Institut für Informationssicherheit

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Analyse der Android-Sicherheitsarchitektur

Carola Krug

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. Ralf Küsters
Betreuer/in:	Dipl.-Inf. Daniel Fett
Beginn am:	7. November 2017
Beendet am:	26. April 2018

Kurzfassung

Android ist eine weit verbreitete Plattform für mobile Geräte, die nach Googles eigener Aussage von über zwei Milliarden Geräten genutzt wird. Dies verlangt nach einem ausreichenden Schutz der Gerätefunktionen und der Daten der Benutzer. Aus diesem Grund wird in dieser Arbeit die Android-Sicherheitsarchitektur analysiert.

Nach einer kurzen Einführung zur Android-Architektur und einer Übersicht der Sicherheitskonzepte geht die Arbeit vor allem auf die Sicherheit des Linux-Kernels, SELinux und Permissions ein. Auch die Binder-Interprozesskommunikation wird näher erläutert.

Der Linux-Kernel setzt dabei eine benutzerbasierte Prozessisolation und Zugriffskontrolle um. Dieser Mechanismus wird durch SELinux erweitert, wodurch die Möglichkeit besteht, Zugriffsrechte genauer zu definieren. Zur Zugriffskontrolle werden außerdem Permissions genutzt, die in verschiedene Sicherheitsstufen eingeteilt werden. Auf deren Zustimmung hat der Benutzer abhängig von der Sicherheitsstufe einen Einfluss. Trotz Prozessisolation müssen Prozesse miteinander kommunizieren können. Dazu ist in Android ein eigener Mechanismus, Binder, implementiert, der Nachrichten zwischen Prozessen über den Kernel leitet.

Durch die Analyse dieser Komponenten der Android-Sicherheitsarchitektur wird verdeutlicht, wie Prozesse gegeneinander abgeschottet werden, der Zugriff auf Ressourcen gesteuert wird und trotz Prozessisolation eine sichere Kommunikation zwischen Prozessen möglich ist.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Abkürzungsverzeichnis

ACL Access Control List. 40

adb Android Debug Bridge. 22

AIDL Android Interface Definition Language. 79

AOSP Android Open Source Project. 15

AOT Ahead-of-time compilation. 23

API Application Programming Interface. 19

APK Android Package. 27

ART Android Runtime. 19

AVC Access Vector Cache. 54

DAC Discretionary Access Control. 39

GID group identifier. 45

HAL Hardware Abstraction Layer. 22

HRT High-Resolution Timer. 21

IPC Inter-Process Communication. 35

JIT Just-in-time compilation. 23

JNI Java Native Interface. 19

LSM Linux Security Module. 7

MAC Mandatory Access Control. 33

MCS Multi-Category Security. 43

MLS Multi-Level Security. 42

NDK Native Development Kit. 19

OM Object Manager. 54

OOM-Killer Out-of-Memory-Killer. 20

POLA Principle of Least Authority. 39

RBAC Role Based Access Control. 44

RTC Real-Time Clock. 21

SDK Software Development Kit. 27

SELinux Security-Enhanced Linux. 15

TE Type Enforcement. 42

UID user identifier. 45

URI Uniform Resource Identifier. 72

VM Virtual Machine. 23

1 Einleitung

Android ist laut Aussage von Google [Rep17] auf über zwei Milliarden aktiven Geräten als Betriebssystem eingesetzt. Um die Daten der Benutzer und die Geräte vor Angriffen zu schützen, muss Android Konzepte bieten, die diese Sicherheitsanforderungen umsetzen.

Das Design von Android soll App- und Benutzerdaten sowie Systemressourcen schützen und Apps vom System, anderen Apps und dem Benutzer isolieren. Dazu bietet Android allen Geräten ein plattformweites Sicherheitsmodell. Diese Arbeit gibt zunächst einen Überblick über die Android-Architektur und deren Sicherheitskonzepte. Im weiteren Verlauf konzentriert sich die Analyse der Sicherheitsarchitektur auf die Umsetzung des *Sandboxings* von Apps, *Permissions* und eine sichere Interprozesskommunikation.

Sandboxing setzt die Isolation von Apps und Systemprozessen um. Dies beinhaltet nicht nur die Isolation der Prozesse, sondern auch die Abschottung ihrer Daten, die durch die Festlegung von Zugriffsrechten geschützt werden. Umgesetzt wird dies sowohl durch benutzerbasierte Mechanismen des Linux-Kernels als auch durch die Erweiterung des Kernels um Security-Enhanced Linux (SELinux). SELinux lässt eine feinkörnige Definition von Zugriffsregeln zu, die nicht nur auf dem Benutzer basiert. Des Weiteren wird die Zugriffskontrolle durch Permissions ergänzt, auf deren Zustimmung auch der Benutzer einen Einfluss hat. In Bezug auf die Isolation der Anwendungen und Prozesse ermöglicht Android eine sichere Interprozesskommunikation durch *Binder*. Vgl. [SOv][Rep17]

1.1 Android

Zunächst wurde Android von der Softwarefirma Android Inc. entwickelt, jedoch noch vor der Veröffentlichung einer ersten Version 2005 von Google aufgekauft. Die erste Version von Android mit API-Level eins wurde dann im Oktober 2008 vorgestellt. In den letzten zehn Jahren bis zum jetzigen Zeitpunkt wurde die Plattform weiter entwickelt, bis hin zum API-Level 27, das mit der Version Android 8.1 assoziiert ist. Vgl. [APILv][Vers][TH16, S. 958]

Android basiert auf dem *Android Open Source Project (AOSP)*, das von Google geleitet wird. Es bietet den zentralen Softwarestapel von Android, zu dem das Betriebssystem, Middleware und Open-Source-Apps gehören. Dadurch, dass das Projekt Open-Source ist, kann der Quellcode kostenlos heruntergeladen, verändert und weitergegeben werden. Damit können Gerätehersteller das AOSP durch Funktionen ihrer Hardwarekomponenten erweitern oder eigene Apps entwickeln, die auf ihren Geräten vorinstalliert zur Verfügung stehen. Vgl. [AndS][AFacts]

Da das AOSP beliebig verändert werden kann, gibt es das *Android Compatibilty Program* [Comp], das eine Definition für die Eigenschaft „Android-kompatibel“ bietet. Dies ist nötig, da Apps auf allen Android-Geräten nutzbar sein sollen. Wird aber das AOSP von Herstellern stark verändert, können

diese Geräte möglicherweise nicht mehr alle Apps verwenden. Zum Android Compatibility Program gehören sowohl die Definition der benötigten technischen Merkmale der Android-Plattform, als auch Tools, die diese Eigenschaften automatisch für die Entwickler überprüfen. Vgl. [AndS][AFacts]

Entwickler können Apps über verschiedene Marktplätze, welche „App-Stores“ genannt werden, veröffentlichen. Dort können Benutzer von Android-Geräten diese dann erwerben. Google bietet mit *Google Play* dafür auch eine Platform, die die am meisten genutzte ist. Über diesen Marktplatz wird mittels *Google Play Protect* weitere Sicherheit für Android geboten. Google Play Protect überprüft Apps regelmäßig automatisch, die dort verbreitet sind genauso wie auf dem Gerät bereits installierte Apps. Dadurch soll verhindert werden, dass böswillige Apps überhaupt verbreitet werden. Vgl. [Abt][SOv][PlPr]

Durch die Tatsache, dass Android ein Open-Source-Projekt ist, sollen viele Sicherheitsrisiken und -lücken früh erkannt werden. Dazu bietet Google ein Android-Sicherheitsteam, das nach dem Entdecken einer Sicherheitslücke diese schließt. Laut Google [Rep17] gab es 2017 dadurch keine öffentlich bekannte Sicherheitslücke, die nicht durch ein Update geschlossen wurde. Zusätzlich bietet Google das *Android Security Rewards Program* [ASR], das Belohnungen auf das Finden von Sicherheitsproblemen aussetzt. Vgl. [UpRes]

1.2 Ziel dieser Arbeit

Ziel der Arbeit ist es, die Sicherheitsarchitektur der Android-Plattform präzise zu beschreiben. Dabei soll analysiert werden, wie in Android verschiedene Prozesse gegeneinander abgeschottet werden (mit Unix-Techniken, SELinux und ggf. anderen Werkzeugen), wie Prozesse gegen Systemdienste abgeschottet werden, wie der Zugriff auf Hardware- und andere Ressourcen gesteuert wird (Benutzer-Sicherheitsabfragen bei Zugriff beispielsweise auf Kamera, Mikrofon und Kontakte), und wie Prozesse dennoch sicher miteinander kommunizieren können (Binder-IPC).

1.3 Aufbau der Arbeit

Das zweite Kapitel (siehe Kapitel 2 auf Seite 19) zeigt eine Übersicht über die Android-Architektur, wobei einzelne Komponenten näher beschrieben werden. Anschließend werden in Kapitel 3 (siehe Kapitel 3 auf Seite 27) noch die Applikationen und deren Bestandteile näher erläutert. Kapitel 4 (siehe Kapitel 4 auf Seite 33) gibt einen Überblick über die Konzepte der Android-Sicherheitsarchitektur. Die Umsetzung dieser Konzepte wird nachfolgend in den Kapiteln 5, 6 und 7 beschrieben. Dabei liefert Kapitel 5 (siehe Kapitel 5 auf Seite 39) Informationen über die Sicherheitsaspekte, die durch die Verwendung vom Linux-Kernel in Android umgesetzt werden. Dazu ist zunächst eine Übersicht über Zugriffskontrollmechanismen gegeben, deren Umsetzung durch den Linux-Kernel und SELinux anschließend verdeutlicht wird. Kapitel 6 (siehe Kapitel 6 auf Seite 65) beschäftigt sich mit den Android-Permissions, deren Management und Durchsetzung. Den grundlegenden Mechanismus zur Interprozesskommunikation beschreibt Kapitel 7 (siehe Kapitel 7 auf Seite 77). Dabei wird vor allem die Implementierung in Android erklärt, aber auch Sicherheitsaspekte werden angesprochen. Kapitel 8 (siehe Kapitel 8 auf Seite 93) diskutiert die zuvor erarbeitete Android-Sicherheitsarchitektur. Abschließend bietet Kapitel 9 (siehe Kapitel 9 auf Seite 97) eine Zusammenfassung und einen Ausblick.

1.4 Testumgebung

Alle in der Arbeit durchgeführten Versuche sind in AndroidStudio mittels Emulatoren ausgeführt. Dabei wurde AndroidStudio in der Version 3.0.1 verwendet und auf Windows10 ausgeführt. Die Zugriffe auf die Emulatoren erfolgte mit adb Version 1.0.9. Der Programmcode der Inventar-App, die zur Veranschaulichung in Beispielen genutzt wird, ist eigens entwickelt. Es wurden zwei verschiedene Emulatoren verwendet. Zum einen ein Emulator mit dem API-Level 16, der damit Android 4.1 ausführt, und auf einem Nexus 5X Gerät von Google mit x86-CPU operiert. Dieser enthält keine Google-APIs, so dass der PlayStore-Zugang nicht aktiviert ist. Im Folgenden ist dieser mit den Worten „in Android 4.1“ gemeint. Der zweite Emulator hat die gleiche Basis, setzt jedoch Androids API-Level 24 und damit Android 7.0 um. Abbildungen zu diesem werden im weiteren Verlauf mit „in Android 7.0“ bezeichnet.

2 Android-Architektur

Android ist für mobile Geräten designed und basiert auf einem Stapel von Open-Source-Software, der auf dem Linux-Kernel aufbaut. Abbildung 2.1 bildet die Android-Architektur ab. Das Schaubild orientiert sich an den Darstellungen von Elenkov [Ele14, S. 2] und Yaghmour [Yag13, S. 33]. Vgl. [PfAr]

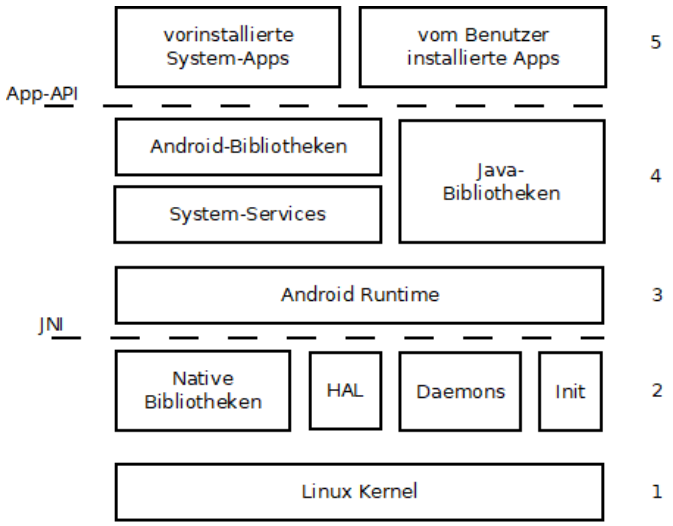


Abbildung 2.1: Android-Architektur

Die Basis des Stapels bildet der Linux-Kernel mit einigen Erweiterungen für die Nutzung von Android. Darauf aufbauend ist eine Schicht aus nativen Komponenten. Um diese mit den darüber liegenden Komponenten zu verbinden gibt es das Java Native Interface (JNI). JNI bildet eine Brücke zwischen Java und nativen Programmiersprachen wie C oder C++. Dies bietet die Möglichkeit aus Java-Code nativen Code aufzurufen und anders herum. Die Funktionalität des JNI benötigen die Java-Bibliotheken, die durch nativen Code ergänzt werden. Auch App-Entwickler können so, zuvor durch das Native Development Kit (NDK) kompilierten nativen Code, aus ihrem Java-Code aufrufen und in ihre App einbinden. Die dritte Schicht bildet die *Android Runtime (ART)*, die Androids Java-VM ART, zuvor bekannt als DalvikVM, und Zygote beinhaltet. In der in Abbildung 2.1 zusammengefassten vierten Schicht befinden sich System-Services sowie Java- und Android-Bibliotheken, welche den darüberliegenden Apps ein Application Programming Interface (API) bieten.

Auf die unteren vier Schichten der Android-Architektur wird im weiteren Verlauf des Kapitels eingegangen. Die Apps werden in Kapitel 3 auf Seite 27 genauer beschrieben.

2.1 Linux-Kernel

Der Linux-Kernel wurde als Grundlage des Android-Systems gewählt, da er wohl bekannt ist, Open Source und viele Systemfunktionen bereits implementiert. Dazu gehören Hardwaretreiber, Netzwerkfunktionen, Speicher-, Energie- und Prozessmanagement. Die Bekanntheit von Linux erlaubt es Herstellern eine einfache Entwicklung von Treibern für ihre Hardware. Außerdem können so die bereits bekannten Sicherheitseigenschaften des Kernels für Android übernommen werden, die in Abschnitt 5.2 auf Seite 45 ausgeführt werden. Vgl. [PfAr][Ele14, S. 2]

Der Linux-Kernel, auf dem das Android-System basiert wurde aus dem „vanilla“-Kernel entwickelt. Dies ist der Referenzkernel von Linux, der Standardkernel ohne Extras. Der Kernel wurde durch einige Funktionen, welche als *Androidisms* bezeichnet werden, erweitert. Dazu gehören der *Low Memory Killer*, *Wakelocks*, *Anonymous Shared Memory*, *Alarms* und *Binder*. Diese Erweiterungen werden anschließend näher betrachtet, während Binder in Kapitel 7 auf Seite 77 genauer untersucht wird. Vgl. [Ele14, S. 2][Yag13, S. 34 f.]

2.1.1 Erweiterungen des Kernels

Der Linux-Kernel wurde durch einige Funktionalitäten erweitert, die es Android vor allem ermöglichen, für mobile Geräte gut geeignet zu sein. Diese Erweiterungen werden nachfolgend beschrieben.

Wakelocks

Das Energiemanagement eines Standard-Linux-Kernels unterscheidet zwischen den zwei Zuständen wach und schlafend. Bei der standardmäßigen Benutzung eines Linux-Systems auf einem Rechner ist das System dabei meistens aktiv, solange der Computer nicht ausgeschaltet wird oder der Benutzer sehr lange Zeit inaktiv ist. Vgl. [TH16, S. 966]

Dieses Prinzip ist für mobile Geräte jedoch nicht optimal. Auf der einen Seite müssen sie viel mehr Energie sparen, da sie durch einen Akku mit Energie versorgt werden. Das bedeutet, dass sie so oft wie möglich in den schlafenden Zustand wechseln sollten. Auf der anderen Seite möchte der Benutzer keine Einschränkungen dadurch, dass das System ständig unerwünscht schläft, wenn beispielsweise der Bildschirm abgeschaltet wird. Der Benutzer möchte trotzdem Nachrichten empfangen oder Musik anhören können. Vgl. [Yag13, S. 36][TH16, S. 966]

Dafür wurden so genannte *Wakelocks* entwickelt, die das System wach halten. Grundsätzlich befindet sich das System im schlafenden Zustand. Nur wenn eine Komponente ein Wakelock hält, bleibt das System aktiv. Diese werden entweder vom System gehalten, solange beispielsweise der Bildschirm an ist oder von Treibern oder Apps, solange diese ihre Operationen ausführen. Das System kann aus dem Schlafmodus durch Ereignisse wie einen zeitbasierten Alarm, ankommende Anrufe oder Hardwarekomponenten wie den An/Aus-Schalter wieder aufgeweckt werden. Vgl. [Yag13, S. 36][TH16, S. 966 f.]

Low Memory Killer

Der Linux-Kernel beinhaltet den *Out-of-Memory-Killer (OOM-Killer)*, der dann einschreitet, wenn der RAM voll ist und Prozesse zwangsläufig beendet werden müssen. Durch die Benutzung von Android durch mobile Geräte, muss Android darauf ausgelegt sein, wenig Speicher zur Verfügung zu haben. Dazu wurde der *Low Memory Killer* in den Kernel eingefügt. Da das Benutzererlebnis eingeschränkt werden kann, falls Apps im Vordergrund zu wenig RAM zur Verfügung haben, greift der Low Memory Killer im Gegensatz zum OOM-Killer bereits ein, wenn eine durch einen Parameter festgelegte Schwelle an Speichernutzung im RAM überschritten wird. Bei der Entscheidung, welcher Prozess beendet werden soll, werden den aktiven Prozessen Parameter zugeordnet. Diese basieren beim Low Memory Killer auf Informationen, die er vom Benutzerraum bekommt. Dadurch soll der optimale Prozess gefunden werden, der lange nicht benutzt wurde, viel Speicher freigibt und dessen Beendigung den Benutzer nicht beeinträchtigt. Vgl. [Yag13, S. 36 f.][TH16, S. 968 f.]

Anonymous Shared Memory

Anonymous Shared Memory bietet eine Möglichkeit Speicherbereiche mit anderen Prozessen zu teilen. Ein Prozess kann eine Region des geteilten Speichert generieren und eine Referenz mittels Binder-Interprozesskommunikation an andere Prozesse weitergeben. Beim Anonymous Shared Memory werden Speicherbereiche durch Referenzzählung überwacht, so dass sie automatisch freigegeben werden, wenn kein Prozess mehr eine Referenz auf diesen besitzt. Außerdem bietet es die Möglichkeit den Speicherbereich zu verkleinern, falls das System ihn anderweitig benötigt und er ungenutzt ist. Vgl. [Yag13, S. 40 f.]

Alarm

Der Alarm-Treiber ist auf der *Real-Time Clock (RTC)* und dem *High-Resolution Timer (HRT)* des Kernels aufgebaut. RTC stellt ein Hardware-unabhängiges Interface bereit, während verschiedene Treiber zur Nutzung von Uhren als Hardwarekomponente existieren können. HRT bietet Aufrufenden die Möglichkeit, zu einem bestimmten Zeitpunkt aufgeweckt zu werden. Die Funktionalität des HRT ist jedoch nicht verfügbar, wenn das System in den schlafenden Zustand wechselt. Aktiviert eine App also einen Alarm mittels HRT und das System wird in den Schlafzustand versetzt, so wird die App erst benachrichtigt, wenn das System wieder aktiv wird. Der RTC-Treiber kann jedoch einen Alarm nutzen, welcher durch die Hardwaregeräte des RTCs aktiviert wird. Dadurch kann der Alarm auch im schlafenden Modus ausgelöst werden. Androids Alarm-Treiber kombiniert diese beiden Funktionen. Zunächst verwendet der Treiber die HRT-Funktion zur Umsetzung von Alarmen. Ist das System dabei sich zu suspendieren, wird RTC programmiert, um die Alarmfunktion zu übernehmen und das System aufzuwecken. Durch diesen Treiber können Apps einen Alarm nutzen, ohne sich über die Aktivität des Systems zu informieren. Vgl. [Yag13, S. 41 f.]

Logger

Das Auslesen von Systemprotokollen in Bezug auf Fehler oder Warnungen ist für Entwickler wichtig. Obwohl der Linux-Kernel Protokollmechanismen hat, sind diese in Android nicht zu finden. Android definiert seinen eigenen *Logger*, der auf einem Treiber basiert. Die Protokolle des Standard-

Linux-Kernels werden in einer Datei gespeichert. Anstatt die Logs in einer Datei festzuhalten, nutzt Android einen zyklischen Speicher im RAM, in dem jedes ankommende Event gespeichert wird. Dadurch wird Speicherplatz gespart, da vor allem keine unbestimmt und unbegrenzt wachsende Log-Datei existiert, sondern der Speicher sofern er voll ist, zyklisch überschrieben wird. Der Protokollmechanismus ist durch ein API für die Entwickler verfügbar, so dass der Log-Buffer ausgegeben werden kann. Vgl. [Yag13, S. 42 f.]

2.2 Native Komponenten

Zu den nativen Komponenten gehören sowohl Bibliotheken, als auch *Daemons* oder der *init*-Prozess. Android bietet außerdem das *Hardware Abstraction Layer (HAL)*, welches eine Schnittstelle für die Nutzung von Hardware bereitstellt.

2.2.1 Native Bibliotheken

Die nativen Bibliotheken von Android sind hauptsächlich in C/C++ geschrieben und werden von höher liegenden Systemkomponenten genutzt. Auch Apps werden einige Funktionalitäten zugänglich gemacht, auf die sie über JNI zugreifen können. Vgl. [PfAr]

2.2.2 Harware Abstraction Layer

Das *Hardware Abstraction Layer (HAL)* bietet Standardinterfaces, welche dem Java-API-Framework Zugriff auf die Hardwarekomponenten des Geräts wie Kamera oder GPS ermöglichen. HAL besteht aus mehreren Bibliotheksmodulen, die jeweils ein Interface zu einer Komponente implementieren. Vgl. [PfAr]

Im Standard-Linux-Kernel werden zur Unterstützung neuer Hardware Treiber geschrieben, die in den Kernel integriert oder zur Laufzeit geladen werden. Diese sind dann zur Laufzeit für den Benutzerraum durch Einträge im Dateiverzeichnis „/dev“ verfügbar. HAL ermöglicht nun einen standardisierten Zugriff auf Hardwarekomponenten, indem der Treiber des Geräts von der API, die dem Entwickler angebotenen wird, getrennt werden. Vgl. [Yag13, S. 46][Hal]

Das HAL-Interface besteht pro Gerät aus den zwei Komponenten Treiber und Modul. Dabei können Hardwarehersteller ihre Treiber flexibel implementieren. Die eigentliche Implementierung der Schnittstelle geschieht durch das Modul, das in der Regel durch ein „shared library module (.so)“ umgesetzt wird. Dieses nutzt die im Treiber implementierten Funktionen. Dabei definiert Android nicht, wie der Treiber und das Modul interagieren. Lediglich die API, die vom Modul den oberen Schichten bereitgestellt wird, ist durch HAL definiert. Im Modul werden Metadaten wie Version, Name und Autor des Moduls gespeichert, die von Android genutzt werden, um das entsprechende Modul zu finden und es zu laden. Die Kommunikation zwischen HAL und dem Android-Framework geschieht entweder durch einen System-Service, der zur Hardwarekomponente passt, oder kann seit Android 8.0 direkt durch Binder-Interprozesskommunikation erfolgen. Vgl. [Yag13, S. 46 ff.][Hal][HalT]

2.2.3 Daemons

Während des Systemstarts werden vom *init*-Prozess mehrere *Daemons* gestartet. Ein Daemon ist ein Programm, das im Hintergrund läuft und bestimmte Dienste zur Verfügung stellt. Diese laufen in der Regel solange das System nicht heruntergefahren wird, andere können auch bei Bedarf gestartet werden. Dazu gehört zum Beispiel der „adbd"-Daemon, der zur *Android Debug Bridge (adb)* gehört und abhört, ob ein angeschlossenes Gerät einen Shell-Zugriff benötigt. Auch *zygote* ist ein Daemon, der für das Starten von Prozessen zuständig ist und in Abschnitt 2.3.2 erklärt wird. Vgl. [Yag13, S. 59][StdAdb]

2.2.4 Init

Der Boot-Prozess von Android-Systemen beginnt wie in einem klassischen Linux-System mit dem Starten des *init*-Prozesses. Dies ist in Abbildung 2.2 zu sehen, da hier der Eintrag der Spalte „PPID", welche die Prozess-ID des Elternprozesses angibt, mit „0" bezeichnet wird. Außerdem bekommt init selbst die Prozess-ID „1", die in der Spalte „PID" steht. Der init-Prozess an sich unterscheidet sich jedoch von dem von anderen Linux-Systemen. Er startet die für Android spezifischen Daemons wie *zygote*. Auch eine Shell wird nicht ausgeführt, sondern der Daemon *adbd* gestartet, da Android-Geräten in der Regel keine Konsole für einen Shell-Zugriff zur Verfügung steht. Außerdem kümmert sich init um die globalen Systemeigenschaften, welche von mehreren Teilen des Systems verändert werden können. Zu diesen gehören Informationen über Netzwerkeigenschaften oder der Batteriestand. Vgl. [Yag13, S. 57 f.][TH16, S. 963 f.]

```
generic_x86:/ # ps
USER      PID   PPID  VSIZE  RSS     WCHAN              PC  NAME
root      1     0     8232   1468    SyS_epoll_ b5e5b424 S /init
```

Abbildung 2.2: Der *init*-Prozess aus der Liste aller aktiven Prozesse in Android 7.0

2.3 Android Runtime und Zygote

Da viele Bestandteile des Android-Systems, darunter auch Apps, in Java programmiert sind, benötigt Android eine Umgebung, die Java-Bytecode ausführen kann. Dazu wird die *Android Runtime (ART)* verwendet. Des Weiteren wird jede Anwendung in ihrem eigenen Prozess ausgeführt, wobei die Erstellung dieser Prozesse vom *zygote*-Daemon übernommen wird.

2.3.1 Android Runtime

Die *Android Runtime (ART)* ist seit Android 5.0 der Nachfolger der DalvikVM, welche eine Java-Virtual Machine (VM) für Android umsetzt. Der große Unterschied zur Java-VM besteht darin, dass ART keine „.class"-Dateien verarbeitet, sondern „.dex"-Dateien, die den Bytecode der ART darstellen. Dieses Bytecode-Format, welches aus dem Java-Bytecode kompiliert wird, ist speicherplatzeffizienter, so dass es für mobile Geräte geeignet ist. Vgl. [PfAr][ARTD]

ART unterstützt *Ahead-of-time compilation (AOT)* und *Just-in-time compilation (JIT)*. AOT kompiliert dabei schon bei der Installation einer App deren „.dex“-Dateien zu einer ausführbaren Datei. Dadurch kann die App ihre Performance beim Start verbessern, da Apps vor ihrer Ausführung nicht erst kompiliert werden müssen. JIT kann durch „code profiling“ dynamisch eine Programmanalyse durchführen, welche den Speicherplatzbedarf und die Performance analysiert. Dadurch kann die Laufzeitperformance von Apps durch JIT optimiert werden. Vgl. [PfAr][ARTD][ArtJ]

2.3.2 Zygote

Der *zygote*-Daemon ist für Android essentiell, da er alle Prozesse des Systems, welche in Java geschrieben sind, mittels ART erzeugt. Dazu gehören sowohl die zum Betriebssystem gehörigen Prozesse wie der Paket-Manager oder die Energieverwaltung, als auch Apps der Benutzer. Die Systemprozesse werden entweder automatisch oder bei Bedarf gestartet. Vgl. [TH16, S. 964]

Das Aufsetzen einer neuen ART dauert sehr lange und braucht viel Speicherplatz, da alle Ressourcen in den RAM geladen werden müssen. Dies soll auf mobilen Geräten vermieden werden. Dazu ist der Daemon zygote entwickelt, welcher für das Aufsetzen und Initialisieren der ART verantwortlich ist. Dies geschieht folgendermaßen: Zygote beinhaltet eine bereits aufgesetzte ART und lädt zusätzlich Teile des Android-Frameworks, die vom System oder Apps häufig verwendet werden, genauso wie Ressourcen. Vgl. [TH16, S. 970]

Ein neuer Prozess wird dann durch einen „fork“-Aufruf erzeugt. Der „fork“-Systemaufruf erzeugt eine Kopie des aufrufenden Prozesses, wobei der aufrufende Prozess zum Elternprozess des neu erzeugten Kindprozesses wird. Danach muss lediglich die Identität des neuen Prozesses geändert und der anwendungsspezifische Code, den die ART ausführen soll, geladen werden. Ein Vorteil dieses Prinzips ist, dass durch die Erzeugung der neuen Prozesse durch den „fork“-Aufruf, die von zygote im Voraus geladenen Klassen und Ressourcen im Speicher von zygote und allen ihren Kindprozessen geteilt werden können und nicht für jede Instanz einzeln in den RAM geladen werden müssen. Vgl. [Zyg][BC00, S. 30][TH16, S. 971]

Abbildung 2.3 zeigt Ausschnitte aus den aktiven Prozessen, welche mit dem Kommando „ps“ ausgegeben werden können. An zweiter Stelle jeder Zeile steht die Prozess-ID des Prozesses, während an dritter Stelle die ID des Elternprozesses ist. Dabei ist zunächst der Prozess zygote selbst zu sehen, der durch den Prozess mit der ID „1“, also den init-Prozess (siehe Abbildung 2.2), gestartet wurde. Zygote erhält selbst in diesem Fall die Prozess-ID 1337. Im Folgenden ist zu sehen, dass der Elternprozess dieser Anwendungen zygote ist. Dazu gehören Systemanwendungen wie im mittleren Teil von Abbildung 2.3 dargestellt sind, genauso wie von Benutzer installierte Apps, wie in diesem Beispiel die Inventar-App.

```
root      1337  1     1310528 69940 poll_sched b519a424 S zygote
u0_a12    2317  1337  1382476 36136 SyS_epoll_ b519a424 S android.ext.services
u0_a50    2360  1337  1384344 37132 SyS_epoll_ b519a424 S com.android.printspooler
u0_a14    2393  1337  1414492 61324 SyS_epoll_ b519a424 S com.android.launcher3
u0_a1     2494  1337  1393220 48448 SyS_epoll_ b519a424 S android.process.acore
u0_a2     2530  1337  1387532 40548 SyS_epoll_ b519a424 S com.android.providers.calendar
u0_a66    7983  1337  1404676 58432 SyS_epoll_ b519a424 S de.caro.inventar
```

Abbildung 2.3: Ausschnitt der Liste aller aktiven Prozesse, die im Zusammenhang mit *zygote* stehen, in Android 7.0

2.4 System-Services und Bibliotheken

System-Services bieten die elementaren Funktionen des Systems. Die Bibliotheken beinhalten die Grundbausteine für die System-Services und Apps. Die Dokumentation von Android umfasst eine Liste über alle verfügbaren Pakete der durch die Bibliotheken angebotenen API [PckId].

2.4.1 System-Services

Durch die System-Services werden die elementaren Features des Systems implementiert. Die meisten von ihnen definieren ein Interface, so dass das System und Apps die Funktionen des Service nutzen können. Vgl. [Ele14, S. 4]

Dazu gehören beispielsweise Services, die das Batterie-Management übernehmen oder die Sensoren der Geräte überwachen. Auch die Touchscreen-Funktionalität oder USB-Funktionen werden hier umgesetzt. Netzwerkverbindungen, Telefonfunktionen wie SMS oder die GPS-Funktion werden genauso durch Services verwaltet. Auch Android-Funktionen wie der Activity-Manager, der Informationen über Komponenten von Apps und ihren Prozessen gibt und mit diesen interagiert oder der Paket-Manager, der sich um die installierten Pakete der Apps kümmert werden durch System-Services umgesetzt.

Der Paket-Manager wird im weiteren Verlauf der Arbeit genauer erklärt. Abschnitt 3.4 auf Seite 31 erklärt seine Funktion während in Abschnitt 6.3.3 auf Seite 73 der Verbindungsaufbau einer App zu ihm näher erläutert wird.

Auf die System-Services kann über Binder-Interprozesskommunikation zugegriffen werden. Dazu können sie sich die Services beim Service-Manager registrieren, so dass andere Services oder Apps sie finden können. Die Funktionsweise von Binder wird in Kapitel 7 auf Seite 77 näher erklärt.

Zusammen mit dem Verknüpfen von Services durch Binder-Interprozesskommunikation bilden sie System-Services ein Objekt-orientiertes Betriebssystem oberhalb von Linux. Vgl. [Ele14, S. 4][Yag13, S. 63]

2.4.2 Java-Bibliotheken

Die Implementierung von Android in Java erfordert einige Bibliotheken, die hauptsächlich aus „java.*“- und „javax.*“-Paketen bestehen. Vgl. [PckId][Ele14, S. 4]

Die Java-Bibliotheken stammen urspünglich vom „Apache Harmony“-Projekt [ApHar], wurden aber an Android angepasst. Sie sind mehrheitlich in Java geschrieben, einige greifen aber auch auf nativen Code zu, das durch JNI möglich gemacht wird. Vgl. [Ele14, S. 4]

Die Java-Bibliotheken bieten zum Beispiel Input/Output-Unterstützung (java.io), Basisklassen von Java (java.lang), Netzwerk-Klassen (java.net), Möglichkeiten für Sicherheitsfunktionen wie Verschlüsselung (java.security), verschiedene Datenstrukturen und weitere Funktionalitäten (java.util). Vgl. [PckId]

2.4.3 Android-Bibliotheken

Die Android-Bibliotheken umfassen die „android.*“-Pakete. Sie sind in Java implementiert.

Alle Funktionen von Android sind durch die APIs der Android-Bibliotheken gegeben. Beispielsweise werden in den Paketen „android.os.*“ die Basis für System-Services oder die Interprozesskommunikation gelegt. Hier liegen zum Beispiel die Klassen der zuvor erwähneten Activity- [AM] und Paket-Manager [PM]. Auch die Basisklassen für Binder [RBin][RIBin], die die Interprozesskommunikation in Android bilden, sind hier zu finden. Vgl. [PckId]

Die Bibliotheken enthalten außerdem Pakete, die spezifisch für Android entwickelt sind, wie „android.app“, die die Basisklassen für Apps wie sie in Android genutzt werden, bieten. Auch Datenstrukturen und Funktionalitäten von „android.util“ sind auf Android zugeschnitten. Vgl. [PckId]

Auch Vereinfachungen zur Implementierung von Systemkomponenten, Services und Apps werden geboten. Ein Beispiel hierfür sind Klassen zur Erstellung der graphischen Benutzeroberfläche, die an Android-Geräte angepasst sind (android.view). Auch für den Zugriff auf Hardwarekomponenten oder andere Ressourcen wie Dateien gibt es Möglichkeiten. Vgl. [PckId][PfAr]

3 Applikationen

Android beinhaltet einige vorinstallierte Apps für das Senden von E-Mails oder SMS, eine Kalenderfunktion oder einen Internetbrowser. Zusätzlich können Benutzer selbst Apps installieren, die auch auf die System-Apps zugreifen und ihre Funktionalitäten nutzen können. Vgl. [PfAr]

Android-Apps können in den Sprachen Kotlin, Java oder C++ geschrieben werden. Der Code und weitere Ressourcen-Dateien werden dann vom Tool des Android-Software Development Kit (SDK) kompiliert. Das Ergebnis wird in einem *Android Package (APK)*, das eine Archiv-Datei mit der Endung „.apk“ ist, gespeichert. Dieses enthält alle Inhalte einer App und wird zur Installation der App verwendet. Vgl. [AppFun]

Applikationen sind aus verschiedenen Komponenten aufgebaut, die beispielsweise durch Intents aktiviert werden können. Die Informationen über die Komponenten und ihre Einstiegspunkte werden in der Datei *AndroidManifest.xml* festgehalten, die außerdem weitere Angaben zur App enthält. Die Verwaltung von Apps in Android übernimmt der Paket-Manager, der eine Datei als Paket-Datenbank führt. Diese Konzepte werden nachfolgend weiter beschrieben.

3.1 Komponenten von Applikationen

Applikationskomponenten sind die Bausteine für Apps, die teilweise abhängig voneinander sind. Sie bieten Einstiegspunkte in Anwendungen, durch die das System oder der Benutzer in die App gelangen kann. Jede Komponente erfüllt einen anderen Zweck. In Android-Apps werden die vier verschiedenen Komponenten *Activity*, *Service*, *Content Provider* und *Broadcast Receiver* verwendet. Vgl. [AppFun]

3.1.1 Activitiy

Activities sind der Hauptbaustein einer App, da sie typischerweise für die Interaktion mit dem Benutzer verwendet werden. Außerdem bietet eine Activity einen Einstiegspunkt in die App. Sie sind in der Regel durch ein Benutzerinterface ausgestattet, da sie ein Fenster bieten, in dem die App eine graphische Benutzeroberfläche darstellen kann. Vgl. [DAct]

Eine App kann mehrere Activities enthalten, die nur lose miteinander verbunden sind. In der Regel arbeiten sie zwar zusammen, können aber auch einzeln verwendet und gestartet werden. Dies ist vor allem bei Aufrufen aus anderen Apps nötig. Öffnet der Benutzer die Kontakte, so wird ihm zunächst eine Liste aller Kontakte angezeigt. Möchte er jedoch einen durch eine E-Mail erhaltenen Kontakt neu hinzufügen, so erwartet er beim automatischen Öffnen der Kontakte-App durch die E-Mail-App, dass direkt der Bildschirm zum Erstellen eines Kontakts angezeigt wird. Vgl. [AppFun] [DAct]

3.1.2 Service

Services sind App-Komponenten, die kein Benutzer-Interface bieten und länger ablaufende Operationen im Hintergrund ausführen können. Dazu gehört das Ausführen eines Downloads, das Abspielen von Musik oder eine Synchronisation von Daten. Diese Aktionen werden dadurch benutzerfreundlich, ohne aktive Apps zu blockieren, ausgeführt . Services werden von anderen Apps gestartet, um eine Aufgabe für sie zu erledigen oder um mittels Interprozesskommunikation zu interagieren. Es gibt zwei verschiedene Benutzungsweisen von Services. Vgl. [AppFun]

Zum einen kann ein Service mit einer Aufgabe gestartet werden wie zum Beispiel ein Download. Ist dieser abgeschlossen, so wird der Service vom System wieder beendet. Dabei muss unterschieden werden, ob der Service im Vordergrund oder Hintergrund läuft. Dies ist insofern wichtig, da vordergründige Services wie zum Abspielen von Musik nicht unterbrochen werden dürfen. Services, die Operationen ausführen, die der Benutzer nicht direkt bemerkt, wie zum Beispiel eine tägliche Synchronisation der Kalenderdaten, können im Hintergrund laufen. Diese können dann auch kurzzeitig unterbrochen werden, da dies den Benutzer nicht beeinflusst. Vgl. [AppSer]

Auf der anderen Seite können sich Komponenten einer App an einen Service binden. Dabei können sich auch mehrere Komponenten gleichzeitig an ihn binden. Dadurch muss der Service aktiv bleiben, bis keine App oder das System die Funktionalitäten mehr nutzen möchten. Ein gebundener Service bietet ein Klient-Server-Interface, über das die gebundene Komponente Anfragen senden, Ergebnisse empfangen und Interprozesskommunikation ausführen kann. Vgl. [DBS]

3.1.3 Broadcast Receiver

Broadcasts sind Nachrichten, die vom System oder anderen Apps beim Eintritt eines bestimmten Ereignisses versendet werden. Die meisten Broadcasts werden jedoch durch das System versendet. Dabei wird gemeldet, dass der Bildschirm ausgemacht wurde, der Batteriestand niedrig ist oder das Gerät jetzt geladen wird. Apps können auch Broadcasts initiieren, um anderen Apps Mitteilungen zu senden, beispielsweise dass ein Download beendet ist und nun die Daten zur Verfügung stehen. Apps können sich für bestimmte Broadcasts registrieren. Das System liefert beim Versenden eines Broadcasts diesen dann an alle Apps aus, die sich für den Typ von Broadcast registriert haben. Vgl. [DBc]

Die Broadcasts werden dann über einen *Broadcast Receiver* empfangen. Dies geschieht außerhalb des Benutzungsflusses. Da Broadcast Receiver auch Eintrittspunkte darstellen, können Broadcasts auch übermittelt werden, wenn die App gerade nicht aktiv ist. Dies kann für verschiedene Funktionen genutzt werden. Zum Beispiel soll eine App den Benutzer jeden Morgen an seine Medikamenteneinnahme erinnern. Wird diese Aufgabe an einen Broadcast Receiver übergeben, kann dieser zum gewünschten Zeitpunkt dem Benutzer eine Benachrichtigung darstellen, ohne dass die App dauerhaft laufen muss, um diese Funktion zu bieten. Vgl. [AppFun]

3.1.4 Content Provider

Ein *Content Provider* kümmert sich um den Zugriff auf eine definierte Menge von Daten. Apps können einen Content Provider definieren, um einen Teil ihrer Daten zu teilen. Durch diesen können andere Apps auf den Datensatz zugreifen und diesen auslesen oder verändern. Da es dadurch keinen

direkten Zugriff auf Daten gibt, können genaue Regeln des Zugriffs für fremde Prozesse von der Besitzer-App definiert werden. So bietet beispielsweise das Android-System einen Content Provider für den Zugriff auf die Kontakte des Benutzers an. Der Zugriff auf diesen wird gewährt, wenn die App vom Benutzer eine Erlaubnis dafür erhalten hat. Vgl. [AppFun]

Während sich auf der Seite der App, die ihre Daten teilen möchte, der Content Provider befindet, hat der darauf zugreifende Prozess einen Content Provider-Klienten. Diese zwei Komponenten stellen ein Interface bereit, in dem sie sich sowohl um Interprozesskommunikation als auch um sicheren Datenzugriff kümmern. Das Interface des Content Providers hat einen standardisierten Zugriff auf seine zugrundeliegenden Daten. Dadurch kann die Art der Datenspeicherung, beispielsweise in einer Datenbank oder als Datei, einfach ausgetauscht werden, ohne dass der Zugriff durch andere Apps beeinflusst wird. Vgl. [AppCP][AppCPB]

3.2 Intent

Intents ermöglichen die Kommunikation zwischen Komponenten in vielerlei Hinsicht. Die drei grundsätzlichen Einsatzbereiche sind jedoch das Starten einer Activity, eines Services oder das Ausliefern eines Broadcasts. Vgl. [DInt]

Ein Intent ist ein Nachrichtenobjekt, das für asynchrone Nachrichtenübertragung genutzt wird. Da es häufig zum Starten von Komponenten verwendet wird, enthält es Informationen über die entsprechende Komponente und Informationen, die die Komponente zur Ausführung der gewünschten Aktion benötigt. Trotzdem ist das Intent-Objekt an sich eine eine passive Datenstruktur, die lediglich eine Beschreibung der auszuführenden Aktion enthält. Für Activities und Services beschreibt die Aktion im Intent eine Operation, während ein Intent auch nur eine Nachricht, welche vom Broadcast Receiver empfangen wird, darstellen kann. Vgl. [RInt][DInt]

Bei den Intents wird zwischen zwei verschiedenen Arten unterschieden. Ein expliziter Intent enthält die Nachricht über eine Operation für eine ganz bestimmte Komponente. Die gewünschte Komponente wird entweder durch den Paket-Namen der App oder den direkten Klassennamen ausgewählt. Dies wird häufig innerhalb einer App verwendet, da der Entwickler die Namen der anderen Komponenten kennt. Vgl. [AppFun][DInt]

Implizite Intents hingegen benennen ausschließlich die Operation, die durch sie ausgeführt werden soll. Das Finden einer passenden Komponente, die in der Lage ist diese Aktion auszuführen, übernimmt dann das System. Wenn mehrere zur Verfügung stehen, wird die Entscheidung an den Benutzer weiter gegeben. Eine App kann in ihrer Manifest-Datei angeben, welche Operationen sie ausführen kann, so dass das System über diese Informationen eine zum Intent passende Komponente identifizieren kann. Vgl. [AppFun][DInt]

3.3 AndroidManifest-Datei

Jede Android-App benötigt eine Datei, welche alle Informationen, die für das System relevant sind, enthält. Diese ist im xml-Format gespeichert und trägt grundsätzlich den Namen „AndroidManifest.xml“. Ein Beispiel hierfür ist in Listing 3.1 zu sehen, welches einen Ausschnitt aus

der Manifest-Datei einer Inventar-App zeigt. Zunächst wird in der Datei der Paketname der App deklariert. Dieser identifiziert die App auf dem Gerät eindeutig und steht im <manifest>-Tag. Vgl. [AppFun][AppMan]

Außerdem wird basierend auf den APIs von Android, die die App benutzt, das minimale API-Level angegeben. Dieses bezeichnet die minimal benötigte Version von Android, die von der App für ihre korrekte und vollständige Funktion benötigt wird. Dies wird durch den <uses-sdk>-Tag definiert und ist im Beispiel „16“. Vgl. [AppMan]

Des Weiteren werden alle Komponenten der App mit ihren Klassennamen aufgeführt, so wie die Information, ob diese als Eintrittspunkt in die App genutzt werden können. Außerdem können hier Intent-Filter angegeben werden, die beschreiben, auf welche Art von Intents die entsprechende Komponente reagieren kann. Im Beispiel ist die Komponente vom Typ „Activity“ mit dem Klassennamen „MainActivity“ aufgeführt, die als Einstiegspunkt in die App dient. Vgl. [AppFun]

Auch die von der App erwünschten Permissions, die die Erlaubnis für bestimmte Systemfunktionen wie den Zugriff auf Kontaktdaten oder den Speicher bieten, müssen in der Manifest-Datei angegeben werden. Diese werden durch den <uses-permission>-Tag identifiziert, wobei Listing 3.1 die Erlaubnis für den Kamerazugriff fordert. Vgl. [AppFun]

Da sehr viele Geräte Android benutzen, jedoch alle unterschiedliche Funktionen und Möglichkeiten bieten, können die benötigten Geräte- und Softwareeigenschaften definiert werden. Dies wird in der Regel nicht vom System ausgelesen, sondern von App-Marktplätzen wie *Google Play*, die dadurch für die Benutzer nur zum Gerät passende Apps anbieten. Dazu wird der <uses-feature>-Tag genutzt. Vgl. [AppFun]

Zusätzlich gibt es noch weitere Tags, die beispielsweise verwendete Bibliotheken definieren oder andere Metadaten. Vgl. [AppFun]

Listing 3.1 AndroidManifest-Datei für die App „Inventar“

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3      package="de.caro.inventar">
4
5      <uses-sdk android:minSdkVersion="16" />
6
7      <uses-permission android:name="android.permission.CAMERA"/>
8                  ...
9
10     <application ... >
11         <activity
12             android:name=".MainActivity"
13             ... >
14             <intent-filter>
15                 <action android:name="android.intent.action.MAIN" />
16                                 ...
17             </intent-filter>
18         </activity>
19     </application>
20
21     <uses-feature android:name="android.hardware.camera" />
22 </manifest>
```

3.4 Verwaltung der Apps

Der Paket-Manager [CdPM] verwaltet alle Apps in Android. Dazu gehört die Installation, Updates oder auch die Deinstallation. In Android gibt es eine zentrale Datei, die alle installierten Pakete aufführt, sowohl vorinstallierte, als auch die vom Benutzer installierten. Diese Paket-Datei ist durch den codierten String „packages.xml“ [CdS, Zeile 439] definiert. Zu finden ist die Datei im Verzeichnis „/data/system/packages.xml“. Die Einträge dieser Datei werden jeweils nur bei Installation, Update oder Deinstallation eine App aktualisiert.

Abbildung 3.1 zeigt den entsprechenden Eintrag zur Inventar-App, deren Manifest in Listing 3.1 dargestellt ist. Wichtige Bestandteile dieser Einträge sind vor allem der Paketname (1) und die zugehörige Benutzer-ID (2). Des Weiteren wird hier auch die Version der App (3), sowie der Dateipfad der Applikation (4) angegeben. Ferner sind Informationen über die Signierung der App (5), die in Abschnitt 4.5 auf Seite 35 erklärt wird, und die Permissions (6), welche in Kapitel 6 auf Seite 65 genauer beschrieben sind, gegeben.

```
    </package>
    <package name="de.caro.inventar" codePath="/data/app/de.caro.inventar-1" nativeLibraryPath="/data/app/de.
caro.inventar-1/lib" publicFlags="944291654" privateFlags="0" ft="162a5ab26c0" it="162347f4506" ut="162a5ab33
aa" version="1" userId="10066">
        <sigs count="1">
         <cert index="4" key="308201dd30820146020101300d06092a864886f70d01010505003037311630140603550403 0c
0d416e64726f69642044656275673110300e060355040a0c07416e64726f6964310b3009060355040613025553301e170d31383033303
83130333630335a170d3438303232393130333630335a3037311630140603550403 0c0d416e64726f69642044656275673110300e0603
55040a0c07416e64726f6964310b3009060355040613025553308 19f300d06092a864886f70d010101050003818d00308189028181 00b
d83c75adeeb2b5a5c2c2e0496d228f66a5c6b31ef8fe811c1068992911b07d55966caa982ef39d9f6333d39734a3f33438ec7a9a05ce0
dee6e0dd1febd547bac0542df9067816a49b9b48b196a3a494a4ff60af0f82f595208ea23c5155d080c8ad457368bb2ef0bddf2f188af
da9d598dfa9829da6d3604464d83f46e3d6450203010001300d06092a864886f70d0101050500038181 00575df70151d3721dcf1a8981
b007d83804327150c14e0257ecbdd2e095420fc965f71b315da96ab76dd0a0986ee3d4b2479e575a600006280febdcfe405ed5ff9399b
251c9c7616bada846c56030eb6c649117f097fdf4171d635692 1fc19113fbd6395019455ad24cd788b20b4e8dc55a011d455e3130c095
f83754a21af032" />
        </sigs>
        <perms>
            <item name="android.permission.BLUETOOTH" granted="true" flags="0" />
        </perms>
        <proper-signing-keyset identifier="9" />
    </package>
```

Abbildung 3.1: Ausschnitt der Datei „packages.xml“ für die Inventar-App in Android 7.0

4 Android-Sicherheitskonzept

Android strebt danach, das sicherste und am besten benutzbare Betriebssystem für mobile Geräte zu sein. Dazu bietet es mehrere Sicherheitskonzepte, auf die im folgenden Kapitel und in weiteren Teilen der Arbeit genauer eingegangen wird. Dazu gehört die Sicherheit, die durch den Linux Kernel ermöglicht wird, die Erweiterung des Sandboxings von Apps durch Mandatory Access Control (MAC), sichere Interprozesskommunikation, Zugriffserlaubnis, die durch den Benutzer mittels Zustimmung zu Permissions gegeben wird und weitere Mechanismen. Vgl. [SOv]

4.1 Application Sandboxing

Das Prinzip *Sandboxing* zielt auf die Isolation von Daten und Code von verschiedenen Prozessen ab. Dabei soll weder Prozessen unberechtigt Zugriff auf Daten anderer Prozesse gegeben werden, noch soll willkürlich Code von anderen Prozessen ausgeführt werden können. Vgl. [TH16, S. 837][SysKS]

Android setzt Sandboxing auf der Ebene des Linux-Kernels um. Dadurch soll der Sicherheitsmechanismus nicht nur für die Apps der Benutzer, sondern auch für die komplette Software oberhalb des Kernels umgesetzt sein. Dazu gehören Bestandteile des Betriebssystems, Systemfunktionen, vorinstallierte Apps sowie Apps von Drittanbietern. Durch die Realisierung auf Kernel-Ebene muss der Kernel kompromittiert werden, um aus dem Rahmen der Sandbox auszubrechen. Vgl. [SeOv][SysKS]

Durch das Sandboxing läuft jede App als eigener Benutzer in einem einzelnen Prozess. So werden die Prozesse isoliert. Außerdem haben Apps dadurch keinen direkten Zugriff auf die Ressourcen anderer Apps oder des Systems. Dieser geschieht über das System, das den Zugriff überwachen und steuern kann. Somit können die Auswirkungen von Schadsoftware deutlich eingeschränkt werden. Vgl. [SC13]

Umgesetzt wird das *Application Sandboxing* in Android durch die UNIX-basierte Benutzerseparierung im Linux-Kernel, die zur Prozessisolation und Zugriffsverwaltung von Dateien genutzt wird. Verstärkt werden die Sandboxes durch SELinux. Vgl. [SysKS][Enh43]

Die Umsetzung des Sandboxings durch den Kernel wird in Abschnitt 5.2 auf Seite 45 näher beschrieben. Die Verbesserungen durch SELinux werden in Abschnitt 5.3 auf Seite 53 erklärt.

4.2 Security-Enhanced Linux

Die Sicherheit von Android-Systemen basiert auf dem zuvor beschriebenen Application-Sandboxing. Um dieses zu verstärken, wird seit Android 4.3 SELinux eingesetzt. SELinux ist eine Erweiterung des Linux-Kernels und bietet die Möglichkeit einer feinkörnigeren Zugriffskontrolle. Vgl. [SeOv]

Ein Unterschied zur Zugriffskontrolle des Linux-Kernels ist es, dass die Entscheidungen nicht basierend auf dem Benutzer getroffen werden, sondern alle Bestandteile des Systems mit verschiedenen Labeln gekennzeichnet werden, auf denen die Zugriffsentscheidung dann basiert. SELinux implementiert die Zugriffskontrolle außerdem so, dass alle Zugriffe explizit erlaubt werden müssen. Dies bedeutet, dass das *principle of least priviledge* verfolgt wird. Damit haben Prozesse nur ihre minimal benötigten Rechte und somit grundsätzlich erstmal zu wenig Rechte als zu viele. Vgl. [AppFun][SeOv]

Da die Entscheidungen über Zugriffe nicht benutzerabhängig sind, bietet SELinux die Möglichkeit auch Rechte von Systemprozessen, welche unter dem Benutzer „root“ laufen, und somit sehr privilegiert sind, genauer zu definieren. Damit werden die Grenzen der Sandboxen exakter bestimmt. Dies bietet eine Verbesserung der Prozess- und Datenisolation. Außerdem kann der Schaden von Angreifern oder Programmfehlern eingegrenzt werden, da auch Prozesse, deren Benutzer hohe Privilegien genießen, eingeschränkt werden können. Vgl. [SC13][SeOv][SysKS]

Eine genaue Beschreibung von SELinux und dessen Umsetzung wird in Abschnitt 5.3 auf Seite 53 geboten.

4.3 Permissions

Durch die Eingrenzung von Apps in Sandboxes, haben Apps sehr beschränkt Zugriff auf Ressourcen. Trotzdem benötigen einige Apps Hardwarekomponenten wie Kamera, Lokalisation oder Netzwerkzugriff oder Zugriff auf private Daten des Nutzers wie Kontakte oder SMS-Funktionen. Diese sind jedoch nicht frei zugänglich, da die vom Nutzer gewünschte Benutzung des Geräts durch unerwünschte Verwendung dieser Funktionen stark verändert werden kann. Damit ist gemeint, dass weder von einer App unerlaubt die Kamera geöffnet werden soll, obwohl der Benutzer gerade einen Text liest, genauso wenig wie das unerlaubte Auslesen von Netzwerkinformationen oder Kontaktdaten durch böswillige Apps. Diese Funktionen sind durch APIs nur über das Betriebssystem verfügbar, welche durch *Permissions* geschützt werden. Vgl. [AppSec]

Permissions werden je nach ihren möglichen Auswirkungen auf das System in verschiedene Sicherheitsstufen eingeteilt. Um Zugriff auf diese Ressourcen zu erhalten, müssen Apps in ihrer Manifest-Datei die gewünschten Permissions deklarieren. Sind diese dort nicht deklariert, so ist ein Zugriff grundsätzlich nicht möglich. Die dort aufgelisteten Permissions werden je nach ihrer Sicherheitsstufe automatisch erlaubt oder der Benutzer nach seiner Zustimmung gefragt. Je nach Android Version geschieht die Zustimmung des Benutzers zur Installations- oder Laufzeit. Wird vom Benutzer nicht zugestimmt, so ist ein Zugriff der App auf die gewünschte Ressource nicht möglich. Zur Verfügung stehen die vordefinierten Permissions für Systemressourcen, jedoch können Entwickler auch eigene Permissions für ihre Apps definieren. Vgl. [AppSec][PermOv]

Permissions bieten Apps damit Zugriff auf Systemfunktionen und private Daten unter Kontrolle des Betriebssystems und des Benutzers, der diesen teilweise erst zustimmen muss. Damit ist ein weiterer Zugriffsmechanismus vorhanden, der die Sicherheit verbessert. Der Sicherheitsmechanismus rund um die Permissions wird in Kapitel 6 auf Seite 65 beschrieben.

4.4 Interprozesskommunikation

Apps und andere Prozesse müssen trotz ihrer Sandboxen miteinander kommunizieren können. Dies ist durch Mechanismen zur *Inter-Process Communication (IPC)* möglich. Android bietet die Möglichkeit traditionelle UNIX-Techniken wie Dateien, lokale Sockets oder Signale zu verwenden. Dies wird allerdings in der Android Dokumentation im Bereich der Sicherheitsbeschreibungen [AppSec] und -empfehlungen [SecTip] nicht empfohlen. Stattdessen hat Android vier neue IPC-Mechanismen: *Intents*, *Services*, *ContentProviders* und *Binder*. Diese sollen mehr Sicherheit in Bezug auf den Schutz der Nutzerdaten und das Vermeiden des Ausnutzens von Sicherheitslücken ermöglichen. Vgl. [AppSec]

Intents stellen asynchrone Nachrichtenobjekte dar. Sie können sowohl Nachrichten übermitteln, als auch Aktionen, die die Empfängerkomponente ausführen soll. Diese werden entweder direkt an Komponenten einer App gesendet oder implizit mit einem Auftrag versandt, für den das System eine passende Komponente findet, die diesen ausführen kann. Eine ausführliche Beschreibung der Intents erfolgte in Abschnitt 3.2 auf Seite 29.

Durch Services können anderen Apps Funktionalitäten angeboten werden. Dies geschieht, indem Apps durch eine Binder-Referenz direkt auf sie zugreifen können. Dieser Zugriff kann jedoch durch die Definition von Permissions geschützt werden. Weitere Details zu den Services finden sich in Abschnitt 3.1.2 auf Seite 28.

Content Provider bieten ein Interface zu App-Daten, so dass auch andere Apps darauf Zugriff erhalten können. Dieser Zugriff ist automatisch nicht möglich, da die Daten von verschiedenen Apps durch Sandboxing geschützt sind. Damit bieten Content Provider eine Möglichkeit für Apps auch Daten zu teilen. Der Content Provider ist in Abschnitt 3.1.4 auf Seite 28 genauer beschrieben.

Binder ist der empfohlene IPC-Mechanismus in Android und liegt den anderen Möglichkeiten zu Grunde. Er kann für Aufrufe innerhalb eines Prozesses sowie zwischen Prozessen genutzt werden. Dieser Mechanismus wird durch Kapitel 7 auf Seite 77 erklärt. Vgl. [SecTip]

4.5 Signierung von Applikationen

Das Vorgehen der Code-Signierung ist dafür gedacht, zu verifizieren, dass eine Anwendung wirklich vom vermeintlichen Anbieter stammt. Dies ist vor allem bei Updates wichtig, da sich böswillige Software als Update einer bereits bekannten App tarnen könnten. Vgl. [AppSec]

Das Signieren von Code basiert auf der Public-Key-Kryptographie. Dabei hat der Entwickler ein Paar aus privatem und öffentlichem Schlüssel. Möchte er nun seinen Code signieren, so berechnet er dessen Hashwert durch eine öffentliche Hashfunktion. Danach verschlüsselt er dieses Ergebnis mit seinem privaten Schlüssel beispielweise durch das RSA-Verfahren. Der entstehende Wert wird

dann als Signatur bezeichnet. Der potentielle Benutzer der Anwendung berechnet dann seinerseits mittels der gleichen Hashfunktion den Hashwert des erhaltenen Codes. Außerdem entschlüsselt er die Signatur mit dem öffentlichen Schlüssel des Entwicklers. Sind die beiden berechneten Werte identisch, so stammt der Code tatsächlich vom erwarteten Entwickler. Vgl. [TH16, S. 750 f.]

In Android muss grundsätzlich jede App von ihrem Entwickler signiert sein, sonst kann diese nicht installiert werden. Dazu nutzt Android die APK-Datei, die zur Installation eine App verwendet wird. In ihr ist die Signierung vom Entwickler, so wie ein Zertifikat über die Identität des Entwicklers enthalten. Dieses Zertifikat enthält den öffentlichen Schlüssel für das Verschlüsselungsverfahren und andere Metadaten zur Identifikation des Entwicklers. Zunächst wird vom System durch die Signierung überprüft, ob das Paket nicht verändert wurde. Dann wird vom Paket-Manager entschieden, ob die zu installierende Anwendung bereits existiert, also nur ein Update erhält, oder neu auf dem Gerät ist. Bei der erstmaligen Installation der App folgt diese mit der Vergabe einer Benutzer-ID vom Paket-Manager. Im Falle eines Updates wird erst kontrolliert ob das Zertifikat der bereits installierten App, mit dem der APK-Datei übereinstimmt und dann das Update durchgeführt. Vgl. [AppSec][BCMV12][Stud]

Android nutzt die Signierung von Anwendungen in mehreren Bereichen zur Verbesserung der Sicherheit. Ein wichtiger Punkt ist, dass der Paket-Manager die Benutzer-ID auf Grund des Zertifikates vergibt. Damit wird der erste Grundstein der App-Sandbox gelegt. Gleichzeitig ist durch dieses Prinzip auch das Teilen einer solchen Benutzer-ID möglich, wie in Abschnitt 5.2.3 auf Seite 51 dargestellt ist. Auch für Updates von Apps bietet die Signierung den Benutzern von Android die Sicherheit, dass das Update vom tatsächlichen Entwickler der Anwendung ist. Des Weiteren werden die Zertifikate der Apps auch im Permission-Mechanismus genutzt, da es hier Sicherheitsstufen von Permissions gibt, die die Nutzung einer Ressource nur für Anwendungen mit dem gleichen Zertifikat erlauben. Dies wird in Abschnitt 6.1 auf Seite 66 erklärt. Vgl. [Sign]

4.6 Verified Boot

Android unterstützt seit Version 6.0 *Verified Boot*. Dies bedeutet, dass die Integrität der Gerätesoftware gesichert sein soll und durch dieses Verfahren bei jedem Start geprüft wird. Seit Android 7.0 wird dies soweit umgesetzt, dass kompromittierte Geräte gar nicht mehr starten können. Vgl. [Enh6][Enh7][SysKS]

Verified Boot wird durch das Linux-Kernel-Feature *device-mapper-verity*, kurz dm-verity, umgesetzt. Dieses prüft die Unversehrtheit von Speicherblöcken auf Basis eines zuvor berechneten Hashwerts. Die Hashwerte werden in einem Baum gespeichert, der in einem Blattknoten den Hashwert eines Blocks enthält. Die inneren Knoten besitzen einen Hashwert, der aus der Berechnung der Werte der Kindknoten entsteht. Vgl. [VeriB][Veri]

Während des Boot-Vorgangs wird die Integrität jeder Stufe des Prozesses zuerst von der darunter liegenden, zuvor geladenen, verifiziert. Dies geschieht dann bis hin zur System-Partition. Da die Berechnung der Hashwerte des ganzen Geräts zu lange dauern würden und viel Energie verbrauchen, geschieht die Verifikation von einzelnen Daten bei Bedarf während sie in den Speicher geladen werden. Vgl. [VeriB][SVB][SVBIm][Veri]

Da die zum Vergleich gespeicherten Hashwerte nicht unbedingt vor Veränderung geschützt sind, muss zumindest die erste Verifikation im Boot-Vorgang gesichert sein, so dass darauffolgend auch die gespeicherten Hashwerte an sich vorher überprüft werden. Dazu wird vom Hersteller ein unveränderbarer Schlüssel in das Gerät eingefügt, der für den ersten Bootloader gedacht ist, der wiederum die Signaturen der darüber liegenden Schichten, bis hin zum Kernel verifiziert. Schlägt die Überprüfung eines Speicherblocks fehl, so wird das Gerät entweder ausgeschaltet oder bei anderen Daten, die nicht die Boot-Partition betreffen, der Benutzer informiert, dass es unbekannte Änderungen an der Software gibt. Vgl. [VeriB][SVB][SVBIm][Veri]

Die Nutzung von Verified Boot mit dm-verity bietet dem Benutzer eines Android-Geräts die Sicherheit, dass das Gerät nach dem Booten im gleichen Zustand ist, wie bei der letzten Benutzung. Es hilft gegen persistente Rootkits vorzugehen, die dauerhaft „root“-Rechte besitzen und dem Gerät und den Daten schaden können. Diese Rootkits können sich oft vor Schadsoftware-Erkennungsprogrammen verstecken, da sie durch die „root“-Rechte besser als die Erkennungsprogramme ausgestattet sind und diese damit über ihre Identität „belügen“ können. Verändert solch ein Rootkit jedoch den Code des Systems, um permanent im Gerät zu bleiben, wird dies bei der Verifikation des Speichers erkannt. Alles in allem kann ein Benutzer durch diese Funktionalität über unerwartete Änderungen der Software seines Geräts informiert werden. Vgl. [VeriB][SVB]

4.7 Weitere Sicherheitskonzepte

Neben den zuvor erläuterten Sicherheitskonzepten bietet Android weitere Möglichkeiten, um die Sicherheit des Systems und der Daten zu verstärken.

Android trennt ganz klar im Speicher die Daten des Systems, wie den Kernel und verschiedene Bibliotheken, von Daten, die unabhängig davon sind oder veränderbar sein müssen, wie Einstellungen oder App-Daten. Die System-Partition ist jedoch grundsätzlich „read-only“, was bedeutet, dass kein Benutzer, auch nicht der „root“-Benutzer, die Zugriffsrechte hat, an diesen Daten etwas zu verändern. Vgl. [SysKS]

Zur Verschlüsselung von Daten bietet Android viele APIs, welche kryptographische Verfahren implementieren und die von Apps genutzt werden können. Außerdem ist seit Android 5.0 die Möglichkeit der *Full-disk Encryption* gegeben. Dazu wird die komplette Partition der Benutzerdaten durch einen Schlüssel verschlüsselt. Während dem Boot-Prozess muss der Benutzer diesen Schlüssel dann eingeben, bevor diese Partition zugänglich ist. Zudem gibt es seit Android 7.0 auch die Möglichkeit der *File-based Encryption*, durch die einzelne Dateien durch verschiedene Schlüssel geschützt werden können. Bei diesen Verfahren erfolgt die Kodierung der Daten automatisch, bevor sie in den permanenten Speicher geschoben werden. Genauso geschieht die Entschlüsselung automatisch nach der Eingabe es Schlüssels beim Laden der Daten für einen Prozess. Vgl. [SysKS][Ecry]

5 Linux-Sicherheit

Android basiert auf dem Linux-Kernel, der eine wichtige Grundlage der Sicherheitskonzepte von Android darstellt. Im folgenden Kapitel werden zunächst in Abschnitt 5.1 allgemeine Konzepte der Zugriffskontrolle beschrieben. Abschnitt 5.2 geht auf die Sicherheitsaspekte des Kernels ein während Abschnitt 5.3 SELinux genauer darstellt.

5.1 Konzepte der Zugriffskontrolle

In einem System gibt es viele Objekte, wozu sowohl Hardwarekomponenten (CPU, Speicher, Drucker) als auch Softwarekomponenten (Prozesse, Dateien, Datenbanken) zählen, die vor unberechtigtem Zugriff geschützt werden müssen. Dazu muss es ein Verfahren geben, welches klar definiert, welcher Prozess auf welches Objekt zugreifen darf und welche Operationen dabei erlaubt sind. Ein grundlegendes Prinzip hierfür ist das *Principle of Least Authority (POLA)*, bei dem es darum geht, nur die tatsächlich benötigten, minimalen Rechte zu verleihen. Vgl. [TH16, S. 727 f.]

5.1.1 Discretionary Access Control

Unter Verwendung von *Discretionary Access Control (DAC)* basiert die Entscheidung über einen Zugriff auf ein Objekt auf der Identität des darauf zugreifenden Subjekts. Mit dem Begriff Subjekt ist in diesem Fall in der Regel der Benutzer gemeint, es können jedoch auch Prozesse, Benutzergruppen oder Gruppen von Prozessen mit dem Begriff verbunden werden. Damit unterscheiden sich alle Zugriffsrechte beim DAC nur je Benutzer. Vgl. [SDV00][Spe08, S. 33]

Somit wird beim DAC-Mechanismus definiert, welche Operationen einem Subjekt in Bezug auf ein Objekt erlaubt sind. Beispielsweise ist einem Mitarbeiter (Subjekt) der Zugriff auf seine Präsentation (Objekt) mittels Lesen und Schreiben (Operationen) erlaubt, während er den Schichtplan (Objekt) ausschließlich Lesen (Operation) darf. Wie dies gespeichert wird, ist im folgenden Abschnitt erklärt.

„Discretionary“ bedeutet ins Deutsche übersetzt willkürlich oder beliebig [UEdis]. Diese Namensgebung für DAC ist darauf zurückzuführen, da Nutzern die Möglichkeit gegeben werden kann, ihre Rechte an andere Nutzer weiter zu geben. Vgl. [SDV00]

Umsetzung

Das Grundprinzip beim DAC ist das *Access Matrix Model*. Dabei sind die Zugriffsrechte in einer Matrix gespeichert. Die Spalten repräsentieren die zu schützenden Objekte während die Zeilen die Subjekte wiederspiegeln. Die paarweise zugehörigen Rechte an Operationen werden dann in den Feldern der Matrix gespeichert. Die Grundidee hierfür wurde von Lampson [Lam74] beschrieben und kurze Zeit später von Harrison, Ruzzo und Ullman [HRU76] verfeinert und weiter formalisiert. Vgl. [SDV00]

Abbildung 5.1 bietet ein Beispiel für solch eine Matrix. Hierbei sind die Subjekte Mitarbeiter Hr. Bauer, Abteilungsleiter und Chef als Zeilen aufgeführt, während zwei Dateien und ein Programm als Objekte in Spalten genannt sind. In den Zellen der Matrix sind die Rechte eingetragen, wobei keine Rechte, ein Recht oder mehrere Rechte möglich sind.

	Präsentation_Bauer.pdf	Schichtplan.xls	Steuerprogramm
Mitarbeiter Hr. Bauer	lesen/schreiben	lesen	
Abteilungsleiter		lesen/schreiben	
Chef		lesen/schreiben	ausführen

Abbildung 5.1: Beispiel einer *Access Matrix*

Das Speichern der Matrix im Ganzen verbraucht zu viel Speicherplatz, da auf ein ganzes System gesehen im Verhältnis zu den möglichen Einträgen nur sehr wenige gefüllt sind. Die Matrix kann durch Tabellen, Zugriffskontrolllisten oder Capabilities gespeichert werden, welche nachfolgend beschrieben sind.

Tabelle Als erste Möglichkeit bietet sich die Speicherung der nicht-leeren Matrixeinträge in einer Tabelle. Dabei wird in einer Spalte der Benutzer, dann das Objekt und in der dritten die erlaubte Operation aufgeführt. Somit gibt es eine Zeile pro Zugriffsrecht, so dass diese Daten als Tabelle in einer Datenbank abgespeichert werden können. Vgl. [SDV00][SS94]

In Abbildung 5.2 wird die Matrix aus Abbildung 5.1 als Tabelle gespeichert. Zu sehen ist hier vor allem, dass pro Zugriffsrecht ein Eintrag in die Tabelle gemacht wird. Mitarbeiter Hr. Bauer hat deshalb in Bezug auf die Datei Präsentation_Bauer sowohl eine Zeile für die Operation Lesen, als auch für die Operation Schreiben.

Subjekt	Objekt	Operation
Mitarbeiter Hr. Bauer	Präsentation_Bauer.pds	lesen
Mitarbeiter Hr. Bauer	Präsentation_Bauer.pdf	schreiben
Mitarbeiter Hr. Bauer	Schichtplan.xls	lesen
Abteilungsleiter	Schichtplan.xls	lesen
...	...	...

Abbildung 5.2: Beispiel einer Tabelle zur Speicherung von Zugriffsrechten

Zugriffskontrolllisten Bei der Umsetzung durch eine Zugriffskontrollliste, die so genannte Access Control List (ACL), wird die Matrix im übertragenen Sinn nach Spalten gespeichert. Dies bedeutet, dass für jedes Objekt eine Liste geführt wird, in der alle Subjekte gespeichert sind, die auf das Objekt zugreifen dürfen. Zusätzlich dazu werden für jedes Subjekt dann die entsprechend erlaubten Operationen aufgelistet. Vgl. [SDV00][SS94][TH16, S. 730 f.]

Möchte man bei dieser Umsetzung eine Auskunft über die Zugriffsrechte auf ein bestimmtes Objekt, so ist dies einfach durch das Auslesen der zugehörigen Zugriffskontrollliste möglich. Dies ist in Abbildung 5.3 zu sehen. Als Besitzer einer Datei ist es außerdem leicht möglich die Zugriffsrechte anzupassen, da nur die Liste verändert werden muss. Sollen jedoch alle Rechte eines Benutzers dargestellt werden, so müssen alle Listen der Objekte durchsucht werden, was sehr aufwändig ist.

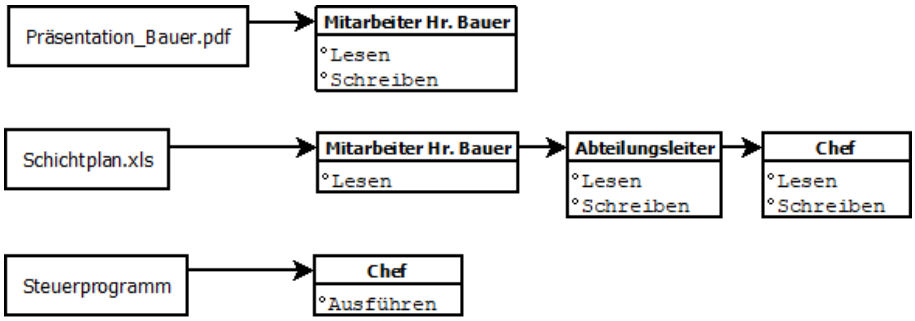


Abbildung 5.3: Beispiel von Zugriffskontrolllisten zur Speicherung von Zugriffsrechten

Capabilities Zur Speicherung der Zugriffsrechte können auch *Capabilities* genutzt werden. Dabei werden die nicht-leeren Einträge der Matrix nach Zeilen in so genannten *Capability-Listen* gespeichert. Für jedes Subjekt wird eine Liste erstellt, in der für jedes Objekt die entsprechenden Zugriffsrechte aufgeführt sind. Vgl. [SDV00][TH16, S. S.733]

Abbildung 5.4 zeigt die Umsetzung der Matrix als Capability-Listen. Möchte man in dieser Umsetzung alle Zugriffsrechte für ein Objekt abfragen oder verändern, stellt sich dies schwieriger dar, da alle Listen durchsucht und bearbeitet werden müssen. Alle Möglichkeiten eines Benutzers aufzuzeigen ist jedoch hier einfach.

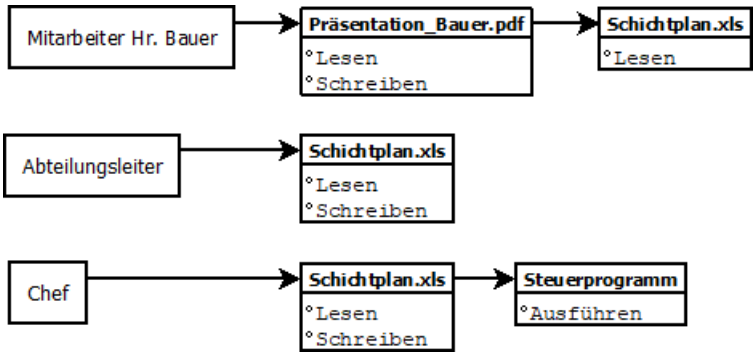


Abbildung 5.4: Beispiel von Capability-Listen zur Speicherung von Zugriffsrechten

Eine Capabilitiy wird im Allgemeinen als Token verstanden, der einem Subjekt den Zugriff auf ein Objekt ermöglicht. Es wird das Zielobjekt referenziert und eine entsprechende Menge an Operationen. Hat nun ein Benutzer die entsprechende Capability, kann er auf das Objekt zugreifen, unabhängig von seiner Identität. So können Zugriffsrechte weiter gegeben werden, in dem die Capability für einen anderen Benutzer kopiert wird. Außerdem muss die Capability-Liste selbst vor der Manipulation durch den Benutzer geschützt werden. Vgl. [SDV00][Ele14, S. 7]

Probleme

Probleme beim DAC ergeben sich dadurch, dass die Zugriffsverwaltung sehr grobkörnig ist. Benutzer und Prozesse werden nicht voneinander getrennt, so dass ein Prozess die Rechte des Benutzers erhält, der ihn startet. Systemprogramme, die unter der „root“-Identität laufen, haben uneingeschränkten Zugriff auf das System. Vgl. [SC13][SDV00]

Führt ein Benutzer einen Prozess aus, der Schadcode beinhaltet, so kann der bösartige Code alle Rechte des Benutzers ausnutzen. Außerdem werden für einige Aktionen wie die Passwortänderung vorübergehend „root“-Rechte benötigt, so dass Schadsoftware bei Sicherheitslücken in solchen Prozessen sogar „root“-Rechte erlangen können. Gelingt dies, ist der Zugriff auf das System und seine Funktionen schrankenlos.

Eine weitere Problematik ergibt sich dadurch, dass DAC nicht kontrolliert, was mit den Daten passiert, nachdem ein Nutzer sie eingesehen hat. Der Benutzer kann die Daten kopieren und an Benutzer weitergeben, die keine Zugriffsrechte besitzen. Bei der Umsetzung von DAC durch Capabilities können sogar die Rechte durch eine Kopie der Capability weiter gegeben werden. Vgl. [SS94][SDV00]

5.1.2 Mandatory Access Control

Die Entscheidung über eine Zugriffserlaubnis bei der Umsetzung von *Mandatory Access Control (MAC)* basiert auf der Entscheidung einer zentralen Autorität, die vorgeschriebene Regeln zur Zugriffskontrolle verwaltet. Der Mechanismus heißt *discretionary*, auf Deutsch obligatorisch oder vorgeschrieben [UEman], da die Zugriffskontrolle durch die zentrale Verwaltung unabhängig von den Benutzern geschieht und somit nicht umgangen werden kann. Dies ist der zentrale Unterschied zu DAC. Vgl. [SDV00][SeCon][LinOv]

Die Entscheidungen von MAC basieren auf der Klassifikation von Subjekten und Objekten im System. Subjekte können in diesem Fall alle bei einem Zugriff aktiven Komponenten wie Benutzer aber auch einzelne Prozesse oder Gruppen sein. Objekte sind die passiven Bestandteile wie Dateien, Geräte oder Prozesse. Allen Subjekten und Objekten ist ein Sicherheitskontext zugeordnet, auf dessen Basis Zugriffsentscheidungen getroffen werden. Der Sicherheitskontext unterscheidet sich in der Implementierung von MAC durch Multi-Level Security (MLS) oder Type Enforcement (TE). MLS und TE, auf welche in den nachfolgenden Abschnitten genauer eingegangen wird, können auch kombiniert werden. Da bei MAC die Zugriffsrechte feinkörniger gestaltet sind, in dem auch einzelne Prozesse andere Sicherheitseigenschaften als ihr Benutzer bekommen können, können Rechteausweitungen von Schadprogrammen deutlich eingeschränkt werden. Vgl. [SS94][SeCon]

Ein Beispiel für die Umsetzung von MAC ist SELinux, das im Abschnitt 5.3 auf Seite 53 genauer beschrieben wird.

Multi-Level Security/Multi-Category Security

Bei Multi-Level Security (MLS) und Multi-Category Security (MCS) wird die Zugriffsrechteeinteilung durch Sicherheitsstufen oder Kategorien abgebildet. Beide Mechanismen ordnen sowohl den Subjekten, als auch den Objekten eine Sicherheitsstufe oder Kategorie zu, auf deren Basis die Zugriffsentscheidung getroffen wird. Bei MLS geschieht die Einteilung in Stufen, die hierarchisch angeordnet, während bei MCS Kategorien ohne Ordnung definiert sind. Die Zugriffssicherheit bei MLS bezieht sich dann auf den „Top-down"- und den „Bottom-Up"-Informationsfluss. Die jeweilige Sicherheitsstufe der Objekte nennt sich Klassifizierung während die der Subjekte Freigabe genannt wird. Vgl. [Spe08, S. 34 ff.][Hai14, S. 48 f.]

Eine Umsetzung von MLS ist das Bell-LaPadula-Modell [BL76]. Dieses Modell verfolgt das Prinzip der Vertraulichkeit durch die Zugriffsregeln „no-read-up" und „no-write-down". Dies bedeutet, dass Lesen nur dann erlaubt ist, wenn die Freigabe des Subjekts gleich oder höher als die Klassifizierung des Objekts ist. Außerdem ist Schreiben nur dann gestattet, wenn die Klassifizierung des Objekts gleich oder höher als die Freigabe des Objekts ist. Somit können keine vertraulichen Daten in Stufen gelangen, deren Sicherheitslevel zu niedrig ist.

Als Beispiel für das Modell dient Abbildung 5.5. Durch die „top-down"-Leserichtung kann der Chef alle Objekte seiner Firma anschauen, wie etwa den Schichtplan oder den aktuellen Stand der Projekte seiner Mitarbeiter. Außerdem können die Mitarbeiter keine sensiblen Informationen wie die Gehaltsliste einsehen. Durch das „no-write-down"-Prinzip kann der Chef auch nicht aus Versehen diese Objekte für niedrigere Stufen freigeben, was die Vertraulichkeit der Daten schützt.

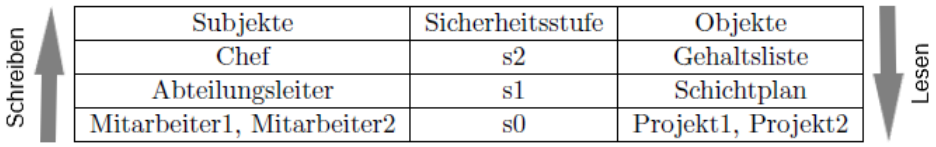


Abbildung 5.5: Beispiel einer Umsetzung des Bell-LaPadula-Modells

Gleichzeitig entstehen jedoch zwei Probleme. Zum einen kann der Schichtplan, den der Abteilungsleiter erstellt hat, weder von den Mitarbeitern eingesehen werden (no-read-up), noch vom Abteilungsleiter an die Mitarbeiter weitergegeben werden (no-write-down). Außerdem könnte ein Mitarbeiter eine manipulierte Gehaltsliste in die Sicherheitsstufe des Chefs speichern.

Type Enforcement

TE ist eine weitere Möglichkeit MAC umzusetzen. Es bedeutet, dass allen Subjekten und Objekten ein *type identifier*, auch Label genannt, zugeordnet wird, auf deren Basis dann die Zugriffsentscheidung getroffen wird. Die Typbezeichner, zu denen Subjekte zugeordnet werden können, werden auch als Domäne bezeichnet. Subjekte können während ihrer Ausführung auch die Domäne wechseln, falls dies durch das implementierte Regelwerk erlaubt ist. Vgl. [Hai14, S. 25][Spe08, S. 37 f.]

In Abbildung 5.6 sind drei Domänen und drei Objekttypen definiert. Dabei ist zu sehen, dass Subjekte ihre Rechte je nach Domäne haben. Im Beispiel hat der Chef durch die Zuteilung zu „AllesErlaubt“ Zugriff auf alles, während die Mitarbeiter nur Zugriff zum jeweiligen Projekt haben, an dem sie arbeiten. Trotzdem kann ein Mitarbeiter auch durch eine erlaubte Domänentransition (Doppelpfeil zwischen Netzwerkprojekt und Datenbankprojekt) zum Netzwerk-Projekt, falls diese Unterstützung benötigen. In die Domäne des Chefs ist jedoch keine Transition erlaubt.

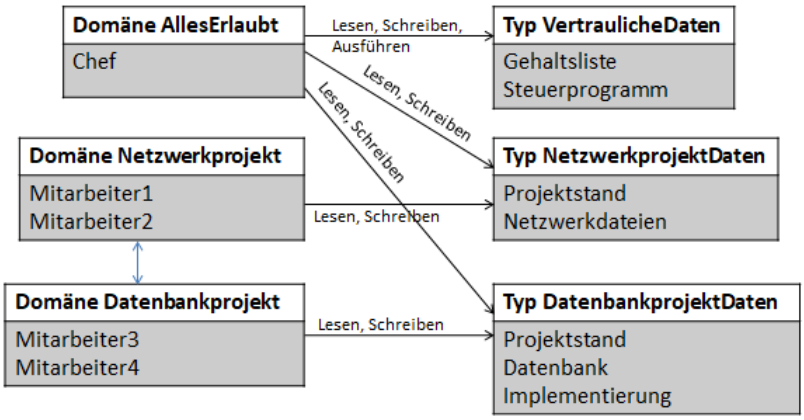


Abbildung 5.6: Beispiel einer Umsetzung von Type Enforcement

5.1.3 Role Based Access Control

Role Based Access Control (RBAC) benutzt Rollen zur Zugriffskontrolle, die zwischen den Subjekten und Objekten stehen. Dabei können Subjekten eine oder mehrere Rollen zugeordnet sein. Die Rollen sind dann wiederum mit verschiedenen Zugriffsrechten auf Objekten assoziiert. Vgl. [FCK95][SDV00]

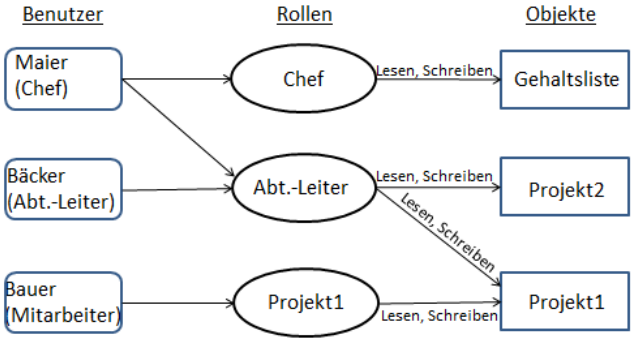


Abbildung 5.7: Beispiel einer Umsetzung von Role Based Access Control

Abbildung 5.7 zeigt ein Beispiel für die Zuteilung von Subjekten zu Rollen und die Ausstattung der Rollen mit Zugriffsrechten. Hierbei ist sichtbar, dass der Chef mehreren Rollen zugeteilt ist, sodass er den Zugriff auf alle Objekte erhält. Der Abteilungsleiter kann alle Projekte einsehen, während ein Mitarbeiter nur genau einem Projekt zugeordnet ist.

5.2 Linux-Kernel

Android basiert auf dem Linux Kernel. Dieser setzt die grundlegenden Sicherheitsmechanismen um. Dazu gehören ein benutzerbasiertes Zugriffsmodell, Prozessisolation, erweiterbare Mechanismen für die Interprozesskommunikation und die Möglichkeit Teile des Kernels zu entfernen, hinzuzufügen oder zu ersetzen. Vgl. [SysKS]

Da Linux als Klon von UNIX entwickelt wurde, beinhaltet es vor allem in Sicherheitsfunktionen wie DAC die wesentlichen Bestandteile von UNIX. Vgl. [LinOv]

Im Folgenden werden die von Android genutzten Funktionen des Kernels näher erläutert.

5.2.1 Discretionary Access Control im Linux Kernel

DAC wird in Linux standardmäßig umgesetzt. Dabei existiert ein Konzept der Eigentumsrechte, so dass der Eigentümer einer Ressource die Zugriffserlaubnis auf diese kontrollieren kann. Dies ist im allgemeinen sehr grobkörnig und kann zu unbeabsichtigten Rechteerweiterungen führen. Vgl. [SeCon][LinOv]

Android nutzt das DAC-System von Linux hauptsächlich zum Isolieren von Apps und Systemfunktionen sowie Ressourcen. Durch die Vergabe der Benutzer-IDs wird der Grundbaustein des Sandboxings gelegt, was auf der Kontrolle von Zugriffsrechten und Prozessisolation basiert. Dies wird im folgenden genauer erklärt. Vgl. [SC13]

Grundkonzept

Das Linux System ist für eine Nutzung von mehreren Benutzern ausgelegt. Dazu erhält jeder Benutzer einen eindeutigen user identifier (UID) (Benutzer-ID). Auch Dateien und andere Prozesse werden mit der UID ihres Besitzers markiert. Benutzer können in Gruppen eingeteilt werden, die durch die group identifier (GID) (Gruppen-ID) gekennzeichnet werden. Ein Benutzer kann Teil mehrerer Gruppen sein. Vgl. .[DPS02][TH16, S. 952]

Der Benutzer mit UID „0“ ist der „root“-Benutzer. Er hat auf alle Dateien im System Zugriff und die Möglichkeit geschützte Systemaufrufe auszuführen. Vgl. .[DPS02][TH16, S. 953]

Android hingegen ist als Betriebssystem für mobile Geräte gedacht, die in der Regel von einem einzelnen Benutzer genutzt werden. Somit kann Android die UIDs/GIDs anders nutzen, indem jeder App eine einzigartige UID und optional ein oder mehrere GIDs zugewiesen werden. Dies geschieht zur Installationszeit.

Die Festlegung von UIDs und GIDs zu bestimmten Zwecken erfolgt in der Datei *android_filesystem_config.h* [CdFCf]. Ein Ausschnitt dieser Datei findet sich in Tabelle 5.1 wieder. Dabei sind die UIDs für Prozesse in Android bereits festgelegt, während Apps laufende UIDs zwischen 10000 und 19999 zugeordnet werden.

Ein Beispiel für den root-Benutzer ist der init-Prozess, der in Abbildung 5.8a zu sehen ist. Der System-Server, der unter der UID „system“ läuft ist in Abbildung 5.8b dargestellt. Als Beispiel für eine App dient der Kalender. Hierzu ist die UID 10026 in Abbildung 5.8c gezeigt. Die Darstellung

ID	Name	Aufgabe
0	root	root-UID
1000	system	UID des System-Server
1001-1068		UIDs oder GIDs für Hardwarenahe Prozesse
2000	shell	UID der ADB- und Debug-Shell
3001-3011		GIDs, denen Kernel-Capabilities zugeordnet sind
10000-19999		UIDs für Apps
ab 100000		verschiedene Benutzer-IDs

Tabelle 5.1: Zuordnung der IDs zu Aufgaben in Android

„u0_a26“ hat dabei folgenden Grund. Für die Benutzung von mehreren Benutzern auf einem Android-System sind die UIDs ab 100000 gedacht. Dabei nummeriert die erste Stelle den Benutzer, während die weiteren fünf Stellen wie bei den anderen Benutzern vergeben werden können. Somit steht bei der Kalender-App „u0“ dafür, dass die erste Stelle eine 0 ist, da man die UID auch als 010026 in Bezug auf mehrere Benutzer sehen könnte. „a26“ kürtzt dabei lediglich 10026 ab. Würde es mehrere Benutzer auf dem System geben, so würde die Kalender-App bei Benutzung eines anderen Nutzers unter der UID 110026 laufen.

```
generic_x86:/ # ps
USER      PID   PPID  VSIZE  RSS     WCHAN              PC  NAME
root      1     0     8232   1468    SyS_epoll_ b5e5b424 S /init
```

(a) root-UID

```
system    1693  1337  1471348 106600 SyS_epoll_ b22c3424 S system_server
```

(b) system-UID

```
u0_a26    2357  1337  1390428 39800 SyS_epoll_ b22c3424 S com.android.calendar
```

(c) UID für die Kalendar-App

Abbildung 5.8: Beispiele für die Zuordnung von UIDs zu Prozessen in Android 7.0

Zugriffskontrolle

Im UNIX-Dateisystem ist jede Datei mit ihrem Besitzer sowie mit Schutzbits versehen, die den Zugriff auf diese spezifizieren. Diese sind durch neun Bit dargestellt. Die ersten drei Bit beziehen sich auf das Zugriffsrecht des Besitzers der Datei, die nächsten drei auf Mitglieder der Gruppen, der der Besitzer angehört und die restlichen drei Bit auf alle anderen Benutzer. Die drei Bit stehen dabei für *r, read* (Lesen), *w, write* (Schreiben) und *x, execute* (Ausführen). Ist jedoch ein Recht nicht gegeben, wird die entsprechende Stelle mit einem „-“ markiert. Vgl. [GM84]

Je nach Dateiart oder Verzeichnis können die Operationen der Schutzbits andere Bedeutungen haben. So ist für ein Verzeichnis die jeweils dritte Komponente der Ausführung das Recht das Verzeichnis zu öffnen. Die Bits für das Lesen und Schreiben bieten die Möglichkeit den Inhalt des Verzeichnisses einzusehen und zu verändern. Vgl. [DPS02]

Die zugewiesenen Schutzbits sowie der Besitzer können durch das Kommando „ls -l“, das eine detaillierte Liste der Inhalte eines Verzeichnisses ausgibt, eingesehen werden.

Jeder App wird bei ihrer Installation ein Verzeichnis erstellt, das sich unter „/data/data/<*packageName*>“ befindet, erstellt. Alle Dateien dieses Verzeichnisses haben die selbe UID wie die App. Vgl. [Ele14, S. 12 ff.]

In der Regel sind die Zugriffsrechte der Verzeichnisse komplett auf den Besitzer, also die App beschränkt. Dies ist für die Kamera-App in Abbildung 5.9 zu sehen, indem ausschließlich die ersten drei Bit gesetzt sind. Das „d“ an erster Stelle bedeutet lediglich, dass es sich um ein *directory* [OrUnix], also Verzeichnis handelt. Anders verhält es sich hierbei mit der Kalender-App. Abbildung 5.9 zeigt, dass auch Mitglieder der Gruppen denen die App zugeordnet ist, das Verzeichnis öffnen und den Inhalt einsehen können. Sogar alle anderen Benutzer können das Verzeichnis öffnen.

```
drwxr-x--x 4 u0_a26 u0_a26 4096 2018-03-09 11:41 com.android.calendar
drwx------ 4 u0_a27 u0_a27 4096 2018-03-30 09:53 com.android.camera2
```

Abbildung 5.9: Schutzrechte für die Verzeichnisse der Kalender-App und Kamera-App in Android 7.0

In Abbildung 5.10 wird durch die Kommandos „adb root“ und „adb shell“ die Shell unter dem root-Benutzer geöffnet. Dies ist sichtbar durch das „#“-Zeichen. Da diesem Benutzer alle Rechte gestattet sind, kann das Kamera-Verzeichnis geöffnet werden und auch dessen Inhalt gelesen werden, was sonst nur der Kamera-App an sich erlaubt ist. Diese Verbote werden in Abbildung 5.11 deutlich. Hier wird die Shell unter dem shell-Benutzer gestartet, was durch das „$“-Zeichen deutlich ist. Der Zugriff auf das Verzeichnis der Kamera wird nicht gestattet, was die Ausgabe „Permission denied“ zur Folge hat. Bei der Kalender-App hingegen ist dies anders. Da hier alle Benutzer durch das letzte x-Bit Zugriff haben, kann hier das Verzeichnis geöffnet werden. Der Aufruf „ls“ wird jedoch auch gestoppt, da die Rechte den Inhalt des Verzeichnisses einzusehen nicht vorhanden sind.

```
C:\Users\Caro\AndroidStudioProjects\Inventar>adb root

C:\Users\Caro\AndroidStudioProjects\Inventar>adb shell
generic_x86:/ # cd data/data
generic_x86:/data/data # cd com.android.camera2
generic_x86:/data/data/com.android.camera2 # ls
cache lib shared_prefs
generic_x86:/data/data/com.android.camera2 #
```

Abbildung 5.10: Zugriffsversuch des root-Benutzers auf die Verzeichnisse der Kalender-App und der Kamera-App in Android 7.0

Die Zugriffskontrolle spielt eine große Rolle beim Sandboxing. Die Daten einzelner Apps und des Systems werden so voneinander abkapselt. Dies geschieht zum einen, indem die Dateien des Systems im eigenen Verzeichnis gespeichert sind und als „read-only“ geladen werden. Um sie zu verändern benötigt es „root“-Rechte, mit denen dann das Verzeichnis neu gemounted wird.

```
C:\Users\Caro\AndroidStudioProjects\Inventar>adb unroot

C:\Users\Caro\AndroidStudioProjects\Inventar>adb shell
generic_x86:/ $ cd data/data/
generic_x86:/data/data $ cd com.android.camera2
/system/bin/sh: cd: /data/data/com.android.camera2: Permission denied
2|generic_x86:/data/data $ cd com.android.calendar
generic_x86:/data/data/com.android.calendar $ ls
ls: .: Permission denied
```

Abbildung 5.11: Zugriffsversuch des shell-Benutzers auf die Verzeichnisse der Kalender-App und der Kamera-App in Android 7.0

Außerdem werden alle veränderbaren Dateien im eigenen „/data“-Verzeichnis gespeichert. Dort wird auch für jede App ihr eigenes Verzeichnis angelegt, zu dem in der Regel ausschließlich die App selbst Zugriffsrechte besitzt. Dadurch werden die Daten der App vor dem Zugriff anderer Apps geschützt. Auch Systemressourcen sind durch verschiedene Lese- und Schreibrechte geschützt. Vgl. [SFK+09][SysKS]

Änderung der Zugriffsrechte

Zugriffsrechte können mit dem Kommando „chmod <*Änderungen*> <*Datei*>“ verändert werden. Als Beispiel hierfür bietet sich das Shell-Verzeichnis an, welches dem Benutzer „shell“ gehört. Startet man die Shell, so läuft diese standardmäßig unter diesem Benutzer.

```
drwx------ 2 shell  shell  4096 2018-03-09 11:40 com.android.shell
```

(a) Die Standard-Zugriffsrechte für das Shell-Verzeichnis

```
generic_x86:/data/data $ chmod a-r,a-w,a-x com.android.shell
```

(b) Änderung der Zugriffsrechte, so dass keiner mehr Zugriff auf das Verzeichnis hat

```
d--------- 2 shell  shell  4096 2018-03-09 11:40 com.android.shell
```

(c) Zugriffsrechte des Shell-Verzeichnis nach der Änderung

```
C:\Users\Caro\AndroidStudioProjects\Inventar>adb shell
generic_x86:/ $ cd data/data
generic_x86:/data/data $ cd com.android.shell
/system/bin/sh: cd: /data/data/com.android.shell: Permission denied
2|generic_x86:/data/data $ chmod u+r,u+w,u+x com.android.shell
generic_x86:/data/data $ cd com.android.shell
generic_x86:/data/data/com.android.shell $
```

(d) Ergebnis der Änderung und Setzen der Standard-Werte des Verzeichnisses

Abbildung 5.12: Beispiele für die Änderung von Zugriffsrechten in Android 7.0

Den Ausgangszustand der Zugriffsrechte zeigt Abbildung 5.12a. Als Extrembeispiel werden die Rechte so geändert, dass keiner mehr Zugriffsrechte auf das Verzeichnis hat (siehe Abbildung 5.12b). Dies geschieht, in dem durch „a“ allen drei Bitgruppierungen die Rechte „r,w,x“ durch „-“ entzogen werden. Das Ergebnis ist in Abbildung 5.12c zu sehen. Hier sind alle Rechte-Bits durch ein „-“ gekennzeichnet. Die Auswirkungen sieht man zu Anfang von Abbildung 5.12d. Da alle Rechte entfernt wurden, wird selbst dem Besitzer des Verzeichnisses, der Shell, der Zugriff verweigert, was mittels der Ausgabe „Permission denied“ sichtbar ist.

Für den Besitzer einer Datei gibt es immer die Möglichkeit die Schutzrechte zu ändern, auch wenn er momentan keine Zugriffsrechte besitzt. Dies bestätigt Abbildung 5.12d, in der nach Verweigerung des Zugriffs die Zugriffsrechte für den Besitzer durch „u“ wieder hinzugefügt („+“) werden können. Nach dieser Änderung ist der Zugriff auf das Verzeichnis dann wieder möglich. Vgl. .[TH16, S. 952 f.]

SetUID/SetGID-Bit

Die Schutzbits von Dateien werden außerdem durch das *SetUID*-Bit und das *SetGID*-Bit ergänzt. Ein Benutzer oder Prozess hat immer eine UID, die nicht veränderbar ist und eine *effektive UID*, die für die Zugriffskontrolle verwendet wird. Normalerweise stimmen UID und effektive UID überein. Ist aber das SetUID-Bit in einer ausführbaren Datei gesetzt, so ändert das System vorübergehend für die Ausführung die effektive UID des Benutzers auf die UID des Besitzers der Datei. Wird die Ausführung beendet, wird die effektive UID wieder auf die UID des Benutzers zurückgesetzt. Damit hat der Benutzer während der Ausführung die Rechte des Besitzers der Datei. Mit dem SetGID-Bit verhält es sich analog, jedoch wird hier die effektive GID verändert. Vgl. [RT78][TH16, S. 954]

Im klassischen Linux System wird dies zum Beispiel für die Passwort-Datei „passwd“ verwendet. Jeder Benutzer sollte sein Passwort selbst ändern können, jedoch nicht die der anderen Benutzer. Somit bleiben die Zugriffsrechte der Passwort-Datei dem „root“-Benutzer vorbehalten, während zusätzlich ein Änderungsprogramm erstellt wird, das das SetUID-Bit gesetzt hat. Durch das Programm kann nun jeder Benutzer auf die Datei zugreifen, ohne dass deren Rechte geändert wurden. Außerdem überwacht das Programm dann, dass Benutzer nur ihr eigenes Passwort ändern können. Vgl. [TH16, S. 954]

Das Setzen der SetUID/SetGID-Bits birgt ein Risiko, falls die Zugriffe, die dadurch möglich werden nicht genau geregelt sind oder Fehler und Sicherheitslücken in den Programmen sind. In Android wurden mit Version 4.3 die SetUID/SetGID-Programme im System abgeschafft. Vgl. [Enh43]

Das Programm *su*, was für „substitute user identity“ steht, hat das SetUID-Bit gesetzt. Dies ist in Abbildung 5.13a durch das „s“ in den Besitzer-Rechten zu sehen. Su [CdSu] ist als „Extra“ in Android verfügbar und gehört nicht zu den essentiellen Teilen des Systems. Zu finden ist es in „/system/xbin/“. Außerdem ist in Abbildung 5.13a dargestellt, dass das Programm als Besitzer den „root“-Benutzer hat, während es zur Gruppe „shell“ gehört. Su kann mittels „su [UID] <*command*>“ für einzelne Befehle benutzt werden, die unter einem anderen Benutzer ausgeführt werden sollen. Wird kein Kommando-Argument übergeben, so öffnet es eine neue Shell unter dem angegebenen Benutzer. Vgl. [CdSu]

```
-rwsr-x--- 1 root shell    9692 2017-07-12 20:06 su
```

(a) Zugriffsrechte des *su*-Programms in Android 7.0

```
126|generic_x86:/system/xbin # id  (1)
uid=0(root) gid=0(root) groups=0(root),1004(input),1007(log),1011(adb),1015(sdcard_rw),1028(sdc
text=u:r:su:s0
generic_x86:/system/xbin # su shell  (2)
generic_x86:/system/xbin $ id  (3)
uid=2000(shell) gid=2000(shell) groups=2000(shell),1004(input),1007(log),1011(adb),1015(sdcard_
eadproc) context=u:r:su:s0
generic_x86:/system/xbin $ su 10026  (4)
generic_x86:/system/xbin $ id  (5)
uid=10026(u0_a26) gid=10026(u0_a26) groups=10026(u0_a26),1004(input),1007(log),1011(adb),1015(s
3009(readproc) context=u:r:su:s0
generic_x86:/system/xbin $ su root  (6)
/system/bin/sh: su: can't execute: Permission denied
126|generic_x86:/system/xbin $ exit
126|generic_x86:/system/xbin $ exit  (7)
126|generic_x86:/system/xbin # id  (8)
uid=0(root) gid=0(root) groups=0(root),1004(input),1007(log),1011(adb),1015(sdcard_rw),1028(sdc
text=u:r:su:s0
generic_x86:/system/xbin #
```

(b) Zugriffsbeispiel auf *su* in Android 7.0

Abbildung 5.13: SetUID-Bit am Beispiel von *su* in Android 7.0

Abbildung 5.13b zeigt ein Beispiel für die Umsetzung des SetUID-Bits. Zunächst läuft die geöffnete Shell unter dem „root“-Benutzer, was mit dem ersten Eintrag „uid“ sichtbar ist, der mit dem Befehl „id“ ausgegeben wird (1). Su öffnet dann auf den Befehl „su shell“ (2) hin eine neue Shell, welche unter dem Benutzer „shell“ läuft (3). Nun wird nach dem Befehl „su 10026“ (4) eine weitere Shell geöffnet, welche unter dem Benutzer 10026 (5) läuft, was der Kalender-App entspricht. Da diese jedoch weder zur Gruppe „shell“ noch zur Benuzter-ID „root“ gehört, werden nun die Zugriffsrechte auf su verweigert (6). Dies ist auf die Zugriffsrechte, die in Abbildung 5.13a sichtbar sind, zurückzuführen. Hier wird klar, dass ausschließlich Prozesse der gleichen Benutzer-ID oder Mitglieder der Gruppe Zugriff erhalten. Da su weitere Shells öffnet, können die zwei geöffneten nun durch „exit“ wieder verlassen werden (7), sodass man sich wieder in der Ausgangs-Shell befindet (8).

5.2.2 Prozessisolation

Die Prozessisolation ist ein weiterer Bestandteil des Sandboxings. In Android basiert dies auf der benutzerbasierten Prozessisolation von UNIX. Diese wird dadurch umgesetzt, dass die UID nicht für einzelne Benutzer, sondern für die Anwendungen eingesetzt werden. Dadurch können keine zwei Apps im gleichen Prozess laufen. Das Android-System startet den Prozess einer App, falls eine Komponente ausgeführt werden muss und beendet ihn wieder, sofern die App nicht mehr benötigt wird und Speicher frei gegeben werden soll. Eine Ausnahme der automatischen Prozessisolation ist nur durch eine geteilte UID möglich, was im nächsten Abschnitt beschrieben wird. Vgl. [SFK+09][SysKS][AppFun]

In standardmäßigen Linux-Systemen erben neue Prozesse die UID/GID ihres Elternprozesses. In Android muss das vor allem beim Starten von Apps anders geregelt sein. Diese werden alle vom *zygote*-Prozess gestartet, erhalten aber die UID/GID, die ihrem Paket zur Installationszeit zugewiesen wurde. Die Funktionsweise von *zygote* ist in Abschnitt 2.3.2 auf Seite 24 erklärt, während auf die Rechte zur Änderung der IDs in Abschnitt 5.3.6 eingegangen wird. Vgl. [DPS02]

5.2.3 Shared UID

In Android wird jeder App bei der Installation eine einzigartige UID zugewiesen. Trotzdem ist es für mehrere Anwendungen möglich sich eine UID zu teilen. Dies heißt in Android *shared User-ID*. Voraussetzung hierfür ist jedoch, dass die Apps mit dem gleichen Schlüssel signiert sind. Dadurch wird diese Möglichkeit hauptsächlich von System-Apps genutzt, aber auch Apps vom gleichen Entwickler können eine geteilte UID anfordern. Da die Vergabe der UIDs zur Installationszeit geschieht, müssen die Apps von vorneherein in ihrer AndroidManifest-Datei unter dem „manifest“-Tag mit dem Attribut „sharedUserId“ [AppM] deklarieren mit welchem anderen Paket sie ihre UID teilen möchten. Vgl. [Ele14, S. 40 ff.][BCMV12]

```
generic_x86:/data/system # grep 'shared-user' packages.xml
  <shared-user name="android.media" userId="10010">
  </shared-user>
  <shared-user name="android.uid.systemui" userId="10021">
  </shared-user>
  <shared-user name="android.uid.shared" userId="10001">
  </shared-user>
  <shared-user name="android.uid.system" userId="1000">
  </shared-user>
  <shared-user name="android.uid.phone" userId="1001">
  </shared-user>
  <shared-user name="android.uid.shell" userId="2000">
  </shared-user>
  <shared-user name="android.uid.calendar" userId="10002">
  </shared-user>
  <shared-user name="com.android.emergency.uid" userId="10011">
  </shared-user>
```

Abbildung 5.14: Auszug der geteilten UIDs aus der Datei packages.xml in Android 7.0

Alle *shared UIDs* werden in der Datei „packages.xml“ mit einem Eintrag hinterlegt. Die zur Ausführungszeit dem System bekannten geteilten UIDs sind in Abbildung 5.14 zu sehen. Dass mehrere Anwendungen tatsächlich die selbe UID zugewiesen bekommen, ist am Beispiel der UID

```
generic_x86:/data/system # grep '10010' packages.list
com.android.providers.media 10010 0 /data/user/0/com.android.providers.media default:privapp 2001,1023,1015
com.android.providers.downloads 10010 0 /data/user/0/com.android.providers.downloads default:privapp 2001,1
com.android.providers.downloads.ui 10010 0 /data/user/0/com.android.providers.downloads.ui default 2001,102
com.android.mtp 10010 0 /data/user/0/com.android.mtp default:privapp 2001,1023,1015,3003,1024,3007
```

Abbildung 5.15: Apps, die unter der geteilten UID „media“ in Android 7.0 laufen

10010 („media“) in Abbildung 5.15 dargestellt. Hier zeigen sich durch die Anfrage aller Apps mit dieser UID mehrere Einträge in der packages.list-Datei. Durch die Vergabe von geteilten UIDs können die Apps gegenseitig auf ihre Daten zugreifen und auch im gleichen Prozess laufen, was sonst durch die verschiedenen UIDs von Apps verhindert wird.

5.2.4 Linux Security Modules

Der Vortrag von P. Loscocco [LosSE] über SELinux stellte 2001 SELinux für den Linux Kernel vor und bot einen eigenen Kernel-Patch zur Umsetzung an. Linus Torvalds wollte jedoch die Umsetzung durch ein allgemeines Sicherheits-Framework im Kernel. Dieses Framework sollte es ermöglichen beliebige Kernel-Module hineinzuladen, welche die gewünschte Sicherheitsimplementierung umsetzen. Vgl. [KLsm][WCM+02]

Dieses allgemeine Framework ist durch *Linux Security Modules (LSM)* realisiert. Vorrangig ist dieses Konzept für Zugriffskontrollmodule implementiert. Es bietet allein keine zusätzliche Sicherheit, sondern die Möglichkeit durch Sicherheitsmodule die DAC-Sicherheit des Linux-Kernels zu erweitern. Dieser Mechanismus wird in Android für die Umsetzung von MAC durch SELinux genutzt. SELinux wird im folgenden Abschnitt beschrieben. Vgl. [WCM+02][KLsm]

Die Implementierung von LSM beinhaltet das Einbauen von *hooks* (dt. Haken) an sicherheitskritischen Punkten im Code des Kernels in Bezug auf die Zugriffskontrolle. Ein Schaubild dazu bietet Abbildung 5.16. Dabei wird bei einem Zugriffswunsch eines Nutzers, beispielsweise auf eine Systemdatei, zuerst der DAC-Mechanismus des Kernels aktiv. Gibt dieser die Erlaubnis für den Zugriff, so wird durch den Haken der LSM-API eine Anfrage an das registrierte LSM-Modul gegeben. Dieses prüft den Zugriff nun nach seiner Implementierung und gibt das Ergebnis an den Kernel zurück. Das Zugriffsergebnis wird dann vom Kernel auch durchgesetzt, sodass der Zugriff nur freigegeben wird, falls das Sicherheitsmodul diesem zugestimmt hat. Dies ist in Abbildung 5.16 durch den blauen Pfeil dargestellt. Vgl. [LinOv][KLsm][WCM+02]

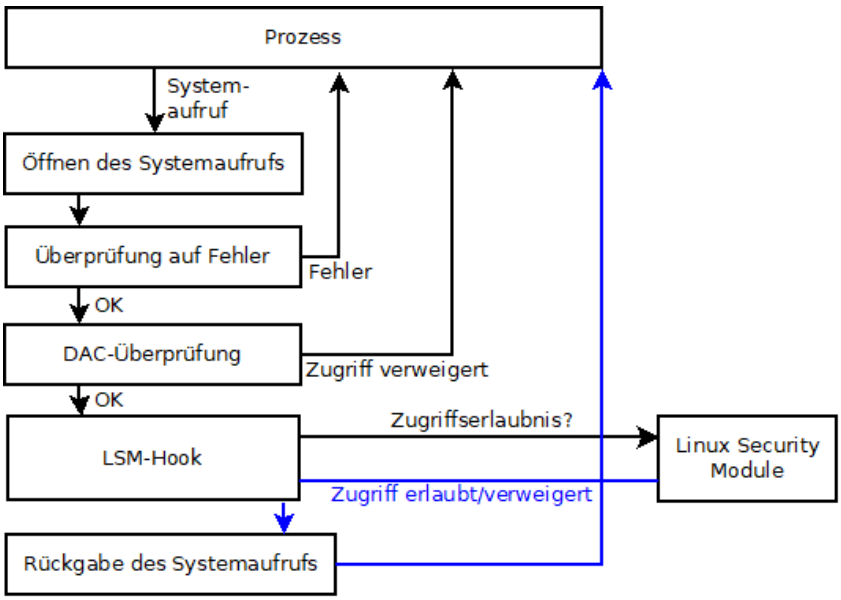


Abbildung 5.16: Implementierung eines LSM

5.3 Security-Enhanced Linux

Security-Enhanced Linux (SELinux) bietet eine Möglichkeit Zugriffskontrolle auf einem feineren Level als der Linux-Kernel umzusetzen. Während der Linux-Kernel DAC zur Zugriffskontrolle umsetzt, erweitert SELinux dies um MAC, das feinkörnigere Zugriffsrechte definieren kann. Es setzt ein systemweites Sicherheitssystem um, das über alle Prozesse, Objekte und Operationen, basierend auf Labels und Sicherheitsregeln den Zugriff bewertet. Als MAC-Mechanismus ist es in der Lage, fehlerhafte oder bösartige Apps einzugrenzen, selbst wenn sie unter „root“ laufen. Vgl. [SC13]

SELinux ist von Mitarbeitern der National Security Agency (NSA) entwickelt und von Peter Loscocco [LosSE] zunächst in einem Vortrag beim „Linux 2.5 Kernel“-Treffen vorgestellt worden. Zusammen mit Stephen Smalley ist das Konzept ausgearbeitet und als Paper [LS01] im Tagungsband der jährlichen USENIX „Technical Conference“ veröffentlicht. Schließlich veröffentlichen Smalley, Vance und Salamon [SVS01] noch ein Paper, welches die genaue Implementierung von SELinux als LSM beschreibt.

Die Implementierung von Android verfolgt das *principle of least priviledge*, was so viel bedeutet, dass Prozessen nur ihre essentiell benötigten Rechte zugeschrieben werden. Für dessen Umsetzung ist SELinux ein wichtiger Bestandteil, da mittels SELinux Zugriffsrechte sehr genau definiert werden können. Vgl. [AppFun]

5.3.1 Motivation

Bereits Loscooco et al. [LSM+98] beschreiben in ihrem Paper die Notwendigkeit einen MAC-Mechanismus auf Ebene des Betriebssystems umzusetzen, da eine Anwendung zur Zugriffskontrolle durch verschiedene Möglichkeiten umgangen werden kann.

Um die Notwendigkeit von SELinux für Android aufzuzeigen, zeigen Smalley und Craig [SC13] mehrere Beispiele auf. Darunter sind mehrere „root exploits“, die Angriffe darstellen, die die Rechte des Angreifers durch eine App auf „root“-Rechte ausweiten. *GingerBreak*, der die Sicherheitslücke CVE-2011-1823 [CVE] ausnutzt, dient als Beispiel für diese Art des Angriffs. Der Android Daemon *vold* ist dafür verantwortlich den externen Speicher in das System einzubinden [SDevCf] und läuft unter dem Benutzer *root*. Dieser hört den *netlink socket* ab, um Benachrichtigungen des Kernels zu erhalten. Dabei prüft vold jedoch nicht, ob die Nachrichten vom Kernel gesendet wurde, so dass nicht berechtigte Apps Nachrichten an vold senden können. Außerdem nutzt der Daemon einen vorzeichenbehafteten Integer aus der Nachricht als Array-Index, von welchem lediglich die obere Grenze überprüft wurde, nicht jedoch die Nicht-Negativität. „GingerBreak“ durchsuchte zunächst das System, um benötigte Informationen für den Angriff auf vold zu erlangen. Danach sendete es ein entsprechendes Paket an vold, das den Daemon eine Binärdatei ausführen lässt. Diese Datei öffnet eine Shell, die durch vold volle root-Rechte besitzt. Vgl. [SC13]

SELinux für Android kann diesen Angriff an verschiedenen Punkten nun unterbinden. Allein das Auslesen verschiedener Systemdateien, um Informationen zu erlangen, hätte die SEPolicy verhindert. Wäre dies dem Angreifer trotzdem möglich gewesen, so hätte GingerBreak keinen netlink socket erstellen können, da Apps diesen Typ von Sockets nicht automatisch erstellen dürfen. Hätte er dies durch entsprechende Permissions trotzdem geschafft, so hätte vold trotzdem nicht die Binärdatei des

Angreifers ausführen können, da Binärdateien, die nicht vom System stammen, von der Policy keine Erlaubnis bekommen, ausgeführt zu werden. Das Ergebnis von Smalley und Craig war letztendlich, dass durch SELinux der Angriff hätte verhindert werden können. Vgl. [SC13]

Dieser Angriff ist auf aktuellen Android-Versionen allein deshalb nicht mehr möglich, da die Implementierung von vold angepasst wurde. Trotzdem ist er ein gutes Beispiel dafür, dass SELinux für Android den Benutzer vor den Einflüssen eines Programmierfehlers im Betriebssystem oder in Apps schützen kann. Im Folgenden wird die genaue Funktionsweise von SELinux erklärt.

5.3.2 SELinux in Android

SELinux für Android wurde von Smalley und Craig [SC13] durch ihr SEAndroid-Projekt umgesetzt. Dabei beschreiben sie, wie es möglich ist das klassische SELinux an die Android Architektur anzupassen und auch wie ihre Lösung in das AOSP eingebaut werden kann.

Seit Android 4.3 wird SELinux im „permissive“-Modus in Android verwendet. Ab Android 4.4 wurde auf teilweise „enforcement“ umgestiegen, wobei einige wenige ausgewählte Domains wie *installd*, *netd*, *vold*, *zygote* „enforced“ wurden. Android 5.0 setzt komplett „enforcement“ um, sodass alles im *enforcing mode* läuft. Die „enforcement level“ werden in Abschnitt 5.3.4 erklärt. Vgl. [SeOv]

SELinux als MAC-Mechanismus bringt mehrere Vorteile und Verbesserungen der Sicherheit mit sich. Zum einen wird das Sandboxing von Apps und Systemprozessen weiter verstärkt. SELinux kann die Interaktionen zwischen Apps und dem Kernel kontrollieren und den Zugriff auf Systemressourcen regulieren. Außerdem ist SELinux in der Lage auch privilegierte Prozesse des Systems zu begrenzen, so dass ihre Rechte nicht durch einen Angriff ausgenutzt werden können. Des Weiteren bietet SELinux eine zentralisierte Policy-Konfiguration, die analysiert werden kann. Dadurch können potentielle Informationsflüsse und Wege zur Rechteausweitung erkannt werden, da Zugriffe während des Betriebs genau protokolliert werden können. Vgl. [SC13][SeOv][SysKS]

Damit ergibt sich eine verbesserte Prozess- und Datenisolation, so dass die Privatsphäre der Daten von Benutzern verbessert wird. Außerdem ist der Schaden, der kompromittierte Systemfunktionen anrichten können, eingegrenzt. Dies bezieht sich nicht nur auf Angriffe auf das System, sondern es wird auch Schutz vor den Folgen von Programmierfehlern geboten.

5.3.3 Architektur

In den Linux Kernel wird SELinux durch ein LSM eingebaut, das in Abschnitt 5.2.4 beschrieben ist. Das LSM-Framework bietet die Möglichkeit weitere Zugriffskontrollmechanismen in den Kernel einzubauen und dadurch den standardmäßigen DAC-Mechanismus zu ergänzen. Vgl. [SeCon]

SELinux basiert auf der Flask-Architektur [SHAL99], die eine Unterstützung für MAC bietet. Sie trennt die Definition der Policy vom Mechanismus, der die Policy durchsetzt. Die SELinux-Architektur ist sehr komplex, besteht aber aus einer abstrahierten Sicht betrachtet aus vier Komponenten. Mehreren *Object Managers (OMs)*, einem *Access Vector Cache (AVC)*, einem *Security Server* und einer *Security Policy*. Der Security Server ist Teil des Kernels, während die Policy aus dem Benutzerbereich geladen wird. OM und AVC können sich sowohl im Kernel- als auch im Benutzerbereich befinden. Vgl. [Ele14, S. 321 f.][LS01][SVS01]

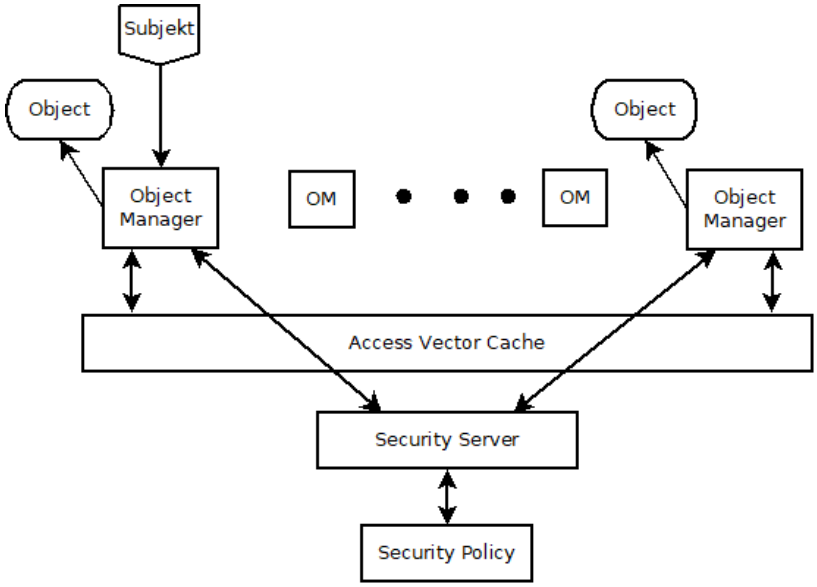


Abbildung 5.17: Zugriffsmechanismus in SELinux

Die Interaktion dieser Komponenten ist in Abbildung 5.17 dargestellt. Der Zugriff auf ein Objekt passiert in der Architektur nicht direkt sondern durch den Objekt-Manager, während die Zugriffsentscheidung vom Security Server getroffen wird. Der Objekt-Manager wird jeweils für jedes zu überwachende Objekt implementiert. Will ein Subjekt auf ein Objekt zugreifen, so muss der Objekt-Manager erst abfragen, ob ein Zugriff gewährt werden kann. Dazu stellt er eine Anfrage an den AVC, der Zugriffsrechte bereits gespeichert haben kann. Ist dies nicht der Fall, so wird der Security Server gefragt, der seine Entscheidung auf Basis der Security Policy fällt. Die Zugriffsrechte, die als Zugriffsvektor dargestellt sind, werden dann im AVC gespeichert, so dass bei einer erneuten Anfrage nicht der Security Server erneut angefragt werden muss. Ein Zugriffsvektor enthält alle Entscheidungen über die Operationen, die das anfragende Subjekt auf dem Objekt ausführen kann. Der AVC wird dabei von allen Objekt-Managern genutzt. Der Objekt-Manager hat drei Aufgaben: Er muss mit dem Security Server kommunizieren, Entscheidungen im AVC speichern und sich beim Security Server registrieren. Dies dient dazu, dass der Security Server bei einer Änderung der Policy den Objekt Manager informieren muss, so dass die passenden Daten im AVC gelöscht werden. Vgl. [SHAL99][Spe08, S. 138 ff.][Ele14, S. 321 f.]

5.3.4 Enforcement Levels

SELinux hat die drei Operationsmodi *disabled*, *permissive* und *enforcing*. Im disabled-Modus wird die Policy erst gar nicht geladen, sondern nur der DAC-Mechanismus zur Zugriffskontrolle verwendet. Ist der permissive-Modus aktiv, so wird bei einem Zugriffsversuch die Policy geladen und der Zugriff überprüft. Die Entscheidung wird protokolliert, jedoch wird der Zugriff trotzdem gewährt, selbst wenn der Zugriff laut Policy verwehrt wurde. Dies ist im enforcing-Modus anders. Hier wird die Policy geladen und ihre Entscheidung protokolliert und durchgesetzt. Der permissive-Modus wird vor allem zur Entwicklung eingesetzt, um zu sehen wie die Policy angepasst werden muss. Vgl. [SeCon][SeOv][Ele14, S. 322]

In Android kann der SELinux-Modus durch „getenforce“ abgefragt werden. Durch „setenforce“ mit dem Argument „0“ oder „1“ kann zwischen permissive und enforcing gewechselt werden. Beides ist in Abbildung 5.18 zu sehen. Wird der Modus durch „setenforce“ geändert, so ist die Änderung nicht dauerhaft. Beim nächsten Boot wird das Gerät wieder im Standard-Zustand gestartet. Trotzdem darf der Modus auch nicht vorübergehend durch Apps geändert werden. Der Zugriff auf SELinux ist für Apps beispielsweise in der Datei „app.te“ [CdSeAp, Zeile 494ff.] durch „neverallow“-Regeln eingeschränkt. Vgl. [Ele14, S. 322]

```
generic_x86:/ # id
uid=0(root) gid=0(root) groups=0(root),1004(input),1007(log),1011(adb),1015(sdcard_rw),1028(sdcar
s),3009(readproc) context=u:r:su:s0
generic_x86:/ # getenforce
Enforcing
generic_x86:/ # setenforce 0
generic_x86:/ # getenforce
Permissive
```

Abbildung 5.18: Veränderung des SELinux-Modus in Android 7.0

Auch wenn SELinux im enforcing-Modus ist, können ausgewählte Domains in der Policy als *permissive* deklariert werden. Damit werden sie so behandelt, dass Zugriffe protokolliert aber immer durchgesetzt werden. Dies wird auch *per-domain permissive*-Modus genannt. Der Rest des System arbeitet dabei im enforcing-Modus. Vgl. [SeOv][Ele14, S. 322]

Ein Beispiel hierfür ist die „su“-Domain mit einem „permissive“-Statement in der Datei „su.te“, das in Listing 5.1 zu sehen ist. Kommentiert wird dies damit, dass der Befehl „setenforce“ damit ermöglicht wird. Dass dies so der Fall ist, zeigt Abbildung 5.18. Hier wird der Befehl „setenforce“ offensichtlich ausgeführt während der Sicherheitskontext der „root“-Shell die Domain „su“ hat. In Abbildung 5.19 zeigt sich, dass die Ausführung trotzdem protokolliert ist. Vgl. [CdSeSu]

Listing 5.1 „permissive“-Statement in der Datei su.te [CdSeSu]

```
19 # su is also permissive to permit setenforce.
20 permissive su;
```

```
11:53:31.993 1334-1334/? I/SELinux: avc:  received setenforce notice (enforcing=0)
```

Abbildung 5.19: Ausgabe von logcat bei der Änderung des SELinux-Modus

5.3.5 Security Context

Das SELinux MAC-Modell hat die drei Hauptkomponenten *subject* (Subjekt), *object* (Objekt), *action* (Operation). Dabei sind Subjekte der aktive Part, die Operationen auf Objekten ausführen. In der Regel sind Subjekte Prozesse, während Objekte meist Ressourcen wie Dateien, Hardware oder Sockets darstellen. Prozesse können aber auch Objekte sein. Operationen unterscheiden sich je nach Objekt. Beispiele für Operationen sind Lesen, Schreiben oder Ausführen. Vgl. [Ele14, S. 321 f.]

Sowohl Subjekte als auch Objekte haben Sicherheitsattribute, den *Security Context*, der durch ein *Label* gespeichert wird. Anhand des Security Contexts wird dann mit den Policy-Regeln entschieden, ob eine Operation ausgeführt wird. Alle Prozesse mit dem gleichen Label werden bei Zugriffsentscheidungen gleich behandelt. Vgl. [SeOv][Hai14, S. 27][Ele14, S. 321 f.]

Ein Label eines Subjekts oder Objekts in SELinux hat die Form:

user:role:type:mls_level [SeCon]

Innerhalb von SELinux ist der Sicherheitskontext als String mit variabler Länge dargestellt. Meist geschieht das als „extended attribute“ in den Metadaten der Datei. Vgl. [Hai14, S. 27][Ele14, S. 324]

Der Sicherheitskontext kann im Terminal durch Abfragen mit dem Zusatz „-Z“ ausgelesen werden. Beispiele für verschiedene Sicherheitslabel bieten Abbildung 5.20 durch „ps -Z“ für Subjekte und Abbildung 5.21 mittels „ls -Z“ für Objekte.

```
generic_x86:/ # ps -Z
LABEL                          USER      PID   PPID  VSIZE  RSS     WCHAN              PC  NAME
u:r:init:s0                    root      1     0     8232   1468    SyS_epoll_ b5e5b424 S /init
u:r:platform_app:s0:c512,c768  u0_a21    1817  1337  1446076 81164  SyS_epoll_ b519a424 S com.android.systemui
u:r:sdcardd:s0                 media_rw  1890  1284  9496   2252    inotify_re aac7e424 S /system/bin/sdcard
u:r:radio:s0                   radio     1899  1337  1413252 59484  SyS_epoll_ b519a424 S com.android.phone
u:r:kernel:s0                  root      2156  2     0      0       worker_thr 00000000 S kworker/2:1H
u:r:platform_app:s0:c512,c768  u0_a12    2317  1337  1382476 36136  SyS_epoll_ b519a424 S android.ext.services
u:r:untrusted_app:s0:c512,c768 u0_a50    2360  1337  1384344 36780  SyS_epoll_ b519a424 S com.android.printspo
u:r:untrusted_app:s0:c512,c768 u0_a66    15491 1337  1404600 58752  SyS_epoll_ b519a424 S de.caro.inventar
```

Abbildung 5.20: Ausschnitt der Sicherheitskontexte der laufenden Prozesse in Android 7.0

```
u:object_r:rootfs:s0          fstab.ranchu-encrypt     u:object_r:rootfs:s0          service_contexts
u:object_r:rootfs:s0          fstab.ranchu-noencrypt   u:object_r:storage_file:s0    storage
u:object_r:init_exec:s0       init                     u:object_r:sysfs:s0           sys
u:object_r:rootfs:s0          init.environ.rc          u:object_r:system_file:s0     system
```

Abbildung 5.21: Ausschnitt der Sicherheitskontexte von Objekten in Android 7.0

SELinux Benutzer (user)

Benutzer sind in der Regel mit einem menschlichen Benutzer assoziiert oder stellen Systemfunktionen dar, wie beispielsweise der Benutzer „root“. SELinux-Benutzernamen stellen jedoch Gruppen oder Klassen von Benutzern dar. Diese werden in einer eigenen SE-Benutzer-Datenbank vermerkt. Die SE-Benuter wiederum können mit einer oder mehreren Rollen assoziiert sein. Vgl. [Hai14, S. 24][Spe08, S. 141][Ele14, S. 322]

Rolle (role)

Die unterschiedlichen Rollen stehen für unterschiedliche Zugriffsrechte auf das System. Durch die Rolle wird RBAC umgesetzt. Vgl. [LS01] [Ele14, S. 322]

Android nutzt jedoch kein RBAC, da nur zwischen zwei Rollen für Subjekte oder Objekte unterschieden wird. Dies wird in Abschnitt 5.3.6 genauer dargestellt.

Typ (type)

Der Typ definiert die Art des Subjekts oder Objekts. Dabei werden Typen von Subjekten auch *domain* genannt. Durch den Typ wird Type Enforcement in SELinux umgesetzt. Dies ist die wichtigste Entscheidungskomponente des Sicherheitskontexts bei den Zugriffsregeln. Während Android die anderen Komponenten wenig nutzt, sind sehr viele verschiedene Typen definiert. Vgl. [LS01][Ele14, S. 322][SeCon]

Eine typische Android App läuft in ihrem eigenen Prozess mit dem Label „untrusted_app“, das ihr nur eingeschränkte Rechte zuschreibt. Plattform-Apps, die ins System geschrieben sind, laufen unter einem eigenen Label. System-Apps, die Teil des Kerns des Android Systems sind, haben das „system_app“-Label mit mehr Privilegien. Vgl. [SeCon]

MLS/MCS (mls_level)

Das *mls_level* ist optional. Durch dieses wird MLS und MCS umgesetzt. Dabei werden die Sicherheitsstufen von MLS durch den Buchstaben „s“ mit einer Zahl dargestellt, während die Kategorien von MCS mit „c“ durchnummeriert sind. Vgl. [LS01]

Wie in Abbildung 5.20 und Abbildung 5.21 an den Labeln zu sehen ist, wird standardmäßig die Sicherheitsstufe „s0“ zugeteilt. Einige Apps oder Verzeichnisse derer, die durch UIDs ab 10000 durchnummeriert sind, haben zusätzlich die Kategorien „c512,c768“ in ihrem Label.

Vergabe des Security Contexts

Der Sicherheitskontext kann explizit für vorhandene Dateien bei der Initialisierung des Dateisystems vergeben werden, jedoch auch implizit bei der Erstellung einer neuen Datei. Für die Vergabe ist auch die SE-Policy verantwortlich. Vgl. [Ele14, S. 324]

In der Regel erben Objekte das Label ihrer Eltern, was beispielsweise für eine Datei bedeutet, dass sie das Label des Verzeichnisses erhält, in dem sie gespeichert wird. Außerdem wird noch die Information des Prozesses, der das Objekt erstellt, hinzugezogen. Subjekte hingegen erben den Sicherheitskontext ihres Elternprozesses. Die Entscheidung kann jedoch von externen Informationen, die in Dateien gespeichert sind, beeinflusst werden. Sowohl Objekte und Subjekte können auch einen anderen Kontext zugewiesen bekommen, falls dies durch Typ- oder Domänentransitionen (siehe Abschnitt 5.3.6) in der Policy erlaubt ist. Vgl. [LS01][Ele14, S. 324]

In Android ist die Vergabe der Sicherheitskontexte durch Definitionen in verschiedene Dateien unterstützt. Diese sind im Quellcode-Verzeichnis „/system/sepolicy/“ zu finden. Als Beispiel bietet die Datei „file_contexts“ [CdSeFC] Zuweisungen für die Kontexte von Dateien und Verzeichnissen. Dort werden sowohl Labels für einzelne Dateien definiert, als auch durch reguläre Ausdrücke allgemeine Regeln festgelegt. Auch für Services gibt es für die Vergabe des Labels eine Datei, „service_contexts“ [CdSeSC], die Services wie Alarm, Batteriestatus oder Bluetooth Sicherheitskontexte zuordnet. Diese sollen auch kontrollieren, welche Prozesse eine Binder-Referenz zu diesen Services finden und diese hinzufügen können. Vgl. [SeImpl]

Vor allem für Apps ist die Vergabe des Sicherheitskontexts wichtig, da diese alle von *zygote* gestartet werden, aber trotzdem einen eigenen Kontext erhalten sollen. Da zygote sich selbst kopiert, muss der Prozess Rechte besitzen, den Sicherheitskontext zu ändern. Dies wird in Abschnitt 5.3.6 genauer erklärt. Bei der Vergabe des Kontexts erhält zygote Angaben zum passenden Kontext durch die Datei „seapp_contexts“ [CdSeAC]. Diese ist sowohl für App-Prozesse als auch für deren Verzeichnis unter „/data/data“ verantwortlich. Vgl. [SeImpl]

5.3.6 Policy

Die *Policy* wird vom Security Server dazu verwendet, über den Zugriff auf Objekte zu entscheiden. Dabei ist die Policy aus Performance-Gründen typischerweise binär gespeichert, nachdem verschiedene Policy-Dateien kompiliert wurden. Die Quelldateien der Policy sind in einer eigenen Policy-Sprache geschrieben, die aus *statements* und *rules* besteht. Die Statements stellen die Policy-Entitäten dar. Dazu gehören Typen, Benutzer und Rollen. Regeln werden weiter unterteilt in *access vector rules*, die Zugriffe erlauben oder verbieten, und *type enforcement rules*, die die Art von Transitionen zwischen Typen und Domänen definieren. Zusätzlich können *attributes* definiert werden, welche Namen für Gruppen von Domänen oder Typen bieten können. Vgl. [Ele14, S. 324][SeCon]

Anweisungen (statements)

Die *statements* sind für die Deklaration verschiedener Benutzer, Rollen, Typen oder anderen Komponeneten in SELinux verantwortlich. Im Folgenden werden die essentiellen Anweisungen erklärt.

Attribute-Statement Attribute werden in SELinux dazu verwendet, Gruppen von Typen zu referenzieren. Dazu wird das *attribute statement* verwendet, das Bezeichner für Attribute definiert. In Listing 5.2 werden sowohl das Attribut „domain“ für alle Typen, die Prozesse betreffen, als auch das Attribut „fs_type“, welches alle Typen, die Dateisysteme betreffen bezeichnet, deklariert. Attribute können direkt bei der Deklaration zugeordnet werden, wie bei den folgenden Statements sichtbar wird, aber auch nachträgtlich durch eine Anweisung der Form „typeattribute type_id attribute_id“. Vgl. [SeT]

Listing 5.2 Deklaration von Attributen für Androids SELinux [CdSeA]

```
5  # All types used for processes.
6  attribute domain;
7  # All types used for filesystems.
8  # On change, update CHECK_FC_ASSERT_ATTRS
9  # definition in tools/checkfc.c.
10 attribute fs_type;
```

User-Statement *User statements* definieren SELinux-Benutzer, wie sie im Sicherheitskontext benutzt werden. Wie in Abbildung 5.20 und Abbildung 5.21 auffällt, ist der Benutzer im Sicherheitslabel ausschließlich „u“. Listing 5.3 zeigt dabei die Datei „users“[CdSeU], die den einzigen SE-Benutzer in Android definiert. Dem Benutzer können in der Anweisung eine Rolle zugewiesen werden, genauso wie ein MLS-Level. Dem Benutzer „u“ in Listing 5.3 wird die Rolle „r“ und das MLS-Level „s0“ zugewiesen. Durch das Wort „range“ wird außerdem definiert, auf welchen MLS-Bereich er Zugriff hat. Vgl. [SeU]

Listing 5.3 Deklaration der SELinux-Benutzer in Android [CdSeU]

```
1 user u roles { r } level s0 range s0 - mls_systemhigh;
```

Role-Statement *Role statements* definieren die im Sicherheitskontext verwendeten Rollen. In Android beschränkt sich dies auf die Standard-Rolle „object_r“, sowie die Rolle „r“, welche für Prozesse genutzt wird. Listing 5.4 zeigt die Deklaration der Rolle „r“ durch die Datei „roles“ [CdSeR] in Android. Durch das Schlüsselwort „types“ können der Rolle dabei Typen oder Attribute zugeordnet werden. In diesem Fall wird das Attribut „domain“ assoziiert, das die Gruppe aller Typen für Domänen, darstellt. Dadurch wird diese Rolle nur Prozessen vergeben. Vgl. [SeR]

Listing 5.4 Deklaration der Rollen von SELinux in Android [CdSeR]

```
1 role r types domain;
```

Type-Statement *Type statements* definieren Typen, wie sie im Sicherheitskontext verwendet werden. Diese werden durch das Schlüsselwort „type“ definiert und können eine Reihe von Attributen zugeordnet bekommen, die durch Kommata getrennt werden. In Listing 5.5 wird der Typ „labeledfs“ für ein Dateisystem, das Label ermöglicht, definiert und mit dem Attribut „fs_type“ assoziiert, das wiederum alle Typen, die für Dateisysteme benutzt werden, beschreibt. Vgl. [SeT]

Listing 5.5 Deklaration der Typen von SELinux in Android [CdSeF]

```
1 # Filesystem types
2 type labeledfs, fs_type;
```

Class-Statement In SELinux werden Objekt-Klassen definiert, die die Art der Objekte genauer spezifizieren. Dies wird für die Regeln in der Policy benötigt, da verschiedene Objektarten auch verschiedene Zugriffsmöglichkeiten bieten. Die Deklaration von Klassen geschieht durch den „class“-Bezeichner. In Android werden die Klassen in der Datei „security_classes“ [CdSeCl] definiert, wofür in Listing 5.6 ein Beispiel gegeben ist. Diesen Klassen werden in Android durch die Datei „access_vectors“ [CdSeAV] ihre Zugriffsoperationen zugewiesen. So sind die möglichen Operationen für die Klasse „filesystem“ in Listing 5.7 dargestellt. Vgl. [SeC]

Listing 5.6 Deklaration von Objekt-Klassen für SELinux in Android [CdSeCl]

```
15 # file-related classes
16 class filesystem
```

Listing 5.7 Deklaration der Zugriffsmöglichkeiten für Objekte für SELinux in Android [CdSeAV]

```
145 class filesystem
146 {
147         mount
148         remount
149         unmount
150         getattr
151         relabelfrom
152         relabelto
153         associate
154         quotamod
155         quotaget
156 }
```

Regeln (rules)

Eine SELinux Policy Regel hat die folgende Form:

RULE_VARIANT SOURCE_TYPES TARGET_TYPES : CLASSES PERMISSIONS.

Hierbei beschreibt „RULE_VARIANT“ die Art der Regel, beispielsweise „allow“ für eine Regel, die einen bestimmten Zugriff erlaubt. „SOURCE_TYPES“ ist ein Label für den Typ eines Prozess oder einer Gruppe von Prozessen. Im Gegensatz dazu stellt „TARGET_TYPES“ Typen für ein Objekt oder eine Menge von Objekten dar. Objekte werden außerdem mit mindestens einer Klasse assoziiert, die durch „CLASSES“ überprüft wird. Die Klassen veranschaulichen dabei die Art des Objekts, wie Datei, Socket oder Hardwarekomponente. Dadurch werden die entsprechenden Operationen, die auf dieser Klasse von Objekten überhaupt ausführbar sind, klargestellt. Die erlaubten Optionen wie Lesen oder Schreiben sind dann in „PERMISSIONS“. Vgl. [SeCon]

Die Regel beschreibt also, was passieren soll, wenn ein Subjekt, das mit einem Typ aus SOURCE_TYPES markiert ist, eine der Aktionen aus PERMISSIONS auf einem Objekt mit einer Klasse aus CLASSES, das einen Typ aus TARGET_TYPES hat, ausführen will.

Acess Vector Rules *Access Vector Rules* definieren, welche Zugriffsrechte für einen Prozess erlaubt sind. Rechte werden durch „allow“-Regeln erteilt. Trotzdem gibt es noch „neverallow“-Regeln, mit denen Zugriffe grundsätzlich verboten werden, auch wenn es eine „allow“-Regel dafür gibt. Alle Regeln vom Typ „neverallow“ werden von einem Policy-Compiler überprüft, der bei einem Verstoß in Bezug auf die definierten „allow“-Regeln die Konfiguration ablehnt. Damit können Fehler in der Policy-Konfiguration entdeckt werden. Vgl. [Sma02][SeAvc]

Ein Beispiel einer „allow“-Regel ist in Listing 5.8 dargestellt. Diese Regel erlaubt es Prozessen unter der „shell“-Domäne ein Verzeichnis auszulesen, das für die Auflistung der aktuell laufenden Prozesse nötig ist. Abbildung 5.22 zeigt, dass diese Regel umgesetzt ist, indem die Darstellung unter der Domäne „shell“ möglich ist. Die Zuweisung der Domäne „shell“ an die Shell wird durch das Label „context“ bei der Ausgabe des Kommandos „id“ deutlich.

Listing 5.8 Beispiel einer „allow“-Regel für die „shell“-Domain [CdSeSh]

```
146 # allow shell to read /proc/pid/attr/current for ps -Z
147 allow shell domain:process getattr;
```

```
generic_x86:/ $ id
uid=2000(shell) gid=2000(shell) groups=2000(shell),1004(input),1007(log),1011(adb),1015(sdcard_rw),102
eadproc) context=u:r:shell:s0
generic_x86:/ $ ps -Z
LABEL                          USER      PID   PPID  VSIZE  RSS   WCHAN            PC  NAME
u:r:init:s0                    root      1     0     8232   1468  SyS_epoll_ 00000000 S /init
u:r:kernel:s0                  root      2     0     0      0       kthreadd 00000000 S kthreadd
```

Abbildung 5.22: Ergebnis der „allow“-Regel aus Listing 5.8 in Android 7.0

Ein Beispiel für eine „neverallow“-Regel ist in Listing 5.9 zu sehen. Diese Regel verbietet Prozessen der Domain „shell“ einen Link auf einen „file“ vom Typ „file_type“ zu erstellen. Abbildung 5.23 zeigt, dass diese Regel auch in der in dieser Arbeit verwendeten Testumgebung von Android 7.0 umgesetzt ist. Die Shell läuft unter der Domain „shell“. Beim Versuch einen Link zu erstellen, wird dies verhindert und die Meldung „Operation not permitted“, Operation nicht erlaubt, ausgegeben.

Listing 5.9 Beispiel einer „neverallow“-Regel für die „shell“-Domain [CdSeSh]

```
200 # Do not allow shell to hard link to any files.
201 # In particular, if shell hard links to app data
202 # files, installd will not be able to guarantee the deletion
203 # of the linked to file. Hard links also contribute to security
204 # bugs, so we want to ensure the shell user never has this
205 # capability.
206 neverallow shell file_type:file link;
```

```
1|generic_x86:/sdcard $ id
uid=2000(shell) gid=2000(shell) groups=2000(shell),1004(input),1007(log),1011(adb),1015(sdcard_rw),102
eadproc) context=u:r:shell:s0
generic_x86:/sdcard $ ln Alarms alarmlink
ln: cannot create hard link from 'Alarms' to 'alarmlink': Operation not permitted
```

Abbildung 5.23: Folgen der „neverallow“-Regel aus Listing 5.9 in Android 7.0

Type Transition Rules *Type transition rules* bieten eine Möglichkeit den Typ eines Subjekts oder Objekts zu ändern. Wichtig hierbei ist, dass die Regel nicht die Erlaubnis für eine Typänderung gibt. Die „type_transition“-Regel besagt nur, dass falls die in der Regel beschriebenen Gegebenheiten eintreffen, der Typ geändert werden soll. In Listing 5.10 wird dies durch ein Makro für eine automatische Domänentransition gezeigt. Hier wird kommentiert, dass zunächst die Erlaubnis gegeben werden muss und durch die „type_transition“-Regel die Transition dann umgesetzt wird. Die Argumente des Makros sind „(olddomain, type, newdomain)“ [CdSeTM]. Somit soll die Typtransition dann ausgeführt werden, wenn ein Prozess vom Typ „olddomain“ einen File vom Typ „type“ ausführt. Die Domäne des Prozesses soll dann zu „newdomain“ geändert werden. Vgl. [Sma02][SeTy]

Listing 5.10 Makro für eine automatische Domänentransition als Ausschnitt aus „te_macros.te“ [CdSeTM]

```
28  define(`domain_auto_trans', `
29  # Allow the necessary permissions.
30  domain_trans($1,$2,$3)
31  # Make the transition occur by default.
32  type_transition $1 $2:process $3;
33  ')
```

Domänentransition Domänentransitionen sind in Android besonders wichtig, da alle Apps vom *zygote*-Prozess gestartet werden, aber nicht dessen Sicherheitskontext erben sollen, sondern einen eigenen, zur App passenden. Dies wird durch Regeln in der Datei „zygote.te“ [CdSeZy] bestimmt, aus welcher ein Ausschnitt in Listing 5.11 gegeben ist. Zygote nutzt dabei die Operationen „setcurrent“ oder „dyntransition“. „setcurrent“ erlaubt das Ändern des Sicherheitskontextes eines Prozesses, welcher hier mit „self“ sich selbst beschreibt. Die Operation „dyntransition“ bietet die Möglichkeit dynamisch den Kontext eines Prozesses zu verändern. Dies kann zygote unter anderem von Prozessen mit den Domänen „system_server“ oder „appdomain“. Vgl. [SePerm]

Listing 5.11 Domänentransition von *zygote* [CdSeZy]

```
15  # Switch SELinux context to app domains.
16  allow zygote self:process setcurrent;
17  allow zygote system_server:process dyntransition;
18  allow zygote appdomain:process dyntransition;
```

„dyntrans“ macht die Domänentransition durch den „setcon(3)“-Aufruf an das System. Dadurch kann jedoch der ganze Kontext unabhängig von Faktoren wie der Ausgangsdomäne geändert werden. Dies ist für den allgemeinen Gebrauch deshalb ungeeignet, da bei einer Kompromittierung Sicherheitskontexte völlig frei geändert werden könnten. Abhilfe dafür schafft das Makro „domain_trans“ für Domänentransitionen in der Datei „te_macros“ [CdSeTM], die auch für andere Typtransitionen Makros bietet. Die Funktion ist in Listing 5.12 abgebildet und hat die drei Argumente „(olddomain, type, newdomain)“. Zunächst muss die zu ändernde Domain die Erlaubnis haben, den File vom Typ „type“ ausführen zu dürfen (siehe Zeile 11), da die Transition beim Ausführen passiert. Des Weiteren muss die Transition von der alten in die neue Domäne erlaubt werden (siehe Zeile 12). Drittens wird ein „entrypoint“ benötigt, der der neuen Domäne die Erlaubnis gibt, in die Domäne „einzutreten“ (siehe Zeile 14). Mit diesen Zugriffsrechten kann durch eine „type_transition“-Regel dann die Domänentransition erfolgen. Vgl. [SePerm]

Listing 5.12 Makro für eine Domänentransition als Ausschnitt aus „te_macros.te“ [CdSeTM]

```
9  define(`domain_trans', `
10 # Old domain may exec the file and transition to the new domain.
11 allow $1 $2:file { getattr open read execute map };
12 allow $1 $3:process transition;
13 # New domain is entered by executing the file.
14 allow $3 $2:file { entrypoint open read execute getattr map };
```

6 Permissions

Zum Schutz der privaten Daten der Android Benutzer können Apps nicht automatisch direkt darauf zugreifen, sondern müssen erst eine Erlaubnis vom Benutzer einholen, die Zusage zu einer sogenannten *Permission*. Verschiedene Bereiche wie Nutzerdaten, spezielle Systemfeatures oder Hardware haben eigene Permissions, die bei Zustimmung den jeweiligen Zugriff erlauben.

Da jede App unter einem eigenen Benutzer läuft, sind die Apps durch die Sandboxen im Zugriff auf Systemressourcen, Hardware oder andere Apps eingeschränkt. Um aus der Sandbox zu gelangen und mit dem System oder anderen Apps zu interagieren, können Apps den Permission-Mechanismus nutzen. Dabei werden die benötigten Permissions vom Entwickler statisch in der Manifest-Datei deklariert. Einige Permissions wie Internetzugriff, Kameranutzung oder Bluetooth sind von Android vordefiniert, Entwickler können aber auch eigene Permissions definieren. Vgl. [SysKS][Yag13, S. 30 f.]

In Android ist eine Permission ein String, der die Fähigkeit eine bestimmte Operation auszuführen repräsentiert. Diese Operation kann der Zugriff auf physische Ressourcen wie die SD-Karte oder die Kamera sein oder auch die Möglichkeit eine Komponente einer anderen App zu starten oder darauf zuzugreifen. Vgl. [Ele14, S. 21 f.]

Je nachdem welches Feature von der Permission angefragt wird, wird die Permission automatisch gestattet oder der Benutzer gefragt. Näheres dazu ist in Abschnitt 6.2.1 beschrieben. Vgl. [PermOv]

Eine detaillierte Liste der dem System bekannten Permissions kann man durch den Befehl „pm list permissions -f“ im Terminal erhalten. Ein Ausschnitt dieser Liste ist in Abbildung 6.1 zu sehen. Hier wird die Permission „ACCESS_WIFI_STATE“ beschrieben. Durch „package“ wird das Paket angegeben, in dem die Permission definiert ist, in diesem Fall „android“, was bedeutet, dass sie in Android vordefiniert ist. Außerdem wird bei dieser Permission ein Label und eine Beschreibung gegeben, die erklären was die Permission für Möglichkeiten bietet. Als letzten Punkt wird das „protectionLevel“ angegeben. Auf dieses wird in Abschnitt 6.1 genauer eingegangen.

```
+ permission:android.permission.ACCESS_WIFI_STATE
  package:android
  label:view Wi-Fi connections
  description:Allows the app to view information about Wi-Fi networking, such as whether Wi-Fi is enabled and
name of connected Wi-Fi devices.
  protectionLevel:normal
```

Abbildung 6.1: Ausschnitt der detaillierten Ansicht der Permissions in Android 7.0

6.1 Sicherungslevel

Um die Permissions zu kategorisieren wird ihnen jeweils ein *Protection Level*, ein Sicherungslevel zugeordnet. „[The protection level] Characterizes the potential risk implied in the permission and indicates the procedure the system should follow when determining whether or not to grant the permission to an application requesting it.“ [ManP] Dies bedeutet, dass das Sicherungslevel das potentielle Risiko darstellt, mit welchem die Ausführung der Operation der Permission verbunden ist. Des Weiteren soll es dem System angeben, wie es vorgehen soll, wenn die Entscheidung getroffen werden soll, einer App die Permission zu erteilen oder nicht.

Bei den Sicherungslevel gibt es die vier Kategorien *normal*, *dangerous*, *signature* und *signatureOrSystem*. Apps und Permissions von Drittanbietern betreffen meistens nur die *normal* und *dangerous* Sicherungslevel. Vgl. [PermOv]

6.1.1 normal

Das Sicherungslevel *normal* ist der Standardwert. Bei diesen Permissions handelt es sich um Operationen wie Zugriffe auf Daten, Ressourcen und Features anderer Apps außerhalb der Sandbox. Es sind jedoch nur Permissions, deren Operationen wenig Risiko für andere Apps, das System oder die Privatsphäre des Benutzers darstellen. Dazu gehören Permissions wie „SET_TIMEZONE“, „BLUETOOTH“, oder „ACCESS_NETWORK_STATE“. Ist eine solche Permission im App-Manifest deklariert, so wird sie bei der Installationszeit vom System automatisch gewährt. Der Benutzer wird nicht explizit nach seiner Zustimmung gefragt und kann die Erteilung der Permission des Systems auch nicht widerrufen. Vgl. [ManP][PermOv]

6.1.2 dangerous

Permissions mit dem Wert „dangerous“ stellen ein höheres Risiko dar. Sie beschreiben Operationen, die auf Daten und Ressourcen zugreifen möchten, in die private Informationen des Benutzers involviert sind. Außerdem können diese Operationen gespeicherte Daten beeinflussen oder auch Funktionen von anderen Apps benutzen. Hierzu gehören Permissions wie „READ_SMS“, die private Daten auslesen, „WRITE_CONTACTS“, die private Daten verändern kann oder „CAMERA“ und „SEND_SMS“, die die Benutzung des Geräts beeinflussen oder möglicherweise kostenpflichtige Aktionen ausführen. Da dieser Typ von Permission ein potentielles Risiko darstellt, erteilt das System nicht automatisch die Erlaubnis, sondern benötigt die Zustimmung des Benutzers. Vgl. [PermOv][RTPerm][ManP]

6.1.3 signature

Permissions mit dem Level *signature* werden vom System nur zugelassen, wenn die anfragende App mit dem gleichen Zertifikat signiert ist, wie die App, die die Permission deklariert hat. Wenn die Zertifikate übereinstimmen, wird die Erlaubnis vom System automatisch erteilt, ohne den Benutzer zu fragen. Da dieser Typ von Permissions zur Benutzung den kryptographischen Schlüssel der definierenden App benötigt, wird diese Art der Permissions von Apps des gleichen Autors benutzt. In der Regel sind dies somit Androids System-Apps untereinander. Ein Beispiel für

dieses Sicherungslevel ist die Permission „ACCESS_PROVIDER“, die einer App den Zugriff auf die Email-Datenbank erlaubt, die empfangene und gesendete Mails, sowie Benutzernamen und Passwörter enthält. Vgl. [ManP][Ele14, S. 26]

6.1.4 signatureOrSystem

Permissions des Sicherungslevel *signatureOrSystem* werden ausschließlich eingeräumt, wenn Apps im Android-System-Image sind oder mit dem gleichen Zertifikat signiert sind wie die App, von der die Permission deklariert ist. Dieses Level wird verwendet, falls Verkäufer Apps ins System-Image eingebaut haben und diese miteinander spezielle Features teilen müssen. Dieses Level sollte laut Dokumentation nicht unbedingt verwendet werden, signature bietet genug Schutz. Vgl. [ManP]

6.2 Permission-Management

In Bezug auf die Benutzung von Permissions müssen einige Aspekte verwaltet werden. Dazu gehört das Einholen der Zustimmung des Benutzers bei einigen Permissions sowohl während der Installation als auch zur Laufzeit. Diese Entscheidungen müssen dann gespeichert werden. Zusätzlich können die Permissions zur einfacheren Handhabung gruppiert und ihnen bestimmte GIDs zugeordnet werden.

6.2.1 Zustimmung der Permission

Apps müssen die Permissions, die sie für ihre Ausführung benötigen, in der AndroidManifest-Datei publizieren, indem sie den <uses-permission>-Tag benutzen. Ein Beispiel hierfür bietet Listing 6.1, in dem die Permissions „WRITE_EXTERNAL_STORAGE“, „READ_EXTERNAL_STORAGE“, „CAMERA“ und „BLUETOOTH“ angefordert werden. Vgl. [PermOv]

Listing 6.1 Deklaration von Permissions im Android Manifest

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3      package="de.caro.inventar">
4      <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
5      <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
6      <uses-permission android:name="android.permission.BLUETOOTH"/>
7      <uses-permission android:name="android.permission.CAMERA"/>
8      <application>
9          ...
10     </application>
11 </manifest>
```

Werden nur normale Permissions benötigt, so stimmt das System diesen automatisch zu. Sind die Permissions jedoch als dangerous eingestuft, so muss der Nutzer seine Zustimmung gegeben haben. Dies geschieht je nach Android Version entweder bei der Installation oder zur Laufzeit. Vgl. [PermOv]

Installationszeit

Läuft auf dem Gerät Android 5.1.1 (API Level 22) oder niedriger und ist die „targetSdkVersion“ der App 22 oder niedriger, so wird der Benutzer zur Installtionszeit zur Zustimmung der gefährlichen Permissions befragt. Hierbei können nur alle Permissions auf einmal gewährt werden. Stimmt der Nutzer zu, so sind alle Permissions zugelassen, lehnt der Benutzer ab, so wird die Installation abgebrochen. Benötigt ein Update weitere Permissions, so werden diese vor dem Update abgefragt und das Update kann nur durchgeführt werden, wenn sie eingeräumt werden. Vgl. [PermOv]

Laufzeit

Hat das Gerät Android 6.0 (API Level 23) oder höher und ist die „targetSdkVersion“ 23 oder höher, so werden keine Permission-Abfragen zur Installationszeit durchgeführt. Wird eine Permission dann von der App zur Laufzeit benötigt, so erscheint ein Dialog-Fenster, welches einen Zustimmungs- und einen Ablehnen-Button beinhaltet. Im Fall der Ablehnung kann die App auch ohne Zustimmung zu den Permissions laufen, jedoch nicht alle ihre Funktionen bieten. Außerdem können den Permissions jederzeit in den Systemeinstellungen einzeln zugestimmt oder diese widerrufen werden. Vgl. [PermOv][RTPerm]

6.2.2 Gruppierung von Permissions

Permissions werden thematisch gruppiert, unabhängig davon ob sie als normal oder dangerous eingestuft sind. Vgl. [PermOv]

Wenn Android 5.1 und „targetSdkVersion“ 22 oder niedriger verwendet wird, so werden die Permissions zur Installationszeit festgelegt. Dabei wird die Beschreibung der Gruppe dem Benutzer gezeigt, jedoch bei Zustimmung nur der entsprechend verlangten Permission zugesagt. Vgl. [PermOv]

Wird Android 6.0 und „targetSdkVersion“ 23 oder höher verwendet, so beeinflusst die Gruppierung nur die dangerous Permissions, die zur Laufzeit abgefragt werden. Hat die App keine Erlaubnis für eine Permission in der Gruppe, so wird dem Benutzer im Dialog-Fenster die Gruppenbeschreibung angezeigt, nicht die genaue Permission, die abgefragt wird. Stimmt der Nutzer zu, so wird auch nur die Zustimmung für die benötigte Permission vergeben. Wurde der App bereits eine andere gefährliche Permission der Gruppe gewährt, so erlaubt das System automatisch den Zugriff auf andere Permissions der Gruppe, ohne den Benutzer erneut zu fragen. Vgl. [PermOv]

```
generic_x86:/ # pm list permissions -g -d
Dangerous Permissions:

group:android.permission-group.CONTACTS
  permission:android.permission.WRITE_CONTACTS
  permission:android.permission.GET_ACCOUNTS
  permission:android.permission.READ_CONTACTS
```

Abbildung 6.2: Ausschnitt der Permission-Gruppen in Android 7.0

Ein Beispiel hierfür ist die Gruppe „CONTACTS“ (siehe Abbildung 6.2). Zu ihr gehören die Permissions „READ_CONTACTS“, „GET_ACCOUNT“ und „WRITE_CONTACTS“. Diese Gruppierungen kann man durch „pm list permissions -g -d“ im Terminal nachlesen, wobei „-g“ die Gruppierung darstellt und „-d“ nur die als dangerous eingestuften.

6.2.3 Speicherung der Permissions

Die Permissions werden seit Android 6.0 an unterschiedlichen Orten gespeichert, abhängig davon, ob sie zur Installationszeit festgelegt sind oder zur Laufzeit verändert werden können. Vgl. [CdPMS, Zeile 6041]

Installationszeit-Permissions

Alle Informationen zu den installierten Paketen befinden sich in der Datei „packages.xml“, und somit auch die Informationen über die zur App gehörigen Permissions. Da diese Datenbank nur aktualisiert wird, wenn eine App installiert, deinstalliert oder ein Update durchgeführt wird, können jedoch nur noch die zur Installationszeit festgelegten Permissions hier hinterlegt werden.

In Abbildung 6.3, die die Datei „packages.xml“ in einer Android Version vor 6.0 darstellt, ist zu sehen, dass hier alle Permissions im <perms>-Tag aufgelistet sind. Dies ist möglich, da in diesen Android Versionen eine Entscheidung des Benutzers über die Permissions zur Installationszeit getroffen werden musste. Im Ausschnitt der Datei „packages.xml“ in Abbildung 6.4, die auf Android Version 7.0 ausgegeben wurde, ist jedoch zu sehen, dass lediglich die Permission „BLUETOOTH“ aufgeführt ist. Auf beiden Android Emulatoren wurde aber die gleiche Version der App, mit dem AndroidManifest aus Listing 6.1 installiert. Da die Permission „BLUETOOTH“ als normal eingestuft wird, und somit zur Installationszeit automatisch zugewiesen wird, kann diese auch in der Datei abgespeichert werden. Die anderen im AndroidManifest aufgeführten Permissions sind hingegen dangerous, sodass diese bei einer Android-Version, die höher als 6.0 ist, erst zur Laufzeit abgefragt werden. Vgl. [PermOv]

```
<package name="de.caro.inventar" codePath="/data/app/de.caro.inventar-2.apk" nativeLibraryPath="/data/
ebe075" version="1" userId="10044">
<sigs count="1">
<cert index="2" key="30820ldd308201460201013000d06092a864886f70d0101050500303731163014060355040300c0d416
2555330le170d313830333038313033363033 5a170d343830323239313033363033 5a303731163014060355040300c0d416e647
330819f300d06092a864886f70d010101050003818d00308189028181 00bd83c75adeeb2b5a5c2c2e0496d228f66a5c6b31ef8
547bac0542df9067816a49b9b48b196a3a494a4ff60af0f82f595208ea23c5155d080c8ad457368bb2ef0bddf2f188afda9d59
75df70151d3721dcf1a8981b007d83804327150c14e0257ecbdd2e095420fc965f71b315da96ab76dd0a0986ee3d4b2479e575
21fc19113fbd6395019455ad24cd788b20b4e8dc55a011d455e3130c095f83754a21af032" />
</sigs>
<perms>
<item name="android.permission.READ_EXTERNAL_STORAGE" />
<item name="android.permission.CAMERA" />
<item name="android.permission.BLUETOOTH" />
<item name="android.permission.WRITE_EXTERNAL_STORAGE" />
</perms>
</package>
```

Abbildung 6.3: Ausschnitt der Datei packages.xml in Android 4.1

```
    <package name="de.caro.inventar" codePath="/data/app/de.caro.inventar-2" nativeLibraryPath="/data/
f9c0" it="16220d64816" ut="1622a67fcf5" version="1" userId="10065">
        <sigs count="1">
            <cert index="4" key="308201dd30820146020101300d06092a864886f70d0101050500303731163014060355
603550406130255553301e170d3138303330383130333630335a170d3438303232393130333630335a303731163014060355040
504061302555330819f300d06092a864886f70d010101050003818d00308189028181 00bd83c75adeeb2b5a5c2c2e0496d228f
ee6e0dd1febd547bac0542df9067816a49b9b48b196a3a494a4ff60af0f82f595208ea23c5155d080c8ad457368bb2ef0bddf2
50003818100575df70151d3721dcf1a8981b007d83804327150c14e0257ecbdd2e095420fc965f71b315da96ab76dd0a0986ee
df4171d6356921fc19113fbd6395019455ad24cd788b20b4e8dc55a011d455e3130c095f83754a21af032" />
        </sigs>
        <perms>
            <item name="android.permission.BLUETOOTH" granted="true" flags="0" />
        </perms>
        <proper-signing-keyset identifier="8" />
    </package>
```

Abbildung 6.4: Ausschnitt der Datei packages.xml in Android 7.0

Laufzeit-Permissions

Die zur Laufzeit erfragten Permissions, die zur Kategorie dangerous gehören, werden in der Datei „runtime-permissions.xml“ [CdS, Zeile 174] abgespeichert. Sie ist im Verzeichnis „/data/system/users/<*userID*>/runtime-permissions.xml“ zu finden. In dieser Datei wird außerdem gespeichert, ob die Permissions gewährt wurden oder nicht. Für die Abbildung 6.5a und Abbildung 6.5b wurde die App mit dem AndroidManifest aus Listing 6.1 auf Android 7.0 installiert. Dabei ist in Abbildung 6.5a sichtbar, dass nicht allen Permissions auf einmal zugestimmt werden müssen. In diesem Fall wurde die Kamera-Permission bewilligt, wobei die Permissions der Gruppe „Storage“ abgelehnt wurden. Nach erneuter Abfrage wurden alle Permissions gewährt, was zu Abbildung 6.5b führt. Vgl. [PermOv]

```
<pkg name="de.caro.inventar">
  <item name="android.permission.READ_EXTERNAL_STORAGE" granted="false" flags="1" />
  <item name="android.permission.CAMERA" granted="true" flags="0" />
  <item name="android.permission.WRITE_EXTERNAL_STORAGE" granted="false" flags="1" />
</pkg>
```

(a) Die Permissions sind nur teilweise genehmigt

```
<pkg name="de.caro.inventar">
  <item name="android.permission.READ_EXTERNAL_STORAGE" granted="true" flags="0" />
  <item name="android.permission.CAMERA" granted="true" flags="0" />
  <item name="android.permission.WRITE_EXTERNAL_STORAGE" granted="true" flags="0" />
</pkg>
```

(b) Die Permissions sind alle genehmigt

Abbildung 6.5: Ausschnitt der Datei runtime-permissions.xml in Android 7.0

6.2.4 Assoziation von Permissions mit Gruppen-IDs

Einige Permissions sind mit GIDs assoziiert. Dies liegt an der Durchsetzung der Permissions, die je nach Permission auf verschiedenen Ebenen erfolgt. Weiteres dazu ist in Abschnitt 6.3.1 beschrieben.

Die Zuordnung von Permissions zu GIDs ist statisch und in der Datei „platform.xml“ [CdPf], die im Verzeichnis „/data/etc/permissions/“ zu finden ist, beschrieben. Listing 6.2 zeigt hierbei die Assoziation von der Permission „BLUETOOTH“, die im AndroidManifest in Listing 6.1 benötigt wird, zur GID „net_bt“. Eine Abbildung dieser wörtlichen GID zu einer Zahl erfolgt durch die Datei „android_filesystem_config.h“ [CdFCf], von der der entsprechende Abschnitt in Listing 6.3 dargestellt ist. In diesem Beispiel wird der GID „net_bt“ nun die Zahl 3002 zugeordnet.

Listing 6.2 Ausschnitt der Datei platform.xml [CdPf]

```
30 <!-- The following tags are associating low-level group IDs with
31          permission names.  By specifying such a mapping, you are saying
32          that any application process granted the given permission will
33          also be running with the given group ID attached to its process,
34          so it can perform any filesystem (read, write, execute) operations
35          allowed for that group. -->
36     <permission name="android.permission.BLUETOOTH_ADMIN" >
37         <group gid="net_bt_admin" />
38     </permission>
39     <permission name="android.permission.BLUETOOTH" >
40         <group gid="net_bt" />
41     </permission>
```

Listing 6.3 Ausschnitt der Datei android_filesystem_config.h [CdFCf]

```
143 /* The 3000 series are intended for use as supplemental group id's only.
144  * They indicate special Android capabilities that the kernel is aware of. */
145 #define AID_NET_BT_ADMIN 3001 /* bluetooth: create any socket */
146 #define AID_NET_BT 3002       /* bluetooth: create sco, rfcomm or l2cap sockets */
147 #define AID_INET 3003         /* can create AF_INET and AF_INET6 sockets */
```

Wird einer Permission eine GID zugeordnet, so werden beim Gewähren dieser Permission dem entsprechen Prozess oder der App alle Rechte, die mit dieser GID verbunden sind, zugeschrieben. Dass die App nach der Zustimmung zur Permission mit der entsprechenden GID agiert, kann man in der Datei „packages.list“ sehen, die sich im Verzeichnis „/data/system/packages.list“ befindet. Der Inhalt dieser Dabei bietet eine Zusammenfassung der Pakete. Abbildung 6.6 zeigt hierzu den Eintrag der App Inventar, deren Manifest Listing 6.1 zeigt. Vgl. [CdPf]

```
generic_x86:/ # cd data/system
generic_x86:/data/system # grep 'de.caro.inventar' packages.list
de.caro.inventar 10065 1 /data/user/0/de.caro.inventar default 3002
```

Abbildung 6.6: Ausschnitt der Paketliste in Android 7.0

Hier wird an zweiter Stelle die UID der App angezeigt, welche in der Paket-Liste der Datei „packages.xml“ aus Abbildung 6.4 unter „userID“ auch zu sehen und in diesem Fall 10065 ist. Außerdem stehen an letzter Stelle die zugeordneten GIDs der App, die in diesem Fall 3002 ist. Sie wurde zuvor durch die Permission mittels „platform.xml“ und „android_filesystem_config.h“ abgebildet.

6.3 Durchsetzung von Permissions

In Android kümmert sich der *PackageManager* um die Verwaltung der Pakete, so dass dieser auch während der Laufzeit befragt wird, ob die Permissions vom System oder dem Benutzer zugesagt wurden oder nicht. Da jedoch Komponenten auf einem niedrigeren Level wie der Kernel oder Daemons in der Regel nicht direkt auf den Paket-Manager zugreifen können, wird dort bei der Entscheidung über Zugriffe mit der UID und den GIDs der App gearbeitet. Vgl. [Ele14, S. 26]

6.3.1 Kernel-Ebene

Der Zugriff auf Systemressourcen und Sockets wird vom Kernel basierend auf dem Besitzer und den UID und GIDs des anfragenden Prozesses gewährt oder verwehrt. Ein Beispiel hierfür ist der Umgang mit Internet-Sockets. Um diese zu erstellen, muss der Prozess zur Gruppe „inet“ gehören. Wie in Listing 6.3 zu sehen ist, wird diese mit der Nummer 3003 verknüpft. Um zur Gruppe „inet“ zu gehören muss die Permission „INTERNET“ einer App erteilt sein. Dies bedeutet, dass ohne diese Permission auch keine Netzwerk-Sockets erstellt werden können. Vgl. [Ele14, S. 26][Ele14, S. 30 f.]

6.3.2 Native-Daemons-Ebene

Native Daemons kommunizieren oft über lokale Sockets. Diese werden durch Files im Verzeichnis „/dev/socket/“ erstellt. Deshalb können die Linux-Zugriffsrechte für Dateien zur Zugriffskontrolle verwendet werden. Die Zugriffsrechte der Sockets beziehen sich jedoch ausschließlich auf den Benutzer und Mitglieder dessen Gruppe. Damit können Prozesse mit anderen UIDs und GIDs nicht auf den Socket zugreifen und über diesen Nachrichten an den Daemon senden. Da einige Permissions jedoch mit GIDs assoziiert sind, kann damit Apps der Zugriff auf Daemon-Sockets ermöglicht werden. Dadurch können auch Apps Nachrichten an Daemons über deren Sockets senden. Vgl. [Ele14, S. 31 ff.]

6.3.3 Framework-Ebene

Komponenten des Android Systems können den *PackageManager* aufrufen, um zu überprüfen, ob eine Permission gewährt ist oder nicht. Ein Beispiel des Vorgangs wird im weiteren Verlauf dieses Abschnitts beschrieben. Die Abfrage der Permissions erfolgt automatisch bei Nutzung einer durch Permissions geschützten Komponente oder Ressourcen. So wird beispielsweise beim Starten einer Activity während den Methoden „Context.startActivity()“ und „Activity.startActivityForResult()“ überprüft, ob die Activity durch die Aufrufenden überhaupt gestartet werden darf. Vgl. [PermOv]

Dieses Prinzip hat jedoch ein Problem im Zusammenhang mit Content Providern. Ein Beispiel dafür sind Anhänge von E-Mails. Auf die E-Mails sollte nicht jeder zugreifen können, da sie private Informationen enthalten. Ist der Anhang zum Beispiel eine pdf-Datei und soll durch ein externes Pdf-Anzeigeprogramm angeschaut werden, so hat dieses nicht die entsprechende Email-Permission wenn ihm der Uniform Resource Identifier (URI) zum Anhang übergeben wird. Hierfür gibt es „per-URI-Permissions“. Wird eine Activity gestartet oder ein Ergebnis empfangen, so kann „Intent.FLAG_GRANT_READ_URI_PERMISSION“ und/oder „Intent.FLAG_GRANT_WRITE_URI_PERMISSION“ gesetzt werden. Dies erlaubt der Activity auf die im Intent übergebenen Daten zuzugreifen, unabhängig davon, ob sie Zugriffsrechte für den zum Intent gehörenden Content Provider hat oder nicht. Vgl. [PermOv]

Beispielvorgang einer Permission-Abfrage

Auch aus dem App-Code, für den ein Beispiel in Listing 6.4 gegeben ist, kann man Abfragen über Permissions machen, indem man den Paket-Manager aufruft. Dies ist durch „PackageManager.checkPermission(String permission, String package)“ möglich. Vgl. [PermOv]

Listing 6.4 Abfrage beim Paket-Manager, ob die Kamera-Permission gewährt ist

```
1  public class MainActivity extends AppCompatActivity
2          implements NavigationView.OnNavigationItemSelectedListener {
3      ...
4
5      @Override
6      protected void onCreate(Bundle savedInstanceState) {
7          ...
8          PackageManager pm = getApplicationContext().getPackageManager();
9          int permissionResult = pm.checkPermission("android.permission.CAMERA", getPackageName());
10         String permissionGranted = permissionResult == PackageManager.PERMISSION_GRANTED ? " =
               Permission granted"
11                 : " = Permission denied";
12         Log.i(LOG_TAG, "Check for camera permission: " + String.valueOf(permissionResult) +
               permissionGranted);
13     }
14     ...
15 }
```

Zunächst muss eine Verbindung zum *PackageManager* hergestellt werden. Dieser Vorgang ist in Abbildung 6.7 skizziert. Die Anfrage „getPackageManager()“ wird durch den App-Kontext, der durch die Klasse „ContextImpl.java“ [CdCI] implentiert ist, realisiert (1). Dieser gibt am Ende eine Instanz des *ApplicationPackageManager* [CdAPM] zurück, der als Attribut eine Verbindung zum *PackageManagerService* [CdPMS] hat (2).

Zunächst ruft der *ContextImpl* die Methode „getPackageManager()“ in der Klasse „ActivityThread.java“ [CdAT] auf (3), die wiederrum beim *ServiceManager* [CdSM] eine Anfrage macht. Der ServiceManager verwaltet eine Liste aller bei ihm registrierten Services, die den Namen des Services und eine Binder-Referenz zum entsprechenden Service enthält (4).

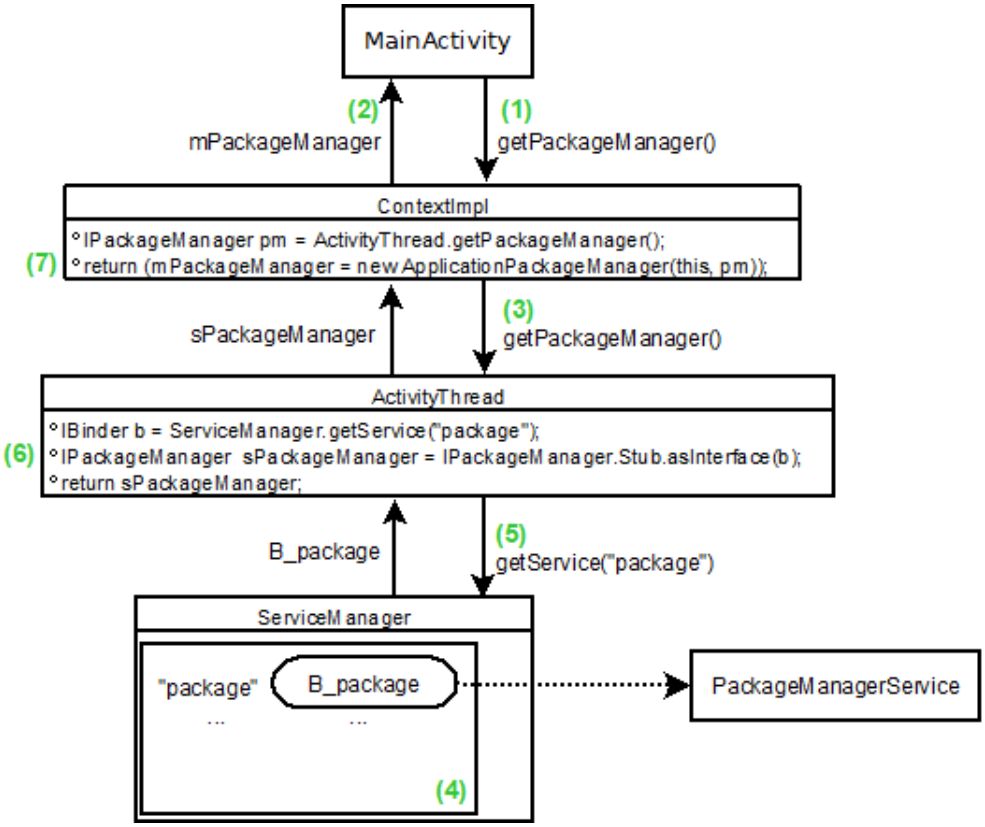


Abbildung 6.7: Herstellen der Verbindung der App zum PackageManager

Der *ActivityThread* möchte dabei eine Binder-Referenz, die auf den Service mit dem Namen „package“ verweist (5). Da der PackageManagerService sich unter diesem Namen registriert hat [CdPMS, Zeile 2325], wird eine Binder-Referenz von ihm zurückgegeben. Der ActivityThread kann dann eine Referenz zum PackageManagerService herstellen (6), die an ContextImpl zurückgegeben wird. ContextImpl erstellt dann eine Instanz des ApplicationPackageManager, die als Attribut den PackageManagerService erhält (7). Das Herstellen einer Verbindung zum PackageManagerService mittels des ServiceManagers durch Binder wird in Abschnitt 7.2.6 auf Seite 88 erklärt.

Die Methode „checkPermission(String permission, String package)“, die auf dem ApplicationPackageManager aufgerufen wird, gibt dieser an die Methode „checkPermission(String permission, String package, int userID)“ des PackageManagerService weiter. [CdAPM, Zeile 567] Diese Transaktion des Aufrufes an den PackageManagerService geschieht mittels Binder-Interprozesskommunikation. Der genaue Vorgang dazu wird in Abschnitt 7.2.5 auf Seite 82 genauer ausgeführt. Wie der *PackageManagerService* die Methode „checkPermission(String permission, String package, int userID)“ [CdPMS, Zeile 5298] behandelt, zeigt Abbildung 6.8.

Der PackageManagerService hat als Attribut ein Verzeichnis aller Paketnamen mit den entsprechend zugeordneten Paketen vom Typ „PackageParser.Package“ [CdPP] (1). Aus dieser wird zunächst das entsprechende Paket herausgesucht, in diesem Fall das Paket der Inventar-App. Dieses hat als Attribut „mExtras“, welches ein Objekt der Klasse „PackageSetting.java“ [CdPSt] hält (2).

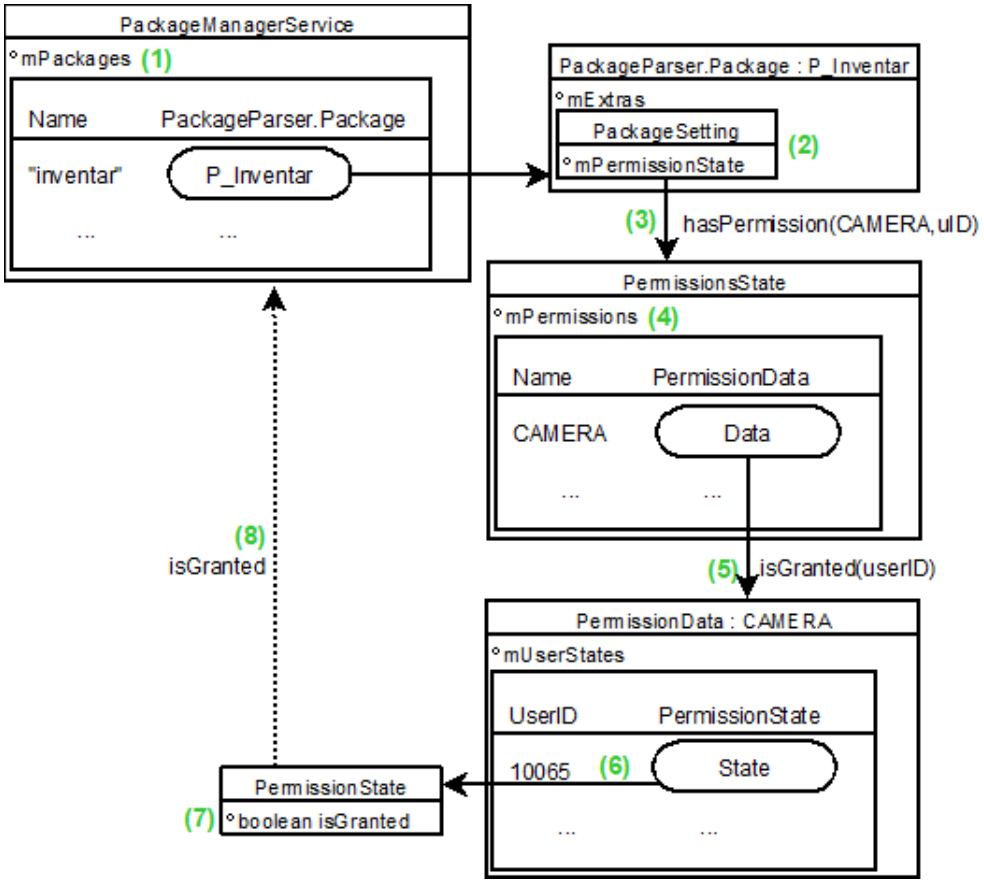


Abbildung 6.8: Durchführung von checkPermission() des PackageManagerServices

PackageSetting wiederum hat das Attribut „mPermissionState“, das einen *PermissionsState* [CdPS] darstellt. Auf dem PermissionsState wird dann die Methode „hasPermission(String permission, int userID)“ aufgerufen (3). PermissionsState verwaltet eine Liste von Permissions, die mit einem Objekt der Klasse „PermissionData.java“ [CdPS] verknüpft sind (4).

Aus dieser Liste wird die Permission „CAMERA“ herausgesucht und auf dem *PermissionData*-Objekt die Methode „isGranted(int userID)“ aufgerufen (5). Das PermissionData-Objekt hält eine Zuordnung von UIDs zu jeweils einem *PermissionState* [CdPS], aus dieser nun der PermissionState der UID der Inventar-App herausgesucht wird (6). Das PermissionState-Objekt hat dann ein boolsches Attribut „isGranted“ (7), das als Ergebnis bis zum PackageManagerService zurückgegeben wird (8), der dies zur MainActivity zurück gibt.

Wie in der Klasse des PackageManagers definiert, ist die Konstante „PERMISSION_GRANTED“ „0“ während für „PERMISSION_DENIED“ „-1“ festgelegt ist. [CdPM, Zeile 535 ff.] Da der Inventar-App der Kamerazugriff erlaubt ist, wie in Abbildung 6.5b sichtbar ist, wird in diesem Fall „PERMISSION_GRANTED“ als Antwort auf checkPermission(...) zurückgegeben (siehe Abbildung 6.9).

```
15:32:48.279 2887-2887/de.caro.inventar I/MainActivity: Check for camera permission: 0 = Permission granted
```

Abbildung 6.9: Logcat Ausgabe der Activity aus Listing 6.4 in Android 7.0

7 Binder-Interprozesskommunikation

Androids *Binder*-Interprozesskommunikation liegt den IPC-Mechanismen von Android zugrunde. Binder setzt sowohl Intraprozess- als auch Interprozesskommunikation um und bietet eine Verwaltung von Lebenszyklen von Objekten durch Referenzzählung. Die Binder-Funktionalität ist durch den Binder-Treiber im Kernel umgesetzt. Android unterstützt keine Linux-IPC-Mechanismen wie Semaphoren, gemeinsame Speichersegmente oder Nachrichtenwarteschlangen. Vgl. [TH16, S. 971 f.][SC13]

Die Ursprünge von Binder liegen in Arbeiten, die für BeOS von Be Inc. begonnen, und von Palm gekauft wurden. Deren Ergebnisse wurden als *OpenBinderProject* [BinDok] veröffentlicht. Google übernahm dann eine der essentiellen Entwickler von OpenBinder, Dianne Hackborn, die dann Binder für Android mitentwickelte. Androids Binder ist jedoch nicht als Erweiterung von OpenBinder geschrieben, sondern basiert nur auf den Ideen und der Basis dessen. Die Implementierung von Binder ist an die Android-Architektur angepasst. Vgl. [Yag13, S. 39][DHack]

Dieses Kapitel beschreibt Binder-IPC zunächst allgemein, während auf die Implementierung in Android in einem weiteren Abschnitt eingegangen wird. Außerdem werden die Funktionen und Sicherheitsaspekte dargestellt.

7.1 Kommunikation durch Binder

Zum Binder-Framework, das die ganze Binder-IPC-Architektur beschreibt, gehören mehrere Komponenten. Diese werden im folgenden Abschnitt benannt und weiterführend die Funktionsweise von Binder erklärt. Die Implementierung der Komponenten und deren Interaktion in Android wird in Abschnitt 7.2 näher beschrieben.

7.1.1 Komponenten

Binder besteht aus einigen Komponenten. Zunächst der **Binder-Treiber**. Dieser ist im Kernel umgesetzt und setzt die Kommunikation zwischen Prozessen auf Kernel-Ebene um. Das **IBinder-Interface** beschreibt Methoden, die ein Binder-Objekt implementieren muss, um Binder-IPC realisieren zu können. Ein **Binder-Objekt** ist dabei eine Implementierung des IBinder-Interfaces. Um Binder-Objekte im System eindeutig zu identifizieren gibt es den **Binder-Token** oder ***Handle***, der ein zwischen Binder-Objekten eindeutiger 32-Bit-Wert ist. Ein **Binder-Service** ist ein Service, der seine Funktionen anderen Prozessen anbietet und dazu eine Implementierung eines Binder-Objekts ist. Der **Binder-Klient** ist dann ein Objekt, das Funktionen eines Binder-Services nutzen möchte. Das Aufrufen einer Funktion eines Binder-Objekts, bei der Daten über das Binder-Protokoll

gesendet werden, nennt sich **Binder-Transaktion**. Um die Binder-Services den Klienten zugänglich zu machen, führt der **Context Manager** eine Übersicht aller Prozesse, zu denen Klienten einen Handle von ihm erhalten können. Vgl. [Gar13][RIBin][RBin][Yag13, S. 69]

7.1.2 Funktionsweise

Da die Android-Sicherheitsarchitektur auf Prozess- und Datenisolation beruht, können Prozesse weder auf andere Speicherbereiche zugreifen, noch Funktionen von anderen Prozessen direkt aufrufen um diese zu nutzen. Deshalb wird der Binder-Treiber des Kernels zur Kommunikation genutzt (siehe Abbildung 7.1. Vgl. [SC13][Gar13]

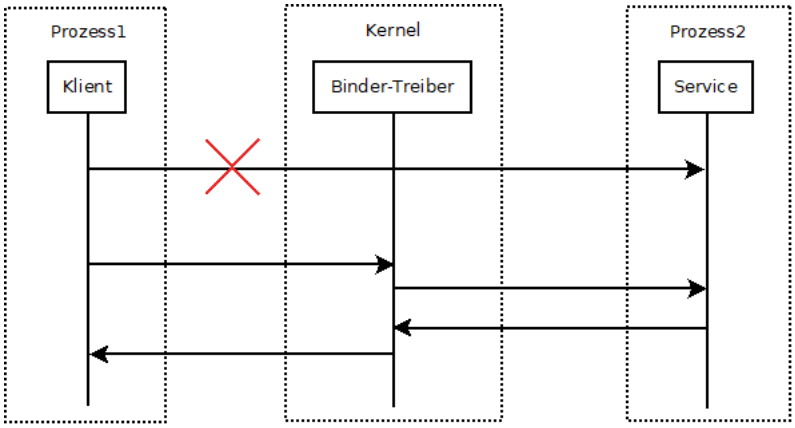


Abbildung 7.1: Kommunikation zwischen Prozessen über den Kernel

Um den Entwicklern die Implementierung von IPC zu erleichtern, wird der Binder-Mechanismus auf verschiedenen Ebenen abstrahiert, so dass der Klient über *Proxies* und *Stubs* mit dem Service kommunizieren kann. Dies wird in Abbildung 7.2 dargestellt. Außerdem sollen Services mehrere Klienten gleichzeitig bedienen können. Dazu hält ein Service mehrere Threads in seinem Prozess, die dann mit verschiedenen Klienten kommunizieren können. Vgl. [RIBin][BBG14][Gar13]

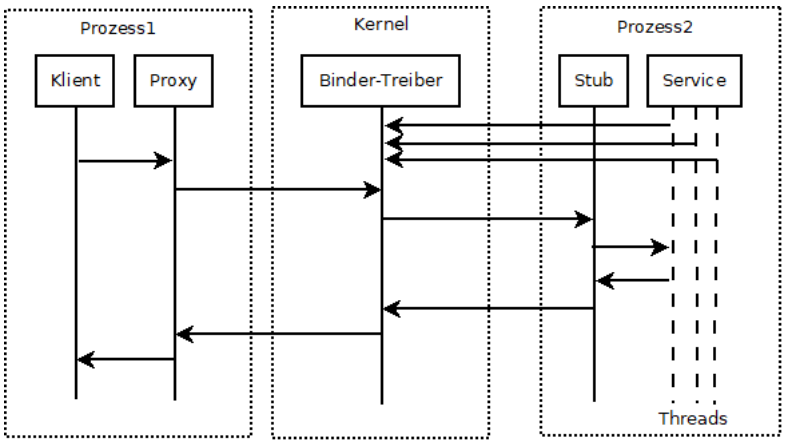


Abbildung 7.2: Erleichterung der Kommunikation über den Kernel mittels Proxy und Stub

Für den Zugriff auf den Service benötigt der Klient einen Handle. Dazu gibt es den Context Manager, bei dem sich Services mit ihrem Namen und ihrem Handle registrieren können. Solch eine Registrierung zeigt Abschnitt 2 der Abbildung 7.3. Benötigt der Klient nun den Handle für einen bestimmten Service, kann er den Context Manager nach diesem Fragen, was Abbildung 7.3 in Abschnitt 3 zeigt. Aus Sicherheitsgründen kann es nur einen Context Manager geben, so dass der Treiber nur einmal die Registrierung als solcher akzeptiert. Die Registrierung als Context Manager ist durch Abschnitt 1 in Abbildung 7.3 umgesetzt. Vgl. [Gar13][CdSMc][Ele14, S. 8 f.]

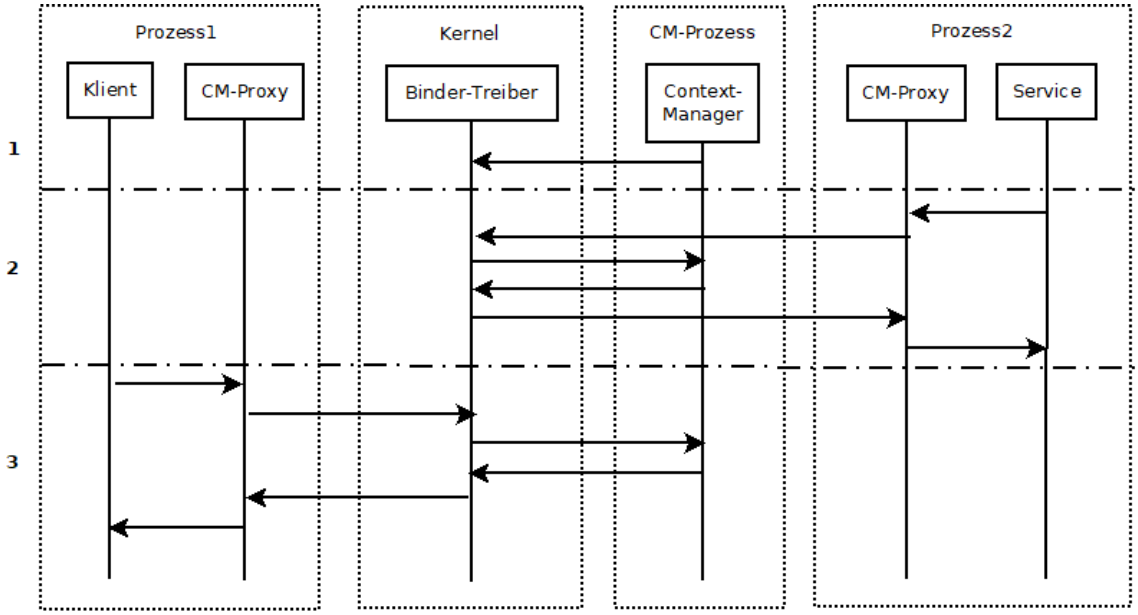


Abbildung 7.3: Die Funktionen des Context-Managers

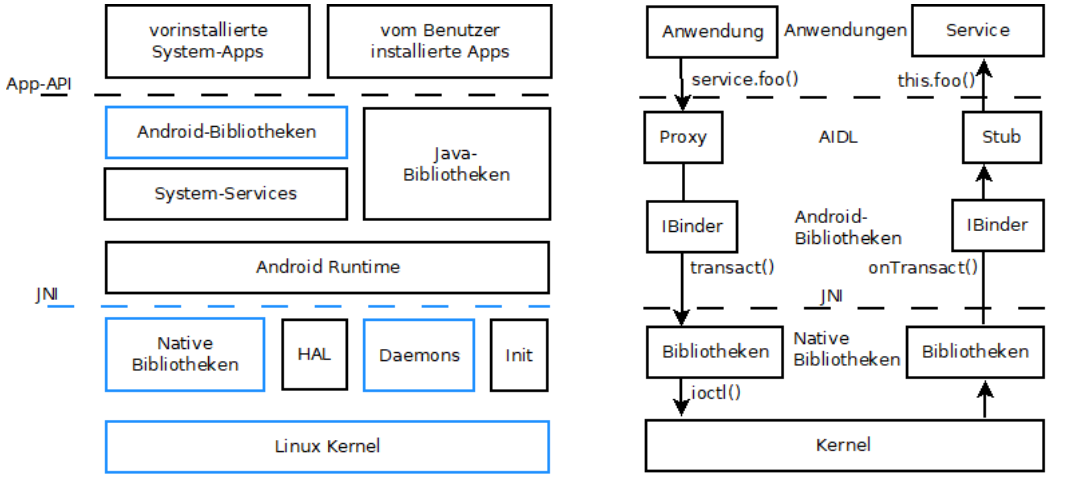
7.2 Implementierung in Android

Zunächst wird im folgenden Abschnitt ein Überblick über die weitreichende Implementierung von Binder in der Android-Architektur gegeben, während nachfolgend einzelne Bereiche und Funktionen genauer erläutert werden.

7.2.1 Binder-Bestandteile in der Android-Architektur

Um Binder in Android umzusetzen sind viele Komponenten der Android-Architektur beteiligt. Die an der Implementierung beteiligten Komponenten sind in Abbildung 7.4a blau hervorgehoben. Eine grobe Übersicht der Interaktion der Instanzen dieser Komponenten bei einer Transaktion bietet Abbildung 7.4b. In der Mitte sind dabei die entsprechenden Komponenten der Android-Architektur und das Android Interface Definition Language (AIDL) genannt, deren Funktion in weiteren Abschnitten genauer erläutert wird.

Auf unterster Ebene ist der Kernel der wohl wichtigste Bestandteil, da dieser die Transaktionen von Binder ausführt und Daten vom sendenden Prozess in den Zielprozess weiterleitet. Dies geschieht mittels „ioctl“-Aufrufen. Der Binder-Treiber ist durch die Datei „binder.c“ [CdBc] implementiert. Die



(a) Komponenten der Android-Architektur **(b)** Interaktion der Binder-Komponenten

Abbildung 7.4: Komponenten der Android-Architektur, welche am Binder-Framework beteiligt sind und deren Interaktion

Implementierung des Context-Managers des Treibers wird durch den Daemon „service_manager“ [CdSMc] umgesetzt. Die nativen Bibliotheken, die zur Kommunikation mit dem Treiber genutzt werden, befinden sich im Verzeichnis „/platform/frameworks/native/libs/binder“ [CdBLib].

Um mit den Klassen der Android-API kommunizieren zu können, benötigt es ein JNI, das durch die Klasse „core/jni/android_util_Binder.cpp“ [CdBJni] umgesetzt ist. Dadurch werden die Aufrufe „transact()“ und „onTransact()“ von Java-Objekten auf Klassen der Bibliotheken und andersherum übertragen.

Darüberliegend befinden sich dann einige Java-Klassen, die dem Entwickler direkt zur Verfügung stehen. Dazu gehören „Binder.java“ [CdBin], „IBinder.java“ [CdIBin], „BinderInternal.java“ [CdBI] und „Parcel.java“ [CdP], die im Verzeichnis „/android/os/“ zu finden sind. Um die Programmierung für die Entwickler zu erleichtern, ist das Ziel, dass diese im Code der Anwendung direkt die Methode des Services durch die darunterliegende Implementierung eines Proxys aufrufen können. Außerdem soll im Service auch nur die entsprechende Methode zu implementieren sein und keine Arbeit mit der Kommunikation mittels Binder entstehen. Dazu gibt es das aidl-Tool, das mittels AIDL-Klassen die Stubs und Proxys automatisch generiert und die Kommunikation mit darunterliegenden Komponenten umsetzt.

7.2.2 Funktionen

Höhere Level von IPC in Android sind auf Binder aufgebaut. Dazu gehören Intents, Content Providers oder Services. Die Hauptfunktion von Binder ist dabei das Übermitteln von Nachrichten. Dies geschieht mittels so genannten Transaktionen. Deren Funktionsweise ist bereits in Abschnitt 7.1.2 dargestellt, während auf die Implementierung dieser im Folgenden weiter eingegangen wird. Obwohl die Transaktionen standardmäßig synchron sind, ist auch eine Implementierung von asynchroner Kommunikation mittels Binder möglich.

Neben Transaktionen zum Versenden von Nachrichten implementiert Binder jedoch auch so genannte *Death Notifications*. Diese sind dafür gedacht, dass Services darüber informiert werden können, falls Klienten, die eine Binder-Referenz von ihnen halten, beendet werden oder abstürzen. Dann können die Services Ressourcen freigeben, indem nicht mehr benötigte Objekte gelöscht oder „Listener“ beendet werden. Außerdem kann sich durch den Thread nun ein anderer Klient an den Services binden. Dazu muss der Service ein Objekt vom Interface „IBinder.DeathRecipient“ halten, welcher als Empfänger der Todesnachricht agiert. Wird der Empfänger auf der Binder-Referenz des Klienten durch die Methode „linkToDeath(IBinder.DeathRecipient dr, int flags)“ registriert, so wird beim Beenden des Prozesses die Methode „DeathRecipient.binderDied()“ aufgerufen. Diese muss vom Empfänger implementiert sein und kümmert sich dann um das weitere Vorgehen. Vgl. [RIBin][Ele14, S. 9][Gar13]

Um die Performance zu verbessern führt Binder außerdem Referenzzählung durch. Dies sorgt dafür, dass Binder-Objekte automatisch gelöscht werden können, falls keine Referenzen auf diese mehr existieren, da sie zu diesem Zeitpunkt dann nicht mehr benötigt werden. Dadurch kann der Speicherplatzbedarf reduziert werden. Vgl. [Ele14, S. 9]

7.2.3 Kernel und native Bibliotheken

Das vom Kernel angebotene IPC-Interface ist mittels eines Treibers umgesetzt, der in „/dev/binder“ zu finden ist. Auf diesen wird mittels einem „ioctl()“-Aufruf zugegriffen. Vgl. [Ele14, S. 5 f.][TH16, S. 973]

Jede Transaktion, die dem Kernel übergeben wird, beinhaltet sowohl das Ziel als auch die Daten der Operation, die ausgeführt werden soll. Vgl. [Ele14, S. 5 f.][TH16, S. 973]

Die Angabe des Ziels unterscheidet sich darin, ob das aufzurufende Objekt sich im aufrufenden Prozess oder einem anderen befindet. Befindet sich das Objekt im gleichen Prozess, so ist es eine virtuelle Adresse direkt zum Objekt. Befindet es sich jedoch in einem anderen Prozess, so ist es ein abstrakter Handle zum Objekt. Dabei hat der Kernel eine Übersicht über alle Objekte und ihre Handles. Somit weiß der Kernel in welchen Prozess die Transaktion weiter gegeben werden muss. Vgl. [AR14][Gar13]

Um eine Transaktion weiter zu leiten kopiert der Kernel die Nachricht und fügt die Identität des Absenders für den Empfänger zur Nachricht hinzu. Dann weckt der Kernel einen Thread des Zielprozesses und übergibt ihm die Nachricht. Vgl. [AR14][Gar13]

Die Weitergabe der Nachrichten durch den Kern erfolgt durch geteilten Speicher. Der Kern erzeugt für jeden Prozess, der Binder-Transaktionen empfangen kann, einen Speicherbereich, den er sich mit diesem teilt. Dadurch kann eine Transaktion direkt in den Speicherbereich des Zielprozesses kopiert werden, der diesen dann weiter verarbeitet. Vgl. [TH16, S. 973 f.]

Der zur Nutzung des Treibers erforderliche „ioctl()“-Aufruf wird von Komponenten der nativen Bibliotheken gemacht. Die entsprechenden Klassen befinden sich in „/libs/binder“ [CdBLib]. Die entscheidende Klasse zur Interaktion mit dem Kernel ist dabei Die Klasse „IPCThreadState.cpp“ [CdIpc]. Hier wird schließlich der Aufruf „ioctl(mProcess->mDriverFD, BINDER_WRITE_READ, &bwr)“ gemacht. Durch das erste Argument des Aufrufs wird der Binder-Treiber als Ziel des „ioctl()“ angegeben. Der Treiber soll mit dem Aufruf wie im zweiten Argument angegeben, verfahren. Die Binder-Nachricht wird durch das dritte Argument übermittelt.

Durch JNI mit der Klasse „android_util_Binder.cpp“ [CdBJni] werden die Komponenten der nativen Bibliotheken auf die der Java-Umgebung abgebildet. Dies ist auf Proxy-Seite eine Instanz der Klasse „BinderProxy“ [CdBin], während auf der Seite des Services eine Instanz des „Binder“-Objekts [CdBin] vorliegt.

7.2.4 Binder und IBinder der Android-API

Die Klasse „IBinder.java“ [CdIBin] bietet das Basis-Interface für die Komminukation mittels Binder. Es beschreibt ein Protokoll, um mit Binder-Objekten zu interagieren. „Binder.java“ [CdBin] ist dabei die Basis-Implementierung des IBinder-Interfaces. Entwickler können von dieser Klasse erben, um Kommunikation mit ihrem Service zu implementieren. Dazu kann jedoch auch das aidl-Tool genutzt werden, das aus einer Interface-Beschreibung in AIDL automatisch die Binder-Subklasse generiert. Vgl. [RIBin][RBin]

Die Schlüsselmethoden der Binder-API sind „transact()“ auf Seiten des Proxys, die auf der anderen Seite „onTransact()“ beim Stub auslöst. Die Aufrufe der Transaktions-API sind standardmäßig synchron, so dass vom transact()-Aufruf nicht zurückgekehrt wird, solange kein Ergebnis von onTransact() erhalten wurde. Vgl. [RIBin]

Daten werden dabei mittels eines *Parcels* [CdP] transferiert. Die Umwandlung der Java-Objekte in Parcels und zurück geschieht dabei im Proxy und Stub. Vgl. [RIBin]

7.2.5 AIDL

Mittels AIDL können Interfaces definiert werden, auf die sich Klienten und Services einigen um die Kommunikation zwischen ihnen umzusetzen. Da Prozesse nicht direkt auf Speicher von anderen Prozessen zugreifen können, müssen Daten auf Kernel-Ebene transportiert werden. Dort können keine Java-Objekte, mit denen im Code gearbeitet wird, verarbeitet werden. Dazu werden diese in Android in Parcels konvertiert. Auf der Server-Seite muss das Parcel dann auch wieder in ein Java-Objekt umgewandelt werden. Diese Konvertierungen werden durch AIDL automatisch implementiert. Vgl. [Daidl]

Aus der „.aidl“-Datei werden außerdem ein Proxy und Stub für das Interface gebildet. Der Proxy bildet dabei die Methodenaufrufe auf den transact()-Aufruf der niederen Level ab, während der Stub den onTransact()-Aufruf, den er erhält wieder in einen Methodenaufruf des Services konvertiert. Vgl. [Daidl]

Beispiel

Dieser Abschnitt gibt ein Beispiel eines einfachen Services zur Addition. Es werden die generierten Komponenten Stub und Proxy erklärt, sowie ein Beispiel einer Implementierung des Interfaces gegeben. Außerdem wird der Nachrichtenfluss bei einem Aufruf der Methode des Services vom Klienten dargestellt. Die „.aidl“-Datei ist dabei in Listing 7.1 gezeigt.

Listing 7.1 „.aidl“-Datei für das Interface IAidlPlusInterface

```
1 interface IAidlPlusInterface {
2     int plus(int a, int b);
3 }
```

Durch das aidl-Tool wird automatisch die Klasse „IAidlPlusInterface.java“ generiert, welche zum einen die Methoden angibt, welche durch das Interface implementiert werden aber auch Implementierungen für einen Stub und Proxy gibt. Diese zwei Teile sind in Listing 7.5 und Listing 7.4 gezeigt und werden im weiteren Verlauf genauer erklärt. Ziel des Stubs ist es, dass der Entwickler des Services nur die Methoden implementieren muss, jedoch nicht die Kommunikation mit Klienten. Dazu hat der Service ein Binder-Objekt, das die Additionsmethode implementiert. Ein Beispiel hierfür ist in Listing 7.2 zu sehen. Wird die Methode onBind() des Services aufgerufen, so gibt diese ein Objekt des Interfaces IBinder zurück. Da der IAidlPlusInterface.Stub von Binder erbt, wie in Zeile vier von Listing 7.5 erkennbar ist, und Binder das IBinder-Interface implementiert, kann dieses Objekt zurückgegeben werden.

Listing 7.2 Implementierung des PlusServices

```
1  public class PlusService extends Service {
2          ...
3      @Override
4      public IBinder onBind(Intent intent) {
5          return binder;
6      }
7
8      private final IAidlPlusInterface.Stub binder = new IAidlPlusInterface.Stub() {
9          @Override
10         public int plus(int a, int b) throws RemoteException {
11             return a + b;
12         }
13     };
14 }
```

Die Implementierung eines Aufrufs der Methode „plus()“ des Services ist in Listing 7.3 dargestellt. Der Klient kann den Service zum Beispiel durch einen Intent mittels einer *ServiceConnection* starten. Diese reagiert auf den onBind()-Aufruf des Services und bekommt den Service als IBinder. Dann kann ein Proxy mittels der Methode „IAidlPlusInterface.Stub.asInterface(service)“ hergestellt werden. Auffällig ist vor allem, dass der Zugriff auf den Service für den Entwickler wie ein Methodenaufruf eines lokalen Objekts wirkt und sich der Entwickler auch hier nicht um die Implementierung der Kommunikation kümmern muss.

Listing 7.3 Beispiel zum Aufrufen des PlusServices

```
1  public class MainActivity extends AppCompatActivity{
2      ...
3      IAidlPlusInterface plusService = null;
4      //bindService(intent, mConnection, flags);
5      ...
6      private ServiceConnection mConnection = new ServiceConnection() {
7          public void onServiceConnected(ComponentName className,
8                                         IBinder service) {
9              plusService = IAidlPlusInterface.Stub.asInterface(service);
10         } ...
11     };
12     ...
13     int result = plusService.plus(1,1);
14     }
```

Proxy Listing 7.4 zeigt die Implementierung des Proxys. Dieser bekommt durch seinen Konstruktor das IBinder-Objekt „remote“, das dann die Methode nach der Weitergabe des Aufrufs implementiert. Der Proxy des PlusServices hat nur die Methode „plus()“ zu implementieren. Dabei sieht man, dass hier die Umwandlung von den Java-Argumenten in ein Parcel geschieht. Dann wird die Methode „transact()“ auf dem „remote“-Binder-Objekt ausgeführt. Dieses Objekt entspricht dem BinderProxy, wie in den nächsten Abschnitten gezeigt wird.

Listing 7.4 Stub.Proxy-Implementierung für IAidlPlusInterface in der „IAidlPlusInterface.java“-Klasse

```
1  public interface IAidlPlusInterface extends android.os.IInterface{
2
3  public static abstract class Stub extends android.os.Binder implements
       de.caro.inventar.IAidlPlusInterface
4  {
5  ...
6  private static class Proxy implements de.caro.inventar.IAidlPlusInterface{
7  private android.os.IBinder mRemote;
8  Proxy(android.os.IBinder remote){mRemote = remote;}
9  ...
10 @Override public int plus(int a, int b) throws android.os.RemoteException{
11 android.os.Parcel _data = android.os.Parcel.obtain();
12 android.os.Parcel _reply = android.os.Parcel.obtain();
13 int _result;
14 try {
15 _data.writeInterfaceToken(DESCRIPTOR);
16 _data.writeInt(a);
17 _data.writeInt(b);
18 mRemote.transact(Stub.TRANSACTION_plus, _data, _reply, 0);
19 _reply.readException();
20 _result = _reply.readInt();
21 }
22 finally {...}
23 return _result;
24 }}...}
```

Stub Die Implementierung des Stubs zeigt Listing 7.5, der von der Klasse Binder [CdBin] erbt. Der erste wichtige Teil besteht aus der Methode „asInterface()“, die dafür genutzt wird, eine Verbindung zum lokalen Service herzustellen oder einen Proxy zu erhalten. Zunächst wird durch die Methode „queryLocalInterface“ überprüft, ob der Service lokal im Prozess eine Instanz hat. Ist dies der Fall, so wird diese dem Klienten zurück gegeben. Andernfalls wird ein neuer Proxy erstellt. Des Weiteren übernimmt der Stub die Implementierung der Methode „onTransact()“, die für die Konvertierung vom Parcel zu Java-Objekten und zurück wichtig ist. Außerdem identifiziert sie, welche Methode des Services aufgerufen wird.

Listing 7.5 Stub-Implementierung für IAidlPlusInterface in der „IAidlPlusInterface.java“-Klasse

```
1  public interface IAidlPlusInterface extends android.os.IInterface{
2  /** Local-side IPC implementation stub class. */
3
4  public static abstract class Stub extends android.os.Binder implements
       de.caro.inventar.IAidlPlusInterface
5  {
6  private static final java.lang.String DESCRIPTOR = "de.caro.inventar.IAidlPlusInterface";
7  ...
8  /**
9   * Cast an IBinder object into an de.caro.inventar.IAidlPlusInterface interface,
10  * generating a proxy if needed.
11  */
12 public static de.caro.inventar.IAidlPlusInterface asInterface(android.os.IBinder obj){
13 if ((obj==null)) {return null;}
14 android.os.IInterface iin = obj.queryLocalInterface(DESCRIPTOR);
15 if (((iin!=null)&&(iin instanceof de.caro.inventar.IAidlPlusInterface))) {
16 return ((de.caro.inventar.IAidlPlusInterface)iin);
17 }
18 return new de.caro.inventar.IAidlPlusInterface.Stub.Proxy(obj);
19 }
20 ...
21 @Override public boolean onTransact(int code, android.os.Parcel data, android.os.Parcel reply,
       int flags) throws android.os.RemoteException{
22 switch (code){
23 case INTERFACE_TRANSACTION:{
24 reply.writeString(DESCRIPTOR);
25 return true;}
26 case TRANSACTION_plus:{
27 data.enforceInterface(DESCRIPTOR);
28 int _arg0;
29 _arg0 = data.readInt();
30 int _arg1;
31 _arg1 = data.readInt();
32 int _result = this.plus(_arg0, _arg1);
33 reply.writeNoException();
34 reply.writeInt(_result);
35 return true;}
36 }
37 return super.onTransact(code, data, reply, flags);
38 }
39 ...}
```

Transaktion Die Transaktion ist durch Abbildung 7.5 skizziert. Sie stellt zur Linken die Binder-Vorgänge auf der Seite des Klienten dar, während zur Rechten alle Binder-Vorgänge auf Service-Seite abgebildet sind.

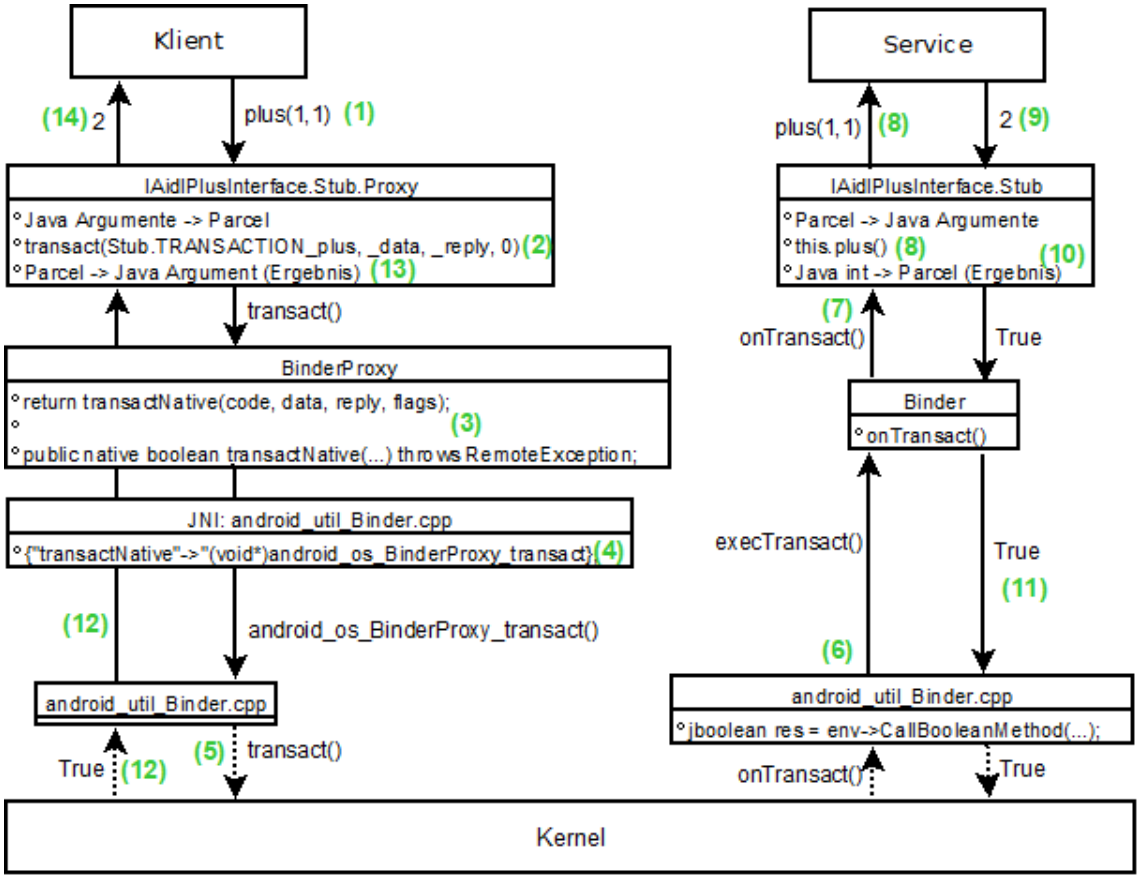


Abbildung 7.5: Vorgang einer Transaktion zwischen Klient und Service oberhalb des JNIs

Zunächst wird der Methodenaufruf des Klienten an den Proxy weitergegeben (1), der die Argumente in ein Parcel umwandelt. Dann wird auf dem „remote“-Objekt, das durch ein BinderProxy-Objekt [CdBin] realisiert ist, die Methode „transact()“ ausgeführt (2). Wichtig ist dabei, dass sowohl ein Parcel für die Argumente, als auch ein „leeres“ Parcel für das Ergebnis der Operation weiter gegeben wird. Außerdem wird ein Identifikator für die Operation übergeben, der dann im Stub zum Auswählen der passenden Methode genutzt wird.

Im BinderProxy [CdBin] wird in der „transact()“- die „transactNative()“-Methode aufgerufen, die „native“ deklariert ist (3). Somit wird diese durch eine JNI-Klasse [CdBJni] auf eine native Methode projiziert (4), die dann den weiteren Verlauf der Transaktion übernimmt (5). Dies ist hier nicht dargestellt.

Auf Seiten des Services bildet JNI [CdBJni] den „onTransact()“-Aufruf zunächst auf die Methode „execTransact()“ in der Binder-Klasse [CdBin] ab (6). Diese ruft die Methode „onTransact()“ auf, die durch das Überschreiben des Stubs auch dort ausgeführt wird (7). Der Stub konvertiert zunächst die Parcels und ruft dann auf sich selbst die Methode „plus()“ auf (8). Da diese vom PlusService überschrieben wurde, wird die Implementierung im Service verwendet (8).

Dieser gibt das Ergebnis zurück (9), das dann in das mitgegebene Antwort-Parcel geschrieben wird (10). Der Kernel kann auf das veränderte Antwort-Parcel durch den mit dem Prozess geteilten Speicher zugreifen. Nun wird nach erfolgreicher Transaktion der Wert „true“ bis zum JNI zurückgegeben (11). Der Kernel transferiert die Ergebnisse nun zurück zum Klienten.

Erhält die JNI-Methode des Klienten nun das Ergebnis der Transaktion mittels des boolschen Wertes, der angibt, ob die Transaktion erfolgreich war, so gibt sie diesen weiter (12), vor allem um zu signalisieren, dass die Transaktion abgeschlossen ist. Da der Stub in das vom Kernel weitergeleitete Ergebnis-Parcel geschrieben hat, das vom Kernel auch zurück-geleitet wurde, kann nun der Proxy dieses wieder in einen Java-Integer umwandeln (13) und an den Klienten zurückgeben (14).

7.2.6 Context Manager

Der *Context Manager* von Binder setzt eine Art Datenbank aller Services mit deren Referenz um. Dadurch können Klienten den Zugang zu Services erhalten. Sie fragen beim Context Manager über den Namen nach einem Service und erhalten den entsprechenden Handle, falls der Service sich bis dahin beim Context Manager registriert hat. Dieser Handle wird durch die Abstraktion in eine objektorientierte Umgebung im Proxy auf nativer Ebene als Attribut gespeichert. Dadurch kann der Entwickler auf dem BinderProxy-Objekt agieren. Während einer Binder-Transaktion wird das Ziel der Transaktion während der Bearbeitung auf Klienten-Seite durch die nativen Komponenten wieder zum Handle aufgelöst, mit dem der Kernel weiter verfährt. Vgl. [SC13][Ele14, S. 9]

Der Context Manager wird in Android durch den *Service Manager* umgesetzt. Dieser wird als nativer Daemon sehr früh im init-Prozess gestartet und durch „service_manager.c“ [CdSMc] implementiert. Ein Ausschnitt aus der main-Funktion dieser Klasse ist in Listing 7.6 gegeben.

Listing 7.6 Ausschnitt der main-Funktion des service_manager-Daemons [CdSMc]

```
365 int main(int argc, char** argv)
366 {
367     struct binder_state *bs;
368     union selinux_callback cb;
369     char *driver;
370     if (argc > 1) {
371         driver = argv[1];
372     } else {
373         driver = "/dev/binder";
374     }
375     bs = binder_open(driver, 128*1024);
376     ...
377     if (binder_become_context_manager(bs)) {
378         ALOGE("cannot become context manager (%s)\n", strerror(errno));
379         return -1;
380     }
381     ...
382     binder_loop(bs, svcmgr_handler);
383     return 0;
384 }
```

Hier ist zu sehen, dass der Service Manager zunächst den Binder-Treiber im Verzeichnis „/dev/-binder“ öffnet. Danach ruft er die Methode „binder_become_context_manager(bs)“ auf, die durch den „ioctl“-Aufruf „ioctl(bs->fd, BINDER_SET_CONTEXT_MGR, 0);“ [CdSMbc, Zeile 149] realisiert ist. Am Ende wird die Methode „binder_loop(bs, svcmgr_handler)“ [CdSMbc, Zeile 389ff.] aufgerufen, die den Service Manager wartend auf Anfragen von anderen Prozessen hinterlässt. Diese werden dann durch die Funktion „svcmgr_handler“ [CdSMc, Zeile 251ff.] verarbeitet. Dort werden die Kommandos „SVC_MGR_GET_SERVICE“, „SVC_MGR_CHECK_SERVICE“, „SVC_MGR_ADD_SERVICE“ und „SVC_MGR_LIST_SERVICES“ verarbeitet, die zur Verwaltung der beim Context Manager gelisteten Services verwendet werden.

Da der Context-Manager von allen Services zur Registrierung gefunden werden muss, erhält dieser den fest definierten Binder-Token „0“ [CdSMbh] (siehe Listing 7.7).

Listing 7.7 Definition der Identifikation des Context-Managers mittels der „0“ [CdSMbh]

```
30 /* the one magic handle */
31 #define BINDER_SERVICE_MANAGER  0U
```

Verbindung zum Service Manager der Android-API

Der Entwickler möchte natürlich nicht direkt mit dem Daemon kommunizieren müssen, um einen Handle zu einem Service zu erhalten. Dazu wird der Service-Manager auch in der Android-API durch die Klasse „ServiceManager.java“ [CdSM] implementiert.

In Abschnitt 6.3.3 auf Seite 73 wurde ein Beispiel für die Nutzung des Service-Managers gezeigt, da die Activity eine Verbindung zum Paket-Manager herstellen wollte, um eine Permission zu überprüfen. Dazu wurde im Verlauf der Service-Manager mit der Methode „getService(String name)“ aufgerufen, was im Beispiel in Abbildung 6.7 auf Seite 74 den Punkt (5) beschreibt. Die in der Abbildung gezeigte Liste der Services (4), auf die der Service-Manager der Android-API zugreift, ist ein Cache der bisher bekannten Services.

Jedoch werden alle Funktionen, die das Hinzufügen oder Auffinden von unbekannten Services betrifft, über die Funktion „getIServiceManager()“ [CdSM, Zeile 33] aufgerufen, die in Listing 7.8 dargestellt ist. Diese gibt einen Binder-Proxy des Service-Manager-Daemons zurück. Wie dies funktioniert wird im Folgenden beschrieben und durch Abbildung 7.6 skizziert.

Der Proxy des Daemons wird durch die „ServiceManagerNative“-Klasse [CdSMN] mittels des Aufrufs „asInterface()“ (1) erstellt. Auf den Bestandteil des Arguments „Binder.allowBlocking()“ wird hier nicht weiter eingegangen, da dies nur Eigenschaften des Proxys verändert, nicht aber das Erstellen und Finden des Daemons. Somit wird auf der Klasse „BinderInternal.java“ [CdBI] die Methode „getContextObject()“ aufgerufen (2).

Die Methode „getContextObject()“ ist jedoch als „native“ deklariert [CdBI, Zeile 88], so dass sie durch JNI auf die Methode „android_os_BinderInternal_getContextObject()“ [CdBJni, Zeile 1000] abgebildet wird (3). Diese bedient sich der nativen Bibliotheken und ruft die Methode „getContextObject(NULL)“ der Klasse „ProcessState.cpp“ [CdPScp].

Listing 7.8 Ausschnitt der Implementierung des Service-Managers in der Android-API [CdSM]

```
30  private static IServiceManager sServiceManager;
31  private static HashMap<String, IBinder> sCache = new                    HashMap<String,
        IBinder>();
32      private static IServiceManager getIServiceManager() {
33          if (sServiceManager != null) {
34              return sServiceManager;
35          }
36          // Find the service manager
37          sServiceManager = ServiceManagerNative
38                  .asInterface(Binder.allowBlocking(BinderInternal.getContextObject()));
39          return sServiceManager;
40      }
```

Durch diese wird die Methode „getStrongProxyForHandle(0)“ aufgerufen [CdPScp, Zeile 112] (5), die durch die „0“ den Handle für den Context-Manager erhält. Dieser wird durch ein „BpBinder“-Objekt [CdBpB] zurück gegeben (6). Die Funktion „javaObjectForBinder()“ sorgt dafür, dass ein Java-Binder-Proxy des Service-Manager-Daemons zurück gegeben wird (7). Dieser wird dann als Service-Manager-Proxy an den Service-Manager der Android-API gereicht (1).

Auffällig ist hier, dass ein Proxy ohne Interaktion mit dem Kernel erstellt wird. Erklärt wird dies durch Kommentare der „ProcessState“-Klasse [CdPScp, Zeile 259 ff.]. Dort wird ausgesagt, dass der Context Manager das einzige Objekt ist, für den ein Proxy erstellt wird, ohne die Referenz vorher zu besitzen. Zuvor wird lediglich überprüft, ob sich ein Context Manager beim Binder-Treiber registriert hat. Da dieser jedoch immer den Handle mit der Zahl „0“ hat, kann dann ein Proxy erstellt werden.

7.3 Sicherheit

Die in Android durch Sandboxing separierten Prozesse können mittels Binder-IPC kommunizieren. Dazu muss jedoch gewährleistet sein, dass Binder die Sicherheit aufrechterhält. Dies ist zum einen durch den Context Manager umgesetzt, zum anderen durch die einzigartige Identifizierung von Binder-Objekten durch Binder-Tokens. Außerdem werden Sender von Binder-Transaktionen durch den Kernel-Treiber identifiziert.

7.3.1 Binder-Transaktionen

Sobald eine Binder-Transaktion im Binder-Treiber vom Sender zum Empfänger weiter geleitet wird, wird der Nachricht die Prozess- und effektive Benutzer-ID angehängt. Dies geschieht automatisch durch den Treiber. Da diese IDs erst durch den Kernel der Nachricht beigefügt werden, können sie nicht vom Sender manipuliert und so eine Rechteausweitung verhindert werden. Vgl. [TH16, S. 973]

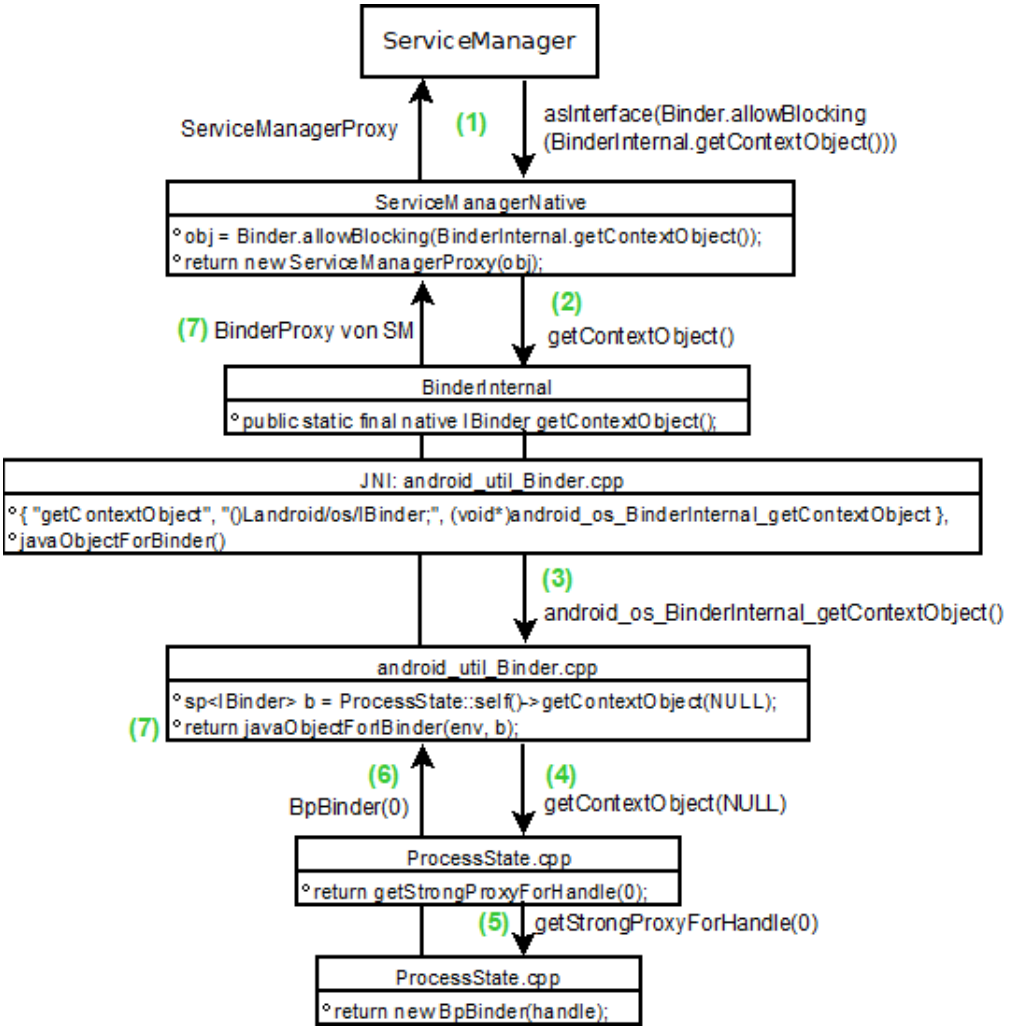


Abbildung 7.6: Verknüpfung des Service-Managers der Android-API mit dem Service-Manager-Daemon

Durch die Informationen über den aufrufenden Prozess kann der Service entscheiden, ob die vom Sender gewünschte Operation ausgeführt wird oder nicht. Dies ist vor allem bei Services, die auf Permissions basieren entscheidend. Durch den Aufruf der Methode „checkCallingPermission()“ kann zunächst abgefragt werden ob der Sender über die entsprechenden Permissions verfügt. Vgl. [SecTip]

7.3.2 Context Manager

Der Context Manager des Binder-Treibers, der in Android durch den Service-Manager implementiert wird, bietet auch weitere Sicherheiten. Zum einen kann nur genau ein Context-Manager beim Binder-Treiber registriert werden. Der Service-Manager-Daemon wird beim Hochfahren des Geräts sehr früh gestartet und registriert sich dann direkt beim Binder-Treiber. Dadurch kann kein später gestarteter böswilliger Prozess mehr diesen Posten übernehmen. Vgl. [CdBc, Zeile 3325ff.]

Zum anderen bietet die Implementierung des Service-Managers weiteren Schutz. Bevor ein Service durch die Methode „do_add_service()“ hinzugefügt wird, wird die Methode „svc_can_register()“ aufgerufen, die in Listing 7.9 gezeigt ist. Diese überprüft zunächst die UID des Services, der sich registrieren möchte. Diese darf nicht größer oder gleich als die in der Datei „Code_android_filesystem_config.h“ [CdFCf] definierte „AID_APP“ liegen, die 10000 beträgt. Alle Apps erhalten UIDs, die größer oder gleich als die „AID_APP“ sind. Damit wird Apps nicht erlaubt sich beim Context-Manager zu registrieren.

Listing 7.9 Ausschnitt der Implementierung des Service-Manager-Daemons [CdSMc]

```
112 static int svc_can_register(const uint16_t *name, size_t name_len, pid_t spid, uid_t uid)
113 {
114     const char *perm = "add";
115     if (multiuser_get_app_id(uid) >= AID_APP) {
116         return 0; /* Don't allow apps to register services */
117     }
118     return check_mac_perms_from_lookup(spid, uid, perm, str8(name, name_len)) ? 1 : 0;
119 }
```

Falls die UID der App die gegebene Grenze nicht überschreitet, wird der Aufruf „check_mac_perms_from_lookup“ gemacht. Durch diesen werden noch zusätzlich die Rechte zum Registrieren des Services in Bezug auf den in Android implementierten MAC-Mechanismus SELinux überprüft. Die entsprechende Regel dafür wird im Allgemeinen durch das Makro „add_service()“ [CdSeTM] (siehe Listing 7.10) definiert. Jeder Service, der sich beim Service-Manager registrieren muss, muss vorher die entsprechenden Regel in seiner „.te“-Datei haben oder ein Verweis auf das Makro. Vgl. [UsBinI]

Das Makro erlaubt sowohl das Registrieren beim Service-Manager, als auch das Finden von anderen Services. Zusätzlich bietet es die „neverallow“-Regel, dass ein Service nur von einer passenden Domäne registriert werden kann. Dadurch kann beispielsweise der Paket-Manager nicht durch den Bluetooth-Prozess registriert werden. Auch durch Einschränkungen auf das Finden von Services beim Service-Manager kann die Sicherheit erhöht werden, da nicht allen Prozessen der direkte Aufruf des Service-Managers gestattet ist. Damit können einige Prozesse keine Handles für einzelne Services erhalten und somit auch nicht auf diese zugreifen.

Listing 7.10 Vergabe der SELinux-Rechte zur Erlaubnis der Registrierung beim Context-Manager [CdSeTM]

```
585 # add_service(domain, service)
586 # Ability for domain to add a service to service_manager
587 # and find it. It also creates a neverallow preventing
588 # others from adding it.
589 define(`add_service', `
590   allow $1 $2:service_manager { add find };
591   neverallow { domain -$1 } $2:service_manager add;
592 ')
```

7.3.3 Binder-Tokens

Binder Objekte werden eindeutig durch ihren Binder-Token identifiziert. Die Assoziation des Tokens mit dem Objekt wird im Kernel abgebildet. Dadurch sind Binder-Objekte einzigartige Objekte, auf die mittels des Tokens der Zugriff ermöglicht wird. Der Binder-Token kann dabei zum einen als bloße Identifikation der Zielressource genutzt werden, indem alle Prozesse ihn auf Anfrage erhalten können. Dann wird erst bei einem Aufruf einer Funktion des Binder-Objekts überprüft, ob der jeweilige Prozess die entsprechenden Zugriffsrechte besitzt. Auf der anderen Seite kann ein Binder-Token auch als *Capability* genutzt werden, während das zugehörige Binder-Objekt ohne weitere Funktionalitäten ist. Die Umsetzung von Zugriffsrechten durch Capabilities ist in Abschnitt 5.1.1 auf Seite 41 erklärt. Zusammengefasst ermöglicht der Besitz einer Capability den Zugriff auf ein Objekt. Vgl. [Ele14, S. 7 f.][Gar13][DHack]

Bei der Nutzung von Binder-Tokens als Capability wird der Token von zwei oder mehreren Prozessen gehalten, während der als Server agierende Prozess diesen zur Identifikation der Klienten nutzt. Dieses Prinzip wird im Hintergrund von Android genutzt, ist aber für Apps größtenteils unsichtbar. Ein Beispiel hierfür wird von Dianne Hackborn [DHack] gegeben. Dabei wird für eine App ein einfaches Binder-Objekt zur Identifikation der App erstellt. Die App erhält dann den Binder-Token, der auch an beispielsweise den Window-Manager und Activity-Manager gereicht wird. Fügt die App nun weitere Fenster hinzu, so übergibt sie dem Window-Manager den Binder-Token zur Identifizierung. Sollen nun alle offenen Fenster der App geschlossen werden, so kann der Activity-Manager dem Window-Manager die Nachricht übermitteln, dass alle Fenster des App-Tokens geschlossen werden sollen. Der Window-Manager kann diese dann eindeutig identifizieren. Durch die Eindeutigkeit von Binder-Tokens kann die Identität der App so nicht von anderen Prozessen angenommen werden, ohne dass diese den Token zuvor erhalten haben. Vgl. [Ele14, S. 7 f.][DHack]

8 Diskussion

Android basiert auf dem Linux-Kernel. Dieser ist weit verbreitet und vielen Entwicklern wohl bekannt. Durch seine lange Existenz und Entwicklung sind viele Sicherheitsprobleme bereits beseitigt. Zu Bedenken ist jedoch, dass er an Android angepasst wurde, so dass vor allem die komplett neuen Erweiterungen des Kernels noch einige gravierende Lücken bergen könnten.

Im Allgemeinen bietet der Kernel allein außerdem nur eine grobkörnige Zugriffskontrolle. Dadurch ist Android nicht ausreichend gesichert. Wird ein Systemprozess, der unter dem „root“-Benutzer läuft, kompromittiert, so kann eine Rechteausweitung erfolgen. Eine Lösung hierfür wurde durch SELinux gefunden. Da hier der Zugriff grundsätzlich verboten ist und nur durch explizite Regeln erlaubt wird, haben Prozesse keine weitreichenden Rechte.

Google [Rep17] erklärt im Bericht über die Android-Sicherheit von 2017, dass alle bekannten Sicherheitslücken durch ein Patch behoben wurden. Dazu bietet Android eine Übersicht der entdeckten Sicherheitslücken [Blt], die gleichzeitig die Codeupdates der Schließungen der Lücken referenzieren. Die Sicherheitslücken werden in folgende Kategorien eingeteilt: *Remote code execution*, *Elevation of privilege*, *Information disclosure*, *Denial of service* und ohne Klassifikation. Beispiele dafür vom April 2018 [BltA] sind:

- CVE-2017-13277 [CVE77]. Hier kann durch einen „out of bounds write“ wegen fehlender Überprüfung der Grenzen Remote-Code ausgeführt werden.
- CVE-2017-13284 [CVE84]. Durch eine ungenügende Überprüfung des Inputs im Code kann eine Rechteausweitung möglich gemacht werden.
- CVE-2017-13275 [CVE75]. Eine fehlende Überprüfung von Grenzen macht einen „out of bounds“-Lesezugriff möglich, durch den Informationen offengelegt werden könnten.
- CVE-2017-13280 [CVE80]. Auch hier fehlt eine Überprüfung von Grenzen eines Lesezugriffs. Dadurch kann ein Denial-of-Service entstehen.

Ohne den genauen Hintergrund der Beispiele zu betrachten, zeigen sie, dass einfache Programmierfehler große Wirkung haben können. Trotzdem müssen Angreifer erst eine Möglichkeit finden, Sicherheitslücken auch auszunutzen.

SELinux soll dies erschweren. Ein Beispiel, dass es trotzdem möglich ist, zeigt ein Ausbruch aus der Sandbox von Chrome [Expl], der durch eine Kette der Nutzung von Sicherheitslücken möglich war. Der Ausbruch an sich war durch SELinux nicht verhindert, da Apps Zugriff auf drei System-Services erhalten [CdSeIA]. Hier zeigt sich das Problem, dass SELinux nicht alles verhindern kann.

System-Prozesse wie init oder zygote benötigen weitreichende Rechte. Auch Apps kann nicht alles verboten werden, da sonst jegliche Interaktionen mit anderen Komponenten nicht möglich wären. Damit ist es nicht möglich allein auf Kernel-Ebene der Android-Sicherheitsarchitektur alle Möglichkeiten einer Rechteausweitung zu beseitigen. Hier muss vor allem darauf geachtet werden, dass keine Programmierfehler in diesen Prozessen vorhanden sind.

Trotzdem bietet SELinux eine große Verbesserung der Prozess- und Datenisolation. Wie das Beispiel der Motivation für SELinux in Abschnitt 5.3.1 zeigt, kann dadurch auch vor Auswirkungen von Programmierfehlern im System geschützt werden. Google [DVers] bietet eine Übersicht der Verteilung der Android-Versionen auf allen Android-Geräten vom Februar 2018. Dort wird klar, dass 17,7% der Android-Geräte eine Version unter Android 5.0 installiert haben, die damit kein SELinux unterstützt. Somit ist mehr als jedes sechste Gerät durch das Fehlen von SELinux nicht weitreichend vor Angriffen geschützt.

Androids Permissions liefern einen weiteren Zugriffsmechanismus auf Ressourcen. Dabei kann vor allem auch der Benutzer entscheiden, ob er einer App Zugriff zu seinen privaten Daten ermöglichen möchte. Die Permissions werden nach Sicherheitsstufen getrennt, wobei die Permissions mit der Stufe „normal“ automatisch vom System zur Installationszeit gewährt werden. Hier stellt sich die Frage, wozu diese Permissions erteilt werden müssen, wenn sie automatisch vergeben werden. Außerdem hat der Benutzer hier keinen Einfluss.

Insgesamt ist bei diesem Sicherheits-Mechanismus der Benutzer das größte Problem. Dieser entscheidet über die Zustimmung der gefährlichen Permissions. Hierbei ist der Benutzer oft nicht ausreichend darüber informiert, welche Zugriffsrechte die Permissions darstellen. Durch das Einführen der Laufzeit-Permissions soll der Unwissenheit entgegen gewirkt werden. Zum einen muss der Benutzer nicht unbedingt zustimmen, um die App überhaupt nutzen zu können. Zum anderen werden die Permissions einzeln zur Laufzeit abgefragt, was eine Assoziation der Permisson mit einer Aktion bringt. Ein Beispiel hierfür wäre das Drücken eines Buttons, der zur Kamera führt, was das Erscheinen des Dialogs zur Abfrage der Permission zur Folge hat.

Durch Permissions ist der Zugriff auf private Daten oder auch kostenpflichtige Dienste wie das Senden von SMS möglich. Ist den Permissions dazu einmal zugestimmt, kann die App darauf zugreifen. Hier ist die große Chance der Malware. Wird eine böswillige App heruntergeladen und stimmt der Benutzer den geforderten Permissions zu, so hat die App sogar die Erlaubnis persönliche Daten auszulesen und zu versenden. An dieser Stelle bietet die Android-Sicherheitsarchitektur keinen Schutz. Der Benutzer ist selbst verantwortlich, nur den „richtigen“ Apps Rechte zuzuschreiben.

Um diesem Problem entgegen zu wirken hat Google die Funktion Google Play Protect ihres App-Marktplatzes Google Play entwickelt. Diese untersucht Apps, die dort zur Verfügung stehen möchten und auch bereits installierte Apps, ob diese böswillige Absichten haben. Dadurch wurden laut Google [Rep17] im Jahr 2017 ungefähr 39 Millionen potentiell gefährliche Apps entdeckt und entfernt. Allerdings bietet die Nutzung dieser Dienste keine Lösung in Bezug auf Android selbst. Der Benutzer ist in diesem Fall an Google Play gebunden, um vor der Installation von Schadsoftware zumindest teilweise geschützt zu sein.

Binder bietet eine sichere Möglichkeit von Prozessen miteinander zu kommunizieren. Dies ist vor allem auf Kernel-Ebene umgesetzt. Der Kernel fügt dabei den Nachrichten die Prozess- und Benutzer-ID des Senders hinzu, so dass der Empfänger der Nachricht entscheiden kann, ob dem

Sender ein Zugriff gewährt oder die gewünschte Operation ausgeführt wird. Dadurch, dass der Kernel diese Informationen anfügt, muss dieser angegriffen werden, um hier eine Änderung zu erwirken.

Im Allgemeinen werden trotzdem immer wieder Sicherheitslücken entdeckt. Unabhängig davon, ob diese durch die Android-Sicherheitsarchitektur abgedeckt werden können, beispielsweise mittels Veränderung einer SELinux-Regel, oder die Implementierung der Komponente, wie eines Daemons verändert werden muss, kann eine Sicherheitslücke nur durch eine Aktualisierung der Software mittels Patchs geschlossen werden. Dies stellt ein weiteres Problem dar, denn Aktualisierungen müssen von den Geräteherstellern an die Android-Geräte übermittelt werden. Vgl. [UpRes]

Lell und Nohl [LN18] fanden in ihren Forschungen heraus, dass einige Hersteller die Patches erst lange Zeit später oder gar nicht an die Benutzer weiterleiten. Teilweise wurden sogar die Versionsnummern von Android in den Geräten gefälscht, um Updates vorzutäuschen. Dadurch sind Android-Geräte trotz existierender Sicherheits-Patches weiter angreifbar. Dies stellt ein schwerwiegendes Sicherheitsproblem außerhalb der Android-Sicherheitsarchitektur dar.

9 Zusammenfassung und Ausblick

Die Arbeit bietet einen Überblick über die Android-Architektur sowie eine Analyse deren Sicherheitsaspekte. Die Hauptthemen der Analyse sind dabei der Linux-Kernel mit der Erweiterung von SELinux, die Android-Permissions und die Interprozesskommunikation mittels Binder.

In Android werden verschiedene Prozesse gegeneinander durch das Sandboxing abgeschottet. Diese Isolation gilt dabei zwischen Apps untereinander aber auch gegenüber Systemdiensten. Den Grundstein dafür legen die Sicherheitskonzepte des Linux-Kernels. Dieser bietet benutzerbasierte Abschottung von Ressourcen und Prozessen. In Android wird in der Regel jeder App ein eigener Linux-Benutzer zugeordnet, sodass jede App in einer eigenen Sandbox agiert. Dies ist bei Systemprozessen jedoch nicht immer möglich. Um weitere Abgrenzungen zu bieten, wird SELinux genutzt. Dieses definiert die Grenzen von Apps genauer und kann auch Systemprozesse, die unter dem privilegiertesten Benutzer „root“ laufen, einschränken. Dadurch wird das Prinzip der geringsten Privilegien umgesetzt, das allen Prozessen nur ihre nötigsten Rechte zuweist. Der Linux-Kernel mit SELinux sorgt damit für eine gute Isolation von Daten und Prozessen.

Der Zugriff auf Hardware und andere Ressourcen wird in Android sowohl mittels der benutzerbasierten Zugriffsrechteverwaltung des Linux-Kernels umgesetzt, die durch SELinux weiter unterstützt wird, aber auch durch die Android-Permissions. Diese erlauben sowohl dem System die Verweigerung des Zugriffs auf seine Ressourcen, bieten aber auch dem Benutzer eine Möglichkeit Entscheidungen über Zugriffsrechte zu treffen. Dazu gehören der Zugriff auf private Daten oder die Nutzung von kostenpflichtigen Diensten.

Da die Prozesse und deren Daten gegeneinander abgeschottet sind, muss eine andere Möglichkeit zur Interprozesskommunikation geboten werden. Android stellt dafür Binder zur Verfügung. Dieser Mechanismus führt die Kommunikation zwischen zwei Prozessen über den Kernel, was zu einer sicheren Kommunikation beiträgt.

Ausblick

Für Androids zukünftige Version, welche momentan ausschließlich als „Android P“ bezeichnet wird, gibt es bereits eine erste Vorschau. Diese geht auch auf Sicherheitsverbesserungen [APSecB][APSecU] ein.

Dazu gehört eine neue Version des Schemas zur Signierung von Apps. Außerdem werden Apps in ihrer Möglichkeit Systemfunktionen aufzufrufen weiter eingeschränkt. Auch die Verschlüsselung der Daten wird erweitert. Für Apps, welche Android P als Ziel haben, wird „Per-app SELinux domains“ [APSecB] genannt. Sollte es eine Spezialisierung der SELinux-Regeln auf einzelne Apps geben, verstärkt das die Sandbox von Apps. Dann können Zugriffsregeln noch spezifischer, auf Apps zugeschnitten, definiert werden. Vgl. [APSecB][APSecU][APdns]

Literaturverzeichnis

[Abt] *Android, the world's most popular mobile platform | Android Developers*. URL: https://developer.android.com/about/android.html (besucht am 24. 04. 2018) (zitiert auf S. 16).

[AFacts] *Android ist für alle da | Wissenswertes*. URL: https://www.android.com/intl/de%7B%5C_%7Dde/everyone/facts/ (besucht am 24. 04. 2018) (zitiert auf S. 15, 16).

[AM] *ActivityManager | Android Developers*. URL: https://developer.android.com/reference/android/app/ActivityManager.html (besucht am 24. 04. 2018) (zitiert auf S. 26).

[AndS] *The Android Source Code | Android Open Source Project*. URL: https://source.android.com/setup/ (besucht am 24. 04. 2018) (zitiert auf S. 15, 16).

[APdns] *Google Online Security Blog: DNS over TLS support in Android P Developer Preview*. URL: https://security.googleblog.com/2018/04/dns-over-tls-support-in-android-p.html (besucht am 24. 04. 2018) (zitiert auf S. 97).

[ApHar] *Apache Harmony - Open Source Java Platform*. URL: http://harmony.apache.org/ (besucht am 24. 04. 2018) (zitiert auf S. 25).

[APILv] *API Level | Android Developers*. URL: https://developer.android.com/guide/topics/manifest/uses-sdk-element.html%7B%5C#%7DApiLevels (besucht am 24. 04. 2018) (zitiert auf S. 15).

[AppCP] *Content Providers | Android Developers*. URL: https://developer.android.com/guide/topics/providers/content-providers.html (besucht am 24. 04. 2018) (zitiert auf S. 29).

[AppCPB] *Content Provider Basics | Android Developers*. URL: https://developer.android.com/guide/topics/providers/content-provider-basics.html (besucht am 24. 04. 2018) (zitiert auf S. 29).

[AppFun] *Application Fundamentals | Android Developers*. URL: https://developer.android.com/guide/components/fundamentals.html (besucht am 24. 04. 2018) (zitiert auf S. 27–30, 34, 50, 53).

[AppM] *App Manifest File <manifest> | Android Developers*. URL: https://developer.android.com/guide/topics/manifest/manifest-element.html (besucht am 24. 04. 2018) (zitiert auf S. 51).

[AppMan] *App Manifest | Android Developers*. URL: https://developer.android.com/guide/topics/manifest/manifest-intro.html (besucht am 24. 04. 2018) (zitiert auf S. 30).

[AppSec] *Application security | Android Open Source Project*. URL: https://source.android.com/security/overview/app-security (besucht am 24. 04. 2018) (zitiert auf S. 34–36).

[AppSer] *Services | Android Developers*. URL: https://developer.android.com/guide/components/services.html (besucht am 24. 04. 2018) (zitiert auf S. 28).

[APSecB] *Security Behavior Changes | Android Developers*. URL: https://developer.android.com/preview/features/security-behav.html (besucht am 24. 04. 2018) (zitiert auf S. 97).

[APSecU] *Security updates | Android Developers*. URL: https://developer.android.com/preview/features/security.html (besucht am 24. 04. 2018) (zitiert auf S. 97).

[AR14] N. Artenstein, I. Revivo. „Man in the binder: He who controls ipc, controls the droid“. In: *Black Hat* (2014) (zitiert auf S. 81).

[ARTD] *ART and Dalvik | Android Open Source Project*. URL: https://source.android.com/devices/tech/dalvik/ (besucht am 24. 04. 2018) (zitiert auf S. 23, 24).

[ArtJ] *Implementing ART Just-In-Time (JIT) Compiler | Android Open Source Project*. URL: https://source.android.com/devices/tech/dalvik/jit-compiler (besucht am 24. 04. 2018) (zitiert auf S. 24).

[ASR] *Android Rewards – Application Security – Google*. URL: https://www.google.com/about/appsecurity/android-rewards/index.html (besucht am 24. 04. 2018) (zitiert auf S. 16).

[BBG14] M. Backes, S. Bugiel, S. Gerling. „Scippa: System-Centric IPC Provenance on Android“. In: *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, 2014, S. 36–45 (zitiert auf S. 78).

[BC00] D. P. Bovet, M. Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. O'Reilly Media, Inc., 2000 (zitiert auf S. 24).

[BCMV12] D. Barrera, J. Clark, D. Mccarney, P. C. Van Oorschot. „Understanding and Improving App Installation Security Mechanisms through Empirical Analysis of Android“. In: *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2012, S. 81–92 (zitiert auf S. 36, 51).

[BinDok] *Open Binder Doku*. URL: http://www.angryredplanet.com/%7B~%7Dhackbod/openbinder/docs/html/ (besucht am 24. 04. 2018) (zitiert auf S. 77).

[BL76] D. E. Bell, L. J. La Padula. *Secure computer system: Unified exposition and multics interpretation*. Techn. Ber. MITRE CORP BEDFORD MA, 1976 (zitiert auf S. 43).

[Blt] *Android Security Bulletins | Android Open Source Project*. URL: https://source.android.com/security/bulletin/ (besucht am 24. 04. 2018) (zitiert auf S. 93).

[BltA] *Android Security Bulletin—April 2018 | Android Open Source Project*. URL: https://source.android.com/security/bulletin/2018-04-01 (besucht am 24. 04. 2018) (zitiert auf S. 93).

[CdAPM] *core/java/android/app/ApplicationPackageManager.java - platform/frameworks/base - Git at Google*. URL: https://android.googlesource.com/platform/frameworks/base/+/master/core/java/android/app/ApplicationPackageManager.java (besucht am 24. 04. 2018) (zitiert auf S. 73, 74).

[CdAT] *core/java/android/app/ActivityThread.java - platform/frameworks/base - Git at Google*. URL: https://android.googlesource.com/platform/frameworks/base/+/master/core/java/android/app/ActivityThread.java (besucht am 24. 04. 2018) (zitiert auf S. 73).

[CdBc] *drivers/staging/android/binder.c - kernel/msm - Git at Google*. URL: https://android.googlesource.com/kernel/msm/+/android-8.1.0%7B%5C_%7Dr0.42/drivers/staging/android/binder.c (besucht am 24. 04. 2018) (zitiert auf S. 79, 90).

[CdBI] *core/java/com/android/internal/os/BinderInternal.java - platform/frameworks/base.git - Git at Google*. URL: https://android.googlesource.com/platform/frameworks/base.git/+/master/core/java/com/android/internal/os/BinderInternal.java (besucht am 24. 04. 2018) (zitiert auf S. 80, 88).

[CdBin] *core/java/android/os/Binder.java - platform/frameworks/base - Git at Google*. URL: https://android.googlesource.com/platform/frameworks/base/+/master/core/java/android/os/Binder.java (besucht am 24. 04. 2018) (zitiert auf S. 80, 82, 85, 86).

[CdBJni] *core/jni/android_util_Binder.cpp - platform/frameworks/base - Git at Google*. URL: https://android.googlesource.com/platform/frameworks/base/+/master/core/jni/android%7B%5C_%7Dutil%7B%5C_%7DBinder.cpp (besucht am 24. 04. 2018) (zitiert auf S. 80, 82, 86, 88).

[CdBLib] *libs/binder - platform/frameworks/native - Git at Google*. URL: https://android.googlesource.com/platform/frameworks/native/+/master/libs/binder (besucht am 24. 04. 2018) (zitiert auf S. 80, 81).

[CdBpB] *libs/binder/BpBinder.cpp - platform/frameworks/native - Git at Google*. URL: https://android.googlesource.com/platform/frameworks/native/+/master/libs/binder/BpBinder.cpp (besucht am 24. 04. 2018) (zitiert auf S. 89).

[CdCI] *core/java/android/app/ContextImpl.java - platform/frameworks/base - Git at Google*. URL: https://android.googlesource.com/platform/frameworks/base/+/master/core/java/android/app/ContextImpl.java (besucht am 24. 04. 2018) (zitiert auf S. 73).

[CdFCf] *libcutils/include/private/android_filesystem_config.h - platform/system/core - Git at Google*. URL: https://android.googlesource.com/platform/system/core/+/master/libcutils/include/private/android%7B%5C_%7Dfilesystem%7B%5C_%7Dconfig.h (besucht am 24. 04. 2018) (zitiert auf S. 45, 71, 91).

[CdIBin] *core/java/android/os/IBinder.java - platform/frameworks/base - Git at Google*. URL: https://android.googlesource.com/platform/frameworks/base/+/master/core/java/android/os/IBinder.java (besucht am 24. 04. 2018) (zitiert auf S. 80, 82).

[CdIpc] *libs/binder/IPCThreadState.cpp - platform/frameworks/native - Git at Google*. URL: https://android.googlesource.com/platform/frameworks/native/+/master/libs/binder/IPCThreadState.cpp (besucht am 24. 04. 2018) (zitiert auf S. 81).

[CdP] *core/java/android/os/Parcel.java - platform/frameworks/base - Git at Google*. URL: https://android.googlesource.com/platform/frameworks/base/+/master/core/java/android/os/Parcel.java (besucht am 24. 04. 2018) (zitiert auf S. 80, 82).

[CdPf] *data/etc/platform.xml - platform/frameworks/base - Git at Google*. URL: https://android.googlesource.com/platform/frameworks/base/+/master/data/etc/platform.xml (besucht am 24. 04. 2018) (zitiert auf S. 71).

[CdPM] *core/java/android/content/pm/PackageManager.java - platform/frameworks/base - Git at Google*. URL: https://android.googlesource.com/platform/frameworks/base/+/master/core/java/android/content/pm/PackageManager.java (besucht am 24. 04. 2018) (zitiert auf S. 31, 75).

[CdPMS] *services/core/java/com/android/server/pm/PackageManagerService.java - platform/frameworks/base - Git at Google*. URL: https://android.googlesource.com/platform/frameworks/base/+/master/services/core/java/com/android/server/pm/PackageManagerService.java (besucht am 24. 04. 2018) (zitiert auf S. 69, 73, 74).

[CdPP] *core/java/android/content/pm/PackageParser.java - platform/frameworks/base - Git at Google*. URL: https://android.googlesource.com/platform/frameworks/base/+/master/core/java/android/content/pm/PackageParser.java (besucht am 24. 04. 2018) (zitiert auf S. 74).

[CdPS] *services/core/java/com/android/server/pm/PermissionsState.java - platform/frameworks/base - Git at Google*. URL: https://android.googlesource.com/platform/frameworks/base/+/master/services/core/java/com/android/server/pm/PermissionsState.java (besucht am 24. 04. 2018) (zitiert auf S. 75).

[CdPScp] *libs/binder/ProcessState.cpp - platform/frameworks/native - Git at Google*. URL: https://android.googlesource.com/platform/frameworks/native/+/master/libs/binder/ProcessState.cpp (besucht am 24. 04. 2018) (zitiert auf S. 88, 89).

[CdPSt] *services/core/java/com/android/server/pm/PackageSetting.java - platform/frameworks/base - Git at Google*. URL: https://android.googlesource.com/platform/frameworks/base/+/master/services/core/java/com/android/server/pm/PackageSetting.java (besucht am 24. 04. 2018) (zitiert auf S. 74).

[CdS] *services/core/java/com/android/server/pm/Settings.java - platform/frameworks/base - Git at Google*. URL: https://android.googlesource.com/platform/frameworks/base/+/master/services/core/java/com/android/server/pm/Settings.java (besucht am 24. 04. 2018) (zitiert auf S. 31, 70).

[CdSeA] *public/attributes - platform/system/sepolicy - Git at Google*. URL: https://android.googlesource.com/platform/system/sepolicy/+/master/public/attributes (besucht am 24. 04. 2018) (zitiert auf S. 60).

[CdSeAC] *private/seapp_contexts - platform/system/sepolicy - Git at Google*. URL: https://android.googlesource.com/platform/system/sepolicy/+/master/private/seapp%7B%5C_%7Dcontexts (besucht am 24. 04. 2018) (zitiert auf S. 59).

[CdSeAp] *public/app.te - platform/system/sepolicy - Git at Google*. URL: https://android.googlesource.com/platform/system/sepolicy/+/master/public/app.te (besucht am 24. 04. 2018) (zitiert auf S. 56).

[CdSeAV] *private/access_vectors - platform/system/sepolicy - Git at Google*. URL: https://android.googlesource.com/platform/system/sepolicy/+/master/private/access%7B%5C_%7Dvectors (besucht am 24. 04. 2018) (zitiert auf S. 61).

[CdSeCl] *private/security_classes - platform/system/sepolicy - Git at Google*. URL: https://android.googlesource.com/platform/system/sepolicy/+/master/private/security%7B%5C_%7Dclasses (besucht am 24. 04. 2018) (zitiert auf S. 61).

[CdSeF] *public/file.te - platform/system/sepolicy - Git at Google*. URL: https://android.googlesource.com/platform/system/sepolicy/+/master/public/file.te (besucht am 24. 04. 2018) (zitiert auf S. 60).

[CdSeFC] *private/file_contexts - platform/system/sepolicy - Git at Google*. URL: https://android.googlesource.com/platform/system/sepolicy/+/master/private/file%7B%5C_%7Dcontexts (besucht am 24. 04. 2018) (zitiert auf S. 59).

[CdSeIA] *private/isolated_app.te - platform/system/sepolicy - Git at Google*. URL: https://android.googlesource.com/platform/system/sepolicy/+/master/private/isolated%7B%5C_%7Dapp.te (besucht am 24. 04. 2018) (zitiert auf S. 93).

[CdSeR] *public/roles - platform/system/sepolicy - Git at Google*. URL: https://android.googlesource.com/platform/system/sepolicy/+/master/public/roles (besucht am 24. 04. 2018) (zitiert auf S. 60).

[CdSeSC] *private/service_contexts - platform/system/sepolicy - Git at Google*. URL: https://android.googlesource.com/platform/system/sepolicy/+/master/private/service%7B%5C_%7Dcontexts (besucht am 24. 04. 2018) (zitiert auf S. 59).

[CdSeSh] *public/shell.te - platform/system/sepolicy - Git at Google*. URL: https://android.googlesource.com/platform/system/sepolicy/+/master/public/shell.te (besucht am 24. 04. 2018) (zitiert auf S. 62).

[CdSeSu] *private/su.te - platform/system/sepolicy - Git at Google*. URL: https://android.googlesource.com/platform/system/sepolicy/+/master/private/su.te (besucht am 24. 04. 2018) (zitiert auf S. 56).

[CdSeTM] *public/te_macros - platform/system/sepolicy - Git at Google*. URL: https://android.googlesource.com/platform/system/sepolicy/+/master/public/te%7B%5C_%7Dmacros (besucht am 24. 04. 2018) (zitiert auf S. 63, 64, 91).

[CdSeU] *private/users - platform/system/sepolicy - Git at Google*. URL: https://android.googlesource.com/platform/system/sepolicy/+/master/private/users (besucht am 24. 04. 2018) (zitiert auf S. 60).

[CdSeZy] *private/zygote.te - platform/system/sepolicy - Git at Google*. URL: https://android.googlesource.com/platform/system/sepolicy/+/master/private/zygote.te (besucht am 24. 04. 2018) (zitiert auf S. 63).

[CdSM] *core/java/android/os/ServiceManager.java - platform/frameworks/base - Git at Google*. URL: https://android.googlesource.com/platform/frameworks/base/+/master/core/java/android/os/ServiceManager.java (besucht am 24. 04. 2018) (zitiert auf S. 73, 88, 89).

[CdSMbc] *cmds/servicemanager/binder.c - platform/frameworks/native - Git at Google*. URL: https://android.googlesource.com/platform/frameworks/native/+/master/cmds/servicemanager/binder.c (besucht am 24. 04. 2018) (zitiert auf S. 88).

[CdSMbh] *cmds/servicemanager/binder.h - platform/frameworks/native - Git at Google*. URL: https://android.googlesource.com/platform/frameworks/native/+/master/cmds/servicemanager/binder.h (besucht am 24. 04. 2018) (zitiert auf S. 88).

[CdSMc] *cmds/servicemanager/service_manager.c - platform/frameworks/native - Git at Google*. URL: https://android.googlesource.com/platform/frameworks/native/+/master/cmds/servicemanager/service%7B%5C_%7Dmanager.c (besucht am 24. 04. 2018) (zitiert auf S. 79, 80, 87, 88, 91).

[CdSMN] *core/java/android/os/ServiceManagerNative.java - platform/frameworks/base - Git at Google*. URL: https://android.googlesource.com/platform/frameworks/base/+/master/core/java/android/os/ServiceManagerNative.java (besucht am 24. 04. 2018) (zitiert auf S. 88).

[CdSu] *su/su.cpp - platform/system/extras - Git at Google*. URL: https://android.googlesource.com/platform/system/extras/+/master/su/su.cpp (besucht am 24. 04. 2018) (zitiert auf S. 49).

[Comp] *Android Compatibility | Android Open Source Project*. URL: https://source.android.com/compatibility/ (besucht am 24. 04. 2018) (zitiert auf S. 15).

[CVE] *CVE-2011-1823 : The vold volume manager daemon on Android 3.0 and 2.x before 2.3.4 trusts messages that are received from a PF_NETLINK s*. URL: https://www.cvedetails.com/cve/CVE-2011-1823/ (besucht am 24. 04. 2018) (zitiert auf S. 53).

[CVE75] *NVD - CVE-2017-13275*. URL: https://nvd.nist.gov/vuln/detail/CVE-2017-13275 (besucht am 24. 04. 2018) (zitiert auf S. 93).

[CVE77] *NVD - CVE-2017-13277*. URL: https://nvd.nist.gov/vuln/detail/CVE-2017-13277 (besucht am 24. 04. 2018) (zitiert auf S. 93).

[CVE80] *NVD - CVE-2017-13280*. URL: https://nvd.nist.gov/vuln/detail/CVE-2017-13280 (besucht am 24. 04. 2018) (zitiert auf S. 93).

[CVE84] *NVD - CVE-2017-13284*. URL: https://nvd.nist.gov/vuln/detail/CVE-2017-13284 (besucht am 24. 04. 2018) (zitiert auf S. 93).

[DAct] *Introduction to Activities | Android Developers*. URL: https://developer.android.com/guide/components/activities/intro-activities.html (besucht am 24. 04. 2018) (zitiert auf S. 27).

[Daidl] *Android Interface Definition Language (AIDL) | Android Developers*. URL: https://developer.android.com/guide/components/aidl.html (besucht am 24. 04. 2018) (zitiert auf S. 82).

[DBc] *Broadcasts | Android Developers*. URL: https://developer.android.com/guide/components/broadcasts.html (besucht am 24. 04. 2018) (zitiert auf S. 28).

[DBS] *Bound Services | Android Developers*. URL: https://developer.android.com/guide/components/bound-services.html (besucht am 24. 04. 2018) (zitiert auf S. 28).

[DHack] *LKML: Dianne Hackborn: Re: [PATCH 1/6] staging: android: binder: Remove some funny && usage*. URL: https://lkml.org/lkml/2009/6/25/3 (besucht am 24. 04. 2018) (zitiert auf S. 77, 92).

[DInt] *Intents and Intent Filters | Android Developers*. URL: https://developer.android.com/guide/components/intents-filters.html (besucht am 24. 04. 2018) (zitiert auf S. 29).

[DPS02] S. De Capitani Di Vimercati, S. Paraboschi, P. Samarati. „Access control: principles and solutions“. In: *Pract. Exper* 00 (2002), S. 1–7 (zitiert auf S. 45, 46, 51).

[DVers] *Dashboards | Android Developers*. URL: https://developer.android.com/about/dashboards/index.html (besucht am 24. 04. 2018) (zitiert auf S. 94).

[Ecry] *Encryption | Android Open Source Project*. URL: https://source.android.com/security/encryption/ (besucht am 24. 04. 2018) (zitiert auf S. 37).

[Ele14] N. Elenkov. *Android security internals: An in-depth guide to Android's security architecture*. No Starch Press, 2014 (zitiert auf S. 19, 20, 25, 42, 47, 51, 54–59, 65, 67, 72, 79, 81, 87, 92).

[Enh43] *Security Enhancements in Android 4.3 | Android Open Source Project*. URL: https://source.android.com/security/enhancements/enhancements43 (besucht am 24. 04. 2018) (zitiert auf S. 33, 49).

[Enh6] *Security Enhancements in Android 6.0 | Android Open Source Project*. URL: https://source.android.com/security/enhancements/enhancements60 (besucht am 24. 04. 2018) (zitiert auf S. 36).

[Enh7] *Security Enhancements in Android 7.0 | Android Open Source Project*. URL: https://source.android.com/security/enhancements/enhancements70 (besucht am 24. 04. 2018) (zitiert auf S. 36).

[Expl] *Google Online Security Blog: Android Security Ecosystem Investments Pay Dividends for Pixel*. URL: https://security.googleblog.com/2018/01/android-security-ecosystem-investments.html (besucht am 24. 04. 2018) (zitiert auf S. 93).

[FCK95] D. Ferraiolo, J. Cugini, D. R. Kuhn. „Role-based access control (RBAC): Features and motivations“. In: *Proceedings of 11th annual computer security application conference*. 1995, S. 241–48 (zitiert auf S. 44).

[Gar13] A. Gargenta. „Deep dive into Android IPC/Binder framework“. In: 2013. URL: https://events.static.linuxfound.org/images/stories/slides/abs2013%7B%5C_%7Dgargentas.pdf (zitiert auf S. 78, 79, 81, 92).

[GM84] F. T. Grampp, R. H. Morris. „The UNIX system UNIX operating system security“. In: *AT&T Bell Laboratories Technical Journal* 63 (1984), S. 1649–1672 (zitiert auf S. 46).

[Hai14] R. Haines. „The SELinux Notebook“. In: (2014). URL: http://freecomputerbooks.com/books/The%7B%5C_%7DSELinux%7B%5C_%7DNotebook-4th%7B%5C_%7DEdition.pdf (zitiert auf S. 43, 57).

[Hal] *Hardware Abstraction Layer (HAL) | Android Open Source Project*. URL: https://source.android.com/devices/architecture/hal (besucht am 24. 04. 2018) (zitiert auf S. 22).

[HalT] *HAL Types | Android Open Source Project*. URL: https://source.android.com/devices/architecture/hal-types (besucht am 24. 04. 2018) (zitiert auf S. 22).

[HRU76] M. A. Harrison, W. L. Ruzzo, J. D. Ullman. „Protection in operating systems“. In: *Communications of the ACM* 19 (1976), S. 461–471 (zitiert auf S. 40).

[KLsm] S. Smalley, T. Fraser, C. Vance. *Linux Security Modules: General Security Hooks for Linux*. 2001. URL: https://www.kernel.org/doc/htmldocs/lsm/index.html (besucht am 24. 04. 2018) (zitiert auf S. 52).

[Lam74] B. W. Lampson. „Protection“. In: *ACM SIGOPS Operating Systems Review* 8.1 (1974), S. 18–24 (zitiert auf S. 40).

[LinOv] *Overview of Linux Kernel Security Features | Linux.com | The source for Linux information*. URL: https://www.linux.com/learn/overview-linux-kernel-security-features (besucht am 24. 04. 2018) (zitiert auf S. 42, 45, 52).

[LN18] J. Lell, K. Nohl. „Uncovering the Android Patch Gap Through Binary-Only Patch Level Analysis“. In: *HIBTSecConf*. 2018. URL: https://conference.hitb.org/hitbsecconf2018ams/materials/D2T1%20-%20Karsten%20Nohl%20%7B%5C&%7D%20Jakob%20Lell%20-%20Uncovering%20the%20Android%20Patch%20Gap%20Through%20Binary-Only%20Patch%20Level%20Analysis.pdf (zitiert auf S. 95).

[LosSE] P. Loscocco. *Security-Enhanced Linux*. Techn. Ber. Information Assurance Research Group, NSA, 2001. URL: https://www.nsa.gov/what-we-do/research/selinux/documentation/assets/files/presentations/2001-kernel-summit-selinux-presentation.pdf (zitiert auf S. 52, 53).

[LS01] P. Loscocco, S. Smalley. „Integrating Flexible Support for Security Policies into the Linux Operating System“. In: *Proceedings of the FREENIX track: USENIX Annual Technical Conference*. 2001 (zitiert auf S. 53, 54, 58).

[LSM+98] P. A. Loscocco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner, J. F. Farrell. „The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments“. In: *Proceedings of the 21st National Information Systems Security Conference*. 1998, S. 303–314 (zitiert auf S. 53).

[ManP] *App Manifest File <permission>| Android Developers*. URL: https://developer.android.com/guide/topics/manifest/permission-element.html (besucht am 24. 04. 2018) (zitiert auf S. 66, 67).

[OrUnix] *Using UNIX Permissions to Protect Files (System Administration Guide: Security Services)*. URL: https://docs.oracle.com/cd/E19120-01/open.solaris/819-3321/secfile-60/index.html (besucht am 24. 04. 2018) (zitiert auf S. 47).

[PckId] *Package Index | Android Developers*. URL: https://developer.android.com/reference/packages.html (besucht am 24. 04. 2018) (zitiert auf S. 25, 26).

[PermOv] *Permissions Overview | Android Developers*. URL: https://developer.android.com/guide/topics/permissions/overview.html (besucht am 24. 04. 2018) (zitiert auf S. 34, 65–70, 72, 73).

[PfAr] *Platform Architecture | Android Developers*. URL: https://developer.android.com/guide/platform/index.html (besucht am 24. 04. 2018) (zitiert auf S. 19, 20, 22–24, 26, 27).

[PlPr] *Android – Google Play Protect*. URL: https://www.android.com/intl/de%7B%5C_%7Dde/play-protect/ (besucht am 24. 04. 2018) (zitiert auf S. 16).

[PM] *PackageManager | Android Developers*. URL: https://developer.android.com/reference/android/content/pm/PackageManager.html (besucht am 24. 04. 2018) (zitiert auf S. 26).

[RBin] *Binder | Android Developers*. URL: https://developer.android.com/reference/android/os/Binder.html (besucht am 24. 04. 2018) (zitiert auf S. 26, 78, 82).

[Rep17] Google. *Android Security 2017 Year In Review*. Techn. Ber. Google, 2018. URL: https://source.android.google.cn/security/reports/Google%7B%5C_%7DAndroid%7B%5C_%7DSecurity%7B%5C_%7D2017%7B%5C_%7DReport%7B%5C_%7DFinal.pdf (zitiert auf S. 15, 16, 93, 94).

[RIBin] *IBinder | Android Developers*. URL: https://developer.android.com/reference/android/os/IBinder.html (besucht am 24. 04. 2018) (zitiert auf S. 26, 78, 81, 82).

[RInt] *Intent | Android Developers*. URL: https://developer.android.com/reference/android/content/Intent.html (besucht am 24. 04. 2018) (zitiert auf S. 29).

[RT78] O. Ritchie, K. Thompson. „The UNIX time-sharing system“. In: *The Bell System Technical Journal* 57 (1978), S. 1905–1929 (zitiert auf S. 49).

[RTPerm] *Runtime Permissions | Android Open Source Project*. URL: https://source.android.com/devices/tech/config/runtime%7B%5C_%7Dperms (besucht am 24. 04. 2018) (zitiert auf S. 66, 68).

[SC13] S. Smalley, R. Craig. „Security Enhanced (SE) Android: Bringing Flexible MAC to Android“. In: *NDSS*. Bd. 310. 2013, S. 20–38 (zitiert auf S. 33, 34, 42, 45, 53, 54, 77, 78, 87).

[SDevCf] *Device Configuration | Android Open Source Project*. URL: https://source.android.com/devices/storage/config (besucht am 24. 04. 2018) (zitiert auf S. 53).

[SDV00] P. Samarati, S. De Capitani, D. Vimercati. „Access Control: Policies, Models, and Mechanisms“. In: *International School on Foundations of Security Analysis and Design*. Springer, 2000, S. 137–196 (zitiert auf S. 39–42, 44).

[SeAvc] *AVCRules - SELinux Wiki*. URL: https://selinuxproject.org/page/AVCRules (besucht am 24. 04. 2018) (zitiert auf S. 62).

[SeC] *ObjectClassStatements - SELinux Wiki*. URL: https://selinuxproject.org/page/ObjectClassStatements (besucht am 24. 04. 2018) (zitiert auf S. 61).

[SeCon] *SELinux concepts | Android Open Source Project*. URL: https://source.android.com/security/selinux/concepts (besucht am 24. 04. 2018) (zitiert auf S. 42, 45, 54, 55, 57–59, 61).

[SecTip] *Security Tips | Android Developers*. URL: https://developer.android.com/training/articles/security-tips.html (besucht am 24. 04. 2018) (zitiert auf S. 35, 90).

[SeImpl] *Implementing SELinux | Android Open Source Project*. URL: https://source.android.com/security/selinux/implement (besucht am 24. 04. 2018) (zitiert auf S. 59).

[SeOv] *Security-Enhanced Linux in Android | Android Open Source Project*. URL: https://source.android.com/security/selinux/ (besucht am 24. 04. 2018) (zitiert auf S. 33, 34, 54–57).

[SePerm] *NB ObjectClassesPermissions - SELinux Wiki*. URL: https://selinuxproject.org/page/NB%7B%5C_%7DObjectClassesPermissions (besucht am 24. 04. 2018) (zitiert auf S. 63, 64).

[SeR] *RoleStatements - SELinux Wiki*. URL: https://selinuxproject.org/page/RoleStatements (besucht am 24. 04. 2018) (zitiert auf S. 60).

[SeT] *TypeStatements - SELinux Wiki*. URL: https://selinuxproject.org/page/TypeStatements (besucht am 24. 04. 2018) (zitiert auf S. 59, 60).

[SeTy] *TypeRules - SELinux Wiki*. URL: https://selinuxproject.org/page/TypeRules (besucht am 24. 04. 2018) (zitiert auf S. 63).

[SeU] *UserStatements - SELinux Wiki*. URL: https://selinuxproject.org/page/UserStatements (besucht am 24. 04. 2018) (zitiert auf S. 60).

[SFK+09] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev. „Google Android: A State-of-the-Art Review of Security Mechanisms“. In: *Neural Networks* 126 (2009). DOI: abs/0912.5101. eprint: 0912.5101 (zitiert auf S. 48, 50).

[SHAL99] R. Spencer, M. Hibler, D. Andersen, J. Lepreau. „The Flask Security Architecture: System Support for Diverse Security Policies“. In: *Proceedings of the 8th USENIX Security Symposium*. USENIX Association, 1999 (zitiert auf S. 54, 55).

[Sign] *Application Signing | Android Open Source Project*. URL: https://source.android.com/security/apksigning/ (besucht am 24. 04. 2018) (zitiert auf S. 36).

[Sma02] S. Smalley. „Configuring the SELinux Policy“. In: *NAI Labs Rep* (2002). URL: https://www.nsa.gov/resources/everyone/digital-media-center/publications/research-papers/assets/files/configuring-selinux-policy-report.pdf (zitiert auf S. 62, 63).

[SOv] *Security | Android Open Source Project*. URL: https://source.android.com/security/ (besucht am 24. 04. 2018) (zitiert auf S. 15, 16, 33).

[Spe08] R. Spenneberg. *SELinux & AppArmor: Mandatory Access Control für Linux einsetzen und verwalten;* Pearson Deutschland GmbH, 2008 (zitiert auf S. 39, 43, 55, 57).

[SS94] R. S. Sandhu, P. Samarati. „Access Control: Principles and Practice“. In: *IEEE Communications Magazine* (1994), S. 40–48 (zitiert auf S. 40–42).

[StdAdb] *Android Debug Bridge (adb) | Android Studio*. URL: https://developer.android.com/studio/command-line/adb.html (besucht am 24. 04. 2018) (zitiert auf S. 23).

[Stud] *Sign Your App | Android Studio*. URL: https://developer.android.com/studio/publish/app-signing.html (besucht am 24. 04. 2018) (zitiert auf S. 36).

[SVB] *Verifying Boot | Android Open Source Project*. URL: https://source.android.com/security/verifiedboot/verified-boot (besucht am 24. 04. 2018) (zitiert auf S. 36, 37).

[SVBIm] *Implementing dm-verity | Android Open Source Project*. URL: https://source.android.com/security/verifiedboot/dm-verity (besucht am 24. 04. 2018) (zitiert auf S. 36, 37).

[SVS01] S. Smalley, C. Vance, W. Salamon. „Implementing SELinux as a Linux Security Module“. In: *NAI Labs Report* 1 (2001), S. 139 (zitiert auf S. 53, 54).

[SysKS] *System and kernel security | Android Open Source Project*. URL: https://source.android.com/security/overview/kernel-security (besucht am 24. 04. 2018) (zitiert auf S. 33, 34, 36, 37, 45, 48, 50, 54, 65).

[TH16] A. S. Tanenbaum, B. Herbert. *Moderne Betriebssysteme*. 4. aktuali. Pearson Deutschland GmbH, 2016 (zitiert auf S. 15, 20, 21, 23, 24, 33, 36, 39, 41, 45, 49, 77, 81, 89).

[UEdis] *discretionary : Englisch » Deutsch | PONS*. URL: https://de.pons.com/%7B%5C%22%7Bu%7D%7Dbersetzung/englisch-deutsch/discretionary (besucht am 24. 04. 2018) (zitiert auf S. 39).

[UEman] *mandatory : Englisch » Deutsch | PONS*. URL: https://de.pons.com/%7B%5C%22%7Bu%7D%7Dbersetzung/englisch-deutsch/mandatory (besucht am 24. 04. 2018) (zitiert auf S. 42).

[UpRes] *Security Updates and Resources | Android Open Source Project*. URL: https://source.android.com/security/overview/updates-resources (besucht am 24. 04. 2018) (zitiert auf S. 16, 95).

[UsBinI] *Using Binder IPC | Android Open Source Project*. URL: https://source.android.com/devices/architecture/hidl/binder-ipc%7B%5C#%7Dselinux (besucht am 24. 04. 2018) (zitiert auf S. 91).

[Veri] *verity.txt\device-mapper\Documentation - kernel/git/torvalds/linux.git - Linux kernel source tree*. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/device-mapper/verity.txt (besucht am 24. 04. 2018) (zitiert auf S. 36, 37).

[VeriB] *Verified Boot | Android Open Source Project*. URL: https://source.android.com/security/verifiedboot/ (besucht am 24. 04. 2018) (zitiert auf S. 36, 37).

[Vers] *VERSION_CODES | Android Developers*. URL: https://developer.android.com/reference/android/os/Build.VERSION%7B%5C_%7DCODES.html%7B%5C#%7DBASE (besucht am 24. 04. 2018) (zitiert auf S. 15).

[WCM+02] C. Wright, C. Cowan, J. Morris, S. Smalley, G. Kroah-Hartman. „Linux Security Modules: General Security Support for the Linux Kernel“. In: *Proceedings of the 11th USENIX Security Symposium*. USENIX Association Proceedings, 2002 (zitiert auf S. 52).

[Yag13] K. Yaghmour. *Embedded Android: Porting, Extending, and Customizing*. O'Reilly Media, Inc., 2013 (zitiert auf S. 19–23, 25, 65, 77, 78).

[Zyg] *Overview of Android Memory Management | Android Developers*. URL: https://developer.android.com/topic/performance/memory-overview.html (besucht am 24. 04. 2018) (zitiert auf S. 24).

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift