramenta *atrace* admite a seleção de categorias de eventos que serão observados durante a coleta de traços. A lista de categorias é obtida a partir do comando `atrace -list_categories`, cujo resultado está apresentado na tabela A.1.

Categoria	Funcionalidade
gfx	Graphics
input	Input
view	View System
webview	WebView
wm	Window Manager
am	Activity Manager
sm	Sync Manager
audio	Audio
video	Video
camera	Camera
hal	Hardware Modules
res	Resource Loading
dalvik	Dalvik VM
rs	RenderScript
bionic	Bionic C Library
power	Power Management
pm	Package Manager
ss	System Server
database	Database
network	Network
adb	ADB
vibrator	Vibrator
aidl	AIDL calls
nnapi	NNAPI
rro	Runtime Resource Overlay
pdx	PDX services
x sched	CPU Scheduling
irq	IRQ Events
i2c	I2C Events
freq	CPU Frequency
idle	CPU Idle
disk	Disk I/O
sync	Synchronization
workq	Kernel Workqueues
memreclaim	Kernel Memory Reclaim
binder_driver	Binder Kernel driver
binder_lock	Binder global lock trace
pagecache	Page cache

Tabela A.1: Categorias de *tracing* disponíveis no *atrace*

APÊNDICE B – CÓDIGO PARA EXECUÇÃO DO PERFETTO

A ferramenta *Perfetto* admite sua utilização por meio de interface gráfica na web, toques na tela no smartphone Android ou por linhas de comando que podem ser enviadas aos aparelhos via `adb`. O comando a seguir habilita a captura de todos os traços disponíveis nela:

```
adb shell perfetto \
  -c - --txt \
  -o /data/misc/perfetto-traces/trace \
<<EOF

buffers: {
    size_kb: 63488
    fill_policy: DISCARD
}
buffers: {
    size_kb: 2048
    fill_policy: DISCARD
}
data_sources: {
    config {
        name: "android.packages_list"
        target_buffer: 1
    }
}
data_sources: {
    config {
        name: "android.gpu.memory"
    }
}
data_sources: {
    config {
        name: "android.power"
        android_power_config {
            battery_poll_ms: 1000
            battery_counters: BATTERY_COUNTER_CAPACITY_PERCENT
            battery_counters: BATTERY_COUNTER_CHARGE
            battery_counters: BATTERY_COUNTER_CURRENT
            collect_power_rails: true
        }
    }
}
data_sources: {
    config {
        name: "linux.process_stats"
```

```
      target_buffer: 1
      process_stats_config {
         scan_all_processes_on_start: true
         proc_stats_poll_ms: 1000
      }
   }
}
data_sources: {
   config {
      name: "android.log"
      android_log_config {
      }
   }
}
data_sources: {
   config {
      name: "android.surfaceflinger.frametimeline"
   }
}
data_sources: {
   config {
      name: "android.game_interventions"
   }
}
data_sources: {
   config {
      name: "linux.sys_stats"
      sys_stats_config {
         meminfo_period_ms: 1000
         vmstat_period_ms: 1000
         stat_period_ms: 1000
         stat_counters: STAT_CPU_TIMES
         stat_counters: STAT_FORK_COUNT
      }
   }
}
data_sources: {
   config {
      name: "android.heapprofd"
      target_buffer: 0
      heapprofd_config {
         sampling_interval_bytes: 4096
         shmem_size_bytes: 8388608
         block_client: true
      }
   }
}
```

```
data_sources: {
   config {
      name: "linux.ftrace"
      ftrace_config {
         ftrace_events: "sched/sched_switch"
         ftrace_events: "power/suspend_resume"
         ftrace_events: "sched/sched_wakeup"
         ftrace_events: "sched/sched_wakeup_new"
         ftrace_events: "sched/sched_waking"
         ftrace_events: "power/gpu_frequency"
         ftrace_events: "gpu_mem/gpu_mem_total"
         ftrace_events: "raw_syscalls/sys_enter"
         ftrace_events: "raw_syscalls/sys_exit"
         ftrace_events: "regulator/regulator_set_voltage"
         ftrace_events: "regulator/regulator_set_voltage_complete"
         ftrace_events: "power/clock_enable"
         ftrace_events: "power/clock_disable"
         ftrace_events: "power/clock_set_rate"
         ftrace_events: "mm_event/mm_event_record"
         ftrace_events: "kmem/rss_stat"
         ftrace_events: "ion/ion_stat"
         ftrace_events: "dmabuf_heap/dma_heap_stat"
         ftrace_events: "kmem/ion_heap_grow"
         ftrace_events: "kmem/ion_heap_shrink"
         ftrace_events: "sched/sched_process_exit"
         ftrace_events: "sched/sched_process_free"
         ftrace_events: "task/task_newtask"
         ftrace_events: "task/task_rename"
         ftrace_events: "lowmemorykiller/lowmemory_kill"
         ftrace_events: "oom/oom_score_adj_update"
         ftrace_events: "ftrace/print"
         atrace_apps: "*"
      }
   }
}
duration_ms: 10000

EOF
```