A-SIT

Zentrum für sichere Informationstechnologie - Austria

Laufzeit-Integritätsprüfung in Android-Applikationen

A-SIT

Laufzeit-Integritätsprüfung in Android-Applikationen

Autor:
Florian Draschbacher:
florian.draschbacher@iaik.tugraz.at

Datum: 08.02.2023

Abstract/Zusammenfassung:
Um das Reverse-Engineering von Anwendungen zu erschweren und Repackaging-Attacken abzuwenden, setzen einige besonders sicherheits-affine Android-Entwickler auf die Integration von Runtime-Integritäts-Checks in ihren Programmcode. Diese Checks stellen zur Laufzeit unter anderem sicher, dass das Betriebssystem in einem unmodifizierten Werkszustand vorliegt (keine Root-Möglichkeiten, originales Firmware-Image, unmodifizierte System-Partition), und dass die Anwendung zwischen der Auslieferung vom Entwickler und der Installation beim Nutzer nicht modifiziert wurde. Obwohl diese Checks in erster Linie die Sicherheit des Nutzers erhöhen sollen, erschweren sie dennoch auch die Sicherheitsanalyse von Android-Anwendungen in der Forschung.

Im Rahmen dieses Forschungsprojektes wird ein Überblick über verschiedene Möglichkeiten zur Integritäts-Prüfung und -Sicherstellung geboten. Dazu werden Vorschläge und Implementierungen aus Wissenschaft und Praxis diskutiert. Zusätzlich präsentieren wir eine Studie zur Prävalenz von Laufzeit-Integritätsprüfung in 99 populären Android-Anwendungen.

Inhalt

1. Einleitung

Parallel zur weiten Verbreitung von mobilen Endgeräten wie Smartphones und Tablets entdeckten auch Parteien mit bösartigen Absichten die Vorteile der neuen Plattformen für sich. Die Allgegenwärtigkeit dieser Geräte, die permanente Anbindung an das Internet und die Dichte an sensiblen Nutzerinformationen, die darauf verarbeitet und gespeichert werden, machen mobile Geräte für Angreifer besonders attraktiv.

Ein andauerndes Problem im App-Ökosystem ist Repackaging. Dabei wird eine legitime Anwendung von einem Angreifer manipuliert und an unbedarfte Nutzer weitergegeben. Über dieses Vorgehen lassen sich etwa Phishing-Angriffe realisieren, bei denen sensible Nutzer-Daten (etwa Anmeldedaten oder Bankdaten) gesammelt werden können. Auch wird das Angriffsszenario genutzt, um beliebigen Schadcode am Gerät auszuführen.

Alternativ zum Repackaging können bestehende Apps auch manipuliert werden, indem die Laufzeitumgebung bzw. das Betriebssystem von einem Angreifer kontrolliert werden. Hier sind zum Beispiel auch Angriffe denkbar, in denen der Kontrollfluss einer Anwendung geändert werden soll, etwa um Funktionalität freizuschalten, die eigentlich hinter einer Bezahlschranke verborgen ist.

Um beide Kategorien von Angriffen unter Android zu unterbinden, stehen verschiedene Schutzvorkehrungen zur Verfügung, die App-Entwickler implementieren können, um zur Laufzeit sicherzustellen, dass die Ausführungs-Umgebung wie auch der Anwendungs-Code selbst nicht manipuliert sind. In diesem Forschungsprojekt soll untersucht werden, welche Möglichkeiten hier zur Verfügung stehen, und inwieweit sie bereits von Android-Anwendungen im Produktiv-Betrieb genutzt werden.

2. Hintergrund

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

2.1. Android

Android ist das populärste Betriebssystem für Mobilgeräte wie Smartphones und Tablets. Das System wird im Rahmen des Android Open Source Project (AOSP) von einigen Geräte-Herstellern unter Federführung von Google entwickelt. Aus technischer Sicht handelt es sich um ein Betriebssystem auf Basis eines Linux-Kernels. Die Userspace-Komponenten weichen jedoch wesentlich von denen jeder anderen Linux-Distribution ab. Sie wurden speziell für die Anforderungen von Mobilegeräten neu entwickelt.

2.2. Android-Apps

Android erlaubt Nutzern die Ausführung von beliebigen Anwendungen von Drittanbietern. Google stellt ein Software-Development-Kit (SDK) zur Verfügung gestellt, mithilfe dessen Entwickler in den Programmiersprachen Java, Kotlin, C und C++ Anwendungen schreiben können. Nach dem Kompilieren werden Programme in eine Android Package (APK) Datei verpackt und können auf einem Android-Gerät installiert werden. In der Regel werden APK-Dateien zur Verteilung an Endnutzer über den Google Play Store vertrieben. Dieser Distributionskanal muss aber nicht zwangsläufig genutzt werden. In den Android-Systemeinstellungen kann die Installation von APK-Dateien aus beliebigen Quellen aktiviert werden.

2.3. APK-Datei

Bei APK-Dateien handelt es sich im Wesentlichen um ein Zip-Archiv mit klar definierter Struktur. Zentrale Elemente innerhalb dieser Struktur sind das Applikations-Manifest, die DEX-Files, native Bibliotheken und App-Ressourcen. Beim Manifest handelt es sich um eine XML-Datei, die beschreibt, welche Betriebssystem-Ressourcen die Anwendung nutzt und welche Funktionalitäten sie dem System bzw. anderen Anwendungen zur Verfügung stellt. Dalvik Executable (DEX)-Dateien enthalten den kompilierten Programmcode, der aus Java oder Kotlin kompiliert wurde. Im Rahmen der Kompilierung wird dieser in Dalvik Bytecode transformiert. Dabei handelt es sich um eine Intermediate Representation, die die plattformunabhängige Ausführung des Codes am Gerät erleichtert. Erwähnenswert ist in diesem Zusammenhang, dass im Dalvik-Bytecode entscheidende Highlevel-Informationen zum ursprünglichen Programmcode erhalten bleibt. Im Unterschied zu Programmen, die im Maschinencode vorliegen, kann hier also die implementierte Funktionalität relativ einfach rekonstruiert und in weiterer Folge verändert werden. Native Bibliotheken enthalten den kompilierten Programmcode jener Applikations-Anteile, die in den nativen Programmiersprachen C oder C++ implementiert wurden. Da sie im CPU-spezifischen Maschinencode vorliegen, sind diese wesentlich schwerer rekonstruier- und manipulierbar.

2.4. App Signing

Bevor eine App im APK-Format auf einem Android-Gerät installiert werden kann, muss sie vom Entwickler signiert werden. Hierzu generiert der Entwickler ein asymmetrisches Schlüsselpaar, bestehend aus einem privaten Schlüssel und einem öffentlichen Zertifikat. Schlüssel und Zertifikat sind kryptografisch so miteinander verbunden, dass sich über das Zertifikat feststellen lässt, dass die APK-Datei vom Besitzer des privaten Schlüssels signiert wurde.

Im Rahmen des Signatur-Prozesses wird ein digitaler Fingerabdruck jeder Datei im APK-File berechnet. Schließlich wird mit dem privaten Schlüssel ein Fingerabdruck dieser Fingerabdrücke verschlüsselt und das Zertifikat mit dem öffentlichen Schlüssel in der APK-Datei hinterlegt. Während der Installation der Anwendung berechnet das Betriebssystem dieselben Fingerabdrücke, und vergleicht den resultierenden Fingerabdruck mit dem über den öffentlichen Schlüssel entschlüsselten Signatur-Wert. Bei Übereinstimmung kann kryptografisch bestätigt werden, dass die APK-Datei vom Eigentümer des zum Zertifikat passenden privaten Schlüssel signiert wurde.

Die Signatur einer APK-Datei soll in diesem Szenario jede Anwendung mit einer Entwickler-Identität verknüpfen. Obwohl im Zertifikat häufig Informationen zum Namen des Entwicklers hinterlegt sind, besteht die Entwickler-Identität aus kryptografischer Sicht nur aus dem privaten Schlüssel. Weil die Ausstellung von Schlüsselpaaren (bzw. Zertifikaten) vom Entwickler selbst erledigt wird und nicht zentral überwacht wird, können im Zertifikat beliebige Informationen hinterlegt werden. Die App-Signatur lässt also keine verlässlichen Schlüsse zur rechtlichen Identität des Entwicklers zu.

Diese Tatsache ist eine entscheidende technische Grundlage für Repackaging-Attacken. Manipulierte APK-Files müssen vom Angreifer neu signiert werden, bevor sie installiert werden können. Weil den Angreifern hierzu der private Schlüssel des ursprünglichen App-Entwicklers in der Regel nicht zur Verfügung steht, müssen sie ein neues asymmetrisches Schlüsselpaar verwenden. Die resultierende Anwendung ist allerdings problemlos installierbar. Einem durchschnittlich versierten Anwender ist nicht ersichtlich, dass die Anwendung manipuliert bzw. mit einem vom Original abweichenden Zertifikat ausgestattet ist.

2.5. Rooting und Hooking

Da Android auf dem Linux-Kernel basiert, gibt es auch hier einen Root-Nutzer, der aus Sicht der Discretionary Access Control (DAC) mit jeder möglichen Berechtigung in Bezug auf Ressourcen-Zugriff (etwa von Dateien) ausgestattet ist. In der Regel wird das Android-System allerdings so konfiguriert, dass es dem Endnutzer nicht möglich ist, beliebige Software mit Root-Berechtigungen auszuführen. Es ist allerdings bei den meisten Android-Geräten möglich, den Root-Zugriff entweder gewollt vom Endnutzer („Rooting“) oder durch Ausnutzung von Schwachstellen zu erlangen. So ist es unter anderem auch möglich, System-Komponenten auszuwechseln, bzw. den Speicherinhalt beliebiger Prozesse zu manipulieren.

Wird eine Android-Anwendung ausgeführt, so wird ihr kompilierter Code im Allgemeinen nicht direkt auf der CPU ausgeführt, sondern innerhalb der Android Runtime (ART). Dieses Vorgehen erlaubt die Ausführung von Android-Anwendungen auf Geräten mit beliebiger CPU-Architektur. Es ermöglich Angreifern aber auch, durch das Ersetzen einiger weniger System-Komponenten, die Ausführung jeder installierten Android-Anwendung zu manipulieren. Wird hier gezielt die Ausführung einer bestimmten Funktion abgefangen, so spricht man von „Hooking“.

Weil mittels Rooting bzw. Hooking die Manipulation des Kontrollflusses von Android-Anwendungen in ähnlichem Ausmaß wie mit Repackaging möglich ist, integrieren einige Entwickler besonders sicherheits-sensibler Anwendungen Laufzeit-Integritäts-Checks. Wird eine manipulierte Laufzeitumgebung (also etwa das Vorhandensein von Rooting-Tools oder Hooking-Lösungen) festgestellt, verweigert die Anwendung den Betrieb.

3. Lokale Lösungen

In diesem Abschnitt wird diskutiert, wie Repackaging ohne Zuhilfenahme externer Parteien erkannt und abgewendet werden kann.

3.1. Zentralisierte Repackage-Erkennung

Die einfachste Möglichkeit, um Repackaging zu erkennen, besteht in statischen Lösungen, die anhand eines Vergleichs einer bekannten Original-Anwendung A und eines Paketes B mit unklarer Provenienz feststellen können, ob B durch Modifikation von A entstanden ist. Eine solche Lösung eignet sich etwa gut zur Integration in Appstores, wo für jede von Entwicklern neu übermittelte App sichergestellt werden kann, dass es sich um ein eigenes Werk handelt. Erkannte Fälle von Repackaging können beim Upload so abgewiesen werden.

Mit dieser Technologie kann jedoch nicht verhindert werden, dass legitime Original-Anwendungen aus diesen Appstores geladen, modifiziert und mit neuer Signatur versehen über andere Kanäle weitergegeben werden. Wie oben erwähnt, können unter Android auch Apps installiert werden, die etwa

von beliebigen Websites geladen wurden. Es existieren etwa auch Websites, die explizit den illegalen Austausch von ursprünglich zahlungspflichtigen Anwendungen zum Zweck haben.

3.2. Dynamisches Repackage-Proofing

Um die Distribution modifizierter Anwendungen auch über Kanäle abseits der etablierten App-Stores zu unterbinden, setzen einige Entwickler auf dynamische Repackage-Erkennung, die sie direkt in ihren Programmcode integrieren können. Das Ziel dieser dynamischen Überprüfung ist in der Regel, die Funktionalität der App bei erkannter Modifikation zu verweigern. Man spricht hier von Repackage Proofing, weil die App gegen Repackaging geschützt werden soll.

Das Application Programming Interface (API), auf das Android-Entwickler zugreifen können, bietet in der `PackageManager`-Klasse mit der Methode `getPackageInfo()` eine Möglichkeit, Informationen (unter anderem) über das APK-File der aktuell ausgeführten Anwendung zu bekommen. Als Teil dieser Informationen kann auch das Signatur-Zertifikat ausgelesen werden, das schon bei der Installation vom System überprüft wurde. Eine simple Möglichkeit zur Laufzeit-Integritäts-Überprüfung besteht also darin, den von dieser Schnittstelle gelieferten Wert mit dem statisch hinterlegten korrekten Signatur-Zertifikat zu vergleichen. Während diese Methode prinzipiell funktioniert, schützt sie in der Praxis nicht effektiv vor Repackaging. Angreifer können als Teil ihrer Modifikationen einfach den Überprüfungs-Code gänzlich entfernen, den Methoden-Aufruf abfangen und das Zertifikat des originalen APKs zurückliefern oder das hinterlegte originale Vergleichszertifikat austauschen.

Aufgrund der bekannten Schwächen dieses Ansatzes, aber der konzeptionellen Alternativlosigkeit für lokale Lösungen in manchen Einsatz-Szenarien, hat eine ganze Reihe von wissenschaftlichen Publikationen untersucht, wie die Überprüfung abgesichert werden kann. Das Ziel ist hier, zu erreichen, dass der Check schwieriger lokalisiert, manipuliert, oder entfernt werden kann, ohne die bestehende Funktionalität der Anwendung zu zerstören.

Eine Möglichkeit, um die Lokalisierung und Manipulation zu erschweren, liegt in der Nutzung von nativem Code. Die wesentliche Schwierigkeit für Angreifer ergibt sich hier aus der Tatsache, dass nativer Code nach dem Kompilieren nur noch in Maschinencode vorliegt, die wesentlich weniger High-Level-Informationen als Dalvik Bytecode enthält, wodurch der ursprüngliche Code schwieriger nachvollzogen werden kann. Häufig wird die Nutzung von nativem Code mit weiteren Obfuscation-Mechanismen kombiniert. Tanner et al. [1] kombinieren den Ansatz einer mehrschichtigen Verschlüsselung von Dalvik Bytecode und nativem Code. Auch Protsenko et al. [2] folgen einem ähnlichen Ansatz.

Verschlüsselung von Bytecode wird auch von einigen anderen Wissenschaftlern verwendet. Zeng et al. [3] nutzen die Technik, um Logik-Bomben zu verschleiern, die in die Kern-Funktionalität der App eingewoben werden und nur während der Nutzung der App durch echte Endnutzer einen Signatur-Check vornehmen. Der Schlüssel kann zur Laufzeit nur dann dynamisch abgeleitet werden, wenn keine Modifikationen vorliegen. Ren et al. [4] kombinieren self-decrypting code mit Watermarking.

Für alle diese Lösungen ist anzumerken, dass ihre Absicherung gegen Angreifer, die die Signatur-Checks entfernen möchten, lediglich in Obfuscation liegt. Es wird Angreifern also erschwert, die implementierte Funktionalität anhand des Kompilats nachzuvollziehen. Weil aber die Signatur-Checks in der Regel bestimmte Android-APIs nutzen müssen, um etwa das APK-File im Dateisystem zu lokalisieren, oder die Signatur auszulesen, muss in vielen Fällen die genaue Wirkungsweise von Schutzmechanismen gar nicht bekannt sein, um sie umgehen zu können. Ma et al. [5] zeigen, dass es häufig ausreicht, gezielt die Laufzeitumgebung von Android-Apps zu manipulieren, um entscheidende Android API-Methoden abzufangen und der wiederverpackten App so eine unmanipulierte Umgebung vorzutäuschen. Das Abfangen dieser Methoden kann entweder über systemweites Hooking mittels Root-Rechten umgesetzt

werden, oder indem eine native Bibliothek in die App eingeschleust wird, die beim App-Start die ART-Runtime manipuliert.

Ganz allgemein kann festgestellt werden, dass mit entsprechendem Aufwand alle lokalen Repackage-Proofing-Verfahren umgangen werden können. Es ist davon auszugehen, dass für jeden Obfuscation-Ansatz ein entsprechendes statisches oder dynamisches Analyse-Tool gebaut werden kann, das die Verschleierungs-Mechanismen rückgängig machen und so auch die Signatur-Checks entfernen kann.

3.3. Remote Code Attestation

Um einen unüberwindbaren Schutz gegen Repackaging zu erreichen, wird neben der Anwendung selbst eine unabhängige externe Instanz benötigt, deren Funktionalität nicht im Ziel-Paket selbst implementiert wird, und die separat ausgeführt wird.

In vielen aktuellen Android-Geräten kann eine solche unabhängige externe Instanz sogar lokal implementiert werden, wenn von der Trusted Execution Environment (TEE; z.B. ARM TrustZone) Gebrauch gemacht wird. Dabei stellt die CPU eine sichere Ausführungsumgebung zur Verfügung, deren Zustand vollständig von der Haupt-Ausführungsumgebung entkoppelt ist. Lediglich die sichere Ausführungsumgebung kann bestimmen, in welchem Maß bei Bedarf ein Datenaustausch stattfindet. Demesie et at. [6] stellen einen Ansatz vor, der mittels TEE einen Schutz gegen Repackaging implementiert.

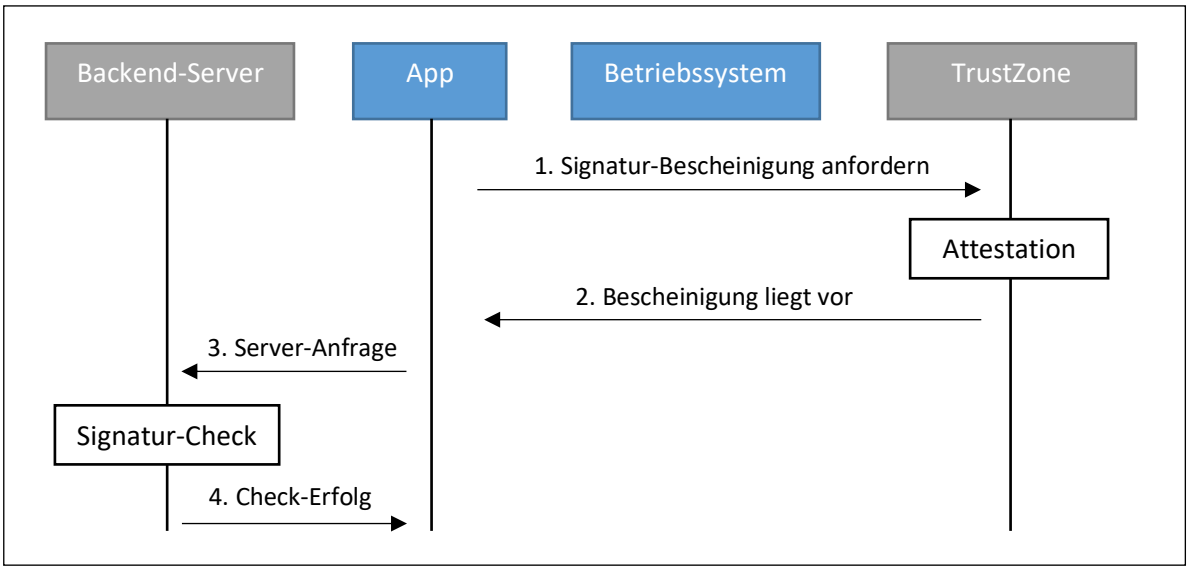

Abbildung 1 Schematische Darstellung simpler Remote Attestation

Während die Überprüfung der Integrität von Laufzeitumgebung und Anwendung mittels TEE lokal und sicher implementiert werden kann, ist ein unüberwindbaren Schutz gegen Repackaging dennoch nur für Anwendungen möglich, die für ihre Kernfunktionalität mit einem Server-Backend kommunizieren müssen. Wird nur lokal in der App entschieden, wie bei einer negativen Integritätsprüfung verfahren werden soll, so lässt sich dieses Verhalten von einem Angreifer selbst bei externer Integritätsprüfung manipulieren. Wird jedoch das Resultat der Integritätsprüfung als Teil jeder Anfrage an den Backend-Server übermittelt, so kann dort außerhalb des Kontrollbereichs des Angreifers entschieden werden, ob die Anfrage beantwortet werden oder die Kommunikation eingestellt werden soll.

Abbildung 1 zeigt den prinzipiellen Ablauf, dem eine solche Implementierung folgt. Die App fordert beim App-Start oder vor kritischen Verarbeitungsschritten (etwa vor Übermittlung eines Highscores) eine Signatur-Bescheinigung von einer entsprechenden Komponente in der Trusted Execution Environment

an. Das Ergebnis wird dann mit der Anfrage an den Backend-Server übermittelt, wo die Bescheinigung evaluiert wird. Nur wenn das bescheinigte Signatur-Zertifikat mit dem originalen Signatur-Zertifikat übereinstimmt, wird die Anfrage beantwortet.

Entscheidend für die tatsächliche Sicherheit der Implementierung sind auf Basis dieses konzeptionellen Aufbaus noch einige weitere Überlegungen. So bedarf es einer Absicherung der Signatur-Bescheinigung gegen Manipulation durch die potentiell attackierte App während der Weiterreichung an den Backend-Server. Hierzu kann asymmetrische Kryptografie verwendet werden. Die Bescheinigung wird von der TEE-Komponente signiert oder verschlüsselt, und am Backend-Server verifiziert oder entschlüsselt. Die signierte (oder verschlüsselte) Bescheinigung wird nun auch Attestation, der gesamte Prozess Remote Attestation genannt. Um Replay-Attacken zu verhindern, muss in die Attestation eine Nonce, also ein einzigartiger Wert, eingebunden werden.

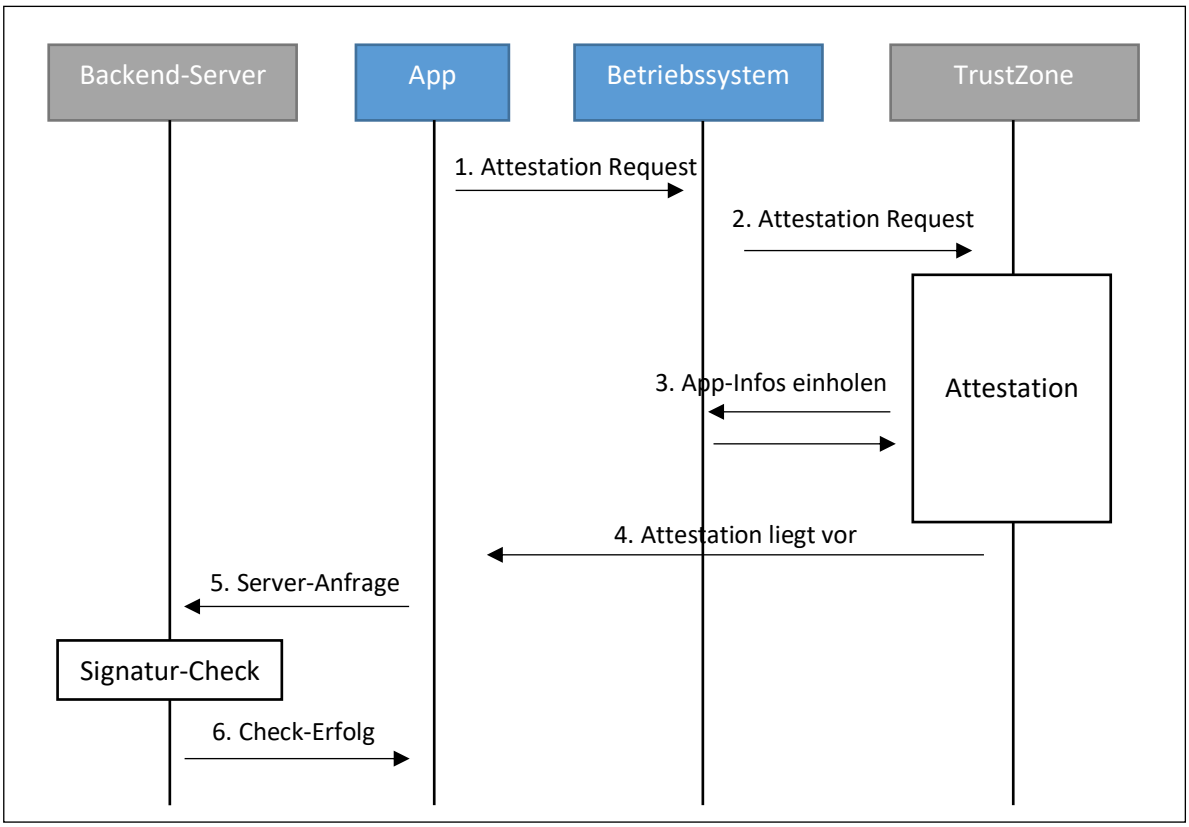

Abbildung 2 Remote Attestation inklusive Betriebssystem-Interaktion

Schließlich ist auch die Rolle des Betriebssystems eine nähere Betrachtung wert. Wie in Abbildung 2 gezeigt, ist die TEE-Komponente auf das Betriebssystem angewiesen, um App-Informationen wie den Paketnamen der ausgeführten Anwendung und die passenden Signaturinformationen zu erhalten. Hat ein Angreifer die Möglichkeit, die Kommunikation zwischen Betriebssystem und TEE-Komponente zu beeinflussen, kann der gesamte Mechanismus ausgehebelt werden. Um dies verhindern, müssen als Teil der Attestation auch Informationen zur Integrität des Betriebssystems eingeholt werden. Hier wird nun sichtbar, weshalb Integritätsprüfung von Anwendungen nicht entkoppelt von der Integritätsprüfung des Betriebssystems betrachtet werden kann. Prünster et al. [7] zeigten als erste, wie mit der seit Android 8.0 verfügbaren Key Attestation die durchgängige Integrität von Bootloader, System und Anwendung garantiert werden kann.

3.4. Google SafetyNet

Um App-Entwicklern ein wirksames Werkzeug gegen Repackaging und ähnliche Attacken auf die Anwendungs-Integrität zur Verfügung zu stellen, bietet Google schon seit einigen Jahren eine Lösung namens SafetyNet Attestation API [1]. Diese funktioniert im Wesentlichen wie die von Prünster et al [7] vorgeschlagene Lösung. Für die Ausstellung der Attestation nimmt die Anwendung Kontakt mit den Google Play Services auf, die für diesen Zweck wiederum einen entsprechenden Google-Server einbinden, der die Attestation signiert. Google Play Services leiten die Antwort dann an die App weiter, die sie an ihren Backend-Server übermittelt. Der detaillierte Ablauf ist in Abbildung 3 ersichtlich.

Die exakte Rolle des Attestation API Backend ist von Google nicht dokumentiert. Es ist unklar, weshalb hier nicht wie von Prünster et al. [7] vorgeschlagen der Attestierungs-Mechanismus von Key Attestation benutzt wird, dessen Signatur lokal (in der TEE) ausgestellt werden kann.

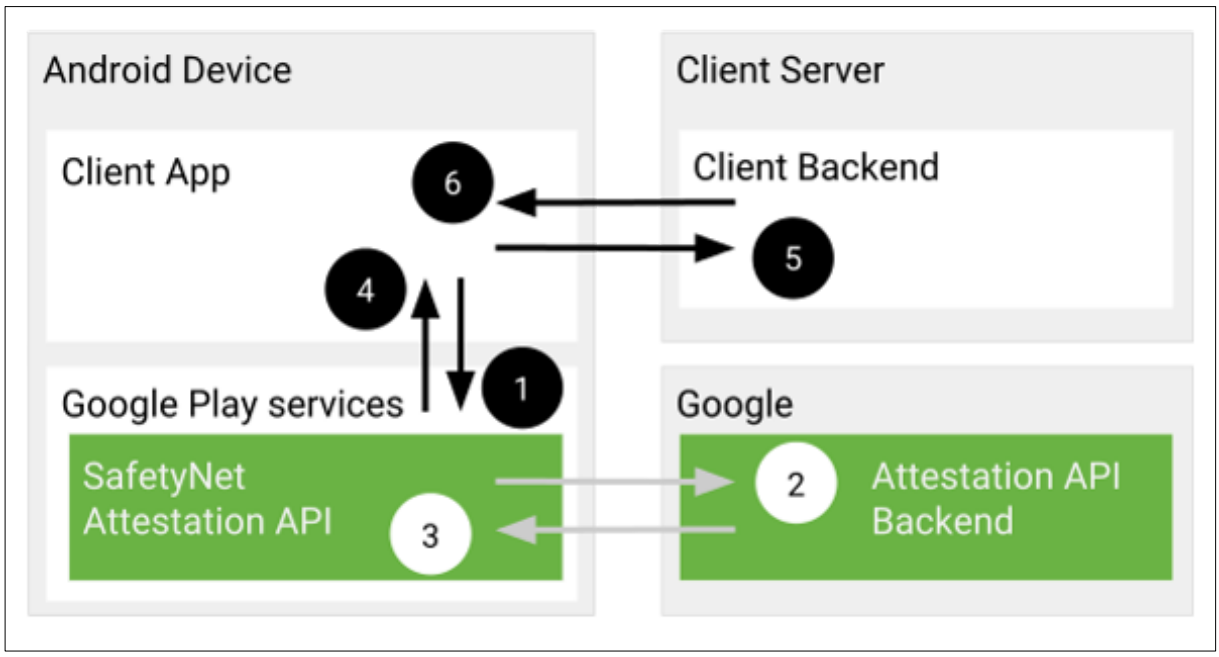

Abbildung 3 Google SafetyNet Remote Attestation (Quelle: Android Developers Documentation)[2]

Ibrahim et al. [8] fanden heraus, dass von 163773 untersuchten Anwendungen nur 62 (0,04 %) die SafetyNet Attestation API nutzten. Von diesen 62 Anwendungen werteten allerdings 32 (52 %) die Attestation lokal aus (leiteten sie also nicht an einen Backend-Server weiter), wodurch der komplette Attestierungs-Prozess von einem Angreifer einfach aus der Anwendung entfernt werden konnte.

Vermutlich als Reaktion auf die häufige falsche Verwendung der SafetyNet Attestation API stellte Google im Juni 2022 den Nachfolger in Form der Play Integrity APIs vor. Im Unterschied zur bisherigen Lösung handelt es sich beim Resultat der API nun um einen verschlüsselten Token, der im Standard-Anwendungsfall vom Backend-Server zur Entschlüsselung an einen Google-Server übermittelt wird. Der Ablauf wird in Abbildung 4 gezeigt. Obwohl laut offizieller Dokumentation die Validierung der Attestation lokal am Gerät nicht mehr möglich ist, entspricht dies nicht den technischen Tatsachen. Weiterhin ist es technisch möglich, den Google-Server zur Entschlüsselung des Tokens aus der App heraus aufzurufen.

[1] Android Developers Documentation: SafetyNet Attestation API. https://developer.android.com/training/safetynet/attestation

[2] Portions of this page are reproduced from work created and shared by the Android Open Source Project and used according to terms described in the Creative Commons 2.5 Attribution License.

Zusätzlich ist es Entwicklern möglich, den AES-Schlüssel zur Token-Dekodierung von Google anzufordern, sodass Tokens auch lokal entschlüsselt werden können. Entwicklern ist es somit möglich, den Token auch im Anwendungscode direkt zu entschlüsseln und zu validieren. Obwohl aus der Dokumentation schwerer ersichtlich, besteht also weiterhin die Möglichkeit, die gleichen Fehler zu begehen, wie mit der älteren SafetyNet Attestation API. Tatsächlich finden sich im Internet auch schon Beispiel-Projekte, die die unsichere lokale Entschlüsselung und Validierung verwenden [3].

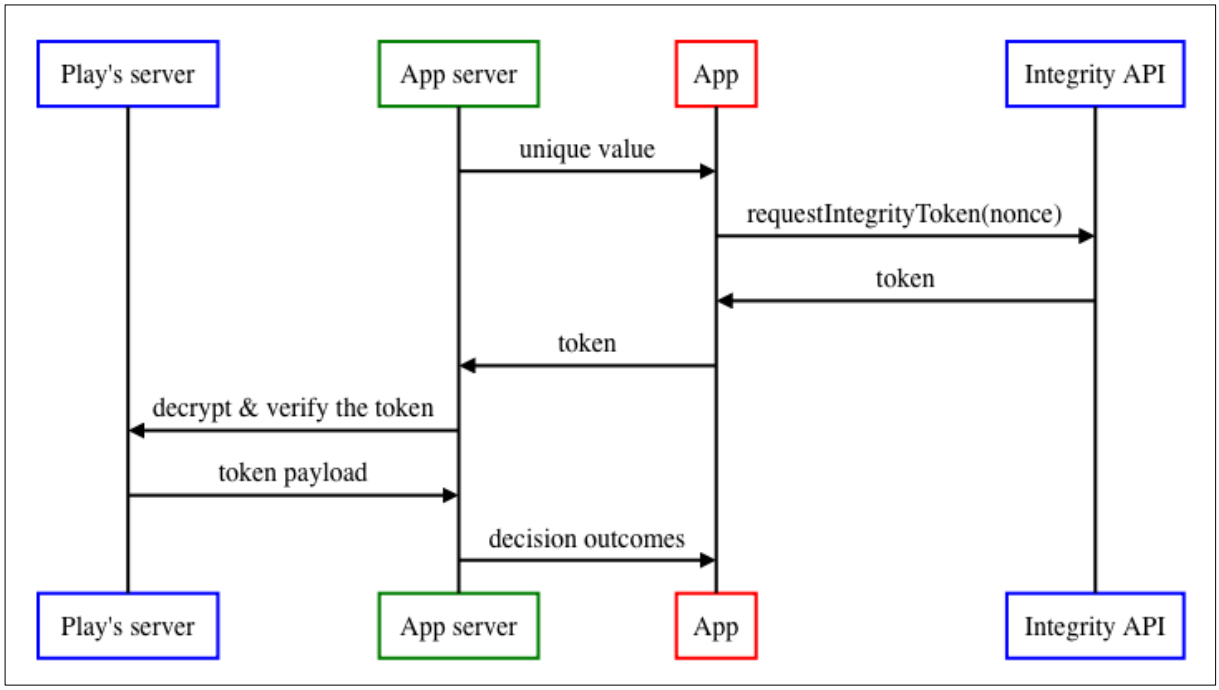

Abbildung 4 Google Play Integrity API Flow (Quelle: Android Developers Documentation) [4]

4. Prävalenz von Laufzeit-Integritätsprüfung

In jüngerer Vergangenheit haben einige wissenschaftliche Studien die Prävalenz von Laufzeit-Integritätsprüfungen in Android-Anwendungen untersucht. Berlato et al. [9] kamen dabei 2019 zum Ergebnis, dass von den beliebtesten 23610 Anwendungen im Play Store 88,8 % Signatur-Checks implementierten. Zirka 11 % der Anwendungen nutzten laut Berlato et al. SafetyNet Attestation. Es muss hier allerdings erwähnt werden, dass das statische Fingerprinting, das zur Analyse genutzt wurde, Schwächen aufweist. Es wurde z.B. nicht analysiert, ob eine implementierte Maßnahme zur Laufzeit tatsächlich ausgeführt wird. Für die Analyse der SafetyNet-Nutzung wurde nicht unterschieden, ob nur die Geräte-Umgebung oder auch die Anwendung selbst überprüft wurde. Bei Signatur-Checks wurde nicht bestimmt, ob Repackage Proofing implementiert wurde (ob ein negatives Ergebnis zur Funktionalitäts-Verweigerung führt).

Um zusätzliche Daten aus der Praxis zu bekommen, wurde im Rahmen dieses Projektes eine eigene Studie zur Prävalenz von Laufzeit-Integritätschecks in populären Android-Anwendungen durchgeführt.

[3] GitHub: Google Play Integrity Demo. https://github.com/chonkyboi-simon/google-play-integrity-demo
[4] Portions of this page are reproduced from work created and shared by the Android Open Source Project and used according to terms described in the Creative Commons 2.5 Attribution License.

4.1. Datensatz

Für die Analyse wurden aus 33 Kategorien im Google Play Store je die 3 populärsten Gratis-Anwendungen herangezogen. Falls eine Anwendung eine kostenpflichtige oder personengebundene Registrierung erforderte, wurde stattdessen die nächste Anwendung im Ranking herangezogen.

4.2. Test-Setup und Methodik

Für die Analyse wurde jede Anwendung einer Repackaging-Attacke ausgesetzt. Sie wurde dekompiliert, sodass Zugriffe auf die PackageManager API zur Zertifikat-Abfrage mit dem ursprünglichen Signatur-Zertifikat beantwortet wurden. So wurde sichergestellt, dass nur Apps erfasst werden, die keine triviale Integritäts-Checks enthalten. Das rekompilierte APK wurde dann mit einem neu generierten Zertifikat signiert und auf einem Google Pixel 3 mit Android 11 (Build RQ3A.210605.005) installiert. Ein menschlicher Anwender navigierte dann 3 Minuten lang durch die Benutzeroberfläche der App, und vollzog dabei den realistischen User-Flow eines wirklichen Nutzers nach. Falls dem Anwender hier eine Fehlermeldung in Bezug auf die Integrität der App auffiel, wurde davon ausgegangen, dass ein nicht-trivialer Signatur-Check implementiert wurde.

4.3. Ergebnis

Von den 99 untersuchten Anwendungen wurde lediglich in 3 Anwendungen (3 %) ein nicht-trivialer Integritäts-Check festgestellt. Es kann abgeleitet werden, dass 97 % der getesteten Anwendungen nicht wirkungsvoll gegen Repackaging geschützt waren. Über den tatsächlichen Sicherheitsgrad der 3 Anwendungen mit nicht-trivialem Integritäts-Check kann keine Aussage getroffen werden.

5. Zusammenfassung

In diesem Bericht beleuchteten wir die Problematik von App-Repackaging im Android-Ökosystem. Neben einer Beschreibung der technischen Ausgangsposition wurden verschiedene Möglichkeiten erläutert, mit denen Apps die Integrität der Laufzeit und ihres Programmcodes überprüfen können. Nach konzeptioneller Diskussion einer unüberwindbaren Implementierung für Remote Attestation wurde eine Studie zur Prävalenz von Repackage Proofing in aktuellen populären Android-Anwendungen präsentiert.

Referenzen

[1] S. Tanner, I. Vogels und R. Wattenhofer, „Protecting Android Apps from Repackaging Using Native Code," in *FPS 2019: Foundations and Practice of Security*, 2019.

[2] M. Protsenko, S. Kreuter und T. Müller, „Dynamic Self-Protection and Tamperproofing for Android Apps Using Native Code," in *International Conference on Availability, Reliability and Security, ARES*, 2015.

[3] Q. Zeng, L. Luo, X. Du, Z. Li, C.-T. Huang und C. Farkas, „Resilient User-Side Android Application Repackaging and Tampering Detection Using Cryptographically Obfuscated Logic Bombs," *IEEE Transactions on Dependable and Secure Computing,* Bd. 18, Nr. 6, pp. 2582 - 2600, 2019.

[4] C. Ren, K. Chen und P. Liu, „Droidmarking: resilient software watermarking for impeding android application repackaging," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, Vasteras, 2014.

[5] H. Ma, S. Li, D. Gao, D. Wu, Q. Jia und C. Jia, „Active Warden Attack: On the (In)Effectiveness of Android App Repackage-Proofing," *IEEE Transactions on Dependable and Secure Computing,* Bd. 19, Nr. 5, pp. 3508 - 3520, 2021.

[6] S. D. Yalew, P. Mendonca, G. Maguire, S. Haridi und M. Correira, „TruApp: A TrustZone-based authenticity detection service for mobile apps," in *IEEE International Conference on Wireless and Mobile Computing, Networking And Communications (WiMob)*, Rome, 2017.

[7] B. Prünster, G. Palfinger und C. Kollmann, „Fides: Unleashing the Full Potential of Remote Attestation," in *Proceedings of the 16th International Joint Conference on e-Business and Telecommunications*, 2019.

[8] M. Ibrahim, A. Imran und A. Bianchi, „SafetyNOT: on the usage of the SafetyNet attestation API in Android," in *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, 2021.

[9] S. Berlato und M. Ceccato, „A large-scale study on the adoption of anti-debugging and anti-tampering protections in android apps," *Journal of Information Security and Applications,* Bd. 52, 2020.