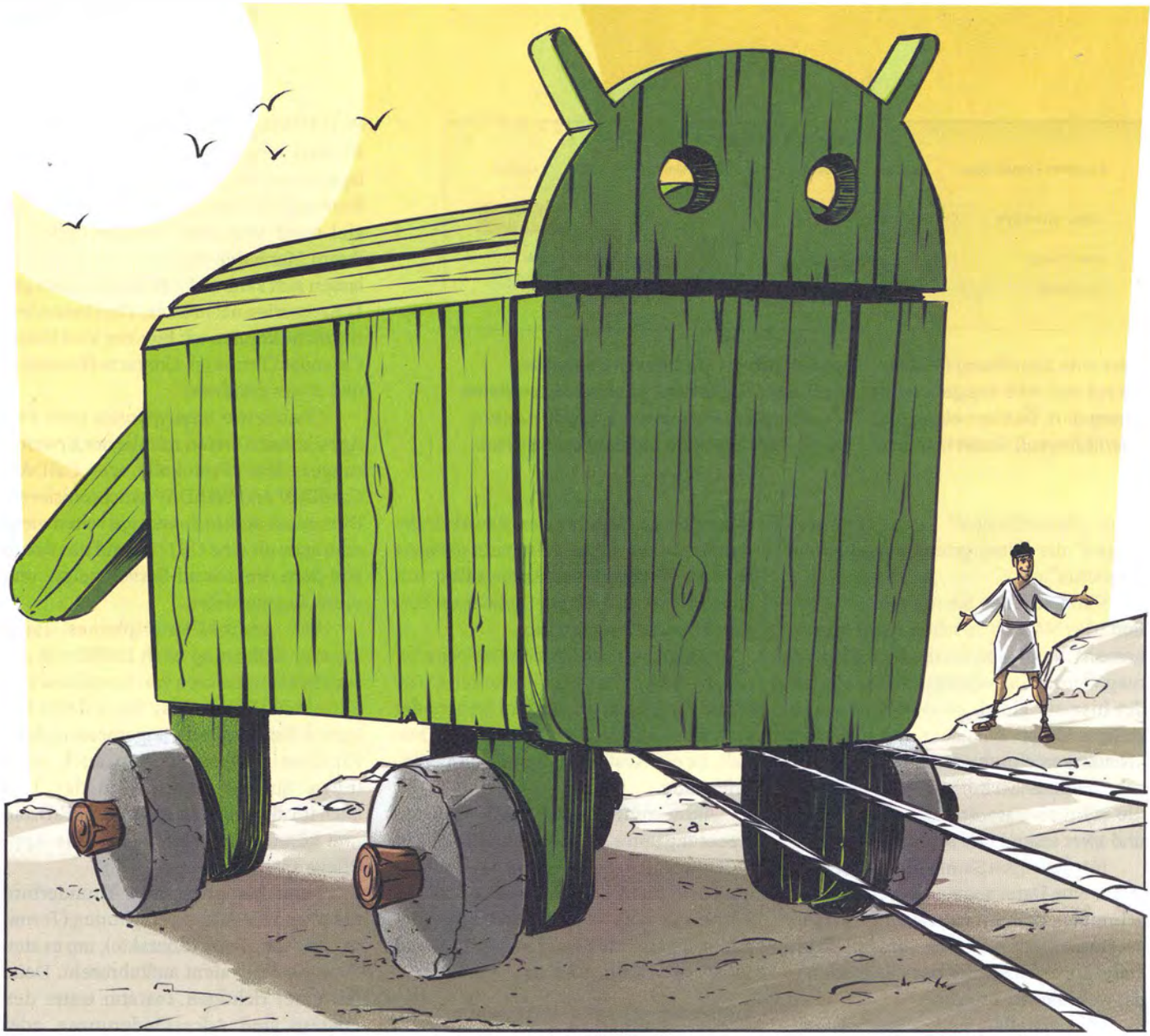


Illustration: Albert Hulm

Android-Trojaner seziert

Verdächtige Android-Apps untersuchen

Unter vielen angeblich nützlichen Apps tummeln sich auch einige schwarze Schafe. Mit Online-Diensten, Analyse-Tools und ein wenig Know-how kann man viele davon enttarnen und einen Blick hinter die App-Kulissen werfen. Die ersten interessanten Einblicke sind nur wenige Klicks entfernt.

Von David Wischnjak und Ronald Eikenberg

Android lässt Nutzern und Entwicklern viele Freiheiten – diese wissen allerdings auch Virenschreiber zu schätzen. Einige Apps führen nach der Installation ein Doppelleben und spionieren ihren Nutzer aus. Wir ließen ein Android-Smartphone infizieren und haben ausprobiert, mit welchen Analysemethoden man Trojanern am besten auf die Schliche kommt – von einfach bis anspruchsvoll. Alle Tools und Links finden Sie unter ct.de/y3zw.

Ein erstes Indiz dafür, ob eine App zwielichtige Absichten hegt, sind ihre Berechtigungen: Passen die eingeforderten Befugnisse so gar nicht zum gebotenen Funktionsumfang, dann ist möglicherweise etwas faul. Jede App benötigt für Aktionen wie das Versenden von SMS oder Zugriff auf den Flashspeicher entsprechende Rechte, die der Nutzer bei der Installation erteilt. Einige besonders heikle Berechtigungen wie das Versenden von SMS oder den Zugriff auf die Kamera muss man seit Android 6 noch mal separat absegnen. Diese Spielregeln gelten auch für Schädlinge, zumindest solange sie das Gerät

nicht rooten oder andere Tricks anwenden. Welche Rechte eine App einfordert, legen Entwickler in der sogenannten Manifest-Datei fest (siehe Kasten „Aufbau von Android-APKs“).

Für einen Überblick über alle installierten Apps und deren Befugnisse haben wir das Tool „Permission Friendly Apps“ von „androidsoft.org“ zurate gezogen. Es zeigt eine Liste an, die nach einem Risikoscore sortiert ist. Dieser wird anhand der erteilten Berechtigungen kalkuliert. Auf unserem Smartphone erschien neben den üblichen Verdächtigen wie Facebook und WhatsApp die Taschenlampen-App „FlashLight“ mit dem Paketnamen „com.zhengjaiu.flight“ ganz oben. Sie nimmt sich unter anderem das Recht, Fotos und Videos aufzunehmen, zu telefonieren, SMS zu verschicken und den Standort abzufragen – für eine vermeintlich simple Taschenlampe ist das schon schwer verdächtig.

Extrahiert und analysiert

Wir entschieden uns, am Rechner einen näheren Blick auf die App zu werfen. Dazu benötigten wir erst mal ihr Application Package (APK). Android hält diese Pakete auch nach der Installation einer App im internen Speicher vor. Mit dem „APK Extractor Lite“ konnten wir die APK-Datei leicht vom Smartphone an den Analyserechner schicken. Das Tool zeigt zunächst eine Liste der installierten Apps an. Ein langes Drücken auf einen Eintrag öffnet das Kontextmenü, der Punkt „Send APK“ ruft den Teilen-Dialog von Android auf. Beim ersten Export mussten wir dem Tool gestatten, auf den Speicher zuzugreifen. Ab Android 8.0 scheint das Tool nicht mehr zu laufen, hier hilft etwa der kostenpflichtige Solid Explorer weiter.

Vom PC aus setzten wir das APK zunächst dem Virenscan-Dienst VirusTotal vor, der es mit etwa 60 Virenscan-Engines auf Schädlingsbefall untersuchte. Offenbar hatten wir mit FlashLight einen Volltreffer gelandet: 28 der Antiviren-Engines hielten die Datei für bösartig. Zumeist wurde sie als „Dropper“ eingestuft – also als Viren-Verteiler, der den eigentlichen Schadcode aus dem Netz nachlädt oder entpackt. Unter „Details“ erfuhren wir, dass die App erstmals am 16. Februar 2016 zur Analyse eingereicht wurde. Zudem wurde sie mit einem äußerst zwielichtigen Zertifikat signiert: Die Angaben „Common Name“ und „Organization“ lauten schlicht „android“ – bei legitimen Apps findet man hier normalerweise den Namen des Entwicklers und die Anbieterfirma.

VirusTotal bestätigt die Diagnose von Permission Friendly Apps: Unter „Permissions“ listet der Analysedienst die insgesamt 53 angefragten Berechtigungen auf. Vom Zugriff auf Kontakte, SMS, Standort, Bluetooth über das Beenden von Prozessen bis hin zum automatischen Starten nach einem Neustart ist alles dabei. Doch VirusTotal verrät noch mehr: FlashLight will unter anderem mitbekommen, wenn das Display eingeschaltet wird, wie die Angabe „android.intent.action.SCREEN_ON“ bei „Intent Filters By Action“ zeigt. Schließlich konnten wir auf der Unterseite „Relations“ auch noch herausfinden, dass die vermeintliche Taschenlampe mit einer Cloudfront.net-URL spricht, die in Verbindung mit zahlreichen Malware-Apps steht. Hierzu nutzten wir VirusTotal Graph (unter Relations), der jedoch nur eingeloggten Nutzern im vollen Umfang zur Verfügung steht.

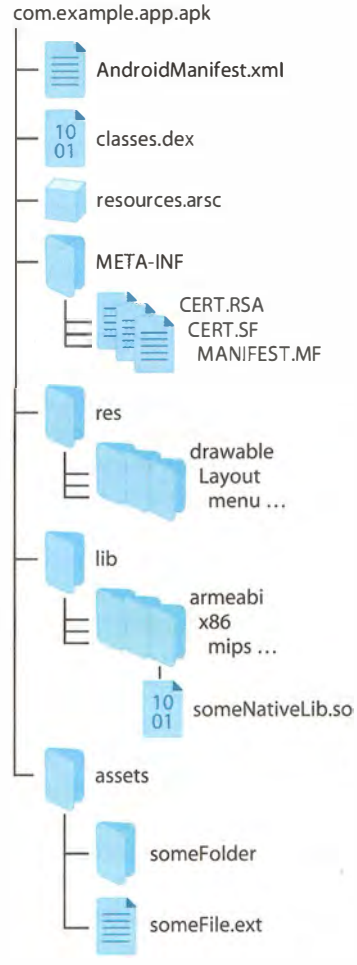
Tiefer buddeln in der Sandbox

Die FlashLight-App ist also höchstwahrscheinlich bösartig. Aber was genau führt sie im Schilde? Um mehr darüber zu erfahren, haben wir das APK bei dem auf Android-Malware spezialisierten Analysedienst Koodous hochgeladen. Es zeigte sich, dass die Datei von anderen Nutzern bereits negativ bewertet wurde. Koodous setzt mehrere Analyse-Werkzeuge wie AndroGuard und Droidbox auf die eingereichten Apps an. Die Ergebnisse findet

Aufbau von Android-APKs

Android-Apps werden in Form von „Android Application Packages“ (APKs) verteilt. Ein APK ist im Grunde nur ein Zip-Archiv und lässt sich mit Packprogrammen wie 7-Zip problemlos öffnen. Im Inneren finden sich üblicherweise an oberster Stelle die Dateien classes.dex und resources.arsc sowie das AndroidManifest.xml. Die classes.dex enthält den Java-Bytecode, und die resources.arsc ist ein komprimiertes Archiv für die Ressourcen der App wie beispielsweise Grafiken, Stringtabellen oder DPI-abhängige Bildschirm-Layouts. Ressourcen können sich aber auch im Ordner res befinden. Einige XML-Dateien in APKs weisen, ungeachtet der Dateiendung, ein binäres, Android-spezifisches Format auf. Sie werden deswegen auch kompilierte XMLs genannt.

Das AndroidManifest.xml ist so eine kompilierte XML-Datei. Sie enthält verschiedene Metadaten wie den Paketnamen und die benötigten Berechtigungen – die App kann später nur auf Schnittstellen zurückgreifen, die hier angefordert sind – außer sie erlangt Root-Rechte. Im Ordner META-INF befinden sich Zertifikate und Prüfsummen und im lib-Ordner vorkompilierter, nativer Code, sortiert in Unterordner nach Prozessorarchitektur. Beliebige weitere Dateien beherbergt der assets-Ordner.



App-Analysedienste im Detail

VirusTotal untersucht Android-Apps mit über 60 Virenscan-Engines auf Schädlingsbefall.

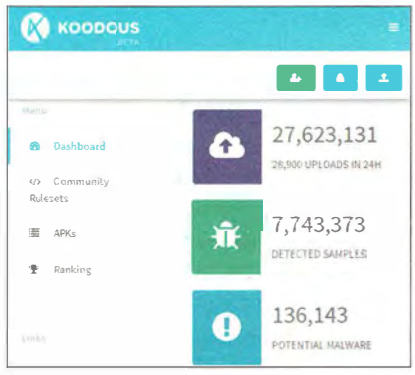
Virustotal ist bekannt für die Analyse von Windows-Malware – hat aber auch bei Android-Apps einiges zu bieten. Es handelt sich bei dem Dienst nicht um einen selbstständigen Scanner, stattdessen greift er auf etwa 60 Virenscan-Engines zurück. Neben den Ergebnissen der Virenscans präsentiert der Dienst unter „Details" Dateiformat-spezifische Metadaten. Darunter befinden sich das Erstellungsdatum, das zum Signieren des APK genutzte Zertifikat und die Berechtigungen. Die Bezeichnungen von Activities (UI-Klassen) und Services (Hintergrunddienste) lassen Rückschlüsse darauf zu, ob sich der Entwickler Mühe gegeben hat, die Interna der App zu verstecken: Handelt es sich lediglich um nichtssagende Bezeichnungen wie a.a.a, dann ist der Code verschleiert (obfuscated).

Über „Intent Filters By Action" erfährt man, bei welchen Systemereignissen die App aktiv wird. Auch ein Blick auf den Registerreiter „Relations" kann sich lohnen: Unten listet VirusTotal die im APK enthaltenen Dateien auf, darüber (Graph Summary) befindet sich eine grafische Darstellung, die etwa kontaktierte Server oder zur Laufzeit entpackte Ressourcen enthalten kann. Der Funktionsumfang von VirusTotal wächst stetig. Seit Kurzem sind auch Sandbox-Analysen für APKs hinzu gekommen. Der Dienst nutzt für APKs derzeit die Sandboxes „Tencent HABO" und „VirusTotal Droidy", weitere sollen folgen. Die Funktion war bei Redaktionsschluss allerdings noch nicht für alle Apps verfügbar.

Koodous hat sich auf das Sezieren von Android-APKs spezialisiert. Der Dienst katalogisiert alle eingereichten APK-Dateien und stellt sie anderen Nutzern zum Download bereit. So ist nach und nach eine riesige Malware-Sammlung entstanden. Die Nutzer können die Apps bewerten und kommentieren und so zur Klassifizierung beitragen. Um nach APKs zu suchen, klickt man links auf „APKs" und benutzt anschließend das Suchfeld. Eine Suche nach `flashlight rating:"-2"` förderte eine beachtliche Auswahl negativ bewerteter und damit potenziell schädlicher Apps zu Tage. Fortgeschrittene können sogenannte YARA-Regeln erstellen und damit die Datenbank nach Apps mit bestimmten Merkmalen oder Verhalten durchsuchen.

Koodous untersucht APKs mit den etablierten Analyse-Tools AndroGuard, Droidbox und Cuckoo. Bei AndroGuard handelt es sich um ein statisches Analysewerkzeug. Es sammelt also Informationen über das APK, ohne es auszuführen. Zu sehen sind unter anderem die Namen der enthaltenen Activity-Klassen, Services und Permissions (App-Berechtigungen).

Droidbox und Cuckoo sind dynamische Analysewerkzeuge und protokollieren das Verhalten der App zur Laufzeit. So findet man etwa heraus, auf welche Dateien eine App zugreift, welche Dienste sie startet und ob sie versucht hat, SMS zu verschicken oder Telefonanrufe zu tätigen. Unter „Strings" findet man eine Übersicht über die in dem APK gefundenen Zeichenfolgen.



Koodous bietet neben diversen Analyse-Tools auch ein großes Archiv verseuchter APK-Dateien.

Joe Sandbox führt Apps in einer Sandbox aus und stuft sie in Schädlingskategorien ein.

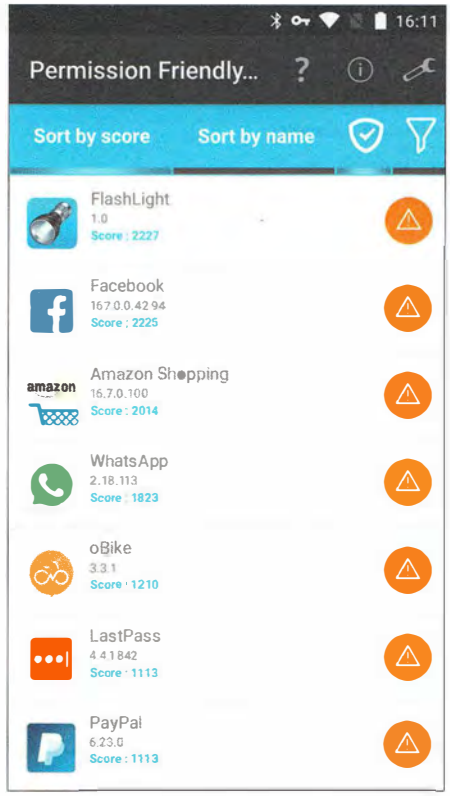
Joe Sandbox versucht anhand des beobachteten Verhaltens einzuschätzen, ob die untersuchte APK-Datei Böses im Schilde führt. Schon in der kostenlosen Basic-Variante führt Joe eine äußerst detaillierte Analyse durch. Die Online-Sandbox kann Programme für Windows, macOS, Android und iOS ausführen. Zur Analyse einer Android-App wechselt man zunächst auf den Registerreiter „Android" und wählt über „Upload Sample" eine APK-Datei aus. Nach einem Klick auf den Analyze-Knopf sind einige Minuten Geduld gefragt. Anschließend hat man die Wahl, ob man den Analysebericht direkt im Browser betrachtet oder als PDF-Datei herunterlädt.

Weit oben im Bericht unter „Detection" findet man das zusammengefasste Testergebnis, das anhand mehrerer Faktoren berechnet wird. Für das Auge gibt es eine schmucke Grafik, welche die Einstufung der App in verschiedene Malware-Klassen wie Ransomware, Bot und Spyware vornimmt. Weiter unten folgt eine Mischung aus dynamischer und statischer Analyse der App. Dort werden verdächtige Eigenschaften nach Oberbegriffen wie „Anti-Debugging", „Privilege Escalation" oder „Data Obfuscation" aufgelistet. Joe Sandbox überwacht auch den Netzwerkverkehr und schlüsselt auf, mit welchen IP-Adressen die App auf welchen Ports kommuniziert hat. Passend dazu erstellt der Dienst eine Weltkarte mit den ungefähren Standorten der Hosts.

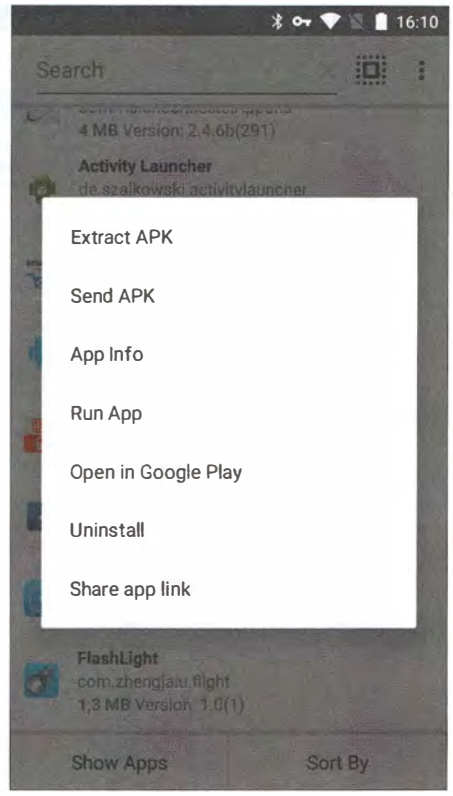
man unter „Analysis Report“. Das AndroGuard-Modul liefert ähnliche Metadaten wie die Details-Seite von VirusTotal. Droidbox (und ähnlich auch Cuckoo) versucht, die App in einer virtuellen Umgebung auszuführen. Bei der Droidbox-Analyse kam heraus, dass die Taschenlampe zur Laufzeit eine APK-Datei namens „protect.apk“ in ihr App-Verzeichnis /data/data/com.zhengjaiu.flight/cache/ geschrieben und anschließend darauf zugegriffen hat. Das sieht so aus, als sei eine weitere App installiert worden – leider erfuhren wir nicht, was sie anstellt. In vielen Fällen erkennt Malware die Sandboxes von Koodous und ändert ihr Verhalten – das führt dann zu unvollständigen Ergebnissen.

Um besser zu verstehen, was da genau passiert, mussten wir mehr über das mysteriöse „protect.apk“ in Erfahrung bringen. Dafür untersuchten wir FlashLight auch noch mit Joe Sandbox. Der Dienst liefert deutlich detailliertere Ergebnisse als VirusTotal und Koodous. Er zeigt nicht nur die aufgezeichneten Aktivitäten an, sondern auch eine anhand des Verhaltens generierte Bewertung. FlashLight ist mit 56 von 100 möglichen Punkten „Malicious“ (also schädlich) und passt vor allem in die Schädlingskategorien „Spyware“ und „Evader“. Letzteres bedeutet, dass die App etwa Schutzmechanismen des Betriebssystems umgeht oder ihr Verhalten verschleiert. Passend dazu erscheinen die „Warnings“ am Ende der ersten Tabelle. Ein Klick auf „Show all“ verrät, dass es zu einem Fehler bei der Ausführung kam („An application runtime error occurred“). Das bedeutet, dass die App abgestürzt ist und der dynamische Teil des Reports wahrscheinlich unvollständig ist. Weiterhin rügte die Sandbox vor allem die weitreichenden Berechtigungen („Has permission to [...]“) und dass eine APK-Datei entpackt wurde („Drops a new APK file“) – was sich mit den Beobachtungen deckte, die wir bereits mit Koodous gemacht hatten.

Außerdem soll die vermeintliche Taschenlampen-App regen Gebrauch vom DexClassLoader („Uses the DexClassLoader“) gemacht haben, was darauf hindeutet, dass die App nachgeladenen Code ausführen kann. Details zu den Anschuldigungen gab es jeweils unter dem Link „Show sources“, wo die entsprechenden API-Aufrufe angegeben sind. Joe Sandbox erstellt bei der Ausführung sogar Screenshots, die in einer animierten Bilderstrecke zusammengestellt werden. In unserem Fall lieferte der Mitschnitt jedoch nur den Beleg dafür, dass die App abgestürzt war. Da FlashLight anscheinend Code enthält, um die Analyse zu erschweren, ist es gut möglich, dass der Absturz kein Zufall war. Der Report bot zudem noch allerlei weitere Details wie eine Auflistung der ausgeführten sowie der nicht ausgeführten Methoden: Von 227 Methoden kamen gerade mal 12 zum Einsatz.



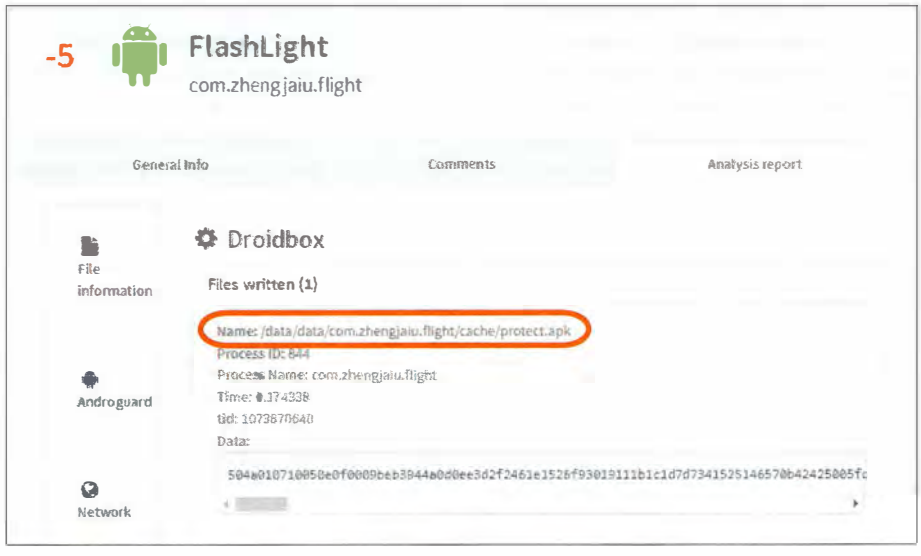
Permission Friendly Apps sortiert Apps danach, wie viele Berechtigungen sie einfordern.



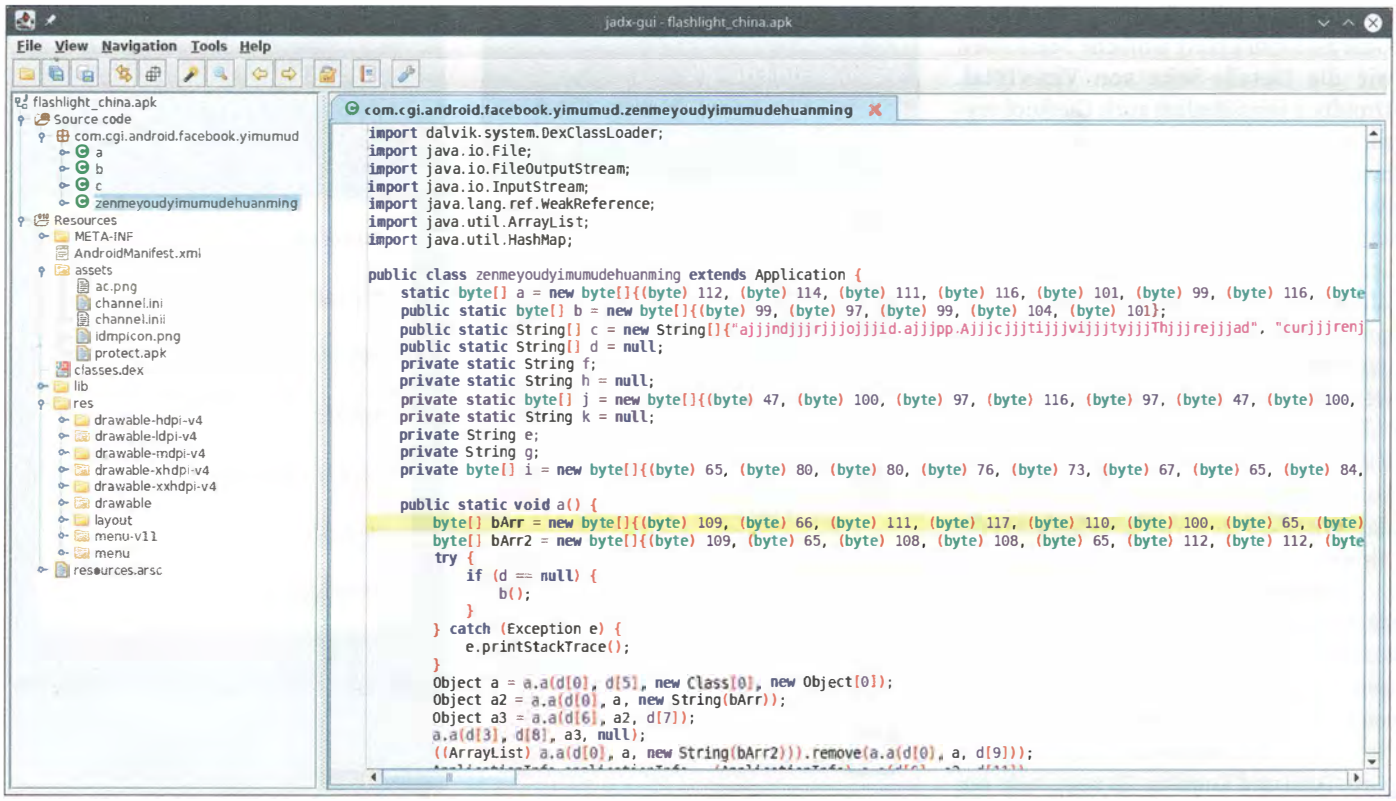
Mit APK Extractor Lite überträgt man APK-Dateien installierter Apps zur Analyse auf den Rechner.

Allzweckwaffe Decompiler

Trotz der umfangreichen Sandbox-Reports wussten wir noch immer nicht



Laut Koodous schreibt die von uns analysierte Taschenlampen-App nach dem Start eine Datei namens „protect.apk“ und greift darauf zu.



Die dekompilierte Activity-Klasse der Taschenlampe in jadx: Imports, Byte-Arrays und verschleierter Quellcode

genau, wofür die weitreichenden Berechtigungen tatsächlich eingesetzt werden und was das im App-Verzeichnis platzierte protect.apk damit zu tun hat. Zum Glück kann man recht leicht einen Blick in den Quellcode von Android-Apps werfen, da sie sich meist gut dekompilieren lassen. Wir öffneten das FlashLight-APK mit dem Reverse-Engineering-Tool jadx, um die Innereien des Schädlings zu erkunden.

Wir bekamen ein Package mit den vier Klassen „a“, „b“, „c“ und „zenmeyoudyimumudehuanming“ zu sehen – letzteres ist Chinesisch und lautet übersetzt „Wie man einige Szenen der Freude hat“. Der Code-Einstiegspunkt ist androidtypisch die onCreate()-Methode der ersten Activity-Klasse – in unserem Fall die Klasse mit dem unaussprechbaren Klassennamen. Um eine Idee davon zu bekommen, was die verschwurbelten Klassen machen, schauten wir uns zuerst ihre Imports an. Diese findet man immer oben am Beginn einer dekompilierten Klasse und sie lassen sich kaum verheimlichen. So verraten sie, wonach man im Codesalat Ausschau halten sollte. Die Activity-Klasse von FlashLight besitzt unter anderem die Imports „dalvik.system.DexClassLoader“ und „java.io.FileOutputStream“, was die Vermutung bestätigt, dass etwas in den Speicher geschrieben und dann ausgeführt wird. Die Klasse „a“ macht laut den Imports etwas mit Java-Reflection („java.lang.reflect.Field“) und „b“ scheint mit Dateien zu arbeiten („java.io.File“). „c“ hat keine Imports und enthält auch kaum Code.

Das konnte aber nicht alles sein, denn wir fanden in den vier Klassen nicht genug Code, der die umfangreichen Berechtigungen erklären würde. Wir entdeckten in den Activity-Klasse außerdem mehrere verdächtige Byte-Arrays, aus denen mit „new String(arr)“ Zeichenfolgen erstellt werden. Offensichtlich bestehen die Arrays einfach nur aus ASCII-Werten. Wir kopierten die entsprechenden Zeilen und übersetzten sie mit einem ASCII-Konverter. Es kamen dabei Zeichenfolgen wie „protect.apk“ oder „/data/data“ heraus.

Innerhalb von jadx schauten wir anschließend in den Ordner Resources/assets der App. Dort liegen unter anderem zwei png-Dateien und eine protect.apk. Doch der Schein trügt, denn nachdem wir diese Dateien mit dem Zip-Entpacker 7-Zip aus dem APK extrahiert hatten, konnten wir weder protect.apk noch die PNGs öffnen. Mit dem *nix-Kommandozeilen-Tool file versuchten wir daraufhin, das echte Dateiformat festzustellen. Doch das entpuppte sich als Sackgasse: Bei allen drei Dateien kam nur „data“ raus, die Dateien sind also irgendwie verschlüsselt oder gepackt.

Virtueller Androide

Die größte Unbekannte blieb also nach wie vor die Datei protect.apk, welche bei der Ausführung von FlashLight im App-Verzeichnis landet und ausgeführt wird. Um an die ominöse Datei zu kommen, hätten wir sie prinzipiell vom infizierten Smartphone ziehen können. Allerdings befindet sie sich in einem Unterverzeichnis von /data/data, auf das man nur mit Root-Rechten zugreifen kann. Da wir das Smartphone nicht rooten konnten, hatten wir im Wesentlichen noch zwei Möglichkeiten: Die eine war, die Entpackroutine aus der dekompilierten Taschenlampen-App zu kopieren und zu versuchen, sie lokal auf dem PC auszuführen. Ersetzt man die Android-Abhängigkeiten so weit wie nötig mit Java-eigenen Methoden, entpackt der Code mit etwas Glück die fraglichen Dateien. Die andere Möglichkeit war, die Malware in einer virtuellen Android-Maschine (VM) auszuführen, um uns anschließend mit Root-Rechten im geschützten Datenverzeichnis der App zu bedienen. Wir entschieden uns für eine

Android-VM in der Virtualisierungs-Software VirtualBox.

Das virtuelle Android war schnell an den Start gebracht: Wir haben ein Android-6-Image von osboxes.org heruntergeladen und in VirtualBox einer Linux-VM als Festplatte zugewiesen. Die genaue Vorgehensweise erklärt der Kasten „Android-VM mit VirtualBox“. Zunächst mussten wir uns über die Android Debug Bridge (ADB) mit der VM verbinden:

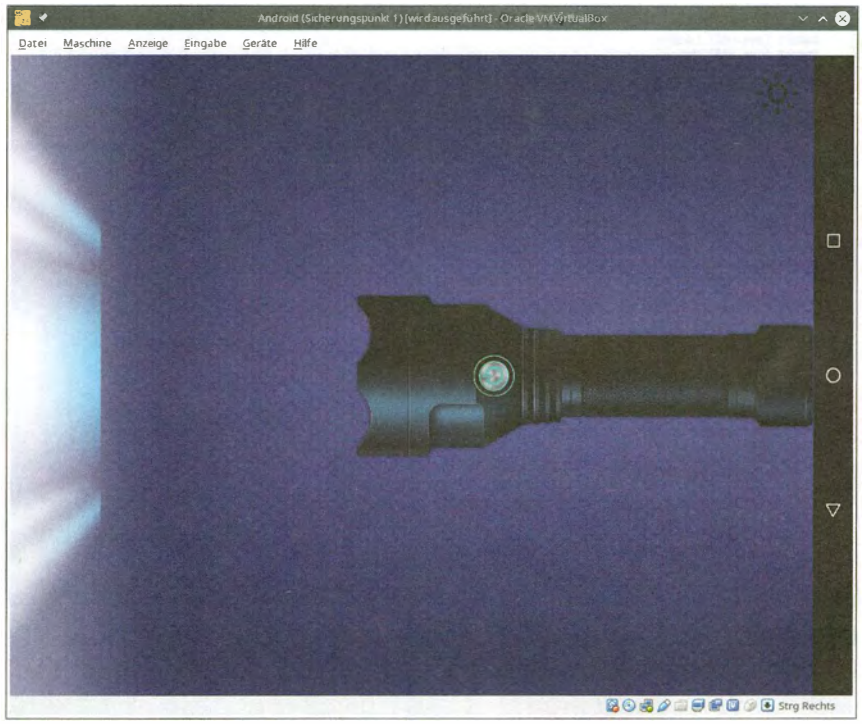
```
adb connect <ip der VM>
```

Anschließend installierten wir FlashLight mit

```
adb install flashlight.apk
```

und starteten die App von Hand. Mit dem Befehl `adb root` verschafften wir uns danach Root-Rechte auf dem virtuellen Android-Gerät. Da der ADB-Server neu startete, mussten wir mit `STRG + c` und `adb connect <ip>` die Verbindung wiederherstellen. Mit dem folgenden Befehl konnten wir schließlich das App-Verzeichnis der Taschenlampe auf das Analysesystem ziehen:

```
adb pull /data/data/com.zhengjaiu↵
  ↳.flight
```



Die **FlashLight-App in der Android-VM – der Wolf im Taschenlampen-Pelz**

Trojanische Taschenlampe

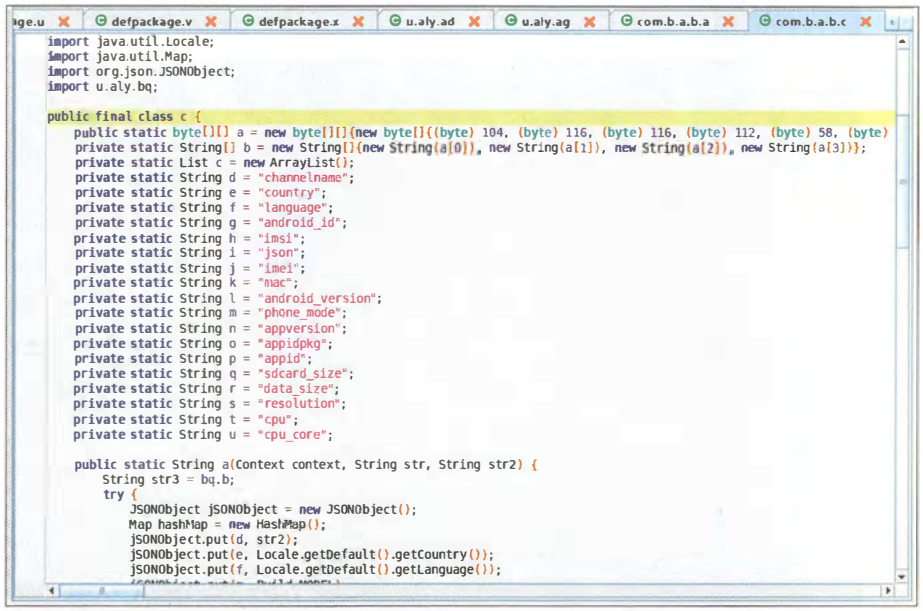
Tatsächlich fanden wir in einem Unterverzeichnis das entpackte protect.apk. Dieses warfen wir wieder dem Decompiler jadx zu – und siehe da, es ist ein Android-Bot! Er enthält eine ganze Menge Klassen, deren Code nur leicht verschleiert ist. Wir entdeckten viele Codeteile, die sich mit der Kommunikation mit den Kontrollservern beschäftigen. Allerlei gerätespezifische Daten wie IMEI, CPU-Typ, WiFi-MAC und die Android-Version werden abgefragt und verschickt. Etwas AES-Cryptocode inklusive hardkodierter Passwörter gibt es auch.

Jetzt war klar, wofür die Berechtigungen gebraucht werden: Der Bot kann beliebige weitere Malware in Form von Dex-Dateien vom Kontrollserver nachladen und ausführen. Da die nachgeladene Mal-

Android-VM mit VirtualBox

Wer eine verdächtige Android-App analysieren möchte, kann innerhalb weniger Minuten eine virtuelle Testumgebung aufsetzen. Dazu benötigt man neben der quelloffenen Virtualisierungssoftware VirtualBox lediglich ein Android-Image. Wir haben gute Erfahrungen mit dem 64-bittigen Android-6-Image von osboxes.org gemacht (siehe ct.de/y3zw), die folgenden Schritte funktionieren aber auch mit anderen Images. Laden Sie zunächst das Android-Image herunter und entpacken Sie die 7-Zip-Datei. Starten Sie VirtualBox und klicken Sie oben links auf „Neu“, um eine neue virtuelle Maschine anzulegen. Klicken Sie im erscheinenden Dialog anschließend auf den Button „Expert-Modus“. Jetzt können Sie einen beliebigen Namen wie „Android 6“ eingeben. Legen Sie den Typ „Linux“ sowie die Version „Linux 2.6 / 3.x / 4.x (64-bit)“ fest. Als Speichergröße (RAM) stellen Sie mindestens 2048 MByte ein. Wählen Sie nun noch die Option „Vorhandene Festplatte verwenden“, als Festplatte Ihr Android-Image und klicken Sie auf „Erzeugen“.

Die VM taucht jetzt in der Liste auf und ist bereit für die Konfiguration: Wählen Sie dafür den gerade erstellten Eintrag aus und klicken Sie oben links auf „Ändern“. Um Probleme mit der Maus-Steuerung zu vermeiden, wählen Sie bei „System / Hauptplatine / Zeigegerät“ die „PS/2-Maus“ aus. Unter „System / Prozessor“ stellen Sie zwei oder mehr CPUs, unter „Anzeige / Bildschirm“ mindestens 64 MByte Grafikspeicher und unter „Netzwerk / Adapter 1“ die „Netzwerkbrücke“ ein. Die Netzwerkbrücke ist nötig, damit Sie wie in unserer Analyse beschrieben über Android Debug Bridge (ADB) auf die virtuelle Maschine zugreifen können, zudem erhält die VM dadurch Internetzugriff. Der Nachteil ist, dass die VM dadurch auch alle Geräte in Ihrem Heimnetzwerk erreichen kann. Uns ist zwar bisher keine Android-Malware bekannt, welche Geräte im lokalen Netzwerk angreift, doch wenn Sie auf Nummer sicher gehen möchten, wählen Sie in den Netzwerk-Einstellungen der VM bei „Adapter 1“ die Option „VirtualBox Host-Only Ethernet Adapter“. Dann befindet sich die VM in einem lokalen Netz mit dem Host-PC, kann aber nicht auf das Heimnetz oder das Internet zugreifen. Ihre virtuelle Maschine ist nun startklar!



Das zur Laufzeit entpackte protect.apk sammelt gerätespezifische Daten, um einen passenden Exploit zu finden.

ware ihre Berechtigungen von der vermeintlichen Taschenlampen-App erbt, werden einfach alle möglichen Berechtigungen im Voraus reserviert. Eine Google-Recherche anhand der im Code gefunden URLs ergab, dass es sich dabei vermutlich um eine Variante der Ztorg-Malware handelt. VirusTotal bestätigte diese Vermutung, nachdem wir protect.apk dort hochgeladen hatten.

Der Schädling fragt diverse Gerätedaten ab, um anschließend einen passenden Exploit vom Kontrollserver zu bekommen und sich so ungefragt Root-Rechte zu verschaffen. Danach hat er freie Hand. Chinesische Werbe-SDKs deuten darauf hin, dass auch noch Werbung nachgeladen wird. Zu guter Letzt entdeckten wir dann auch noch den Code für die Taschenlampe.

Die von uns durchgeführte Analyse mit Online-Sandboxes & Co. eignet sich hervorragend, um in kurzer Zeit herauszufinden, was eine App im Schilde führt. Das Hochladen der APK-Dateien ist unbedenklich, da die Dateien ja zumeist ohnehin aus einer öffentlichen Quelle stammen – nämlich Google Play – und darüber hinaus keine persönlichen Daten enthalten. Professionelle Virenanalysten würden in brenzligen Fällen jedoch davon absehen, den verdächtigen Code öffentlich zu teilen. Bei gezielten Angriffen kann es es sich um individuell erstellte Schädlinge handeln, die nur einmal zum Einsatz kommen. Lädt man eine solche Datei etwa bei VirusTotal hoch, können das auch die Trojaner-Entwickler herausfinden – und werden alarmiert. *(rei@ct.de)* c't

Alle Analyse-Tools und -Dienste:
ct.de/y3zw

Anti-Analyse-Techniken

Malware-Entwickler versuchen, die wahren Absichten ihrer Apps möglichst zu verheimlichen. Denn je länger eine App unentdeckt bleibt, desto größer ist ihr Profit. Typisch für Malware ist verschleierter und mehrstufig verpackter Quellcode. Java-Bytecode, wie er bei Android-Apps zum Einsatz kommt, lässt sich oft zu gut lesbarem Sourcecode dekompilieren. Daher werden vor dem Kompilieren alle Klassen, Methoden und Variablen zu nichtssagenden Zeichenketten umbenannt. Die Entwicklungsumgebung Android Studio bietet mit ProGuard bereits ein vorinstalliertes Plug-in dafür, der Eintrag „minifyEnabled true" in der Datei build.gradle genügt.

Auch Zeichenfolgen wie URLs tauchen in Malware meist nur verschlüsselt auf und werden zur Laufzeit entschlüsselt. Viele Schädlinge gehen oft noch einen Schritt weiter und verstecken den eigentlichen Schadcode in scheinbar harmlosen Dateien wie „background.png". Das funktioniert, weil Android das dynamische Nachladen von Code erlaubt. Beim ersten Starten der App wird der Schadcode entpackt und ausgeführt. Genauso kann Malware auch Code von einem externen Server nachladen.

Android erlaubt das Auslagern von Code in native, meist in C/C++ geschriebene Programmbibliotheken (Libraries). Diese müssen speziell für die verwendete Prozessorarchitektur kompiliert sein und sind eigentlich für performancekritischen Code gedacht. Native Libraries sind deutlich schwerer zu analysieren als Java-Bytecode. Gleichzeitig dient das auch als Anti-VM-Technik: Virtuelle Maschinen (VM) bilden oft die Basis für dynamische Analysewerkzeuge. Aus Performancegründen wird dann meist Android X86 eingesetzt: ein Android, welches auf x86-64 Prozessoren ausgeführt werden kann. Darauf laufen aber Libraries für ARM-Architekturen nicht ohne Weiteres. Malware versucht zudem häufig, das Ausführen in einer VM zu erkennen und verändert dann ihr Verhalten. Herkömmliche VMs verraten sich beispielsweise schon, wenn sie von der App nach den Bezeichnungen der Gerätehardware gefragt werden.

Generell nähern sich Android-Schädlinge in ihrer Komplexität immer mehr ihren Windows-Artgenossen an. Eine fiese Taktik der Malware-Entwickler ist, den Schadcode in einer nativen Library zusätzlich zu verschlüsseln. Dann lohnt sich der Griff zu einem echten Smartphone, auf dem die Analysewerkzeuge auf Kernel-Ebene mitlaufen, oder eine aufwendige statische Analyse, beispielsweise mit IDA Pro. Einige Code-Packer arbeiten sogar mit kleinen virtuellen Maschinen, in deren Sprache dann der Entpacker-Code geschrieben ist. Für solche Fälle kann der android-unpacker von strazzere ein guter Anlaufpunkt sein. Glücklicherweise sind die meisten Malware-Entwickler zu faul für solchen Aufwand.