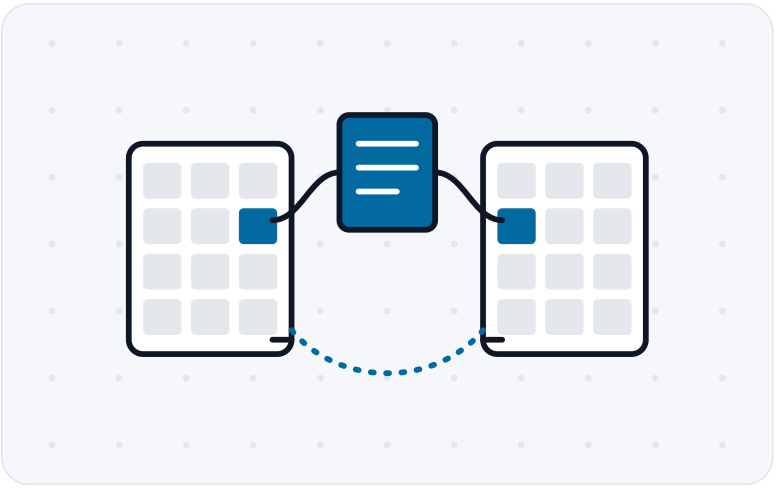
A-SIT Technology
E-Government Innovationszentrum

17. Mai 2024 Überblick · Mobile Sicherheit · 15 Min. Lesezeit

Potentielle Covert Channels durch Kernel Samepage Merging unter Android

Florian Draschbacher



Autorinnen und Autoren	Florian Draschbacher
Stand	2024-05-14
Quelle	https://technology2.a-sit.at/veroeffentlichungen/potentielle-covert-channels-durch-kernel-samepage-merging-unter-android/

Kernaussagen

- KSM ist in der Standard-Android-Kernel-Konfiguration nicht aktiviert, aber einzelne Herstellergeräte (z. B. Samsung Galaxy A51, OnePlus) haben es aktiviert.
- Bei OnePlus lässt sich der KSM-Daemon ohne Root-Rechte über einen Schalter in den Systemeinstellungen starten.
- KSM kann über Timing-Effekte beim Schreibzugriff einen Covert Channel bilden, der pro Speicherseite 1 Bit überträgt.
- Ein solcher Covert Channel könnte die Isolation des Android Virtualization Framework oder der SDK Runtime umgehen.

> - Laut Possemato et al. wichen 20 % von knapp 3000 Android-ROMs vom Google-CDD ab, 190 von 2396 Kernel-Images bei der Kernel-Konfiguration.

Kernel Samepage Merging (KSM) ist ein Mechanismus im Linux-Kernel, der es erlaubt, identische Seiten im Arbeitsspeicher zwischen mehreren Prozessen zu teilen. So wird die Effizienz der Speichernutzung verbessert. Um KSM nutzen zu können, muss ein Programm einen Speicherbereich explizit als mergable markieren. Wird eine Seite gleichen Inhalts von mehreren Prozessen als mergable markiert, so wird eine der Kopien gelöscht und eine gemeinsame physische Speicherseite von den Page Tables beider Prozesse referenziert. Da Android auf den Linux-Kernel aufbaut, kann KSM auch hier zum Einsatz kommen. Zwar gibt es schon Forschung, die die Konsequenzen von KSM auf die Sicherheit von virtuellen Maschinen erforscht, doch Projekte zu KSM unter Mobilgeräten sind in der Literatur kaum vorhanden. Im Rahmen dieses Projektes erläutern wir zunächst den Android-Linux-Kernel, die Möglichkeit zur Modifikation durch Geräte-Hersteller, sowie in der Vergangenheit aufgedeckte dadurch entstandene Sicherheitsprobleme. Anschließend besprechen wir Kernel Same Page Merging unter Android, erläutern bestehende Covert und Side Channels, und diskutieren, wie KSM unter Android neue solcher Kanäle ermöglicht.

1. Einleitung

Mit über 70% Marktanteil ist Android das am weitesten verbreitete Betriebssystem für Tablets oder Smartphones. Entscheidend für diesen Erfolg war auch das Lizenzmodell des mobilen Betriebssystems. Es wird als Open-Source-Projekt unter Federführung von Google entwickelt. Jeder Hersteller von mobilen Geräten kann den Quellcode beliebig modifizieren, etwa um ihn auf die verbaute Hardware abzustimmen. So werden verschiedene Android-Geräte von einer Vielzahl verschiedener Hersteller angeboten.

Dieser Reichtum an Geräten hat sich in der Vergangenheit aber nicht nur als Vorteil der Android-Plattform erwiesen. Zwar sorgt die Möglichkeit für einen nie abbrechenden Strom immer neuer Android-Geräte, allerdings fallen diese immer wieder durch ihren Mangel an Sicherheit auf. Viele Hersteller nutzen die Möglichkeit zur Anpassung auch, um neuartige Funktionen umzusetzen, die als Alleinstellungsmerkmal gegenüber der Konkurrenz dienen sollen. Bei dieser Entwicklung steht meist das sichtbare Nutzererlebnis im Vordergrund, während die Sicherheit oft nur beiläufig bedacht wird. Häufig präsentieren Hersteller außerdem so viele neue Geräte in so kurzer Zeit, dass sie nicht ausreichend Kapazitäten haben, um diese Geräte auch langfristig mit Sicherheitsupdates zu versorgen. Eine einmal bestehende Sicherheitslücke kann so dazu führen, dass das Gerät nie mehr sicher betrieben werden kann.

Ein erst in jüngerer Zeit in den Fokus der Forschung gerücktes Thema sind Side Channels bzw. Covert Channels. Bei Side Channels handelt es sich im Allgemeinen um die Möglichkeit, aus Begleiterscheinungen der Programmausführung (etwa Stromverbrauch oder Zeitdauer), Rückschlüsse auf den internen Programmzustand zu ziehen. Bei einem Covert Channel werden diese Phänomene dazu benutzt, um die Prozessisolation zu brechen, also Prozesse miteinander kommunizieren zu lassen, die eigentlich (bei Außer-Acht-Lassung dieser Begleiterscheinungen) keine Möglichkeit dazu haben. Benutzt werden kann diese Methode etwa von Angreifern, um trotz bestehender Sicherheitsvorkehrungen Daten extrahieren zu können.

In diesem Projekt wurde untersucht, welche Hersteller-Anpassungen sich zur Umsetzung eines Covert Channels eignen. Insbesondere wird hier auf Kernel-Samepage-Merging eingegangen, das in der Vergangenheit schon in anderen Linux-Umgebungen dazu benutzt wurde, Covert Channels umzusetzen, die Virtualisierungs-Grenzen umgehen konnten. Zusätzlich wurde ein Überblick über den Entwicklungsprozess der Linux-Kernel-Trees im Android-Ökosystem erarbeitet, sowie eine Zusammenschau bestehender Literatur bezüglich durch Anpassungen entstandener Sicherheitslücken im Android-Linux-Kernel, sowie bestehender Side bzw. Covert Channels unter Android. Auch das Potential von Covert Channels im Hinblick auf aktuelle Entwicklungen auf der Android-Plattform wurde studiert und wird in diesem Bericht diskutiert.

2. Hintergrund

In diesem Abschnitt sollen jene Technologien erklärt werden, die für das weitere Verständnis der späteren Ausführungen notwendig sind.

2.1. Android

Die Android-Plattform ist eines der am weitesten verbreiteten mobilen Betriebssysteme der Welt und genießt eine enorme Popularität bei Millionen von Nutzern weltweit. Von Smartphones über Tablets bis hin zu eingebetteten Systemen und IoT-Geräten erstreckt sich die Android-Präsenz auf eine Vielzahl von Gerätetypen und -anwendungen.

Die Basis von Android bildet der Linux-Kernel. Alle im Software-Stack darüberliegenden Komponenten („Userspace“) wurden für das Android-Betriebssystem speziell für die Anforderungen von mobilen Geräten (geringer Stromverbrauch, Touch-Display, …) entwickelt. Diese Userspace-Komponenten stehen unter der Apache-2.0-Lizenz zur Verfügung. Die Lizenz erlaubt beliebige Modifikationen, ohne dass der Quellcode des Derivats unter der gleichen Lizenz veröffentlicht werden muss.

Android erlaubt die Ausführung von vom Nutzer installierten Anwendungen. Diese werden jeweils in ihrer eigenen isolierten Umgebung („Sandbox“) ausgeführt, um etwa zu verhindern, dass eine bösartige App Daten einer anderen App stiehlt, oder deren Ausführung beeinflusst. Für Zugriff auf Ressourcen, die mit anderen Apps geteilt werden (etwa das öffentliche Dateisystem oder Sensoren), müssen Apps über eine Permission verfügen. Für besonders gefährliche (sensible) Permissions muss der Nutzer mittels Dialogs explizit die Erlaubnis zum Zugriff erteilen.

Auch unprivilegierte Standard-Anwendungen (ohne Permissions) können mittels Inter-Process-Communication-Funktionalität (IPC) miteinander kommunizieren. Allerdings ist es aufgrund der klar definierten Schnittstellen einfach, auf diese Weise implementierte Kommunikation abzufangen. Soll die Kommunikation verschleiert werden, so können hierfür Covert Channels genutzt werden. Außerdem haben bestimmte Anwendungen keine Möglichkeit, IPC zu nutzen. Diese können auf Covert Channels zurückgreifen, um diese Limitierung zu umgehen und dennoch mit anderen Prozessen zu kommunizieren.

2.2. Linux

Bei Linux handelt es sich um einen Betriebssystem-Kernel, der unter der GNU GPL-Lizenz steht. Er ist daher nicht nur quelloffen, sondern alle abgeleiteten Werke müssen zusätzlich unter derselben Lizenz quelloffen veröffentlicht werden. Aus diesem Grund sind auch Hersteller von Android-Geräten verpflichtet, für jeden

an ein Gerät angepassten Android-Kernel den Quellcode frei zugänglich zu machen. Dies erlaubt etwa auch Einblick in die zum Kompilieren verwendete Konfiguration.

Der Linux-Kernel konfiguriert und verwaltet die Geräte-Hardware und stellt sie in einheitlicher Form für höhere Software-Schichten zur Verfügung. Die einheitlichen Schnittstellen des Linux-Kernels bieten etwa Basis-Funktionalität wie Memory Management, Prozessisolation, Threading, Dateisysteme, Locking, Inter-Process-Communication (IPC), Networking, etc. Hervorzuheben ist hier, dass einige dieser Mechanismen auch verwendet werden, um zentrale Sicherheitsmechanismen im Android-Betriebssystem umzusetzen. Beispielsweise benutzt Android die Prozess-Isolation und das Discretionary Access Control (DAC) von Linux zur Umsetzung der App-Sandbox.

3. Android Common Kernel und Generic Kernel Image

In der Vergangenheit mussten Hersteller für jedes Mobilgeräte-Modell eine eigene Android-Version warten. Diese Version enthielt dann zum Beispiel die passenden Treiber für die verbaute Hardware, etwa für den Hauptprozessor (System-on-Chip, SoC), Kommunikationsmodule (Wifi, Bluetooth, LTE, NFC, ...), Kameras und andere Sensoren, und so weiter. Genauer funktionierte der Prozess wie in Abbildung 1 gezeigt und im Folgenden beschrieben.

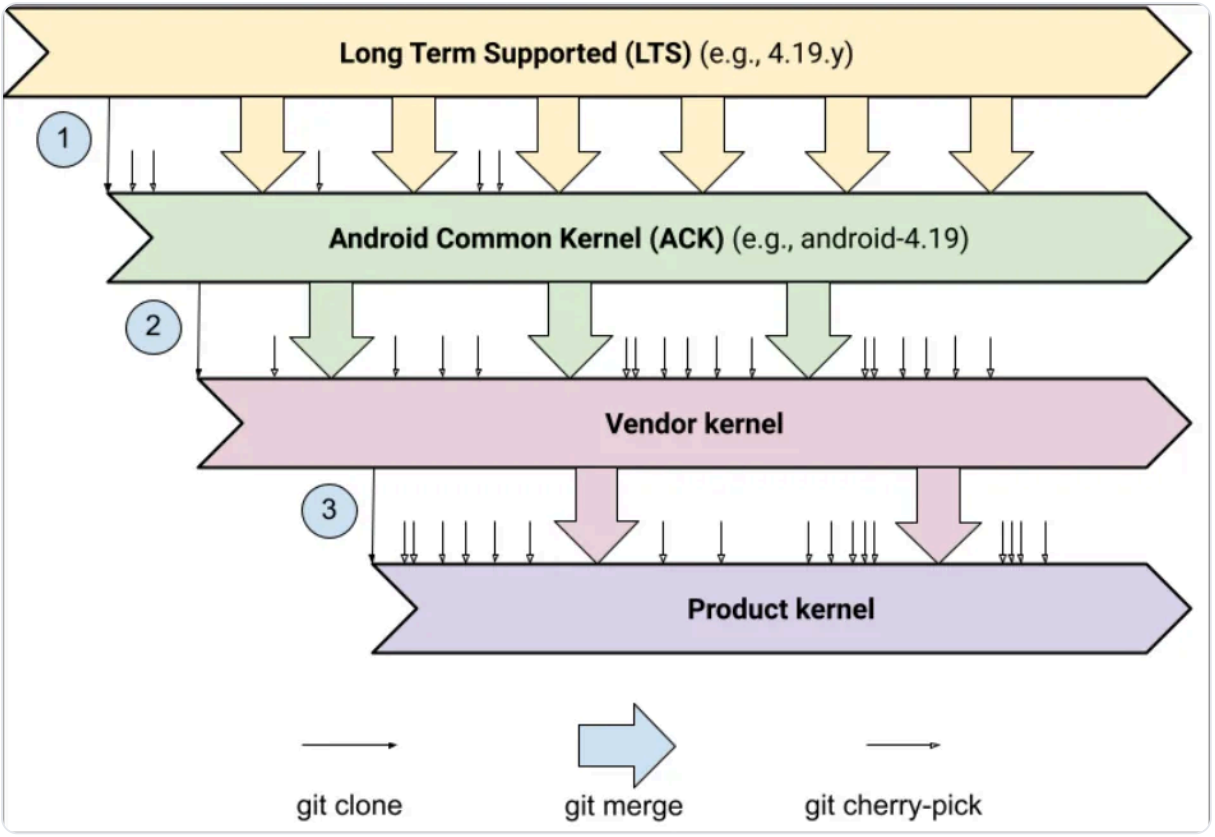


Abbildung 1. Der ursprüngliche Entwicklungsprozess von Android-Linux-Kernels. Quelle: Android Open Source Project [1].

Das Linux-Projekt veröffentlich in regelmäßigen Abständen einen Long Term Stable (LTS) Kernel. Dabei handelt es sich um einen Linux-Kernel, der für einen definierten Zeitraum (meist mindestens 2 Jahre) offiziell unterstützt wird. Wenn eine Sicherheitslücke gefunden wird, wird für alle aktuell offiziell unterstützten LTS-Kernels ein Patch (Behebung der Sicherheitslücke) veröffentlicht.

Für jeden LTS-Kernel wartet das AOSP-Projekt (Google) einen Android Common Kernel (ACK). Bei diesem handelt es sich um eine modifizierte Variante des LTS-Kernels, der um Android-spezifische Funktionalität (etwa spezielle Treiber, die nicht hardware-spezifisch sind) ergänzt wird. Google wartet diese Android Common Kernels über einen längeren Zeitraum (solange sie von einer einigermaßen aktuellen Android-Version offiziell unterstützt werden), pflegt also Sicherheits-Patches vom entsprechenden LTS-Kernel ein.

Jeder Hersteller leitete wiederum von diesen ACK-Kernels eine eigene Variante ab, die um herstellerspezifische Funktionalität erweitert wird (etwa selbst geschriebene Funktionalität, die auf allen Geräten des Herstellers genutzt wird).

Schließlich leitete jeder Hersteller von seiner geräte-agnostischen ACK-Variante einen Geräte-Kernel ab, der speziell an ein konkretes Geräte-Modell angepasst war. Hier flossen dann die zu Beginn erwähnten Hardware-Treiber ein.

Aufgrund der komplexen Ableitung eines Geräte-Kernels über mehrere Ebenen ist es nachvollziehbar, dass die regelmäßige Bereitstellung von Sicherheitsupdates eine Herausforderung für Geräte-Hersteller darstellte. Zu einem Zeitpunkt, als vom bereits verkauften Gerät keine neuen Einkünfte mehr generiert werden konnten, lag es daher für einige Hersteller nahe, keine Ressourcen mehr in Software-Updates zu investieren, selbst wenn diese Reparaturen für kritische Sicherheitslücken enthielten.

Um dieser Problematik entgegenzuwirken, startete Google das Generic Kernel Image (GKI) Projekt. Es sieht vor, dass alle Android-Geräte mit einem von Google bereitgestellten GKI-Kernel-Image (kompiliertem Kernel) betrieben werden. Hardware- und herstellerspezifische Komponenten im Kernel können über ein Kernel Module Interface (KMI) eingebunden werden.

Offiziell müssen Android-Geräte, die mit Android 11 oder später ausgestattet sind, zumindest mit dem GKI-Kernel kompatibel sein. Geräte mit Android 13 oder später müssen laut offiziellen Google-Vorgaben mit dem GKI-Kernel ausgeliefert werden. Diese Kernels können mittels des Uname-Strings identifiziert werden. Bei GKI-Kernels folgt die Versionen-Bezeichnung stets dem Schema `Version.PatchLevel.SubLevel-AndroidRelease-KmiGeneration-suffix.` Der Kernel mit Versions-Bezeichnung 5.4.42-android12-0-00544-ged21d463f856 ist also zum Beispiel ein GKI-Kernel. Mittels dieser Informationen kann das Kernel-Paket (und der dazugehörige Quellcode) auch von Googles Website heruntergeladen werden.

Eine Analyse der Kernel aktueller Geräte zeigt allerdings, dass Google für manche Hersteller Ausnahmeregelungen erlaubt. So hat der Kernel eines aktuellen Samsung-Geräts (Galaxy S23) die Versionsbezeichnung 5.15.94-android13-8-27763874-abS911BXXU3BWJM. Nähere Untersuchung zeigt, dass es sich tatsächlich um keinen GKI-Kernel handelt. Das Image enthält Symbole, die nicht im GKI-Kernel enthalten sind. Auch sind noch immer Geräte im Umlauf, die Android-Versionen (und Kernel-Versionen) nutzen, für die GKI noch nicht genutzt werden kann bzw. muss.

Da also der einheitliche GKI-Kernel trotz entsprechender Vorgaben nicht einheitlich genutzt wird, sind weiterhin hersteller-spezifische Änderungen an der Kernel-Konfiguration möglich.

4. Hersteller-spezifische Kernel-Anpassungen

Mehrere Studien kamen schon in der Vergangenheit zur Erkenntnis, dass herstellerspezifische Anpassungen am Kernel schwerwiegende Konsequenzen für die Gerätesicherheit haben.

Possemato et al. [2] analysierten 2021 knapp 3000 Android-ROMs im Hinblick auf deren Einhaltung der Kompatibilitäts-Richtlinien von Google. Diese Richtlinien sollen sicherstellen, dass Android-Geräte verschiedener Hersteller gewisse Mindeststandards in Bezug auf Sicherheit und Benutzererfahrung erfüllen. Sie werden als Compatibility Definition Document (CDD) veröffentlicht. Einige Aspekte des Dokuments können auch mittels automatisierter Tests erfasst werden. Soll ein Gerät als Android-Gerät verkauft werden, so muss es (bzw. das vom Hersteller angepasste Betriebssystem) diese CDD erfüllen.

Die Forscher fanden heraus, dass in 20% der untersuchten ROMs dennoch Abweichungen zum CDD vorlagen. In knapp 8% (190 von 2396 Kernel-Images) wurden Richtlinien in Bezug auf die Linux-Kernel-Konfiguration nicht eingehalten, die im CDD als verpflichtend hinterlegt waren. Es handelt sich hier um Konfigurations-Flags, die Hardening-Funktionalität gegen Memory-Exploits aktivieren. Bei betroffenen Kernels ist es also für Angreifer einfacher, eine Memory-Management-Schwachstelle auszunützen. Bei 10% der untersuchten Kernels wurden außerdem dringende Empfehlungen des CDD nicht umgesetzt. In vielen Fällen wurde zum Beispiel Kernel Address Space Layout Randomization nicht aktiviert. Auch diese Funktion erschwert Angreifern das Ausnützen einer Speicher-Schwachstelle.

Bei diesen Abweichungen vom CDD handelt es sich jeweils um Kernel Customizations. Die Hersteller haben eigenmächtig die Entscheidung getroffen, dringende Empfehlungen oder unbedingte Vorgaben nicht zu erfüllen. Ein möglicher Grund ist hier der Performance-Overhead der deaktivierten Kernel-Funktionalität.

5. Side Channels und Covert Channels unter Android

Auch zu Side bzw. Covert Channels unter Android gibt es in der Literatur schon einige Beiträge.

5.1. Side Channels

Side Channels sind alternative Kommunikationskanäle, die Informationen durch unkonventionelle Mittel offenbaren, häufig durch die unbeabsichtigte Ausnutzung physikalischer oder logischer Eigenschaften eines Systems. Diese Kanäle können Schwachstellen offenbaren, die von Angreifern genutzt werden können, um sensible Daten auszulesen, ohne die eigentlichen Sicherheitsmechanismen zu durchbrechen. Häufige Beispiele für Side Channels sind:

- Timing-Angriffe : Ausnutzung der Zeit, die ein System für bestimmte Operationen benötigt, um geheime Informationen zu gewinnen.
- Elektromagnetische Lecks : Analyse der elektromagnetischen Strahlung, die von elektronischen Geräten emittiert wird, um Informationen zu extrahieren.
- Power Analysis : Überwachung des Stromverbrauchs eines Geräts, um Verschlüsselungsschlüssel oder andere sensible Daten zu erlangen.

Unter Android wurden in der Vergangenheit Arbeiten zu allen drei Kategorien von Side Channels veröffentlicht. Palfinger et al. [3] stellten in der Vergangenheit AndroTIME vor, eine Lösung, die automatisiert das Timing von Android-APIs untersucht, um Side Channels zu identifizieren. Sie konnten dabei mehrere Side Channels finden, die unter anderem verwendet werden konnten, um ohne entsprechende Erlaubnis die installierten Apps, aktiven Nutzeraccounts, Existenz von Dateien oder Browser-Logins zu erfassen.

Schon 2016 zeigten Genkin et al. [4], dass die elektromagnetischen Emissionen eines Smartphones dazu genutzt werden können, Schlüssel zu extrahieren, die gerade in kryptografischen Operationen verwendet werden. Sie platzierten dazu eine magnetische Sonde in der Nähe des Mobilgeräts. Als Beispiel für einen praktischen Angriff nennen die Forscher die Möglichkeit, diese Sonde etwa unterhalb eines Tisches anzubringen, auf dem das Opfer sein Smartphone ablegt. Der Ansatz wurde von den Forschern unter anderem dazu genutzt, Teile eines ECDSA-Schlüssels aus OpenSSL unter Android auszulesen.

Ansätze zur Power Analysis nutzen unter anderem manipulierte Ladegeräte. Cronin et al. [5] stellten einen Power Line Side Channel vor, der anhand des Stromverbrauchs eines ladenden Mobilgeräts Touchscreen-Interaktionen erfassen kann. Dazu wird die Tatsache ausgenutzt, dass Touch-Events häufig zu visuellem Feedback führen, das Fluktuationen im Stromverbrauch des Displays verursacht. Wird der Stromverbrauch genau genug erfasst, so kann anhand des genauen Zeitpunkts im Screen-Refresh festgestellt werden, an welcher Stelle am Bildschirm das visuelle Feedback gezeigt wurde bzw. wo also das Touch-Event stattfand. Wie die Forscher zeigen, lässt sich mit diesem Ansatz etwa der Unlock-Pin eines Smartphones nur anhand dessen Stromverbrauchs rekonstruieren.

5.2. Covert Channels

Covert Channels sind Kommunikationswege, die genutzt werden, um Informationen unter Umgehung der normalen Sicherheitsrichtlinien zu übertragen. Sie werden oft absichtlich geschaffen, um Daten zu verstecken oder zu verschleiern. Covert Channels sind eine Methode, um Informationen zwischen Prozessen zu übertragen, die eigentlich keine Kommunikationsrechte miteinander haben sollten.

Auch hier lassen sich ähnliche physikalische oder logische Effekte nutzen wie schon für Side Channels. Gasior et al. [6] implementierten einen Covert Channel, der durch Variationen im Timing von TCP Nachrichten versteckte Inhalte kommuniziert. Spolaor et al. [7] nutzten eine am Smartphone installierte App, um durch gezielte Ausführung CPU-intensiver Aufgaben den Stromverbrauch so zu steuern, dass damit Informationen kodiert werden können, die von der Ladestation gelesen werden.

6. Kernel Same Page Merging unter Android

Linux Kernel Samepage Merging (KSM) ist ein Mechanismus im Linux-Kernel, der den Speicherverbrauch optimiert, indem er identische Speicherinhalte (speicher-seitenweise, also üblicherweise in Blöcken zu 4096 Bytes) zwischen verschiedenen Prozessen oder virtuellen Maschinen identifiziert und zusammenführt. Um KSM nutzen zu können, müssen mehrere Bedingungen erfüllt sein:

- KSM muss in den Kernel kompiliert sein. Dazu muss zur Compile-Zeit des Kernels in der Kernel-Konfiguration das CONFIG_KSM-Flag gesetzt sein.
- Die Speicherbereiche müssen jeweils als zusammenführbar markiert werden. Dazu muss der Prozess den Speicherbereich mittels madvise-Aufruf und dem Argument MADV_MERGEABLE markieren.
- Der KSM-Daemon muss laufen. Dieser scannt in regelmäßigen Abständen die als zusammenführbar markierten Bereiche, um Duplikate zu erkennen und zusammenzuführen. Sowohl der Abstand als auch die Anzahl der pro Durchgang überprüften Speicherseiten sind konfigurierbar.

6.1. KSM unter Android

Obwohl in der Standard-Android-Kernel-Konfiguration KSM nicht aktiviert ist, stehen aktuelle Android-Geräte zur Verfügung, bei denen das der Fall ist. Es handelt sich um ein Ergebnis von Anpassungen des Geräteherstellers.

Ein Beispiel eines solchen Geräts ist das Samsung Galaxy A51, das meistverkaufte Android-Smartphone im ersten Quartal 2020. In der Konfigurationsdatei (entweder vom Gerät oder aus dem Quellcodepaket des Kernels extrahiert) zeigt sich, dass CONFIG_KSM aktiviert ist. Tatsächlich sind auch die entsprechenden sysfs-Einträge zur Konfiguration zu finden. Es stellt sich allerdings heraus, dass der Daemon nicht läuft. Weshalb Samsung das Flag aktiviert, um dann den Daemon nicht zu starten, ist nicht nachvollziehbar. Eventuell wird der Daemon nur unter bestimmten Bedingungen gestartet. Zu Forschungszwecken lässt sich der Daemon dennoch starten, indem das Gerät zunächst gerootet wird. Erst der Root-Nutzer hat die Möglichkeit, die Konfiguration des Daemons zu ändern, etwa um seine Ausführung zu starten.

Des Weiteren wurden einige Geräte des Herstellers OnePlus mit KSM ausgestattet. Dort lässt sich der Daemon auch ohne Root-Rechte starten. Konkret geschieht dies über einen Schalter in den Systemeinstellungen.

Schließlich muss hier noch eine Entwicklung erwähnt werden, die dazu führen könnte, dass in Kürze weitere Hersteller (eventuell sogar Google im GKI) KSM aktivieren. Mit Android 13 wurde das Android Virtualization Framework (AVF) eingeführt [8]. Dieses erlaubt das Starten von einfachen Android-Instanzen in virtuellen Umgebungen innerhalb eines Android-Geräts. Das System soll dazu dienen, einzelne Komponenten noch weiter vom Rest des Systems zu isolieren, als dies durch die App-Sandbox geschieht. Nachdem KSM ursprünglich geschaffen wurde, um den Speicherbedarf von virtualisierten Umgebungen zu minimieren, ist nicht auszuschließen, dass KSM im Rahmen des AVF Einzug im GKI-Kernel hält.

6.2. Anwendung für Covert Channels

Wie andere Memory-Deduplication-Mechanismen auch, kann KSM zur Verwirklichung von Side Channels und Covert Channels genutzt werden. Da es für einen Prozess relativ einfach ist, einen Side Channel auf Basis von KSM zu unterbinden (indem sensible Speicherbereiche nicht als zusammenführbar markiert werden), soll hier nur auf Covert Channels eingegangen werden.

Covert Channels lassen sich mittels KSM über Timing-Effekte umsetzen. Als naiver Ansatz kann etwa von Prozess A eine von zwei möglichen Speicherseiten definierten (aber verschiedenen) Inhalts als zusammenführbar markiert werden. Prozess B markiert beide Speicherseiten als zusammenführbar. Anschließend misst Prozess B, wie lange der Schreibzugriff auf die beiden Speicherseiten benötigt. Für die Seite, die auch von Prozess A als zusammenführbar markiert wurde, sind längere Schreibzeiten zu erwarten. Dies liegt daran, dass der Kernel die Zusammenführung auflösen muss, also die zusammengeführte Seite erst kopieren und neu in den Prozess mappen muss. Die Information, welche der beiden Seiten von Prozess A als zusammenführbar markiert wurde, überträgt also 1 Bit an Information an Prozess B. Die Übertragungsrate lässt sich wie für andere cache-basierte Covert Channels optimieren [9].

6.3. Anwendungsfälle unter Android

Wie schon zu Beginn erwähnt, können Android-Apps unter Android auch ohne besondere Permissions mittels IPC miteinander kommunizieren. Dennoch ist es in manchen Szenarien wünschenswert, für die Kommunikation keine klar definierten Programmierschnittstellen zu nutzen. Außerdem besteht die Möglichkeit der Nutzung von IPC in manchen Szenarien nicht zur Verfügung.

Erwähnenswert ist in diesem Zusammenhang das Android Virtualization Framework (AVF), auf das auch schon weiter oben Bezug genommen wurde. Ein Covert Channel (zum Beispiel auf Basis von KSM) eignet sich hier dazu, die Isolation der einzelnen virtualisierten Umgebungen zu umgehen.

Darüber hinaus ergibt sich eine neue Anwendungsmöglichkeit im Zusammenhang mit der SDK Runtime als Teil des größeren Projekts Privacy Sandbox, die Google schrittweise in das Android-Betriebssystem integriert. Dabei sollen Werbebibliotheken nicht mehr innerhalb des App-Prozesses ausgeführt werden, sondern in einem separaten Prozess. Auch diese Isolation könnte mittels Covert Channels durchbrochen werden.

7. Zusammenfassung

In diesem Bericht beleuchteten wir den Entwicklungsprozess des Android-Linux-Kernels, sowie verschiedene Möglichkeiten von durch Hersteller-Anpassungen verursachte Sicherheitsproblematiken. Wir konzentrierten die Schilderungen dann auf Side Channels und Covert Channels unter Android, und beschrieben wie Linux Kernel Samepage Merging dazu benutzt werden kann, Covert Channels zu implementieren, die die Isolation von virtuellen Umgebungen des Android Virtualization Framework (AVF) oder der SDK-Sandbox zu umgehen.

Referenzen

[1] Google, „The Generic Kernel Image (GKI) project,“ 03 2024. [Online]. Available: https://source.android.com/docs/core/architecture/kernel/generic-kernel-image.

[2] A. Possemato, S. Aonzo, D. Balzarotti und Y. Fratantonio, „Trust, But Verify: A Longitudinal Analysis Of Android OEM Compliance and Customization,“ in IEEE Symposium on Security and Privacy (SP), 2021.

[3] G. Palfinger, B. Prünster und D. J. Ziegler, „AndroTIME: Identifying Timing Side Channels in the Android API,“ in IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), 2020.

[4] D. Genkin, L. Pachmanov, I. Pipman, E. Tromer und Y. Yarom, „ECDSA Key Extraction from Mobile Devices via Nonintrusive Physical Side Channels,“ in Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, 2016.

[5] P. Cronin, X. Gao, C. Yang und H. Wang, „Charger-Surfing: Exploiting a Power Line Side-Channel for Smartphone Information Leakage,“ in 30th USENIX Security Symposium, 2021.

[6] W. Gasior und L. Yang, „Exploring Covert Channel in Android Platform,“ in International Conference on Cyber Security, 2012.

[7] R. Spolaor, L. Abudahi, V. Moonsamy, M. Conti und R. Poovendran, „No Free Charge Theorem: A Covert Channel via USB Charging Cable on Mobile Devices," in Applied Cryptography and Network Security, 2017.

[8] S. Patil, „Virtual Machine as a core Android Primitive," 05 12 2023. [Online]. Available: https://android-developers.googleblog.com/2023/12/virtual-machines-as-core-androidprimitive.html.

[9] Y. Yarom und K. Falkner, „FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack," in 23rd USENIX Security Symposium, 2014.