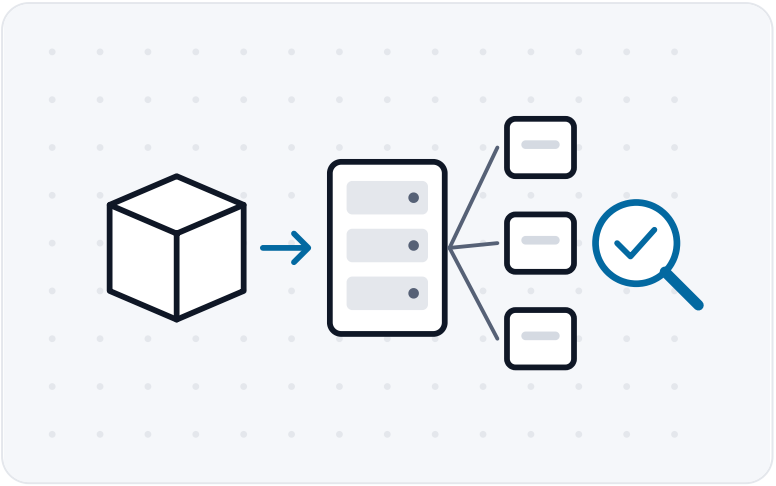
A-SIT Technology

E-Government Innovationszentrum

9. Mai 2023 Analyse · Transparenz · Mobile Sicherheit · 16 Min. Lesezeit

Analyse von Android Code Transparency

Florian Draschbacher



Autorinnen und Autoren	Florian Draschbacher
Stand	2023-04-25
Quelle	https://technology2.a-sit.at/veroeffentlichungen/analyse-von-android-code-transparency/

Kernaussagen

- Von 1.528.310 als AAB eingereichten Apps nutzten nur 21 Code Transparency, veröffentlicht von 6 Entwicklern.
- Bundletool erfasst DEX- und SO-Dateien in Assets und Ressourcen nicht; der Integritäts-Check ist dort außer Kraft gesetzt.
- Beim App Archiving signiert der Entwickler unbemerkt ein Google-DEX-File mit, dessen Funktionalität ihm unbekannt ist.
- Bundletool prüft nicht, ob für JWT-Signatur und APK-Signatur derselbe Schlüssel verwendet wird.
- Konzeptionell fehlen Manifest, JS-/HTML-Code und Konfigurationsdateien; diese Lücken stärken die Position der App-Store-Betreiber.

2018 hat Google ein neues Dateiformat, das Android Application Bundle (AAB) vorgestellt, in dem Entwickler Anwendungen zur Veröffentlichung an Google Play übermitteln können. Der Hauptunterschied zum bestehenden APK-Format ist, dass hier alle Ressourcen und Code-Module als Ganzes auf Googles Server geladen werden, wo danach erst die APK-Dateien für die Distribution auf Zielgeräte gepackt und mit dem Zertifikat des Entwicklers signiert werden. Dadurch kann der Endkunde mit einem schlanken APK beliefert werden, das nur genau die Ressourcen und Module enthält, die in der aktuellen Gerätekonfiguration (CPU, Bildschirmauflösung, Sprache) benötigt werden.

2021 verkündete Google dann, neue Anwendungen nur noch im AAB-Format entgegenzunehmen. Dadurch wurden vermehrt kritische Stimmen laut, die darauf hinwiesen, dass der neue Mechanismus Google zumindest theoretisch die Möglichkeit verleiht, Programmcode ohne das Wissen des Entwicklers zu manipulieren. Als Reaktion auf die Kritik stellte Google mit Code Transparency ein System vor, das es Entwicklern und Nutzern erlauben soll, die Integrität des ausgelieferten Programmcodes zu überprüfen.

Im Rahmen dieses Forschungsprojektes analysieren wir Code Transparency, um festzustellen, welche Möglichkeiten App-Store-Betreiber dennoch haben, Einfluss auf die Programmlogik von übermittelter Software zu nehmen.

1. Einleitung

Im Unterschied zu traditionellen Desktop-Betriebssystemen wurden die heute dominierenden mobilen Betriebssysteme für Smartphones und Tablets bewusst als relativ geschlossene Ökosysteme konzipiert. Als Argument und Grund für die Abgeschlossenheit wird in der Regel auf die sicherheitstechnische Sonderstellung dieser Gerätekategorie hingewiesen. Die Dichte an sensiblen Daten, die auf mobilen Geräten gesammelt und verwaltet werden, soll eine besonders strenge Kontrolle der Plattform-Anbieter bedingen. Eine besondere Rolle nahm hier traditionell der jeweils offizielle App-Store einer mobilen Plattform ein. Unter iOS ist der Apple App Store die einzige Möglichkeit, um Software zu beziehen. Unter Android kann der offizielle Google Play Store zwar umgangen werden, dennoch erhält er durch spezielle Lizenz-Abkommen mit Geräteherstellern eine privilegierte Position auf den meisten Android-Geräten.

Nachdem die marktdominierende Position der offiziellen App-Stores lange Zeit als Notwendigkeit akzeptiert worden war, regte sich im Laufe der letzten Jahre langsam Widerstand. Nach einiger Zeit des öffentlichen Diskurses wurde schließlich von der EU im Jahr 2022 der Digital Markets Act (DMA) verabschiedet, der die Vormachtstellung von Apple und Google brechen soll. Primäres Ziel dieser Bemühungen ist es, App-Entwickler aus dem Distributions-Monopol der öffentlichen App-Stores zu befreien, die für jede Transaktion (App-Verkäufe, In-App-Käufe und App-Abos) beträchtliche Kommissionen verlangen.

Bisher kaum beleuchtet wurde jedoch die technische Machtposition, die offizielle App-Stores innehalten. Unter iOS werden alle Apps von Apple signiert, sodass Apple seit jeher die technische Möglichkeit hat, Apps zwischen der Übermittlung vom Entwickler und der Auslieferung an den Kunden beliebig zu manipulieren. Da wesentliche Teile des Betriebssystems allerdings ohnehin unter vollständiger Kontrolle von Apple stehen (was im Wesentlichen hingenommen werden muss), ergibt sich daraus kein zusätzliches Sicherheitsrisiko.

Im Gegensatz sind die Machtverhältnisse unter Android deutlich differenzierter gestaltet. Obwohl der Play-Store-Betreiber Google auch die Entwicklung des Android Open Source Project (AOSP) vorantreibt, wird die Firmware jedes einzelnen Android-Geräts vom Geräte-Hersteller signiert (in der Regel also nicht von Google). Zusätzlich wurden Apps (genauer APK-Files) unter Android traditionell vom Entwickler signiert, sodass nach der Einreichung durch den Entwickler keine Möglichkeit bestand, eine App ohne Verlust der originalen Signatur zu manipulieren. Wenn Apps zur Laufzeit ihr Signaturzertifikat überprüfen (was bei sensiblen Apps routinemäßig implementiert wird), konnten Nutzer also sicher sein, die App so auszuführen, wie sie vom Entwickler an Google übermittelt worden war.

Diese Integritäts-Garantien für Anwendungen auf der Android-Plattform wurden 2018 mit der Einführung des optimierten Formats Android Application Bundles (AAB) abgeschafft. Apps werden nun von Google signiert, welches nun technisch dazu in der Lage ist, Apps vor der Auslieferung zu manipulieren. Um diese Gefahr zu bannen, führte Google gleichzeitig mit der verpflichtenden Nutzung von AAB für neue Apps 2021 Code Transparency ein. Hierbei wird vom Entwickler eine zusätzliche Signatur über alle Code-Bestandteile einer App erzeugt und in die App eingebettet.

In diesem Bericht wird analysiert, ob AAB mit Code Transparency dieselben Integritäts-Garantien erfüllt wie die frühere App-Distribution über vom Entwickler signierte APK-Files.

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 können Programme in eine Android Package (APK) Datei verpackt und so auf einem Android-Gerät installiert werden.

2.3. APK-Dateien

Das Android-System erlaubt die Installation von Anwendungen einzig im APK-Format. Dabei handelt es sich im Wesentlichen um ein ZIP-Archiv mit klar definierter Struktur. Zentrale Elemente innerhalb dieser Struktur sind:

Das Applikations-Manifest Eine vom App-Entwickler erstellte XML-Datei, die als „Vertrag“ zwischen App und Android-System betrachtet werden kann. Hier spezifiziert der Entwickler unter anderem die Permissions und Features, die die App zur Laufzeit benötigt, und gibt bekannt, welche Funktionalität die App dem Betriebssystem bzw. anderen Apps am System zur Verfügung stellt. Die häufigsten exportierten Komponenten sind hier Activities (UI-Bildschirme), ContentProvider (geregelter Zugriff auf Daten der App) oder Services (Hintergrundfunktionalität).

DEX-Dateien 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.

SO-Dateien (Native Bibliotheken) Entwickler können Teile einer App in nativen Programmiersprachen wie C und C++ entwickeln. Im APK-Paket werden diese dann als ELF Shared Object (Dateiendung .so) abgelegt. Mittels Java Native Interface (JNI) können sie aus Java-Code heraus aufgerufen werden.

Ressourcen Innerhalb eines APK-Pakets werden alle Ressourcen in einem zentralen Index-File (resources.arsc) eingetragen. Value-Daten, (Strings, Integers, Styles, ...) werden direkt im Index hinterlegt, während alle restlichen Daten (UI-Layouts, Bitmaps, Konfigurationen, ...) in separaten Dateien im APK-Paket abgelegt werden.

2.4. APK 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.

Als Sicherheitsmechanismus dienen App-Signaturen dennoch, indem das Betriebssystem sicherstellt, dass ein App-Update nur installiert werden kann, wenn es mit demselben Entwickler-Schlüssel signiert wurde, wie auch die zuvor installierte App-Version. So wird sichergestellt, dass ein bösartiges App-Update (von einem Angreifer) nicht auf die vom Nutzer in der App abgelegten Daten zugreifen kann. Bei diesen Daten könnte es sich unter Umständen um sensible Daten handeln, etwa um Zugangsdaten für einen Web-Service.

2.5. Android Application Bundles

Das APK-Format, das die Android-Plattform für die Installation auf Endgeräten nutzt, wurde lange Zeit auch als Distributionsformat genutzt, in dem Entwickler ihre Anwendung an Google Play übermittelten. Allerdings zeigte sich, dass das Format hier ineffizient war. Da der Entwickler in der Regel eine einzelne APK-Datei an Google Play übermittelte, mit der eine möglichst große Zahl an Android-Geräten unterstützt werden sollten, musste diese Ressourcen und native Bibliotheken für alle unterstützten Geräte enthalten. So benötigte die Datei etwa auf x86-Geräten Speicherplatz für native Bibliotheken, die nur auf ARM-Geräten tatsächlich verwendet wurden.

Um dieses Problem zu beheben, stellte Google 2018 ein neues Format namens Android Application Bundle (AAB) für die Distribution von Anwendungen vor [1]. Seit 2021 müssen neue Anwendungen verpflichtend dieses Format nutzen [2].

Das AAB-Format besteht wie auch das APK-Format aus einem ZIP-Archiv mit klar definierter Struktur der enthaltenen Dateien. Diese beinhalten kompilierten Programmcode in DEX- und SO-Dateien, sowie Ressourcen und Assets. Zusätzlich werden in Metadaten Details zum Erstellen von APK-Dateien aus dem AAB-File hinterlegt.

Nach dem Upload der AAB-Datei auf Googles Server werden dort optimierte APK-Pakete für die Installation auf konkreten Endgeräten generiert. Um hier nicht für jedes mögliche Gerät ein eigenes APK-Paket generieren zu müssen, werden Split-APKs erzeugt. Dazu wird das Paket in verschiedene Teile („Splits“) aufgeteilt, die jeweils in verschiedenen Varianten (optimiert für verschiedene Geräte) vorliegen. So liegt der Split für native Bibliotheken etwa für x86 und ARM-CPUs vor. Die App-Installation auf einem konkreten Gerät kann dann aus verschiedenen Splits (etwa die ARM-Variante für den Native-Bibliotheken-Split, Englisch für den Sprach-Split und XHDPI für den Bildschirm-Auflösungs-Split) zusammengestellt werden. Es werden zur Installation dann also mehrere APK-Dateien zusammen installiert.

Die Generierung von APK-Dateien auf Googles Servern wird offiziellen Angaben zufolge mittels bundletool [3] durchgeführt. Der Quellcode dieser Software wird unter einer Open-Source-Lizenz der Öffentlichkeit zur Verfügung gestellt, sodass auch Drittanbieter-Appstores das AAB-Format unterstützten können. Außerdem können App-Entwickler das Tool auch lokal verwenden, um aus ihren AAB-Dateien APK-Pakete für Testzwecke zu generieren.

2.6. Code Transparency

Als Ergebnis der Umstellung von APK auf AAB als Format zur Übermittlung von Apps an Google Play werden generierte APK-Dateien nun von Google signiert. Für neue Apps bedeutet das in der Praxis, dass Google das APK-Entwickler-Zertifikat erzeugt und anschließend verwaltet. Für bestehende Apps, die auf

AAB migrieren, müssen Entwickler den Schlüssel für ihr bestehendes Entwicklerzertifikat an Google übermitteln.

Durch diese Änderung befindet sich Google nun technisch in der Lage, Apps ohne Wissen des Entwicklers vor der Auslieferung an Endkunden zu modifizieren. Im Unterschied zum früheren Szenario, wo APK-Files schon vom Entwickler signiert und der Schlüssel nie an Google bekannt gegeben wurde, ist die APK-Signatur also gegen Angriffe von Seiten Googles (bzw. allgemein des App-Store-Betreibers) wirkungslos.

Um dieses Problem zu beheben, führte Google 2021 ein Schema namens Code Transparency ein [2] [4]. Dabei signiert der App-Entwickler DEX- und SO-Dateien innerhalb des AAB-Files mit einem Schlüssel, der nie an Google bekannt gegeben wird. Die Signatur wird in Form eines Javascript Web Token (JWT)-Files im AAB an Google übermittelt und in jedes generierte APK-Paket eingebettet. Zur Generierung des JWT-Files für ein AAB und zur Verifikation der Code Transparency für ein APK-Paket oder eine AAB-Datei wurde neue Funktionalität in bundletool integriert.

Um mit Code Transparency zu überprüfen, ob Modifikationen von DEX- oder SO-Dateien vorliegen, muss ein Entwickler die APK-Datei von Google Play herunterladen und lokal mittels bundletool überprüfen, ob das enthaltene Code-Transparency-JWT-File konsistent ist, d.h., dass die vermerkten Hash-Werte von DEX- und SO-Dateien mit denen der referenzierten Dateien übereinstimmen und das Signatur-Zertifikat der JWT-Datei dem Zertifikat des Entwicklers entspricht.

Erwähnenswert ist hier noch, dass die Code Transparency zu keinem Zeitpunkt vom Android-Betriebssystem geprüft wird oder überhaupt Voraussetzung für die Installation eines APK-Pakets ist.

Außerdem ist die Überprüfung der Code Transparency für Endnutzer zwar technisch möglich, allerdings müssen diese über einen zusätzlichen Informationskanal erst das legitime Zertifikat des Entwicklers erhalten.

3. Prävalenz von Code Transparency

Als ersten Schritt in der Analyse von Android Code Transparency erfassen wir in diesem Abschnitt die Prävalenz von Code Transparency unter den Anwendungen, die auf Google Play verfügbar sind.

3.1. Setup

Da Google weder einen offiziellen Datensatz aller Android-Anwendungen zu Forschungszwecken, noch einen Index aller Paketnamen zur Verfügung stellt, haben wir für unsere Analyse auf den Datensatz von AndroZoo [5] zurückgegriffen. Dieser enthält APK-Dateien von mehreren Millionen Anwendungen verschiedener Quellen. Zunächst mussten wir den Datensatz filtern, um unsere Analyse auf das Subset an Apps zu beschränken, die von Google Play bezogen wurden. Wir erhielten so eine Liste von über 6 Millionen Paketnamen, die zu irgendeinem Zeitpunkt auf Google Play verfügbar waren.

Weil AndroZoo kaum Metadaten enthält, haben wir im nächsten Schritt ein eigenes Tool entwickelt, das über die vom Aurora-Projekt [6] reverse-engineerte Schnittstelle App-Metadaten von Google Play herunterlädt.

Mittels der gewonnenen Metadaten konnten wir unseren Datensatz auf 3.265.096 Apps beschränken, die im April 2023 kostenlos von Google Play bezogen werden konnten.

3.2. Code Transparency Erkennung

Ob ein APK-File vom Entwickler mit Code Transparency ausgestattet wurde, lässt sich feststellen, indem der Inhalt des Archivs auf die Existenz einer Datei unter `/META- INF/code_transparency_signed.jwt` überprüft wird.

Um diesen Prozess effizient für 3 Millionen Apps ausführen zu können, haben wir ein Tool entwickelt, das jeweils nur das ZIP Central Directory eines APK-Archivs vom AndroZoo-Datensatz herunterlädt, um das Vorliegen vom JWT-File zu bestimmen.

3.3. Ergebnisse

Von den 3.265.096 Apps in unserem Datensatz wurden 1.528.310 gemäß ihren Metadaten von Entwicklern als AAB-Files an Google übermittelt. Nur 21 davon waren mit Code Transparency signiert worden. Diese 21 Apps wurden von 6 verschiedenen Entwicklern veröffentlicht. Außerdem lässt sich anhand von Ähnlichkeiten in Benutzeroberfläche und Funktion vermuten, dass 4 Apps vom selben Auftragsentwickler geschrieben und nur von unterschiedlichen Publishern veröffentlicht wurden, sodass in Summe nur von 4 verschiedenen Entwicklern gesprochen werden kann, die Code Transparency in ihre Builds integrieren.

Angesichts dieser alarmierenden Zahlen kann vermutet werden, dass die breite Masse der Entwickler entweder kein Bewusstsein für die Problematik haben, dass App-Store-Betreiber mit AAB technisch in der Lage sind, Apps zu manipulieren, oder sie sind nicht überzeugt von Code Transparency als Lösung dieses Problems.

4. Fehler in der Umsetzung von Code Transparency

Im Rahmen unserer Analyse von Android Code Transparency haben wir uns zunächst der Implementierung in Form des quelloffenen bundletool-Projekts gewidmet. Hier konnten wir drei Probleme feststellen.

4.1. Probleme mit DEX- und SO-Files in Assets und Ressourcen

Bei der Generierung des Code Transparency JWT-Files in einem Android Application Bundle werden nur DEX- und SO-Dateien berücksichtigt, die in den Standardpfaden für diese Dateiformate im AAB-File liegen. Damit sind jene Dateien abgedeckt, die von der Android-Runtime beim Start der App geladen werden. Apps können jedoch DEX- und SO-Files auch etwa in den Ressourcen oder Assets ablegen und zur Laufzeit von dort laden. Dies ist zum Beispiel bei Apps der Fall, die das Facebook Audience Network SDK einbinden.

Wenn ein AAB-File einer solchen App auf Google Play geladen wird, werden ohne Fehlermeldung APK-Dateien generiert. Beim späteren Versuch, die Code Transparency eines generierten APK-Pakets zu validieren, warnt bundletool allerdings vor einer Manipulation des APK-Files, da enthaltene DEX- oder SO-Files nicht in der Signatur erfasst sind.

Effektiv ist hier der Integritäts-Check außer Kraft gesetzt. Entwickler haben keine Kontrolle über die vom Facebook Audience Network (oder anderen ähnlich aufgebauten Bibliotheken) in den Assets ausgelieferte DEX-Datei, sodass sie den Fehler in der Validierung von Code Transparency hinnehmen müssen. Einem Angreifer (mit Kontrolle über die Infrastruktur des App-Store-Betreiber, etwa Google), wäre es möglich, die DEX-Datei zu manipulieren, ohne dass sich eine Änderung in der Code-Transparency-Validierung ergeben würde.

4.2. App Archiving

Seit Dezember 2022 erlaubt Google Play Nutzern die Archivierung einer App-Installation [7]. Dabei wird das installierte APK-Paket durch einen minimalen Platzhalter ersetzt, der den gleichen Paketnamen trägt und mit dem gleichen Entwickler-Key wie die volle APK-Datei signiert ist und als einzige Funktionalität den Wiederdownload der vollen APK-Datei anstößt. Ein Nutzer kann selten genutzte Apps archivieren, um den Speicherplatz der vollen APK-Datei einzusparen, ohne dabei die in der App abgelegten Dateien zu verlieren.

Um den Wiederdownload der vollen APK-Datei zu implementieren, muss die Platzhalter-App ein DEX-File enthalten, das nicht vom eigentlichen App-Entwickler, sondern von Google kommt. Während die ersten Versionen der Archivierungs-Funktion dazu Code Transparency einfach ignorierten, fand Google schließlich eine Lösung, um Code Transparency trotz App Archiving zu unterstützen.

Dazu wird das DEX-File der Platzhalter-App in bereits kompilierter Form als Ressource von bundletool ausgeliefert. Wenn ein Entwickler ein Code Transparency JWT für ein AAB erstellt, signiert er nun implizit (ohne darauf aufmerksam gemacht zu werden) ein DEX-File mit, dessen Quellcode und genaue Funktionalität ihm nicht bekannt sind. Es kann hier also davon gesprochen werden, dass das ursprüngliche Ziel von Code Transparency (sicherzustellen, dass APK-Dateien den vom Entwickler bereitgestellten Code enthält) verfehlt wird.

4.3. Wiederverwendung des APK-Zertifikats

Für die Effektivität des Code Transparency Verfahrens ist essenziell, dass der Signaturschlüssel des JWT-Files dem App-Store-Betreiber (meist Google) nicht bekannt ist. In der Dokumentation für Code Transparency weist Google deshalb darauf hin, dass für Code Transparency und APK-Signatur nicht derselbe Schlüssel verwendet werden darf.

In der bundletool-Implementierung wird diese Einschränkung allerdings nie überprüft. Bei der Erstellung von APK-Dateien sollte die Implementierung überprüfen, ob der übergebene Signaturschlüssel dem der JWT-Signatur entspricht. In diesem Falle sollte die Operation abgebrochen werden.

5. Konzeptionelle Fehler von Code Transparency

Bei der konzeptionellen Untersuchung von Code Transparency zeigen sich bedeutende Schwächen.

5.1. Nicht-Erfassung des Android-Manifest

Das Code Transparency Schema beschränkt sich auf DEX- und SO-Files einer App. Damit wird das Android-Manifest im APK nicht erfasst. Dass diese Datei für die Funktionalität einer Anwendung eine zentrale Rolle spielt, ist aus der offiziellen Entwickler-Dokumentation[1] bereits augenscheinlich. Bei genauerer Analyse im Quellcode von Android lässt sich allerdings auch feststellen, dass die Datei nicht nur essentielle Informationen über die App enthält, sondern auch Kontrolle über den Classpath (Quelle der automatisch geladenen Klassen) und die Ausführung der App hat. Beispielsweise kann mittels Manifest-Einträgen ein DEX-File aus einem separaten APK-Paket in die App injiziert werden, sodass damit effektiv die Code Transparency umgangen werden kann.

5.2. Ausführbarer Code in Ressourcen

Auch wird von Code Transparency nicht berücksichtigt, dass Anwendungen ausführbaren Code neben SO- und DEX-Dateien auch in anderen Formen enthalten können. Erwähnenswert sind hier etwa Apps, die einen Teil ihrer Logik in Javascript implementieren, in JS- oder HTML-Dateien, die nicht von Code Transparency berücksichtigt werden. Die Code Transparency bleibt hier intakt, obwohl gravierende Änderungen der Programmlogik vorgenommen werden können.

5.3. Sicherheitskritische Rolle von Konfigurations-Dateien

Auch auf die Rolle von Konfigurations-Dateien in den App-Ressourcen muss hier eingegangen werden. In der Network Security Configuration kann eine App etwa hinterlegen, welchen CA-Zertifikaten für TLS-Verbindungen sie vertraut. Manipuliert ein Angreifer diese Datei, kann er ein eigenes Zertifikat hinterlegen, mit dem er MITM-Angriffe auf die Serverkommunikation der App ausführen kann.

Während für diese MITM-Angriffe der Angreifer neben der Möglichkeit zur Manipulation des APKs auch jene zum Abfangen der Netzwerkverbindung braucht (üblicherweise etwa durch Angriffe auf den Link Layer, wofür physische Nähe zum Opfer nötig ist), sind je nach App noch deutlich drastischere Angriffe auf Serverkommunikation mittels Konfigurations-Dateien möglich. Bei einigen Apps werden etwa URLs von Backend-Servern als String-Ressource hinterlegt. In diesen Fällen könnte ein bösartiger App-Store-Betreiber die URL auf einen vom Angreifer kontrollierten Proxy-Server umschreiben und so Netzwerkverbindungen ohne physische Nähe zum Opfer abhören.

6. Zusammenfassung

In diesem Bericht beleuchteten wir das neue Distributionsformat Android Application Bundle und das dafür entwickelte Code Transparency Schema im Hinblick auf Sicherheitsaspekte. Neben einer ausführlichen Beschreibung der technischen Umsetzung und der konzeptionellen Funktionsweise wurde die Prävalenz von Code Transparency in Apps auf Google Play erfasst. Außerdem wurde auf konzeptionelle Schwächen von Code Transparency hingewiesen, die insgesamt die Position von App-Store-Betreibern im Android-Ökosystem stärken und dadurch Nutzern und App-Entwicklern potentiell schaden können.

Referenzen

[1] T. Lim, „I/O 2018: Everything new in the Google Play Console,“ 08 05 2018. [Online]. Available: https://android-developers.googleblog.com/2018/05/io-2018-everything-new-in-googleplay.html. [Zugriff am 2023].

[2] D. Elliott, „The future of Android App Bundles is here,“ 29 06 2021. [Online]. Available: https://android-developers.googleblog.com/2021/06/the-future-of-android-app-bundles-is.html. [Zugriff am 2023].

[3] Google, „Bundletool is a command-line tool to manipulate Android App Bundles,“ [Online]. Available: https://github.com/google/bundletool.

[4] Google, „Code transparency for app bundles,“ 28 07 2021. [Online]. Available: https://developer.android.com/guide/app-bundle/code-transparency. [Zugriff am 2023].

[5] K. Allix, T. Bissyande, J. Klein und Y. Le Traon, „AndroZoo: Collecting Millions of Android Apps for the Research Community,“ Proceedings of the 13th International Conference on Mining Software Repositories, 2016.

[6] A. Gupta, „Aurora Apps Open-Source Software. Inspired by you. Built for the community.,“ [Online]. Available: https://auroraoss.com. [Zugriff am 2023].

[7] L. Gaymond und V. Amin, „Freeing up 60% of storage for apps,“ 08 03 2022. [Online]. Available: https://android-developers.googleblog.com/2022/03/freeing-up-60-of-storage-for-apps.html. [Zugriff am 2023].

---

1. https://developer.android.com/guide/topics/manifest/manifest-intro ↩