GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN

INSTITUT FÜR INFORMATIK

HOT TOPICS IN COMPUTER SECURITY

SOMMERSEMESTER 2012

Schadsoftware für Android

Frühzeitige Erkennung von Schadsoftware in App-Märkten

Malte Hübner

Angewandte Informatik

2. Semester

“Ich hab ‘ne App dafür…”

Inhaltsverzeichnis

Abkürzungsverzeichnis

App	-	Application / Applikation / hier: Software für Smartphones
CA		Certificate authority

1 Einleitung

Das Smartphone ist – mit steigender Tendenz - für viele Menschen zum alltäglichen Begleiter geworden (vgl. *heise* online 2012a). Für verschiedene Anwendungszwecke existieren unzählige Apps, die zumeist aus den - zum mobilen Betriebssystem gehörenden - App-Stores heruntergeladen und installiert werden können.

Mit steigendem Interesse der Menschen an Smartphones wächst jedoch auch das damit verbundene Gefahrenpotential. Um diesem entgegenzukommen haben, die Hersteller der mobilen Betriebssysteme zahlreiche Sicherheitsmechanismen integriert.

Da ein Großteil der Gefahr von den Apps ausgeht, müssen auch die *Android*nutzer vor der Installation einer App die von der App angeforderten Rechte (Permissions) bestätigen, damit die App auf vom System verwaltete Funktionen - wie z. B. das Internet - zugreifen kann. An dieser Stelle ist der Smartphonesnutzer dazu aufgefordert, sich bei einer App-Installation mit den angeforderten Rechten vertraut zu machen und ggf. die Installation abzubrechen.[1]

Das Ziel dieser Arbeit ist es, Mechanismen – so wie deren Vor– und Nachteile - vorzustellen, mit der Apps - vor der Installation auf dem Endgerät - auf „böswillige Absichten“ hin untersucht werden können. Hierzu werden im nächsten Kapitel Hintergründe und Probleme dargestellt.

Das darauf folgende dritte Kapitel stellt je einen Ansatz zum Erkennen von bereits bekannter und bisher unbekannter Schadsoftware aus der Studie (*Zhou* et. al 2012) vor. Im vierten Kapitel werden die Vor- und Nachteile der vorgestellten Methoden genannt. Diese Arbeit endet mit einem Fazit und Verbesserungsvorschlägen für die dargestellten Probleme.

Nicht im Fokus dieser Arbeit stehen die Sicherheitsmechanismen in *Android* und weitere Angriffsvektoren wie z. B. über Drive-By-Downloads (vgl. *heise* online 2012b), wobei weitere Details zur Sicherheit in *Android* unter (*android* open source project 2012) und (*Enck* 2009) nachzulesen sind.

[1] „Social Engineering“ ist in diesem Zusammenhang die Disziplin eines Angreifers einen Nutzer dazu zu verleiten einer App mehr Rechte einzuräumen, als für den eigentlichen Betrieb nötig wäre.

2 Hintergrund und Problemstellung

Um einen tieferen Einblick in die Problematik zu erhalten, wird im ersten Abschnitt dieses Kapitel auf die bestehenden Gefahren von Smartphones und insbesondere von *Android*-Smartphones eingegangen. Der anschließende Abschnitt thematisiert überdies die durchgeführten Sicherungsmaßnahmen der App-Stores und zeigt die aktuellen Probleme auf.

2.1 Angriffsziele und Gefahrenpotenziale

Mit dem hohen Gehalt an persönlichen Daten, der nahezu dauerhaften Verbindung zum Internet und der Möglichkeit direkt über die Telefonrechnung bzw. online einkaufen können, stellt das Smartphone ein besonders attraktives Angriffsziel dar. Für den Angreifer lässt sich der Nutzen eines kompromittierten Smartphones in drei verschiedene Bereiche einteilen:

Zum einen kann das kompromittierte Smartphone den Angreifer mit personenbezogenen Daten versorgen. Insbesondere sind dabei E-Mail-Adressen, sicherheitskritische Passwörter (z. B. für den automatischen Login bei einschlägigen Webdienstleistern), Bankzugangsdaten und alle weiteren personenbezogenen Daten, die z. B. für Werbezwecke nutzbar gemacht werden können von Interesse.

Zum anderen kann der Angreifer dem Besitzer des Smartphones direkten finanziellen Schaden - meist zugunsten des Angreifers - zufügen, indem dieser beispielsweise kostenpflichtige Nachrichten versenden lässt, gebührenpflichtige Telefonanrufe tätigt oder Online-Banking-Sessions abhört und sich gewinnbringend daran beteiligt.

Unter anderem kann ein kompromittiertes Smartphone auch Mitglied in einem Botnetz werden und sich als Gateway einsetzen lassen, um Zugriff auf geschlossene Netzwerke zu schaffen.

Die Gefahren sind umso höher, wenn ein Angreifer einen (bekannten) Exploit nutzt um *root*-Rechte auf dem *Android*phone zu erhalten. Hat eine App *root*-Zugriff, kann sie nahezu alle Sicherheitsmaßnahmen in *Android*, wie z. B. das Sandboxing umgehen (vgl. *Jiang* 2012).

2.2 Androidmärkte und deren Sicherheitsmaßnahmen und -probleme

Die Softwarebeschaffung über App-Stores ist nutzerfreundlich und bietet den Herstellern der Betriebssysteme die Möglichkeit den verfügbaren Softwarebestand zu kontrollieren und nötige Sicherheitsmaßnahmen durchzuführen. Für *Android* gibt es neben dem „*Google Play Store*“ noch diverse weitere App-Stores.

Um als Entwickler eine App im *Play Store* zu veröffentlichen, muss zunächst gegen eine Gebühr von 25 US-$ ein Entwickleraccount angelegt werden (vgl. *android* developers 2012a). Die Zahlung der 25 US-$ soll verhindern, dass weniger minderwertige Apps veröffentlicht und gesperrte Accounts unter anderem Namen wiedereröffnet werden (vgl. *Google Play* 2012a, *Google Play* 2012b).

Alle veröffentlichten Apps müssen vom Autoren signiert werden, um später jede App ihrem Autor zuordnen zu können. Die Signaturen für die Apps können vom Autoren selbst erstellt werden und müssen nicht durch eine CA signiert werden (vgl. *android* developers 2012b). Es ist somit nicht sichergestellt, welche konkrete Person sich hinter einem Entwickleraccount verbirgt. Falls es einem Angreifer möglich sein sollte, den privaten Schlüssel der Signatur zu erlangen, kann er unter dessen Namen Apps mit bösartigen Absichten in den *Google Play Store* einstellen (vgl. *android* developer 2012b), was eine strafrechtliche Verfolgung erschweren könnte.

Seit 2011 hat *Google* eine neue Ebene für die *Android*-Sicherheit mit dem Codenamen „*Bouncer*“ hinzugefügt. Die Funktion von *Bouncer* wird vom „Vice President of Engineering *Android*“ *Hiroshi Lockheimer* wie folgt erklärt.

> „(...)[O]nce an application is uploaded, the service immediately starts analyzing it for known malware, spyware and trojans. It also looks for behaviors that indicate an application might be misbehaving, and compares it against previously analyzed apps to detect possible red flags. We actually run every application on *Google*’s cloud infrastructure and simulate how it will run on an *Android* device to look for hidden, malicious behavior. We also analyze new developer accounts to help prevent malicious and repeat-offending developers from coming back.“ (*Lockheimer* 2012)

Kurz darauf sind mehrere Presseartikel veröffentlicht worden, in denen beschrieben wird, dass *Googles Bouncer* zu umgehen ist (vgl. *Jesus* 2012; *heise* Security 2012a). Aktuell (am 04. Juni 2012) haben *Jon Oberheide* und *Charlie Miller* gezeigt, dass es möglich ist, für eine App die virtualisierte Umgebung von *Bouncer*

zu erkennen und der App in dieser Umgebung somit ein anders (nicht bösartiges) Verhalten zu geben (vgl. *Oberheide* 2012; *heise* Security 2012a).

Eine Möglichkeit für den Benutzer die Sicherheit bzw. das Verhalten einer App vor der Installation zu beurteilen, ist die Bewertungs- und Kommentarfunktion der App-Stores.

Häufig wird eine Schadsoftware über verschiedene Apps verteilt, um eine große Menge Smartphones zu infizierten. Um diesem Problem nachzukommen, kann man - ähnlich wie beim Computer – das Smartphone mit Antivirensoftware vor Angriffen schützten. Aktuelle Antivirensoftware ist in der Regel signaturbasiert und arbeitet entsprechend effektiv; bringt aber die dazugehörigen Nachteile bei unbekannter bzw. neuer Malware mit sich (vgl. *Schmidt* et. al 2009, S. 1). Ein weiterer Nachteil bei Antivirensoftware auf dem Smartphone ist die Mehrbelastung der Akkus und die dadurch verringerte Laufzeit.

Als letzte Instanz behält sich *Google* die Möglichkeit der Fernlöschung von Apps vor, die bisher nur auf zwei Apps zur Demonstration von Sicherheitslücken angewandt wurde - bzw. werden musste (vgl. *heise* Security 2012b).

Google verfolgt mit *Bouncer* den Ansatz Malware gar nicht erst auf die Androiden gelangen zu lassen. Dadurch, dass *Google* aber - im Gegensatz zu *Apple* - eine offene und aktive Community betreiben möchte, darf der Zeitraum bis zur Veröffentlichung einer App nicht zu groß werden. Diese (Quell-)Offenheit - und all ihre Vorteile - bringt bzgl. der Sicherheit den Nachteil, dass die Angreifer genau wissen, was auf der Seite des „Opfers“ für Quellcodes ausgeführt werden. Außerdem sind die Zulassungsbeschränkungen nicht so restriktiv wie bei vergleichsweise *Apple*.

3 Zwei Untersuchungsmethoden für die Erkennung von Schadsoftware in Androidmärkten und deren Ergebnisse[2]

Im nun folgenden Kapitel werden Methoden zur Erkennung von Malware - außerhalb des mobilen Gerätes - vorgestellt, die aus der Veröffentlichung von

[2] Vgl. Zhou et. al 2012.

(*Zhou* et al. 2012) stammen. *Zhou* et. al haben ein System namens *DroidRanger* entwickelt, mit dem es möglich ist Apps auf bösartige Absichten hin zu untersuchen. Die im Folgenden beschriebenen Methoden von *DroidRanger* nennen *Zhou* et. al „permission-based behavioral footprinting“, zur Untersuchung von bekannter Malware und „heuristic-based filtering scheme“, zur Erkennung von bislang unbekannter Malware.

3.1 Erkennung bekannter Malware

Die Menge an Apps in den Stores ist inzwischen so groß, dass eine detaillierte Untersuchung nur mit erheblichem Aufwand zu leisten ist. Aus diesem Grund wenden *Zhou* et. al zunächst einen Filter an, der auf den angeforderten Permissions einer App basiert. Im zweiten Untersuchungsschritt - für bekannte Malware - werden drei weitere Mechanismen genutzt, um aus dem Verhalten der App einen Fußabdruck zu erstellen.

Permission-based filtering

Dadurch, dass auch die infizierten Apps aus den Stores - zumindest zunächst - keinen *root*-Zugriff haben, müssen sie bei der Installation die benötigten Permissions anfordern. Diese werden bei *Android* im Manifest der App deklariert und lassen sich dort effizient auslesen. Oft brauchen die bösartigen Apps eine Kombination aus verschiedenen Permissions, um ihre Funktionalität ausüben zu können.

Die Idee von *Zhou* et. al ist es, die benötigten Permissions von bekannten schädlichen Apps auf die essenziellen Bestandteile hin zu prüfen und alle zu untersuchenden Apps auf diese Bestandteile hin zu filtern. Nur die Apps werden im zweiten Schritt bearbeitet, die für Malware bekannte Kombinationen aus Permissions in ihrem Manifest vereinen. Um die „false negativ“-Rate gering zu halten, ist es notwendig ausschließlich die essenziellen Permissions für das Filtern auszuwählen, wie das folgende Beispiel verdeutlichten soll:

Beispiel:

Eine bösartige Software, die kostenpflichtige SMS an Sonderrufnummern verschickt und automatisch die zugehörigen Antworten abfängt, um den Nutzer nicht auf sich zu aufmerksam zu machen – wie z. B. bei der Malware *Zsone* (vgl.

Zsone 2011) - benötigt die *Android*-Permissions SEND_SMS und RECEIVE_SMS. Gäbe es zudem noch eine Variante der Malware, die Bookmarks in den Browser einfügen kann, benötigt diese außerdem die Permission WRITE_HISTORY_BOOKMARKS. Der Filter für die Malware - *Zsone* in ihren verschiedenen Ausprägungen - würde so nur die Permissions SEND_SMS und RECEIVE_SMS enthalten.

Behavioral footprint matching

Durch das permission-based Filtering wird die Menge an verbliebenen (zu untersuchenden) Apps drastisch reduziert, so dass im zweiten Schritt auch aufwendigere Untersuchungsmethoden angewandt werden können. Konkret haben *Zhou* et. al die folgenden drei Ansatzpunkte umgesetzt, um aus den *.apk-Paketen[3] das Verhalten der Apps zu extrahieren.

1. Eine detailliertere Betrachtung des Manifestes, um beispielsweise herauszufinden, ob die App sich als *BroadcastReceiver* angemeldet hat. Dies ist nötig, wenn die App über Dinge informiert werden will, wie z. B. das Eintreffen einer SMS oder dass das Ende der Akkulaufzeit naht.
2. Eine Analyse des Bytecodes der App, hauptsächlich mit dem Ziel die Aufrufe der genutzten APIs zu extrahieren und statische Parameter von Funktionen zu finden. Als Beispiel nennen *Zhou* et. al hier das Senden einer SMS an kostenpflichtige Dienste.
3. Als Drittes analysieren *Zhou* et. al das Layout der App, indem sie das *.apk-Archiv entpacken, um den Aufbau der App, d. h. die Baumstruktur und die angelegten Packages sehen zu können.

3.2 Erkennung unbekannter Malware

Um neue Apps mit bekannter Malware zu erkennen, haben die Entwickler von *DroidRanger* einen Datensatz mit infizierten Apps genutzt und konnten aus diesem das Verhalten der verschiedenen Malwaretypen ableiten. Für unbekannte Malware ist es nicht möglich bereits bestehende Datensätze zu analysieren. *Zhou* et. al haben sich aus diesem Grund für eine heuristische Methode entschieden, die zunächst das Filtern der zu analysierenden Apps vornimmt. Eine detaillierte

[3] *Android*-Apps werden in *.apk-Paketen bereitgestellt.

Untersuchung der verbleibenden Apps erfolgt - ebenso wie für bekannte Malware – in einem zweiten Schritt.

Heuristics-based filtering

Um bisher unbekannte Malware zu entdecken, haben *Zhou* et. al die Annahme getroffen, dass das Nachladen von unbekanntem Code in böswilligen Apps genutzt wird. In ihrer Veröffentlichung weisen sie darauf hin, dass diese Annahme nur eine von vielen Möglichkeiten ist und dass mit weiteren Annahmen andere Zero-Day-Malware gefunden werden könnte (vgl. *Zhou* et. al 2012).

Wie bei reinem *Java* - mit dem ClassLoader - kann bei der Programmierung in *Android* über den DexClassLoader Code zur Laufzeit in die Applikation geladen werden. Wird der nachzuladene Code von einem entfernten Server geladen, erscheint dieses Verhalten als „sehr merkwürdig“ und ist ein prinzipieller Angriffsvektor. Apps, die dieses Verhalten aufweisen werden von *Zhou* et. al im zweiten Schritt genauer analysiert.

Die zweite Annahme für das heuristische Filtrieren ist, dass das Nachladen von nativem, lokal liegendem Code zur Laufzeit genutzt werden kann, um Sicherheitslücken auf Maschinencodeebene auszunutzen und dies zu verschleiern. Da aus Performace- oder Kompatibilitätsgründen das Nutzen von nativem Code für manche Applikationen notwendig oder vorteilhaft ist, nehmen *Zhou* et. al an, dass dieser Code sich in dem dafür vorgesehenen Verzeichnis (lib/armeabi) der App befinden sollte. Insbesondere weil der native Code für Exploits auf Kernelebene genutzt werden kann. Für die heuristische Filterung wird somit angenommen, dass nativer Code aus anderen Speicherorten außer dem Verzeichnis lib/armeabi als „merkwürdig“ einzustufen ist. Auch diese Applikationen werden im zweiten Schritt genauer betrachtet.

Dynamic execution monitoring

In zweitem Schritt werden die verbliebenden Apps ausgeführt um deren Verhalten zu protokollieren. Der Fokus liegt dabei auf dem Verhalten des nachgeladenen Codes und insbesondere ob dieser APIs nutzt, für die *Android*-Permissions benötigt werden. Beispielsweise könnte dies ein Aufruf der Systemfunktion zum Versenden von einer SMS sein, die für den Nutzer kostenpflichtig ist.

Für das Nachladen von nativem Code legen *Zhou* et. al den Fokus auf Systemaufrufe, die aus bekannten *root*-Exploits stammen oder nur mit *root*-Zugriff möglich sind. Als Beispiel nennen sie den sys_mount Systemaufruf, da über diesen mit *root*-Rechten die Partition aus ihrem Lesezugriff in den Lese-/Schreibmodus versetzt werden kann.

Hat *DroidRanger* eine Zero-Day Malware gefunden, wird diese für die Analyse bekannter Malware in die Datenbank hinzugefügt.

3.3 Ergebnisse

Um die Effektivität der vorgestellten Methoden aufzuzeigen, sollen in diesem Abschnitt kurz die Ergebnisse von *Zhou* et. al zusammengefasst werden.

DroidRanger wurde auf einem Datensatz von 182.823 unterschiedlichen Apps getestet, die mittels eines Crawlers aus verschiedenen App-Märkten heruntergeladen wurden. Die zu analysierende Menge zeigt die Notwendigkeit eines effektiven Filters, der dennoch eine kleine Rate von „falschen Negativen“ zulässt.

Für das Filtrieren von bekannter Malware haben *Zhou* et. al zehn bekannte Malware-Familien analysiert und die essenziellen Permissions zusammengestellt. Die Ergebnisse zeigen, dass der Filter für die meisten Malware-Familien sehr gut funktioniert. Die Ausnahme bilden diejenigen, die weitverbreitete Kombinationen von Permission anfordern, wie z. B. INTERNET und READ_PHONE_STATE.

Auch mit dem *behavioral footprint matching* haben *Zhou* et. al erfolgreich neue Apps mit bekannter Malware entdeckt. Es kann festgehalten werden, dass die Infektionsrate im *Google Play Store* weitaus geringer ist, als in den inoffiziellen Märkten von dritten Anbietern. In der von *Zhou* et. al durchgeführten Studie liegt die Infektionsrate im *Play Store* bei 0,02% und in den alternativen Stores zwischen 0,2% bis 0,47%.

Unbekannte Malware wurde von *Zhou* et. al in zwei verschiedenen Ausprägungen gefunden. Namentlich handelt es sich um *Plankton* und *DroidKungFu*, die in mehreren Apps nachgewiesen wurden (vgl. *DroidKungFu* 2011; *Plankton* 2011).

4 Vor- und Nachteile zur Untersuchung von Androidmärkten und der vorgestellten Methoden

Zhou et. al motivieren ihre Arbeit hauptsächlich dadurch, dass es bis dato keine Studie gab, die für einen gesamten App-Store untersucht hat, wie hoch die Infektionsrate des App-Stores ist.

Aus den folgenden Gründen ist der Ansatz von *Zhou* et. al als positiv zu bewerten:

1. Schadsoftware könnte bei einem Einsatz von *DroidRanger* frühzeitig erkannt werden und würde nicht - oder nur wenige Smartphones - infizieren können. Der Angriffsvektor über Apps (aus den Märkten) könnte so weitgehend reduziert werden. Bekäme man die Infektionsrate auf nahezu null abgesenkt, könnte Antivirensoftware für das mobile Endgerät die Observation von Applikationen einstellen und so die Akkulaufzeit verlängern.
2. Dadurch, dass alle Apps für eine Untersuchung in Betracht kommen, entsteht eine Übersicht über das Gefahrenpotential, was die App-Märkte beinhalten. Für die Smartphonenutzer, die bisher ihren Apps die angeforderten Permissions ohne Überprüfung gewährt haben, sinkt das Gefahrenpotential bei fallender Infektionsrate besonders.
3. Die Untersuchung ist effizient, vor allem weil effektive Filterungsmethoden angewandt werden und es so möglich ist, ohne großen finanziellen Aufwand einen dauerhaften Betrieb zu gewährleisten.

Aus den folgenden Gründen ist ein Ansatz wie von *Zhou* et. al oder *Bouncer* als kritisch zu betrachten:

1. Bei einer dynamischen Analyse setzt *Bouncer* auf Virtualisierung der *Android*umgebung. Wie bereits im Abschnitt 2.2 erwähnt, lässt sich diese Umgebung aus einer App heraus erkennen, was wiederrum der App die Möglichkeit gibt das Verhalten in dieser Umgebung entsprechend zu ändern. Konkret bedeutet das, dass die App sich in einer virtualisieren Umgebung unauffällig verhält und nur auf „echten“ Betriebssystemen die böswilligen Absichten durchführt. Eine Antivirensoftware auf den mobilen

Geräten würde zwar zur zusätzlichen Belastung der Ressourcen führen, ist dafür aber nicht anfällig für den genannten Nachteil der Virtualisierung.

2. Für die Erkennung von unbekannten Malwaretypen haben *Zhou* et. al einen heuristischen Ansatz gewählt, der lediglich das Nachladen von Code aus unbekannter Quelle beachtet. Sie haben in ihrer Studie gezeigt, dass dieser Ansatz von Erfolg ist, erwähnten zudem allerdings dass Apps, die keinen Code nachladen, nicht erkannt werden würden. Zusätzliche Annahmen könnten diesem Problem nachkommen, was allerdings weitere Nachteile generiert. Zum einen wird durch die höhere Anzahl an Apps die dynamische Analyse zeitaufwendiger. Zum Anderen kann dennoch nicht mit Sicherheit davon ausgegangen werden, dass die richtigen Annahmen getroffen wurden.
3. Selbst wenn die heuristische Filtrierung Schadsoftware für die dynamische Analyse passieren lässt, ist mit dem Ansatz von *Zhou* et. al nur ein kleiner Teil der möglichen Angriffsvektoren abgedeckt. Hier wäre eine Generalisierung des Ansatzes nötig, um aufwendige manuelle Heuristiken zu vermeiden.

5 Fazit und Ausblick

Die Studie von *Zhou* et. al zeigt, dass die Infektionsrate insbesondere beim *Google Play Store* gering ist, jedoch die Gefahr aufgrund der hohen Nutzerzahlen und des wachsenden Marktes dennoch Handlungsbedarf erfordert (*Zhou* et. al 2012). Ihr Ansatz den Markt zu untersuchen ist gut, wenn auch mit Nachteilen behaftet. Mehr Sicherheit könnten die Ansätze bieten, die auf dem Endgerät selbst laufen, wobei der größte Nachteil die beschränkten Kapazitäten wie Akku und Leistung sind.

Wie häufig - aus vielen andere Bereichen auch – tritt hier ein Trade-Off zwischen Nutzbarkeit und Sicherheit in Erscheinung.[4] Somit erscheint eine hybride Lösung die beste zu sein, um die Ressourcen auf dem mobilen Gerät nicht unnötig zu belasten. Rechenintensive Prozesse können wie bei (*Schmidt* et. al 2009) auf

[4] Ein weiteres Beispiel ist, dass *Google* zugunsten einer aktiven Community auf eine restriktivere Kontrolle der Entwickleraccounts verzichtet.

externe Server ausgelagert werden. Für das mobile System bleibt ein leichtgewichtiger Prozess, der Dinge überwacht wie:

- den Stromverbrauch je App (vgl. *Dixon* et. al 2011),
- das Nutzerverhalten abhängig vom Standort und der Netzwerkanbindung,
- das Verhalten des Nutzers bezüglich kostenpflichten Diensten (nimmt ein Nutzer diese fast nie wahr, lernt das System dies und warnt den Nutzer ggf.),
- die Vergabe von Permissions bzw. Kombinationen von Permissions und
- den normalen Netzwerktraffic um Botnetzkommunikation, ungewollte Bankingsession und die Übertragung von personenbezogenen Daten zu erkennen.

Die gewonnen Daten könnten mittels Machine-lerning-Algorithmen auf externen Servern ausgewertet und an den mobilen Client zurück übertragen werden. Dieser reagiert entsprechend des Nutzerverhaltens mit dem Blockieren gewisser Aktionen und expliziertem Nachfragen nach deren Erlaubnis.

Um den mobilen Client zu entlasten, könnten nur neue Apps überwacht - und bereits bekannte – d. h. sich als ungefährlich erwiesene - Apps aus der Überwachung ausgeschlossen werden.

Da sich eine flächendeckende Sicherheitslösung dieser Art wahrscheinlich nicht durchsetzen wird, könnte man derartige Services für Menschen mit höherem Sicherheitsbedürfnis als Zusatzleistung anbieten.

Literaturverzeichnis

(*android* developers 2012a): o. V.: Publishing on *Google Play*. http://developer.android.com/guide/publishing/publishing.html, (2012-06-18).

(*android* developers 2012b): o. V.: Signing Your Applications. http://developer.android.com/guide/publishing/app-signing.html, (2012-06-18).

(*android* open source project 2012): o. V.: Tech Info. http://source.android.com/tech/security/index.html, (2012-06-20).

(*Dixon* et. al 2011): *Dixon, Bryan; Jiang, Yifei; Jaiantilal, Abhishek*; *Mishra, Shivakant*: Location based power analysis to detect malicious code in smartphones. In: CSS-SPSM'11, New York 2011.

(DroidKungFu 2011): *ThreatSolutions* at F-Secure: Another Android Malware Utilizing a Root Exploit. http://www.f--secure.com/weblog/archives/00002177.html, 2011-06-06, (2012-06-23).

(*Enck* 2009*): Enck, William*; *Ongtang, Machigar*; *McDaniel, Patrick*: Understanding *Android* Security. In: IEEE Computer Society, Pennsylvania 2009, S. 50 – 57.

(*Google Play* 2012a): o. V.: *Google Play – Android* Developer Console. https://play.google.com/apps/publish/signup, (2012-06-18).

(*Google Play* 2012b): o. V.: *Google Play* – Developer Registration. http://support.google.com/googleplay/android-developer/bin/answer.py?hl=en&answer=113468, (2012-06-18).

(*heise* online 2012a): o. V.: Smartphones: *Samsung* und *Apple* überholen *Nokia*. http://www.heise.de/newsticker/meldung/Smartphones-Samsung-und-Apple-ueberholen-Nokia-1561160.html, 2012-04-27, (2012-06-22).

(*heise* online 2012b): o. V.: *Android*-Malware öffnet Hintertür ins Internet. http://www.heise.de/newsticker/meldung/Android-Malware-oeffnet-Hintertuer-ins-Intranet-1566356.html, 2012-05-03, (2012-06-23).

(*heise* Security 2012a): o. V.: *Googles Android*-Malware-Scanner *Bouncer* gepwned. http://www.heise.de/security/meldung/Googles-Android-Malware-Scanner-Bouncer-gepwned-1590542.html, 2012-06-05, (2012-06-19).

(*heise* Security 2012b): o. V.: *Google* löscht *Android*-App auf Smartphones aus der Ferne [Update]. http://www.heise.de/security/meldung/Google-loescht-Android-App-auf-Smartphones-aus-der-Ferne-Update-1028907.html, 2010-06-25, (2012-06-19).

(*Jesus* 2012): *Jesus*, *Yago*: *Bouncer* ¿Sirve para algo?. http://www.securitybydefault.com/2012/02/bouncer-sirve-para-algo.html, 2012-02-21, (2012-06-19).

(*Jiang* 2012): *Jiang, Xuxian*: Security Alert: New *RootSmart Android* Malware Utilizes the *GingerBreak Root* Exploit. http://www.csc.ncsu.edu/faculty/jiang/RootSmart/, 2012-02-13, (2012-06-23).

(*Lockheimer* 2012): *Lockheimer, Hiroshi*: *Android* and Security. http://googlemobile.blogspot.de/2012/02/android-and-security.html, 2012-02-02, (2012-06-18).

(*Oberheide* 2012): Oberheide, Jon: Dissecting *Android's Bouncer*. http://blog.duosecurity.com/2012/06/dissecting-androids-bouncer/, 2012-06-04, (2012-06-19).

(Plankton 2011): o. V.: *Plankton*. http://www.gdata.de/securitylab/mobile/mobile-malware.html?tx_gdavlvisualisation_pi1%5Blink%5D=%2Fmobile%2Fprofile%2FAndroid.Backdoor.Plankton.A%2F&tx_gdavlvisualisation_pi1%5Baction%5D=mobileMalwareDetail&tx_gdavlvisualisation_pi1%5Bcontroller%5D=Visualisation&cHash=8c79fcd2818d82b512004445f249e202, 2011-06, (2012-06-23).

(*Schmidt* et. al 2009): *Schmidt, Aubrey-Derrick*; Bye, *Rainer*; *Schmidt*, *Hans-Gunther*; *Clausen*, *Jan*; *Kiraz*, *Osman*; *Yüksel*, *Kamer Ali*; *Camtepe*, *Seyit Ahmet*; *Albayrak*, *Sahin*: *Static* Analysis of Executables for Colloborative Malware Detection on *Android*. In: IEEE International Conference on Communications (ICC) 2009, Dresden 2009.

(*Zhou* et. al 2012): *Zhou*, *Yajin*; *Wang*, *Zhi*; *Zhou*, *Wu*; *Jiang*, *Xuxian*: Hey, You, Get Off My Market: - Detecting Malicious Apps in Official and Alternative *Android* Markets. In: NDSS 2012.

(*Zsone* 2011): o. V.: Trojan: *Android*/*Zsone*.A. http://www.f-secure.com/v-descs/trojan_android_zsone_a.shtml, 2011-05, (2012-06-23).