Die approbierte Originalversion dieser Diplom-/Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at

TU UB WIEN Universitätsbibliothek

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

TU WIEN

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Malvertising und Malware im Kontext von Android

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Nikolai Fraihs, BSc
Matrikelnummer 0826420

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Edgar Weippl
Mitwirkung: Dr Markus Huber, MSc

Wien, 12. September 2017

Nikolai Fraihs　　　Edgar Weippl

Erklärung zur Verfassung der Arbeit

Nikolai Fraihs, BSc
Stefaniegasse 3, 2391 Kaltenleutgeben

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 12. September 2017

Nikolai Fraihs

Danksagung

An dieser Stelle möchte ich allen Personen danken, welche mich tatkräftig bei dieser Diplomarbeit sowohl fachlich als auch persönlich unterstützt haben.

Zu Beginn möchte ich mich bei Dipl.-Ing. Mag.rer.soc.oec Dr.techn. Edgar Weippl für die konstruktive Kritik an dieser Diplomarbeit und seiner fachlichen Unterstützung bedanken.

Mein besonderer Dank gilt Dr.techn. Markus Huber MSc welcher mich aktiv bei der wissenschaftlichen Ausarbeitung dieses Themas unterstützt hat. Mithilfe seiner professionellen Fragen und Anregungen, trug er zu der fachlichen Ausarbeitung meiner Ideen bei.

Ein großer Dank geht an meine Familie und Freunde, welche viel Zeit in die Korrektur meiner Arbeit investiert haben. Sie wiesen mich unermüdlich auf etwaige Verständnisprobleme hin, welche für fachfremde Personen ein Hindernis beim Lesen dieser Arbeit darstellten.

Kurzfassung

Smartphones erobern die Welt und der Großteil der Bewohner dieser Erde wird in naher Zukunft einen dieser intelligenten Wegbegleiter besitzen. Von dem ursprünglich puren Kommunikationsmittel, entwickelte sich das Handy bis heute zu einem wichtigen Helfer bei der täglichen Informationsbeschaffung von Internetnutzern. Malware-Entwickler erkennen diesen Wandel und kreieren demzufolge ausgeklügelte Mechanismen, um mobile Betriebssystem wie Android oder iOS mit schädlichem Code zu infizieren. Anlässlich der Unabhängigkeit angesichts der nötigen Benutzerinteraktion, bilden Drive-by-Downloads für mobile Endgeräte eine attraktive Alternative zu herkömmlichen Angriffsvektoren wie Social Engineering Kampagnen.

Trotz der Tatsache, dass bisher keine Drive-By-Downloads für Android *in-the-wild* beobachtet wurden, steht die Lokalisierung dieses Angrifftyps im Fokus dieser Arbeit. Zu Beginn wird ein Überblick über aktuelle und vergangene Angriffstechniken gegeben, um ein Grundwissen über Malware und deren Angriffsvektoren zu vermitteln. Anschließend illustriert die detaillierte Darstellung und Erklärung einzelner Sicherheitslücken, deren Verwendung in Drive-by-Downloads auf böswilligen Webseiten im Kontext des Androidbetriebssystems. Anlässlich der davon ausgehenden Gefahr für Internetnutzer mit Androidgeräten, wird ein neuartiger Webcrawler präsentiert, welcher Drive-by-Downloads in Malvertising-Kampagnen aufspürt.

Damit der Crawler bei Fingerprinting-Prozessen von böswilligen Skripten nicht als solcher erkannt wird, werden ausgewählte Android- und Browseremulatoren sowie zwei Echtgeräte anhand webbasierten Erkennungsmerkmalen verglichen. Die Parameter zur Differenzierung der überprüften Systeme beinhalten unter anderem verfügbare Javascript-Objekte, Sensordaten, Bildschirmauflösung und unterstützte Schriftarten. Angesichts der gewonnen Informationen wird eine Konfiguration für den implementierten Webcrawler präsentiert, welche ihn als perfekte Imitation eines Androidsmartphones erscheinen lässt.

Abstract

Smartphones are on the rise and the majority of the people on this planet, will be in possession of such a small helper in the near future. From a pure tool of communication, mobile phones have changed their daily usage to a valuable assistant in the sustaining quest of information by users. To address this transformation, malware developers issue sophisticated paths to compromise mobile operating systems like iOS or Android. Due to the independence of user interactions, drive-by downloads on smartphones provide an excellent choice in contrast to conventional attack vectors like social engineering campaigns.

The automatic detection of drive-by downloads builds the main focus of this work, despite the fact, that there hasn't been such an attack for android in the wild yet. To impart knowledge of malware and related attack vectors, there is an outline of current and past attack techniques. Furthermore, some chosen vulnerabilities of the android operating system and their utilization in potential drive-by downloads are declared. In consequence of posing a threat to Android users, a novel web crawler is presented, which recognizes drive-by downloads in malvertising campaigns.

In order that the implemented crawler is not spotted by malicious fingerprinting web pages, selected Android and browser emulators are compared with two real devices by web-based characteristics. The examined list of parameters includes without limitation available Javascript objects, calculated accelerometer data, screen resolutions and supported fonts. Regarding the acquired information, a crawler configuration is showcased, which obtains a superb illusion of a real android device in respect of available web parameters.

Inhaltsverzeichnis

Kurzfassung **vii**

Abstract **ix**

Inhaltsverzeichnis **xi**

1 Einführung **1**
1.1 Problemstellung 1
1.2 Ziel der Arbeit 2
1.3 Methodisches Vorgehen 3
1.4 Forschungsfragen 4

2 Hintergrund **5**
2.1 Malware 5
2.2 Anatomie von Malwareattacken 8
2.3 Emulatoren und Imitatoren 14

3 Stand der Technik **17**

4 Evaluierung **23**
4.1 Remote-Exploits für Android 23
4.2 Android-Emulatoren 45

5 Resultate **49**
5.1 Emulatoren 49
5.2 DryCrawl - Design 59
5.3 DryCrawl - Module 60
5.4 DryCrawl- Implementierung 62

6 Disskusion **77**
6.1 Zukünftige Forschungen 79

7 Zusammenfassung **81**

Abbildungsverzeichnis **83**

Tabellenverzeichnis **83**

List of Algorithms **86**

Literaturverzeichnis **87**

KAPITEL 1

Einführung

1.1 Problemstellung

Smartphones erobern die Welt und die Anzahl der Smart-Devices steigt laufend. Aktuell liegt die Zahl der benutzten Geräte bei rund 1.8 Milliarden [1], Tendenz steigend zu geschätzten 2.6 Milliarden im Jahr 2019 [1]. Künftig wird nahezu jeder Mensch ein Smartphone oder ein ähnliches Gerät besitzen, mit dem der Zugriff auf das Internet möglich ist. Damit werden Mobile-Betriebssysteme wie Android, iOS und Windows Phone zu den meist benutzten Systemen aufsteigen. Deshalb werden sie für Malware-Entwickler stetig interessanter, insbesondere Android mit einem Marktanteil von knapp 86% im zweiten Quartal 2016.

Anfangs machten die Benutzer überwiegend Fotos und teilten diese mit ihren Freunden. Später wurde das Smartphone mehr und mehr zum Surfen im Internet benutzt, sodass die Anzahl der mobilen Geräte im World Wide Web stieg. Heute benutzen rund 63.1% [2] der Smartphonebesitzer ihren täglichen Begleiter, um das Internet zu durchforsten.

Malware für mobile Geräte existiert in unterschiedlichen Formen: AdWare, Trojan-SMS, Trojan-Banking und viele mehr. Die Mehrheit der böswilligen Programme muss aktiv durch einen Benutzer installiert werden, oder ist Teil von einem anderen, oft free-to-use, Programm. Das beschränkt die Verbreitung von Malware nach und nach, weil die Abhängigkeit auf eine Benutzerinteraktion die Infizierung verhindert oder verlangsamt. Malware als Teil eines anderen Programms wird häufig durch Anti-Viren Software entdeckt, und limitiert somit die Verbreitung ebenfalls. Daher brauchen Malware-Entwickler einen anderen Weg, ohne Hilfe von Benutzern oder Drittprogrammen, Smartphones zu infizieren: Drive-By-Downloads.

Drive-By-Downloads existieren bereits seit langer Zeit für Desktop-PCs und sind ein ausgeklügelter Angriffsvektor um Malware zu verteilen. Für eine Infizierung ist es ausreichend, eine kompromittierte Webseite zu besuchen, die eine Sicherheitslücke im Betriebssystem

ausnutzt. Jüngstes Beispiel für einen Drive-By-Exploit unter Android ist Stagefright. Die Sicherheitslücke besteht in den Versionen 2.3 bis 5.1.1. Laut Schätzungen benutzen rund 95% [3] aller Androidgeräte weltweit, das sind etwa 950 Millionen Smartphones [3], ein System mit dieser Lücke.

Während Würmer wie Sasser und Blaster aus der Windows-XP-Zeit sich selbstständig über Netzwerke verteilten, brauchen Drive-By-Downloads eine Verteilungsplattform, die ein Benutzer willentlich oder unwillentlich besucht. Dazu eignen sich kompromittierte, hoch frequentierte Seiten oder Werbenetzwerke.

Da Ad-Networks auf vielen tausenden Seiten gleichzeitig sichtbar sind, sind diese für Malware-Entwickler besonders interessant. Prominente Netzwerke wie *Google AdWords*, *Bing Ads* oder *Facebook Audience Network* haben hohe Sicherheitsanforderungen an die eingeblendete Werbung, um Missbrauch bestmöglich zu unterbinden. Trotz erheblicher Bemühungen ihrerseits, gelingt es Angreifern immer wieder die festgelegten Sicherheitsrichtlinien zu umgehen, um ihre Malware zu verbreiten, wie die Verteilung eins Banking-Trojaners[4] auf einer russischen Nachrichtenseite beweist.

1.2 Ziel der Arbeit

Das Ziel der Arbeit ist die Entwicklung eines neuartigen Crawlers, welcher automatisch Drive-By-Downloads für Mobilgeräte und andere Mobile-Malvertising-Kampagnen erkennt. Um sicherzustellen, dass der Crawler nicht frühzeitig durch webbasierte Fingerprinting-Prozesse von Malwareverteilern als solcher identifiziert und dadurch ausgeschlossen wird, soll zusätzlich eine Konfiguration erarbeitet werden, welche ihn als perfekte Imitation eines Android-Smartphones auftreten lässt.

Im Konkreten soll ein Überblick über aktuelle Remote-Sicherheitslücken, deren Machbarkeit und Ursprung gegeben werden. Die dadurch gewonnenen Informationen sollen analysiert und anschließend in einzelne Module zur automatischen Erkennung der Sicherheitslücken verpackt werden. Außerdem soll *Pagehijacking* in Bezug auf Mobile-Browser und Browser-APIs untersucht und mögliche Angriffsvektoren analysiert werden. Darüber hinaus soll der Crawler als modulares Framework entworfen werden, sodass eine spätere Ergänzung zur Erkennung von weiteren Angriffen oder betrügerischen Aktivitäten gewährleistet ist.

Weiters soll die Arbeit auf die Identifizierung von Android-Betriebssystemen und Emulatoren durch clientseitige Browserskripte und die Aufgabe, die sie in einem Angriffsvektor erfüllen, eingehen. Dazu wird zu Anfang, eine Liste möglicher Emulatoren auf Betriebssystem- und Browserebene erstellt und deren Einsatzmöglichkeiten als Smartphoneimitatoren werden bewertet. Danach sollen die über Javascript verfügbaren Eigenschaften, Objekte und Funktionen von Browsern auf Android-Smartphones und Emulatoren, sowie von Browseremulatoren gesammelt und auf eindeutige Unterschiede geprüft werden. Die daraus resultierenden Differenzen sollen anschließend aufgezeigt und zur Täuschung der webbasierten Identifizierung durch Malware verwendet werden.

In Summe sollen der Crawler und seine gesammelten Daten dazu beitragen, Malvertising-Kampagnen und Mobile-Malware zu bekämpfen, um User vor schädlichen Webinhalten zu schützen.

1.3 Methodisches Vorgehen

Das methodische Vorgehen ist unterteilt in ***Literaturrecherche***, ***Evaluierung*** und ***Proof-of-Concept Implementierung*** des Crawlers.

Die Literaturrecherche gliedert sich in folgende Unterpunkte:

- Literaturrecherche zum Thema Drive-By-Exploits, Malvertising und Mobile-Malware sowie deren webbasierte Erkennung
- Analyse der Methoden für die Erkennung von Drive-By-Exploits für Desktop-PCs und Überprüfung ihrer Verwendbarkeit für mobile Betriebssysteme
- Sammeln von Informationen über Remote-Exploits und Pagehijacking für Android
- Untersuchung von bestehenden Techniken zur webbasierten Identifizierung von Emulatoren

Die gesammelten Informationen sollen einen Überblick über den aktuellen Stand der Technik vermitteln sowie das Design und die Implementierung des Crawlers maßgeblich beeinflussen.

Die Evaluierung enthält

- die Auswahl und Detailanalyse einzelner Remote-Exploits sowie die Feststellung ihrer automatischen Erkennbarkeit.
- das Erstellen einer Applikation, welche via Javascript zur Verfügung gestellte Eigenschaften, Objekte und Funktionen von Androidgeräten und Emulatoren ausliest.

Die Auswahl und Detailanalyse der Android-Sicherheitslücken soll auf den gewonnen Informationen der Literaturrecherche basieren und die Grundlage für die Implementierung des Webcrawlers und seiner einzelnen Module bilden. Darüber hinaus sollen bestehende Techniken zur Erkennung von Emulatoren in die Implementierung der Applikation einfließen, sodass webbasierte Differenzen von Smartphones und Emulatoren erkennbar werden.

Die Proof-of-Concept Implementierung des Crawlers gliedert sich in

- das Entwerfen eines Crawler-Frameworks zur automatischen Erkennung von Drive-By-Downloads

- die Implementierung der einzelnen Framework-Module zur Erkennung von vorher analysierten Angriffen
- das Untersuchen der ermittelten Daten auf Differenzen zwischen Smartphones und Emulatoren
- die Entwicklung einer Smartphone-Imitation, damit der Crawler unerkannt bleibt

Um die fehlerfreie Arbeitsweise des Crawlers zu garantieren, sollen die einzelnen Framework-Module anhand von veröffentlichten Proof-of-Concept-Exploits der zugehörigen Sicherheitslücke getestet werden. Stehen zu einer Schwachstelle keine öffentlichen Testdateien zur Verfügung, werden eigene Proof-of-Concepts - ohne eigentlichem Schadcode - zum Auslösen des Software-Fehlers erzeugt.

1.4 Forschungsfragen

In dieser Arbeit sollen einige Forschungsfragen beantwortet werden. Insbesondere wie schädliche Mobile-Werbung automatisiert erkannt werden kann. Im Detail sollen hier folgende Unterfragen beantwortet werden:

- Welche Remote-Exploits für Android sind bekannt?
- Wie können Remote-Exploits für Android automatisch erkannt werden?
- Welche Android-Emulatoren gibt es und welche Funktionalitäten können damit emuliert werden?
- Wie können Androidemulatoren durch Webskripte erkannt werden?
- Welche Pagehijacking-Attacken gibt es für Android?

Die Herausforderungen bei der Beantwortung dieser Fragen liegen in der Thematik der mobilen Endgeräte, der Analyse der Sicherheitslücken und der daraus resultierenden Erkennung etwaiger Angriffe. Es existieren zwar einige wissenschaftliche Forschungen bezüglich der Erkennung von Emulatoren, Werbung und Webangriffen für Desktop-PCs, allerdings bedeutend weniger für den mobilen Bereich. Damit die Forschungsfragen ausführlich und korrekt beantwortet werden können, müssen eigene Tests erstellt und durchgeführt werden, um auf Bezug der webbasierten Emulatorenerkennung eindeutige Ergebnisse zu liefern.

KAPITEL 2

Hintergrund

Dieses Kapitel soll Hintergründe zu Malware, Angriffsvektoren und Emulatoren erläutern. Dazu wird Hintergrundwissen aufgebaut, welches dem Leser dabei helfen soll, die Materie dieser Arbeit besser zu verstehen. Begonnen wird mit einem allgemeinen Überblick über die verschiedenen Arten von Malware und deren Charakterisierung. Anschließend wird auf die Anatomie von Malwareattacken und deren Angriffsvektoren mithilfe einiger Beispiele eingegangen. Zum Schluss erfolgt eine kurze Einleitung in Emulatoren und deren grobe Einordnung in die Kategorien Betriebssystem- und Webbrowseremulator.

2.1 Malware

Malware ist ein Überbegriff für Computerprogramme die unerwünschte oder schädliche Funktionen ausführen.

Unter anderem, werden folgende Arten unterschieden[5]:

- Virus
- Wurm
- Trojanisches Pferd
- Backdoor
- Spyware
- Ransomware

Unabhängig vom Betriessystem, egal ob Desktop oder Mobilsystem, können diese Schadprogramme überall vorkommen. Allerdings werden bestimmte Systeme von Malware-Entwicklern aufgrund ihrer großen Verbreitung präferiert.

Malware besteht meistens aus einem Angriffsvektor und einem Payload. Der Angriffsvektor sorgt dafür, dass sich die Malware verbreitet, während der Payload andere Malware oder Schadcode beinhaltet. Eine Abgrenzung von Angriffsvektoren und Payloads erweist sich in der Regel als schwierig, da beide Malware-Parts Bestandteile des jeweils anderen enthalten können. Im Allgemeinen nutzt der Angriffsvektor Schwachstellen oder Programme aus, um sich zu verbreiten. Der Payload wiederum verrichtet die Arbeit, welche auf einem infizierten System stattfinden soll. Dies kann unter anderem das Löschen von Dateien, eine Rekonfiguration von Sicherheitssoftware wie Antivirensoftware oder Firewalls, das Ausspionieren des Benutzers oder das Fernsteuern von Computern verursachen.

2.1.1 Virus

Ein Virus ist ein Schadprogramm, das sich selbstständig auf unterschiedliche Arten weiterverbreitet. Unter anderem erfolgt eine Ausbreitung durch Veränderung von Programmen oder Dokumenten, in die der eigene Code injiziert wird. Die meisten Viren existieren innerhalb ausführbarer Wirtprogramme. Das heißt sie können physisch auf einem PC vorhanden sein, ohne dabei das System zu infizieren, da diese Art von Malware immer eine Benutzerinteraktion benötigt, um aktiv zu werden. Im Detail bedeutet das die Ausführung eines infizierten Programms oder das Teilen von infizierten Dateien und Applikationen mit Freunden und Bekannten. Neben dem selbständigen Reproduzieren führen Viren meistens anderen Code aus, der dem betroffenem System schaden kann.

Ein prominentes Beispiel für eine weltweite Virenattacke ist CIH[6] aus dem Jahr 1998. CIH überschrieb die komplette Festplatte des befallenen Systems, wodurch das Betriebssystem nicht mehr gestartet werden konnte. Weiters versuchte der Virus das BIOS zu überschreiben und dadurch den PC unbrauchbar zu machen. Schätzungen zufolge wurde in Korea durch diesen Angriff ein Schaden von rund 250 Millionen Dollar verursacht.

2.1.2 Wurm

Ein Computerwurm ähnelt einem Computervirus, allerdings ist dieser nicht auf eine Benutzerinteraktion angewiesen. Würmer verbreiten sich selbstständig unter anderem über Netzwerke, wodurch eine hohe Zahl an möglichen Neuinfektionen gegeben ist. Die Verbreitung eines Wurms kann durch das Versenden von E-Mails an alle Kontakte, mit sich selbst als Anhang, oder durch das Versenden von Dateien in Messengern, wie beispielsweise ,Skype, Facebook-Messenger oder Whatsapp, erfolgen. Allerdings können auch Sicherheitslücken von PCs ausgenutzt werden, wie der Sasser.A-Wurm[7] beweist. Dieses Schadprogramm nutzte eine Bufferoverflow-Schwachstelle von Microsoft Windows aus, um seine Opfer zu infizieren.

2.1.3 Trojanisches Pferd

Ein trojanisches Pferd, auch Trojaner genannt, ist ein oft unscheinbares Programm, das sich als nützliche Applikaton tarnt, dabei allerdings schädlichen Code beinhaltet.

Trojanische Pferde verbreiten sich nicht selbst, vielmehr werben sie mit der Nützlichkeit ihres Wirtprogramms, um User dazu zu bewegen, sie zu installieren.

Im Gegensatz zu Viren und Würmern repliziert sich ein Trojanisches Pferd nicht, wodurch keine Neuinfektionen stattfinden.

Die Auswirkungen von Trojanern können variieren. So haben einige nur lästige Angewohnheiten, wie zum Beispiel das Verändern von Desktop-Oberflächen, während andere, sogenannte *Dropper*, Malware wie Backdoors oder Spyware installieren.

Der Infektionsweg ist vielfältig, da Trojaner unter anderem über Dateien im Internet oder durch Drive-By-Downloads verteilt werden können.

2.1.4 Backdoor

Als Backdoor[8] bezeichnet man ein eigenständiges Programm oder Teil einer Applikation, das Fernzugriff auf den Computer oder einzelne Bereiche ermöglicht, der aber nicht zwangsweise schädlich sein muss. Beispielsweise können auch Universalpasswörter oder Wartungszugänge als Backdoors im weiteren Sinne klassifiziert werden. Natürlich können diese als harmlos erscheinenden Zugänge auch missbraucht werden, um Schadprogramme zu installieren oder das Gerät fernzusteuern. Die Auswirkungen von Backdoors sind im Grunde immer die Gleichen: Eine externe Person, Unternehmen oder Regierung erhält unerlaubten Zugriff auf das infizierte System.

Backdoors können bei der Produktion von Standardsoftware oder Standardkomponenten als Wartungszugang eingebaut, oder durch Malware installiert oder eingerichtet werden. Computer mit Backdoors werden oft als Spamverteiler oder für Denial-of-Service Angriffen missbraucht.

Ein bekanntes Beispiel aus dem Jahr 2014 sind die Hintertüren, die die NSA in Cisco-Produkten hinterlassen hat. Laut Glenn Greenwald[9] wurden Cisco-Produkte, die in andere Länder exportiert wurden, ohne das Wissen von Cisco abgefangen, um darauf Backdoors einzubauen und danach an den Empfänger weiterzuleiten.

2.1.5 Spyware

Spyware[10] ist eine Software, die Daten über die Benutzer eines infizierten Systems, Unternehmen oder Regierungen sammelt. Sie wird über andere Software mit-installiert und durch Drive-by-Downloads oder andere Malware verteilt. Wird das Programm von einer externen Person betrieben, werden die gesammelten Daten an Werbetreibende oder andere interessierte Personen verkauft.

Software, die Daten sammelt, wird nur dann als Spyware bezeichnet, wenn sie ohne das Wissens des Ausspionierten agiert. Ist der User des PCs oder Mobilgeräts über die datensammelnde Software aufgeklärt und wurde diese mit seinem Zustimmen installiert, handelt es sich nicht um Spyware. Allerdings erweist sich die Abgrenzung von Spyware

und Nicht-Spyware als schwierig. So neigen Programme von Werbeanbietern wie AdSense, auch AdWare genannt dazu, Nutzerverhalten und Daten ohne das Wissen des Benutzers zu sammeln. Da sich Marketing-Unternehmen jedoch weigern ihre Software als Spyware zu deklarieren, wird von einigen Anti-Viren-Herstellern die Bezeichnung *potentiell unerwünschtes Programm* benutzt.

Jüngster Spyware-Ableger ist die iOS-Software Pegasus[11]. Diese Spyware wird durch das Ausnutzen einiger Sicherheitslücken in iOS installiert. Aufgrund der Tiefe der Sicherheitslücken wird das komplette Sicherheitskonzept von Apple Smartphones ausgehebelt und das Programm erhält Zugriff auf GPS-Positionsdaten, Telefongespräche und vieles mehr. Pegasus wird von einem Unternehmen entwickelt, welches eng mit Militärs und Regierungen zusammenarbeitet.

2.1.6 Ransomware

Ransomware[12] versucht durch Erpressung der Benutzer, Geld zu lukrieren. Hierbei werden verschiedene Techniken angewandt, um den Nutzer zu überzeugen, das *Lösegeld* zu überweisen. Beispielsweise wird der User aus seinem PC oder Mobilgerät ausgesperrt und danach angeboten, das System gegen eine oft geringe Zahlung, im Durchschnitt von 270 €[13], wieder zu entsperren. Teilweise werden auch Strafverfolgungsbehörden oder ähnliches imitiert, um den Nutzer einzuschüchtern und zum Bezahlen der Forderung zu bewegen.

Eine Technik, die in letzter Zeit populärer wurde, um mehr Geld zu ergaunern, ist das Verschlüsseln der Daten von betroffenen Usern. Dazu wird ein einmaliger Schlüssel berechnet, durch welchen die Userdaten unnützbar gemacht werden. Dieser Key wird dann über eine gesicherte Public-Private Key-Verbindung zu einem *Command and Control Server* gesendet, wodurch der Malware-Entwickler alle notwendigen Informationen erhält, um das befallene System zu identifizieren und zu entschlüsseln. Danach wird das Opfer per Nachricht darüber informiert, dass seine Dokumente, Bilder und Videos unzugänglich gemacht wurden und zur Wiederherstellung besagter Dateien eine Kontaktaufnahme über eine spezielle E-Mail-Adresse erfolgen muss. Kontaktiert der Nutzer diese Adresse, wird ihm die oben erwähnte *Lösegeldforderung* mitgeteilt und die Daten entschlüsselt, sobald den Anweisungen folge geleistet wurde.

Ein neuer Ransomewareableger für Mobilgeräte ist Dogspectus[14], welcher Daten verschlüsselt und die Identität einer Strafvollzugsbehörde vortäuscht.

In Bild 2.1 ist der Screenshot eines Geräts, welches mit Dogspectus infiziert wurde, abgebildet.

2.2 Anatomie von Malwareattacken

Ein Angriff auf ein Computersystem besteht grundsätzlich aus unterschiedlichen Phasen, welche laut John Zorabedian et al.[15] grob in fünf Schritte gegliedert werden können:

Abbildung 2.1: Infiziertes System durch Dogspectus[14]

- Der *Einstiegspunkt* kann eine infizierte E-Mail oder bei Webattacken der Besuch einer infizierten Seite sein.
- Bei der *Verteilung* wird der betroffene Benutzer identifiziert und abhängig von seinem Betriebssystem, dessen Version und der darin installierten Komponenten auf eine speziell präparierte Seite weitergeleitet.
- Danach wird das Zielsystem durch ein Exploit-Kit auf mögliche Schwachstellen überprüft.
- Weist das attackierte Gerät oder installierte Komponenten bestimmte Sicherheitslücken auf, wird es durch einen entsprechenden Exploit angegriffen und mit einer Malware infiziert.
- Anschließend verrichtet die Malware ihre Arbeit und kommuniziert mit ihrem *Command & Control Server* oder attackiert weitere Systeme.

Diese Vorgehensweise ist bei den meisten Angriffen ähnlich, wobei der Einstiegspunkt oder die Verteilung, abhängig von der Attacke variieren können. Bei manchen Würmern wie dem Sasser.A [7] wurden beispielsweise alle Computer im Internet auf eine einzelne Sicherheitslücke überprüft, wodurch die ersten beiden Phasen entfallen sind, welche den Angriffsvector bilden.

Der Angriffsvektor bezeichnet also den Weg, den ein Angreifer wählt, um sein Ziel, nämlich das Ausführen von Schadprogrammen oder Einrichten von Hintertüren, zu erreichen. Im Bereich der Websecurity gehören Social-Engineering und Drive-By-Downloads zu den

prominenteren Vertretern von Angriffsvektoren. Malvertising wiederum verbindet einen oder beide Ansätze mit dem Missbrauch von Werbenetzwerken, um die Wahrscheinlichkeit einer Infektion zu erhöhen.

2.2.1 Social Engineering

Laut Radha Gulati et al.[16] versteht man unter Social Engineering versteht die Kunst des Täuschens und Manipulierens von Benutzern, um diese zur Herausgabe von vertraulichen Daten, wie zum Beispiel Passwörtern, oder zur Installation von Malware oder ähnlichen Schadprogrammen zu bewegen.

Social Engineering Attacken können im Allgemeinen in zwei Typen gegliedert werden, technisches und menschliches Social Engineering.

Menschliches Social Engineering enthält alle Arten von Täuschungen des Opfers durch eine reale Person. Bei dieser Methode imitiert der Angreifer eine vertrauenswürdige Instanz, um Zugangsdaten, Passwörter oder andere sensible Informationen zu erhalten. Beispielsweise könnte sich ein Angreifer als IT-Administrator eines Unternehmens ausgeben, um die Passwörter der Mitarbeiter zu erfahren und dadurch Zugang zum Firmennetzwerk zu erlangen oder der Anruf eines vermeintlichen Bankberaters veranlasst ein gutgläubiges Opfer zur Bekanntgabe seiner Kontodaten.

Technisches Social Engineering hingegen bedient sich an technischen Tricks, um User zur erneuten Eingabe von Zugangsdaten oder zum Installieren einer schädlichen Software zu verleiten. Dabei können Login-Seiten wie von Facebook, Google oder Amazon optisch imitiert werden, um Nutzer zum Log-In aufzufordern. Gibt das Opfer Username und Passwort ein, werden diese Informationen an den Angreifer geschickt, welcher dadurch Zugriff auf das Konto und alle darin gespeicherten Inhalte erhält. Ein anderes Beispiel für technisches Social Engineering wäre ein Browser-Popup, welches die Infizierung des Systems durch einen oder mehrere Viren anzeigt, und zur Installation einer schädlichen Software rät, um den Virenbefall zu entfernen.

2.2.2 Drive-By-Downloads

Bei Drive-By-Downloads handelt es sich um Angriffsvektoren, bei denen der eigentliche Angriff auf das Zielsystem vom User unentdeckt bleibt. Dabei ist es ausreichend eine schädliche oder kompromittierte Webseite zu besuchen, um Opfer dieser Attacke zu werden.

Laut Van Lam Le et al.[17] besteht ein Drive-By-Download-Angriff aus 4 Stufen:

- Schädlicher Code wird auf einer Webseite durch den Angreifer platziert
- Das Opfer besucht die manipulierte Webseite, welche dann durch den Browser heruntergeladen wird
- Client-Browser-, System- oder Plugin-Sicherheitslücken werden ausgenutzt

- Malware oder anderer schädlicher Code wird platziert und ausgeführt

Platzieren des Codes

In der ersten Phase unterscheidet man zwei Möglichkeiten, wie schädlicher Code platziert werden kann. Entweder erstellt der Angreifer seine eigene Webapplikation mit bösartigem Inhalt oder es wird eine bestehende, populäre Seite einer dritten Person dazu missbraucht. Beide Typen haben Vor- und Nachteile für den Angreifer. So ist zum Erstellen einer eigenen Webseite zur Platzierung eines Drive-By-Download-Codes nur geringer Aufwand nötig, jedoch ist eine weite Verbreitung des schädlichen Codes nicht garantiert, da die User aktiv auf die manipulierte Webseite gelockt werden müssen. Soll der Code auf einer häufig frequentierten, bereits bestehenden Webseite platziert werden, muss diese erst durch komplizierte Attacken infiltriert werden, was nicht immer gelingt. Hat der Angreifer jedoch Erfolg und kann seinen Code in die Drittseite einschleusen, ist mit einer höheren Verbreitungsrate seines Drive-By-Downloads zu rechnen.

Besuch der kompromittierten Webseite

Die nächste Phase wird durch den Besuch des Users auf der manipulierten Webseite eingeleitet. Dies kann unbewusst, durch Weiterleitung einer anderen Applikation, oder aktiv, durch den Besuch einer veränderten, populären Seite, erfolgen, wodurch die Webseite über den Browser heruntergeladen wird und die Client-Skripte ausgeführt werden. Eine Reihe von Überprüfungen zur Bestimmung des Betriebssystems, des Browsers und installierten Plugins finden statt und werden oft durch eine Kette von Weiterleitungen auf verschiedene Seiten begleitet. Skript 2.1 zeigt eine browserabhängige Auslieferung von schädlichem Inhalt[17].

Algorithm 2.1: Brwoserabhängige Verteilung von schädlichem Inhalt[17]

```
1  if $browser == 1 then
2  |  ... if $config['spl2'] ==' on'&&$vers[0] == 6||||$vers[0] == 7) then
3  |  |  include ëxploit/x12.php";
4  |  end
5  |  if $config['spl3'] ==' on'&&$os ← 7&&$os! = 3 then
6  |  |  include ëxploit/x3.php";
7  |  end
8  |  ...
9  end
10 if $browser == 2 then
11 |  if $config['spl6'] ==' on'&&$os == 2&&$vers[0] == 7 then
12 |  |  include ëxploit/x6.php";
13 |  end
14 |  ...
15 end
```

Wird während des Besuchs der manipulierten Webseite festgestellt, dass einzelne Komponenten der installierten Software angreifbar sind, wird der Client-Browser auf die Landing-Page, auf welcher der eigentliche Angriff stattfindet, weitergeleitet. Weist das Opfer keine nutzbaren Sicherheitslücken auf oder entspricht nicht den Anforderungen des Malwareschreibers, wird der Besucher auf eine harmlose Werbeseite gelotst, um den Angriffscode vor möglichen Sicherheitsforschern zu verstecken.

Ausnützen der Sicherheitslücke

In dieser Phase nutzt der Angreifer aktiv ein oder mehrere Schwachstellen des Zielsystems aus. Sie können einzeln oder in einer Reihe von aufeinander folgenden Attacken verwendet werden, um in das System einzudringen. Auch hier kann eine erneute Überprüfung auf spezielle Komponenten, Geräte oder Versionen stattfinden, um auf unterschiedliche Systemkonfigurationen reagieren zu können. Ist der Angriff erfolgreich, wird Code injiziert und ausgeführt und damit der nächste Schritt eingeleitet.

Ausführung von schädlichem Code oder Malware

Konnten alle vorherigen Schritte erfolgreich ausgeführt werden, werden nun, in der finalen Phase, schädlicher Code oder böswillige Programme installiert. Hierbei ist es abhängig von der Intention des Angreifers, welche Art von Malware gestartet wird. Dieser Schritt stellt das Ende eines ausgeklügelten Angriffsvektor dar, der ohne Userinteraktion und meist unerkannt innerhalb von Sekunden erfolgreich ein Fremdsystem kompromittiert.

2.2.3 Advertising und Malvertising

Während man unter Advertising das Schalten von Werbung auf Internetseiten versteht, wird bei Malvertising dieser eigentlich ungefährliche Inhalt durch Malware ausgetauscht.

Advertising: Werbung im Internet

Laut Zhou Li et al.[18] gibt es zusätzlich zu Nebendarstellern wie *Trackern*, drei Hauptakteure bei online-Werbung: den *Veröffentlicher*, den *Inserent* und den *Konsument*. Werbenetzwerke spielen bei Malvertising eine entscheidende Rolle, da sie die Gruppe der Veröffentlicher mit der Gruppe der Inserenten indirekt verbinden. Einige Netzwerke verkaufen oder vermieten ihre Werbeflächen an Drittbetreiber, die eigene Kunden haben können. Daher sind Werbenetzwerkbetreiber nicht ausschließlich als Veröffentlicher tätig, sondern können auch als Inserenten agieren.

Der *Veröffentlicher* platziert Code, meist HTML mit Javascript, auf seiner Seite, welcher dynamisch beim Webseitenaufruf passende Werbung von seinem Werbenetzwerk bezieht. Daher ist es für den Publizierer meist ungewiss, welches Inserat geschaltet wird. Er erhält Geld von seinem Ad-Netzwerk für die Anzeige der Werbebanner oder für Klicks auf diese.

Der *Inserent* erstellt Werbung und wird meistens durch kleine oder große Werbenetzwerke mit den Veröffentlichern verknüpft. Dies erlaubt ihm eine große Anzahl an Personen zu

erreichen, ohne mit den Betreibern einer Seite in direkten Kontakt zu treten. Manche Netzwerke bieten dem Inserenten eine Auswahl von Veröffentlichern an, um ihre Zielgruppe bestmöglich zu erreichen.

Der *Konsument* wiederum sieht beim Besuch einer Webseite nur das Resultat der Werbeportale, welche ihm personalisierte Werbung aus mehreren Einzelteilen von verschiedenen Inserenten präsentieren. Im Detail wird beim Besuch einer Webseite eine Anfrage pro Werbefläche an das jeweilige Netzwerk gesendet, welches ihm, abhängig von den gesammelten Daten über den Besucher, die Werbung eines passenden Inserenten liefert. Durch einen Klick auf einen der eingeblendeten Inhalte, wird der Nutzer auf die Seite des entsprechenden Werbenden weitergeleitet.

Malvertising: Malware in Werbung

Unter Malvertising versteht man den Miss- oder Gebrauch von Werbenetzwerken als Verteiler von schädlichen Programmen. Dies geschieht durch Werbeportale, die Inhalte von Inseraten nicht einschränken und Skripte zulassen, von Angreifern attackiert und kompromittiert werden oder deren Werbungsverteilungsketten unterbrochen werden, wodurch nicht kontrollierte Inhalte eingeschleust werden können. Große Werbenetzwerke wie Google AdWords oder Facebook sind allerdings bemüht, Angriffe auf ihre Konsumenten zu verhindern und verbieten daher teilweise oder komplett die Nutzung von Scripten oder ähnlichem Inhalt in geschalteten Anzeigen.

Laut Apostolis Zarras et al. [19] wird zwischen drei unterschiedlichen Typen schädlicher Werbung differenziert:

- Drive-By-Downloads
- Deceptive Downloads
- Link Hijacking

Wie im Abschnitt 2.2.2 beschrieben, wird bei *Drive-By-Downloads* Malware ohne Userinteraktion installiert.

Deceptive Downloads verzichten hingegen auf das Suchen nach Sicherheitslücken und deren Missbrauch und täuschen den Benutzer, indem schädliche Software als unabdingbar und wichtig tituliert wird, ähnlich zu Social-Engineering-Angriffen. Dabei wird dem User vermittelt, dass er den gewollten Inhalt nur durch die Installation der angebotenen Software erhält oder bereits installierte Software durch ein eigens bereitgestelltes Aktualisierungsprogramm auf die neueste Version upgedatet werden muss. Die vermeintlich nützlichen Programme beinhalten allerdings Malware oder andere schädliche Applikationen, die der User nun (un)freiwillig installiert.

Bei *Link Hijacking* wird versucht, dem Opfer eine andere Webseite, als die von ihm geforderte, zu liefern. Werbung wird meistens in einem iframe auf einer Seite eingeblendet.

Da jedoch alle modernen Browser den Zugriff auf Seitenelemente aus iframes aufgrund der Same-Origin-Policy-Regeln verhindern, kann die aktuelle Seite nicht verändert und der schädliche Inhalt nicht eingefügt werden. Um dies zu umgehen, wird durch das Setzen der Browservariable *top.location* noch vor der Anzeige der eigentlichen Seite, der User auf eine willkürliche, durch den Angreifer kontrollierte Webseite weitergeleitet.

2.3 Emulatoren und Imitatoren

Es gibt verschiedene Möglichkeiten ein Androidsystem zu emulieren, wobei die gewählte Technologie abhängig vom gewünschten Emulationsumfang ist. Entwickelt man eine Android-App und will diese auf unterschiedlichen Geräten oder Androidversionen testen, empfehlen sich virtuelle Geräte, welche die komplette Funktionalität des Androidbetriebsystems abbilden. Dadurch kann kostengünstig auf einer Vielzahl von Gerätekonfigurationen getestet werden, ohne ein echtes Gerät für jede gewünschte Software/Hardware-Kombination zu benötigen. Erstellt man allerdings eine Website, welche abhängig von Bildschirmauflösung oder Useragent einen bestimmten Inhalt liefert, ist eine Webbrowserimitation ausreichend, solange keine gerätespezifischen Funktionalitäten erforderlich sind. Möchte man Benutzerverhalten auf Webseiten simulieren oder einen WebCrawler implementieren, bieten sich kopflose Webbrowser als Emulatoren an. Darüber hinaus eignet sich grundsätzlich jeder der genannten Emulatortypen für die beschriebenen Szenarien, wobei der Aufwand der Anwendung stark variieren kann.

2.3.1 Betriebssystem-Emulatoren

Android Virtual Device Manager

Der Android Virtual Device Manager, oder kurz AVD-Manager, ist ein Manager von frei verfügbaren Emulatoren, welche von Google angeboten werden. Er ist Teil des Android SDK und bietet unterschiedliche vordefinierte Gerätekonfigurationen an, welche das komplette Betriebssystem von Android emulieren. Weiters können eigene virtuelle Geräte mit Hardware- und Software-Eigenschaften, wie unter anderem Bildschirmauflösung, Arbeitsspeicher oder Androidversion, ausgestattet werden, um jeglichen Anforderungen zu entsprechen. Darüber hinaus unterstützt der AVD-Manager mehrere Architekturen, wodurch Entwickler von nativen Apps auf x86, x86_64, MIPS und ARM testen können.

Genymotion

Bei Genymotion handelt es sich um einen kommerziellen, kostenpflichtigen Anbieter für Androidemulatoren, welcher eine Plattform zur Verwaltung, ähnlich der von Google, anbietet. Vom Funktionsumfang gleicht Genymotion dem AVD-Manager, mit der Einschränkung, dass neu erstellte Geräte immer von bestehenden Gerätekonfigurationen von Genymotion ableiten müssen. Die virtuellen Genymotion-Geräte emulieren das komplette Androidbetriebssystem, allerdings nur mit der Architektur x86.

AWS Device Farm

Die Amazon Web Services Device Farm ist eine kommerzielle und kostenpflichtige Plattform von Amazon, welche im Gegensatz zu Genymotion und AVD-Manager, echte Androidgeräte zum Testen anbietet. Die Geräte befinden sich in der Amazon-Cloud und können von jedem PC aus bedient werden.

2.3.2 Webbrowser-Imitatoren

Webbrowser-Imitatoren bieten die Möglichkeit, eine Webseite mit unterschiedlichen Browsern und unabhängig vom eigenen Gerät zu testen. Dabei können unter anderem Bildschirmauflösung oder Useragent konfiguriert werden, wodurch der eigene Webauftritt für eine Vielzahl an Geräten getestet werden kann.

Riddle Emulator

Riddle Emulator ist ein Chrome-Plugin, welches nach Auswahl eines Geräts, den Bildschirmausschnitt des aktuellen Chrome-Tabs an die Geräteauflösung anpasst. Er befindet sich im Beta-Stadium und unterstützt die Emulation von Webview-Frameworks wie Phonegap oder Apache Cordova, um Webseiten aus Sicht einer Webview-Applikation testen zu können. Zusätzliche Merkmale sind beispielsweise das Verändern des GPS-Standorts, das Simulieren von Beschleunigungssensoren und das Feuern von speziellen Javascript-Events.

User-Agent Switcher

Der User-Agent Switcher ist als Firefox- und Chromeerweiterung erhältlich und lässt per Knopfdruck einen Wechsel des Useragent-Strings zu. Neben einigen vorgefertigten Konfigurationen, können auch eigene Useragents definiert werden.

Chrome-Browser

Der Chrome-Browser bietet die Möglichkeit, mithilfe der mitgelieferten Entwicklerwerkzeuge, in den Gerätemodus eines Smartphones zu wechseln. Dadurch wird der Bildausschnitt des aktuellen Chrome-Tabs abhängig vom gewählten Gerät verändert. Zusätzlich wird der passende Useragent-String gesetzt und festgelegt, ob Berührungs-Events, wie bei richtigen Smartphones, gefeuert werden. Neben einer großen Auswahl an vordefinierten Geräten, können auch eigene Gerätekonfigurationen hinzugefügt werden.

2.3.3 Kopflose Browser

Kopflose Browser wie PhantomJS oder SlimerJS eignen sich für die Simulation von Benutzerverhalten oder Arbeitsabläufen auf Webseiten. Im Gegensatz zu herkömmlichen Browsern besitzen sie kein Userinterface, um geladene Seiten zu betrachten, bilden allerdings einen Teil der Funktionalität einer Browserengine, wie Javascript-Interpretation, Nachladen von Webseitenressourcen, DOM-Verarbeitung oder CSS-Selektoren ab.

PhantomJS

PhantomJS ist ein kopfloser Browser, der zum Anzeigen und Verarbeiten von Webseiten das QtWebkit-Framework, welches ein Wrapper um die Webkit-Engine von Apple ist, verwendet. Er ist komplett über Javascript steuerbar und bietet neben einigen Webstandards auch das eventabhängige Ausführen von Javascript im Kontext von Webseiten oder Teilen von Webseiten an. Dazu wird in einen Webseitenkontext eine Javascript-Variable, *callPhantom*, eingefügt, welche die Interaktion und Steuerung von PhantomJS zulässt.

SlimerJS

SlimerJS ist momentan noch kein echter kopfloser Browser, da es in der aktuellen Version 0.10.2 eine grafische Oberfläche zum Arbeiten benötigt. SlimerJS arbeitet ähnlich zu PhantomJS mit dem Unterschied, dass es auf der Browserengine Gecko von Mozilla Firefox aufbaut und deshalb den gesamten Funktionsumfang davon anbietet. Wie auch sein Verwandter PhantomJS, wird es durch Javascript gesteuert und bietet eine reibungslose Ausführung von PhantomJS-Skripten, um einen nahtlosen Wechsel zu SlimerJS zu ermöglichen.

KAPITEL 3

Stand der Technik

Die Thematik von Drive-By-Downloads, Malvertising und Emulatorenerkennung wurde bereits in einigen wissenschaftlichen Artikeln analysiert, allerdings mit Fokus auf Mechanismen oder Eigenschaften, welche nicht in dieser Arbeit vorkommen.

Fingerprinting und Erkennung von Emulatoren

Gunes Acar und sein Team [20] analysierten das Fingerprinten von Benutzern durch Webseiten und Werbenetzwerke sowie dadurch entstehende Probleme bezüglich Privatsphäre und Sicherheit. Des Weiteren entwickelten die Autoren das Framework FPDetective, welches unter anderem das Identifizieren von Benutzern anhand installierter Schriftarten erkennt. Die Funktionsweise der webbasierten Erstellung von Gerätefingerabdrücken wurde zusätzlich von Nick Nikiforakis et al. [21] mithilfe des Quelltextes von drei führenden Browserfingerprintern untersucht, um aktuelle Techniken der Benutzeridentifikation aufzuzeigen. Darüber hinaus wurden Mechanismen zur Umgehung von HTTP-Proxys und zur Installation von ausspähenden Browser-Plugins beschrieben, welche von Webseiten, zur Feststellung der richtigen IP-Adresse eines Benutzers, verwendet werden. Hristo Bojinov et al.[22] hingegen präsentierten zwei Algorithmen zur Hardwareidentifikation von Smartphones mithilfe von Gerätesensoren. Während die erste Implementierung den Fingerabdruck aus der Amplitude und der Frequenzverzerrung der aufgezeichneten Signale des Gerätelautsprechers erzeugt, verwendet die zweite Implementierung die Daten des Beschleunigungssensors, um einen Hardwarefingerabdruck zu erstellen. Im Gegensatz zu den webbasierten Identifikationsmerkmalen dieser Arbeit benötigten Hristo Bojinov et al. eine Android-App mit speziellen Berechtigungen, um Geräte mithilfe des Mikrofons zu kategorisieren. Keaton Mowery et al.[23] präsentierten zwei neue Wege zur Identifizierung von Benutzern und deren Browsern, unabhängig von möglicherweise versteckten oder modifizierten Erkennungsmerkmalen, wie dem Useragent. Während ihr erster Ansatz die Performancesignaturen von Javascript-Engines, welche die Zuordnung von Browserversionen, Betriebssystemen und Prozessorarchitekturen ermöglicht, betraf, wurde beim zweiten

versucht, das Firefox-Plugin NoScript zu missbrauchen, um die Surfgewohnheiten eines Benutzers zu bestimmen. Ähnlich dazu, ist der Ansatz zur browserbasierten Erkennung von virtuellen Maschinen und Prozessoremulatoren, vorgestellt von Grant Ho et al.[24], unter Verwendung von Zeitanalysen hinsichtlich Standardbrowserfunktionen. Die Autoren maßen die Zeitspanne eines Browsers, die er zur Ausführung simpler Grundfunktionen, I/O-Operationen oder Anzeigen und Berechnen von komplexen Grafiken benötigt. Die so gesammelten Daten ermöglichten Ho et al. anschließend Emulatoren anhand statistischer Berechnungen zu erkennen und Techniken zur Verteidigung gegen entsprechende Angriffe zu entwickeln. Im Gegensatz zu den Methoden von [23] und [24], basiert die Geräteidentifizierung dieser Arbeit auf der Existenz von Javascript-Objekten und -Events, weshalb sie keinen statistischen Schwankungen oder Performance-Veränderungen durch aktive Benutzerprogramme unterliegt. Großteils befasst sich Fingerprinting mit der Identifizierung von Desktop-PCs, während gleichzeitig die Erkennung von Emulatoren häufig auf nicht-webbasierte Techniken zurückgreift. In Folge dessen ist die webbasierte Charakterisierung mobiler Endgeräte von [22], [23] und [24] für diese Arbeit maßgeblich. Darüber hinaus behandeln bestehende Arbeiten das Thema Fingerprinting häufig im Kontext von Privatsphäre, während der Fokus dieser Arbeit das Thema vielmehr im Kontext von Sicherheit betrachtet.

Erkennung von Drive-by-Downloads

Van Lam Le et al. [17] untersuchten den Aufbau und die Struktur eines Drive-by-Downloads und unterteilten diese Faktoren in vier Stufen, um diese Informationen in die Entwicklung eines Frameworks, zur Beschreibung von Funktionen zur Erkennung entsprechender Angriffe, einfließen zu lassen. Anschließend wurden aktuelle Möglichkeiten der Erkennung von Drive-by-Angriffen hinsichtlich Performance und Verwendbarkeit bewertet. Passend dazu ergab die Untersuchung von Moheeb Abu Rajab et al.[25] über eine Zeitspanne von 4 Jahren, dass die Erkennungssysteme, *Virtual Machine client honeypots*, *Browser Emulator client honeypots*, *Classification base on domain reputation* und *Anti-Virus engines*, für webbasierte Malware, isoliert betrachtet, nicht sehr effektiv sind. Terry Nelms et al. [26] erforschten indes den Ursprung von webbasierten Attacken, durch die Analyse der von Benutzern besuchten Webseiten und daraus resultierenden Pfaden. Dafür entwickelten die Autoren das System WebWitness, welches automatisch den Pfad einer Webinfektion zurückverfolgt. Marco Cova et al.[27] hingegen, präsentierten einen neuartigen Ansatz zur Erkennung und Analyse von böswilligem Javascript-Code, welcher unter anderem zum Starten von Drive-By-Downloads verwendet wird. Dazu wurde ein System entwickelt, welches Javascript emuliert und verdächtige Aktivitäten erkennt. Im Gegensatz zum Crawler in dieser Arbeit, kann die Applikation von Marco Cova et al. beispielsweise einen Angriff auf die Stagefright-Sicherheitslücke mithilfe einer MP4-Datei nicht orten, da ihre Erkennung bei Angriffen ohne schädlichen Javascript-Code fehlschlägt. Ali Ikinci u. a.[28] entwickelten 2008 MonkeySpider, einen Webcrawler, zur automatischen Erkennung und Identifizierung von Malware, welcher aus dem Webcrawler Heritrix und einem eigens entwickelten Modul zur Erkennung von Malware besteht. Das Modul analysiert mithilfe von bestehender Antivirensoftware, wie ClamAV, und

Malwareanalysetools, wie CWSandbox, die gesicherten Daten. Das Team um Yi-Min Wang [29] hingegen entwickelte HoneyMonkey, einen Webcrawler welcher Angriffe auf Schwachstellen in fehlerhaften Browsern mithilfe des Vergleichs von Speicherabbildern und Registry-Einträgen, genommen vor und nach dem Besuch einer Webseite, erkennt. Das Design und die Implementierung von HoneyMonkey bestehen aus mehreren Programmen, welche jeweils unterschiedliche Browser, Browserversionen und Ebenen von Sicherheitspatches widerspiegeln. Ähnlich zu HoneyMonkey, präsentierten Niels Provos et al.[30] ein Netzwerk aus Honey-Clients, welches mit verwundbaren Windows XP Versionen das Internet, auf der Suche nach böswilligen Inhalten, durchforstet. Zur Identifizierung von Drive-by-Downloads wurde eine Kombination aus Laufzeitanalysen und Ergebnissen von Anti-Viren-Software benutzt. Alexander Moshchuk et al.[31] und Konrad Rieck et al. [32] entwickelten die Webproxies SpyProxy und CUJO, um Benutzer vor Malware, verteilt durch böswillige Webseiten, zu schützen. Während SpyProxy den angeforderten Inhalt in einer virtuellen Maschine ausführt, um anschließend seine Aktivitäten anhand von Laufzeitanalysen zu bewerten, versucht CUJO mithilfe von statischen und dynamischen Codeanalysen sowie dem Einsatz von maschinellem Lernen böswilliges Javascript zu erkennen. Während [28] - [32] versuchen Angriffe mithilfe von Laufzeitanalysen, Systemveränderungen, Codeanalysen und Malware-Signaturen zu erkennen, identifiziert der vorgestellte Crawler in dieser Arbeit Drive-by-Downloads bereits beim Auslösen der entsprechenden Sicherheitslücke. In Folge dessen ist seine Erkennungsrate, im Gegensatz zur Erfolgsquote der anderen Crawler, unabhängig von Angriffsskripten, enthaltenem Schadcode oder Laufzeitverhalten von Malware. Darüber hinaus beschäftigen sich bestehende Arbeiten häufig mit der Erkennung von Angriffen auf Desktop-PCs, während der Fokus dieser Arbeit auf Drive-by-Downloads im Kontext von Android liegt.

Android Remote Exploits

Sujal und Sushma Verma [33] beschrieben die 2015 entdeckte Android Sicherheitslücke Stagefright und präsentierten einen Angriff, bei dem sie mithilfe von Heap-Spraying, den eigentlichen Schadcode der Attacke direkt hinter dem überlaufenden Speicherbereich platzierten. Im Gegensatz zu Sujal und Sushma Verma, geht diese Arbeit auf alle Stagefright-Sicherheitslücken im Detail ein und beweist darüber hinaus deren Erkennbarkeit mithilfe eines Webcrawlers. Tongbo Luo et al.[34] präsentierten Angriffe auf das Webview-Interface der Betriebssysteme Android und iOS und skizzierten deren Ursprung in allen Einzelheiten. Dabei beschrieben sie zwei essentielle Schwachstellen der WebView-Infrastruktur bezüglich des Ausbrechens aus der Browsersandbox und des Verletzens der Same-Origin-Policy eines Browsers. Auf Basis der Forschungsergebnisse von Tongbo Luo et al.[34] bezüglich der Android Webview Sicherheitslücken, entwickelten Erik Chin und David Wagner [35] das Tool Bifocals. Bifocals erkennt Android-Apps, welche anfällig für die Webview Schwachstellen *excess authorization* und *file-based cross-zone scripting* sind. Während [34] und [35] die Android Webview Schwachstellen im Detail charakterisieren, wird in dieser Arbeit zusätzlich auf die automatische Erkennung zugehöriger Angriffe eingegangen. Daniel R. Thomas et al. [36] analysierten die Zeitspannen zwischen dem ersten Auftauchen, dem Beheben und dem Veröffentlichen der entsprechenden Sicherheits-

updates von Android API Schwachstellen. Für eine Messung dieser Zeitspanne wurde die Javascript-zu-Java Schwachstelle des Android Webview Interfaces und ihr Missbrauch im Bezug auf Werbenetzwerken betrachtet. Passend zu den Webview Schwachstellen, befassten sich Martin Georgiev et al. [37] mit der Sicherheit und etwaigen Risiken von Hybrid Android Apps auf Basis von Frameworks wie Phongap. Dabei präsentierten sie NoFrak, welches Webinhalten mit fremdem Ursprung den Zugriff auf das lokale Dateisystem und Geräteressourcen verweigert. Das Team um Victor van der Veen[38] entwickelte indes einen auf Rowhammer basierenden Angriff, genannt Drammer, welcher den bekannten Rowhammer-Hardware-Fehler auf einem mobilen Betriebssystem mit ARM-Prozessorarchitektur ausnutzt. Drammer bedient sich dem Mechanismus des Bit flippings, um kritische Rechte auf dem System zu erhalten und stellt somit den ersten Android Root Exploit dar, welcher weder spezielle Benutzerberechtigungen noch eine Softwaresicherheitslücke benötigt. Bestehende Arbeiten bezüglich Android Remote Exploits konzentrieren sich häufig auf die Präsentation von Proof-of-Concept Exploits und die technischen Maßnahmen zur Absicherung des verwundbaren Systems. Diese Arbeit hingegen fokussiert sich auf die Beschreibung generischer Erkennungsmerkmale für Angriffe auf aktuelle Android Sicherheitslücken, um den Urheber eines Drive-by-Downloads zu identifizieren.

Malvertising

Laut Niels Provos et al. [39] steigt das Interesse von Malware-Entwicklern an der webbasierten Verteilung von Malware, weshalb die Autoren dazu vier Mechanismen zur Verbreitung von böswilligen Webinhalten auf hoch frequentierten Webseiten definierten: *web server security*, *user contributed content*, *advertising* und *third-party widgets*. Darüber hinaus präsentierten die Autoren die Entwicklung von Malware über einen längeren Zeitraum und erforschten währenddessen die von Malware-Entwicklern genutzten Techniken, böswilligen Code zu verschleiern. Passend dazu untersuchte das Team von Adrienne Porter Felt [40] im Jahr 2011 den Anreiz hinter mobiler Malware, anhand von 46 im Umlauf befindlichen iOS, Android und Symbian Malware. Darüber hinaus analysierten die Autoren anhand dieser Malware-Proben die Effektivität von aktuellen Erkennungs- und Schutzmechanismen. In einer Studie von Apostiolis Zarras et al. [19] wurden Werbenetzwerke und deren Sicherheitsauswirkungen für Benutzer, mithilfe eines Crawlers sowie dem Honey-Client Wepawet, untersucht. Während Wepawet Webseiten anhand von Laufzeitanalysen von Javascript klassifiziert, werden zum Download angebotene Dateien zur Analyse bei VirusTotal hochgeladen. Im Gegensatz zu Apostiolis Zarras et al. erkennt die in dieser Arbeit präsentierte Applikation auch bislang unbekannte Angriffe auf die Stagefright-Sicherheitslücken. Bezüglich des Themas Malvertising analysierten Zhou Li et al. [18] in einer umfangreichen Studie über drei Monate den Zusammenhang von Werbenetzwerken und böswilligen Webseiteninhalten mithilfe eines Webcrawlers. Dabei wurden Webauftritte und Ad-Netzwerke auf illegale Aktivitäten wie *Scamming*, Klickbetrug oder die Verteilung von Malware überprüft und anschließend in MadTracer, einem automatischen Erkennungssystem für webbasierte Attacken, eingebaut. Zur Erkennung von angebotener Malware und Drive-by-Downloads benutzt MadTracer das Microsoft

Produkt Forefront, welches im Gegensatz zu den vorgestellten Mechanismen dieser Arbeit, Android Malware mit unbekannten Signaturen oder unauffälligem Laufzeitverhalten nicht erkennt. Gegenüber der allgemeinen Klassifizierung von schädlichen Webseiten durch [18], konzentrierten sich Sean Ford et al. [41] auf Werbung basierend auf Flash-Skripten und welche Rolle sie in der Verteilung von Malware spielt. Während die Autoren auf Basis dieser Forschung ein Tool zur automatischen Erkennung von böswilligen, in Werbung eingebetteten Flash-Skripten implementierten, entwickelten Justin Ma et al. [42] einen neuen Ansatz zur automatischen Klassifizierung von URLs, um Benutzer vor böswilligen Webseiten zu schützen. Dabei verwendeten sie statistische Methoden, um eine URL sowohl lexikographisch, als auch auf Basis ihres Hosts zu kategorisieren. Vaibhav Rastogi et al. [43] beschäftigten sich indes mit mobilen Apps und der beinhalteten App-Web-Schranke, welche durch Klicks auf In-App Werbung existiert. Anhand der zugehörigen Angriffsvektoren, entwickelten die Autoren ein neuartiges UI-Framework, welches Werbung und Links in Apps anklickt und damit eine Kette von Weiterleitungen auf möglicherweise böswillige Webseiten auslöst. Terry Nelms et al. [44] präsentierten die erste webbasierte Studie bezüglich Social Engineering Angriffen, welche einen Benutzer zum Download einer schädlichen Datei bewegen. Nach der Überprüfung von über 2000 Beispielen aktueller Social Engineering Kampagnen kamen die Autoren zu dem Schluss, dass ein überwiegender Teil der Social Engineering Angriffe durch *low tier* Werbenetzwerke verbreitet werden. Im Gegensatz zu Terry Nelms et al. analysierten Dae Wook Kim und sein Team [45] vorgetäuschte Anti-Viren-Kampagnen, welche Webseitenbesucher zum Download einer versteckten Malware verleiten sollten. Infolgedessen entwickelten die Autoren DART, welches mithilfe von Bildverarbeitung, Webanalysen und Benutzerbewertungen gefälschte Kampagnen erkennt. Gegenüber der Charakterisierung von Anti-Viren-Kampagnen durch Dae Wook Kim et al., untersuchten Sevtap Duman et al. [46] die automatische Erkennung von falschen Download- und Play-Buttons, welche durch ihr visuelles Auftreten versuchen einen Klick des Benutzers zu provozieren. Um dem entgegen zu wirken, entwickelten die Autoren das Tool TrueClick, welches mithilfe von Bildverarbeitung und maschinellem Lernen böswillige Webseitenelemente erkennt. Bestehende Arbeiten beziehen sich großteils auf Social Engineering Kampagnen sowie die Kategorisierung von Malvertising im Kontext von Desktop-PCs, während sich diese Arbeit auf die Identifizierung von Malvertising-Kampagnen mit Fokus auf mobile Endgeräte konzentriert. Infolgedessen sind die Resultate von [18], [19], [40] und [43] für diese Arbeit von großer Bedeutung.

KAPITEL 4

Evaluierung

Um einen Crawler entwickeln zu können, welcher Drive-by-Downloads erkennt und sie involvierten Webseiten und Advertising-Kampagnen zuordnet, müssen zuerst einige Schwachstellen des Androidbetriebssystems definiert und analysiert werden. Das ermöglicht der Applikation zwar die erfolgreiche Identifizierung von Angriffen, allerdings muss dafür auch sichergestellt sein, dass sie nicht vorzeitig durch einen Fingerprinting- oder anderen Identifizierungsprozesser erkannt wird und somit der Zugriff auf die eigentlichen Malware-Samples verwehrt bleibt.

In den folgenden Kapiteln werden einige Androidsicherheitslücken ausgewählt, welche anschließend im Detail betrachtet und bezüglich automatischer Erkennung beschrieben werden. Nachfolgend wird näher auf das Thema Androidemulatoren eingegangen und in welcher Form sie von einander unterschieden werden können. Da die Erkennung von Emulatoren für den Erfolg des Webcrawlers eine entscheidende Rolle spielt, werden nur Möglichkeiten zur Identifizierung und Differenzierung via Webtechnologien wie Javascript und Flash betrachtet.

4.1 Remote-Exploits für Android

Zu Beginn dieser Arbeit stellte sich die Frage, nach welchen Kriterien Sicherheitslücken gewählt werden sollen. Da die geplante Applikation zur Erkennung von Angriffen auf Basis eines Webcrawlers implementiert werden sollte, wurde das erste Kriterium bereits implizit definiert. In Frage kommende Sicherheitslücken müssten Remote, um genauer zu sein via den Webbrowser oder ähnlichem, angreifbar sein, um einen nichts ahnenden Benutzer beim täglichen Informationsmarathon durch diverse Blogs und News-Seiten infizieren zu können. Das schließt somit Schwachstellen bezüglich offenen oder unzureichend gesicherten Sockets, sowie nur lokal nutzbare aus. Da aus Sicht eines Malware-Entwicklers die Wahrscheinlichkeit für einen erfolgreichen Angriff mit der Anzahl der durch Schwachstelle

betroffenen Geräte-Versionen steigt, fällt das zweite Kriterium auf nicht herstellerspezifische Fehler. Damit sind Sicherheitslücken in Android gemeint, die ausschließlich im Androidkernsystem vorkommen und somit nicht nur einen einzelnen Hersteller wie Samsung, HTC oder LG betreffen. Darüber hinaus sollten die Sicherheitslücken nicht älter als zwei Jahre sein, da sie, trotz der oft schwerfälligen Updatepolitik[47] mancher Hersteller, nach diesem Zeitraum meist behoben sind und ein Angriff somit nur einen geringen Anteil an Geräten erreichen würde. Nachdem DoS und DDoS-Angriffe auf Sicherheitslücken zwar grundsätzlich ebenfalls gefährlich und kostspielig sein können, sie für einen einzelnen Benutzer jedoch recht wenige Auswirkungen haben, müssen die Sicherheitslücken zusätzlich die Ausführung von Schadcode erlauben. Als letztes, einschränkendes Kriterium wird die ursprüngliche Charakteristik eines Drive-By-Downloads, nämlich das Ausnutzen einer Schwachstelle ohne Benutzerinteraktion, gewählt. Aus Sicht der Malware-Entwickler erhöht sich dadurch die Wahrscheinlichkeit einer Infizierung, da die nötige Ausführung einer schädlichen Datei durch einen Benutzer, grundsätzlich die Verbreitungsrate, durch Misstrauen gegenüber einer heruntergeladenen Datei, senkt. Darüber hinaus wird heruntergeladene Malware bei installierter Antivirensoftware, rasch erkannt und isoliert, wodurch eine Infizierung nicht mehr möglich ist. Zu guter Letzt, als nicht ausschließendes Kriterium, werden Sicherheitslücken mit vorhandenem Proof-of-Concept favorisiert, da diese die Analyse, Entwicklung und Tests des geplanten Programms erheblich erleichtern.

Die Suche nach geeigneten Sicherheitslücken gestaltete sich zu Beginn schwierig, da die Fülle der Informationen des Internets und einschlägigen Webseiten teilweise verhindert, einen strukturierten Überblick zu erhalten. Es war naheliegend, dass unter anderem die 2015 unter dem Namen Stagefright weltweit bekannt gewordene Schwachstelle in dieser Arbeit analysiert werden würde. Durch die enorme Medienpräsenz von Stagefright und der lokalen Sicherheitslücke Quadroot[48], war allerdings bei einer Suche nach *Android remote exploits* oder verwandten Suchanfragen an Google, kaum ein Ergebnis ohne die genannten Schwachstellen zu finden. Ich musste daher nach einer anderen Möglichkeit suchen, um passende Schwachstellen zu entdecken, wodurch sich meine Überlegungen in Richtung Android-Bug-Tracker entwickelten. Da Android, wie die meisten großen Open-Source-Projekte, einen öffentlichen Bug-Tracker besitzt, und eine Sicherheitslücke nur ein speziellen Typ von Bug bezeichnet, sollte dieser, in der Hoffnung fündig zu werden, als Hauptquelle für passende Schwachstellen dienen. Aufgrund der Tatsache, dass Security-Bugs unter Android gesondert gemeldet werden und der eigentliche Bug-Report sowie die verwundbare Stelle aus Sicherheitsgründen erst nach Behebung dieser veröffentlicht werden, verlagerte sich der Fokus ein weiteres Mal auf den Android Security Bulletin[49].

Die einzelnen Patches jedes Security Bulletins enthalten Informationen zum ursprünglichen Problem, beispielsweise *Remote Code Execution Vulnerablilty*, die gemeldete CVE-Nummer, der Schweregrad der Schwachstelle sowie die betroffenen Geräte und Androidversionen. Darüber hinaus kann zu jedem behobenen Fehler eine detaillierte Beschreibung und der Quelltext, der die Schwachstelle behebt, eingesehen werden, wodurch bereits Sicherheitslücken gefunden werden, die einem Großteil der vorher definierten Kriterien

entsprechen. Durch die Referenz auf eine CVE-Nummer, kann in weiterer Folge der ursprüngliche Veröffentlicher der Lücke und somit häufig ein Proof-of-Concept oder eine Detailanalyse, mit Informationen über mögliche Angriffsvektoren, gefunden werden.

Anhand dieser Informationen gestaltete sich die Auswahl geeigneter Sicherheitslücken als unkompliziert, da Schwachstellen mit der Bezeichnung *Remote Code Execution* und dem Schweregrad *Critical* ausgewählt wurden. Anschließend wurden mithilfe der weiteren Details einige Sicherheitslücken selektiert, zu denen besonders viele Informationen und Proof-of-Concepts verfügbar waren. Neben den Schwachstellen, die in den nachfolgenden Kapiteln beschrieben werden, existieren noch unzählige weitere, deren Analyse jedoch den Rahmen dieser Arbeit übersteigen würde.

Obwohl die Schwachstelle des Android Webview Interface bereits 4 Jahre alt ist, wurde sie im Zuge dieser Arbeit dennoch analysiert, da sie nicht auf den Browser beschränkt ist, sondern vielmehr auch Apps von Drittanbietern betrifft.

4.1.1 Stagefright

Stagefright ist eine der aktuellsten Android-Sicherheitslücken und besteht aus mehreren einzelnen Schwachstellen. Sie wurde ursprünglich von Josuha Drake entdeckt und auf der Blackhat USA 2015 erstmals präsentiert. Laut Schätzungen[3] sind rund 95% aller Android-Smartphones weltweit, das entspricht circa 950 Millionen Geräten, durch Stagefright angreifbar.

Stagefright ist streng genommen der Name der Android Multimedia Framework Library, welche für die Verarbeitung sämtlicher Audio- und Videodateien zuständig ist. Um Verwirrungen zu vermeiden, werden anschließend die einzelnen Sicherheitslücken als *Stagefright* und das Android Multimedia Framework als *Stagefright Library* betitelt. Die Stagefright Library ist seit der Androidversion 2.3 standardmäßig im Betriebssystem verankert und bis Android 5.1.1 verwundbar. Verwendet wird sie in einem von Android bereitgestellten Service, dem Mediaserver, welcher wiederum von etlichen Standard-Apps, Services und Drittanbietern genutzt wird.

Die Stagefright-Schwachstellen sind bei der Verarbeitung von MP4 Dateien in der Methode *parseChunk* der Klasse *MPEG4Extractor* präsent. *parseChunk* ist für das Lesen von einzelnen MP4-Atomen zuständig, wobei jedes Atom aus seiner Länge, dem Typ und dem eigentlichen Inhalt besteht. Abhängig vom Typ des Atoms, muss der MPEG4Extractor unterschiedliche Aktionen durchführen, um die Atom-Daten zu verarbeitet. Bei der Zerlegung und Verarbeitung der unterschiedlichen MP4-Atom-Typen, weist die Stagefright Library die korrespondierenden Schwachstellen mit den folgenden CVE-Nummern auf:

- CVE-2015-1538
- CVE-2015-1539
- CVE-2015-3824

Tabelle 4.1: Kopf eines MP4-Atoms

Feld	Datentyp	Kommentar
Größe	uint32_t	Größe des ganzen Atoms inklusive des Kopfes
Typ	char	stss, stts, etc.
Erweiterte Größe	uint64_t	existiert nur wenn Größe 1 ist

- CVE-2015-3826
- CVE-2015-3827
- CVE-2015-3828
- CVE-2015-3829

Nach den ersten Security-Patches von Google im August 2015 kam ans Licht, dass die Schwachstelle, beschrieben in CVE-2015-3824, nur teilweise behoben wurde. Der verbleibenden Lücke wurde die neue CVE-Nummer CVE-2015-3864 zugewiesen.

Die verwundbaren Quelltexte der ursprünglich gemeldeten Stagefright-Schwachstellen stimmen teilweise sinngemäß überein, weshalb nicht alle im Detail veranschaulicht werden. Um Angriffe auf sämtliche Stagefright-Lücken automatisch zu erfassen, werden sie zumindest kurz bezüglich ihrer Erkennbarkeit skizziert.

MP4-Dateiformat

Die separaten Atom-Typen und deren Aufgaben in einem Video werden nicht näher ausgeführt, da für die Erkennung von Angriffen auf Stagefright nur der Aufbau der einzelnen Atome von Bedeutung ist. Ein MP4-Atom besteht aus einem Kopf und einem Array seiner Daten, wobei der Kopf wiederum aus der Atom-Größe und dem Atom-Typ besteht. Falls 32-Bit für die Länge der Daten nicht ausreichen, werden die auf den Atom-Typ folgenden acht Bytes für eine erweiterte Größe mit 64-Bit verwendet. Gleichzeitig wird die eigentliche Atom-Größe vor dem Atom-Typ auf den Wert 1 gesetzt, um die Verwendung der erweiterten Größe anzudeuten. Die Atom-Daten können zusätzliche typenspezifische Felder beinhalten, bevor die eigentlichen Werte folgen, welche bei der Beschreibung der verwundbaren Atom-Typen erklärt werden. Der Aufbau des Kopfs ist in Tabelle 4.1 ersichtlich.

In weiterer Folge wird für die ersten 4 Bytes oder die Bytes 8 bis 16 eines Atoms, Atom-Größe oder Größe des Atoms verwendet, wobei die eigentliche Daten-Größe abhängig vom Typ des Atoms ist und somit später in den Daten neu gesetzt sein kann. Atom- und Daten-Größe können vom Angreifer komplett kontrolliert werden, weshalb sie nicht die eigentliche Größe des Atoms oder der Daten widerspiegeln müssen.

CVE-2015-1538

Diese Schwachstelle betrifft die Atom-Typen *STSC*, *CTTS*, *STTS* und *STSS*, bei denen jeweils ein Zahlenüberlauf möglich ist.

STSS-Atom Der verantwortliche Quelltext, welcher beim Verarbeiten eines STSS-Atoms einen Fehler produziert, ist in Skript 4.1 ersichtlich.

Algorithm 4.1: Integer Overflow in SampleTable.cpp in der Funktion setSyncSampleParams bei Version 5.0.0_r2 [50]

```
1  uint32_t *mSyncSamples;
2  uint32_t mNumSyncSamples;
3  ...
4  mNumSyncSamples = U32_AT(&header[4]);
5  uint64_t allocSize = mNumSyncSamples * sizeof uint32_t;
6  if allocSize > SIZE_MAX then
7  |  return ERROR_OUT_OF_RANGE;
8  end
9  mSyncSamples = new uint32_t [mNumSyncSamples];
10 size_t size = mNumSyncSamples * sizeof uint32_t;
11 ...
12 for i = 0 to mNumSyncSamples do
13 |  mSyncSamples[i] = ntohl(mSyncSamples[i]) - 1;
14 end
```

Die ersten 4 Bytes von *header* beinhaltet die Version des Atoms und spezielle Flags, gefolgt von der Anzahl an beinhalteten SyncSamples. Die Anzahl an SyncSamples wird anschließend in *mNumSyncSamples* gespeichert, um später für die Kalkulation des benötigten Speichers zur Verfügung zu stehen. Da ein einzelnes SyncSample einer Größe von vier Bytes entspricht, muss im nächsten Schritt die benötigte Größe des Speichers, welcher schlussendlich sämtliche Samples erhalten soll, berechnet werden. Dazu wird in Zeile 5 eine vorzeichenlose 64-Bit-Zahl, *allocSize*, generiert, welche das Produkt der Anzahl an SyncSamples und der Speichergröße einer vorzeichenlosen 32-Bit-Zahl erhält. In Zeile 6 wird dann die berechnete Größe, allocSize, mit *SIZE_MAX* der maximalen Größe einer Zahl verglichen. Ist der zu allozierende Speicher zu groß, wird ein Fehler geworfen und die Verarbeitung abgebrochen. Ansonsten wird der nötige Speicher reserviert, und die Daten aus dem Atom hinein kopiert.

In Zeile 5 bei der Kalkulation von allocSize kann es durch die gezielte Manipulation von mNumSyncSamples zu einem Zahlenüberlauf kommen. Ist die enthaltene Zahl *0x40000000* oder Größer, ist der Wert in allocSize immer kleiner als SIZE_MAX und die Abfrage in Zeile 6 schlägt fehl. Beinhaltet mNumSyncSamples beispielsweise die Zahl *0x40000002*, ist die *Daten-Größe*, *mNumSyncSamples * sizeof(uint32_t)*, die Zahl 8, da hinsichtlich des Zahlenüberlaufs die Gleichung *0x40000002 * 4 = 0x08* gilt. Da Zeile 9 gleichbedeutend

mit *malloc(mNumSyncSamples * sizeof(uint32_t))* ist, zeigt *mSyncSamples* demnach auf einen allozierten Speicher der Größe *0x08*. In Zeile 12 wird dann über die Anzahl an mNumSyncSamples, welche dem Wert 0x40000002 entspricht, iteriert. Da mSyncSamples nur eine Größe von acht Bytes besitzt, werden die Grenzen des allozierten Speichers nach der zweiten Iteration überschritten und infolgedessen in Speicherbereiche, außerhalb der eigenen Schranken geschrieben.

Automatisch erkennen kann man einen entsprechenden Angriff durch die Überprüfung der Daten-Größe des STSS-Atoms. Ist sie größer oder gleich *0x40000000*, wurde die MP4-Datei vermutlich manipuliert und ein Angriff kann nicht ausgeschlossen werden.

CTTS-Atom Die Lücke beim CTTS-Atom ist ähnlich der im STSS-Atom, weshalb nicht näher auf den Sourcecode eingegangen wird. Zu beachten ist, dass die nötige Größe des Speichers, welcher die einzelnen Einträge erhält, diesmal zusätzlich mit 2 multipliziert wird. Die kalkulierte Zahl zum Allozieren des Speichers entspricht deshalb dem Produkt der *Daten-Größe* und der Zahl 2, weshalb zum Erkennen eines Angriffs die Daten-Größe mit *0x20000000* statt mit *0x40000000* verglichen werden muss.

STTS-Atom Der Fehler beim STTS-Atom ist identisch zu dem beim CTTS-Atom, weshalb erneut nicht näher darauf eingegangen wird. Analog zum CTTS-Atom, muss beim STTS-Atom die Daten-Größe mit *0x20000000* vergleichen um eine modifizierte MP4-Datei zu erkennen.

STSC-Atom Joshua Drake[51] präsentierte im September 2015 einen funktionierenden Exploit, welcher den Zahlenüberlauf in Kombination mit eine Überlauf des Heap-Speichers beim *STSC*-Atom nutzt, um eigenen Code auszuführen. Um einen entsprechenden Angriff zu erkennen, muss das STSC-Atom und der zugehörige Sourcecode genauer analysiert werden. In Skript 4.2 ist der Quelltext, der die Lücke verursacht, ersichtlich.

Algorithm 4.2: Ausschnitt aus der Methode setSampleToChunkParams in der Klasse SampleTable.cpp [50]

```
1 uint32_t mNumSampleToChunkOffsets;
2 ...
3 mNumSampleToChunkOffsets = U32_AT(&header[4]);
4 if data_size < 8 + mNumSampleToChunkOffsets * 12 then
5 |  return ERROR_MALFORMED;
6 end
7 mSampleToChunkEntries = new SampleToChunkEntry
    [mNumSampleToChunkOffsets];
```

Die Variable *header* beinhaltet die Version des STSC-Atoms gefolgt von der Daten-Größe, in diesem Fall die Anzahl an Einträgen. Demnach enthält *mNumSampleToChunkOffsets* die Zahl an Einträgen, die dieses STSC-Atom beinhaltet. Der Zahlenüberlauf passiert in

Zeile 4 und wiederholt sich bei der Instanzierung eines neuen *SampleToChunkEntry*-Arrays in Zeile 7.

Bei der Gleichung *8 + mNumSampleToChunkOffsets * 12* in Zeile 4 liegt der Ursprung der Sicherheitslücke. Wenn mNumSampleToChunkOffsets größer als *0x15555555* ist, kommt es zu einem Zahlenüberlauf, womit das Produkt niedriger sein kann als *data_size*, obwohl mNumSampleToChunkOffsets eigentlich größer wäre. Der Wert in data_size, spiegelt dabei die Größe des Atoms minus acht, jeweils vier Bytes für den Wert der Atom-Größe und für den Atom-Typ, wieder. Der Ausdruck *new SampleToChunkEntry* in Zeile 7 ist ein Synonym für *malloc(mNumSampleToChunkOffsets * sizeof(SampleToChunkEntry))*, welcher erneut zwei 32-Bit Zahlen miteinander multipliziert und infolgedessen einem Überlauf provoziert.

Das allozierte Array, welches durch den Zahlenüberlauf nun zu klein erstellt wurde, wird später als Kopierziel für die Einträge in STSC verwendet. Durch das erneute iterieren über mNumSampleToChunkOffsets, welches weitaus höher ist als die Größe von *mSampleToChunkEntries*, wird über die Grenzen des Puffers geschrieben.

Erkennen kann man einen zugehörigen Angriff, anhand der Daten-Größe des STSC-Atoms. Ist die Anzahl an Samples 0x15555555 oder größer, ergibt die Multiplikation mit 12, dies entspricht *sizeof(SampleToChunkEntry)*, ein Produkt größer als den maximale Integer *0xFFFFFFFF* und eine böswillige Manipulation der MP4-Datei kann nicht ausgeschlossen werden.

CVE-2015-1539

Diese Schwachstelle betrifft das ESDS-Atom bei der Verarbeitung des Elementary-Stream-Descriptors, kurz ESDS, wobei der Ursprung des Fehlers ein Zahlenunterlauf ist. Der Aufbau des ESDS-Atoms bis zum Feld OCR_ES_ID ist in Tabelle 4.2 ersichtlich.

Der Zahlenunterlauf tritt bei der ESDS-Größe in Kombination mit einem oder mehreren der Flags streamDependanceFlag, URL_Flag oder OCRStreamFlag auf. Der fehlerhafte Quelltext ist in Skript 4.3 ersichtlich, wobei *offset* auf die ESDS-ID zeigt und *size* die verbleibende Größe beinhaltet.

Zu Beginn werden zwei Bytes, die ESDS-ID, übersprungen und direkt mit dem Auslesen der Flags gestartet. Anschließend kann es an insgesamt drei unterschiedlichen Passagen zu einem Zahlenunterlauf kommen. Bei der Subtraktion des Subtrahenden, die Zahl 2, von der aktuellen Größe in Zeile 14, kann es zu einem Unterlauf kommen, falls das vorher gelesene streamDependenceFlag gesetzt war. Ist das URL_Flag aktiviert, besteht in Zeile 22 eine weitere Möglichkeit für einen Zahlenunterlauf, vorausgesetzt die zuvor berechnete URL-Länge ist größer als der verbleibende Wert in size. Die letzte problematische Passage ist in Zeile 26 ersichtlich, wobei dafür das OCRstreamFlag des Elementary-Stream-Descriptors gesetzt sein muss.

Um eine MP4-Datei mit manipuliertem Elementary-Stream-Descriptor zu erkennen, müssen mehrere Fälle unterschieden werden, wobei size zu Beginn der Methode mindestens

Algorithm 4.3: Ausschnitt aus der Methode parseESDescriptor in der Klasse ESDS.cpp [50]

```
1  ESDS::parseESDescriptor size_t offset, size_t size
2      if size < 3 then
3          return ERROR_MALFORMED;
4      end
5      offset += 2;
6      size -= 2;
       /* mData beinhaltet die Daten des ESDS-Atoms                */
7      unsigned streamDependenceFlag = mData[offset] & 0x80;
8      unsigned URL_Flag = mData[offset] & 0x40;
9      unsigned OCRstreamFlag = mData[offset] & 0x20;
10     ++offset;
11     –size;
12     if streamDependenceFlag then
13         offset += 2;
14         size -= 2 ;
15     end
16     if URL_Flag then
17         if offset >= size then
18             return ERROR_MALFORMED;
19         end
20         unsigned URLlength = mdata[offset];
21         offset += URLlength + 1;
22         size -= URLlength + 1;
23     end
24     if OCRstreamFlag then
25         offset += 2;
26         size -= 2;
27         ...
28     end
29     if offset >= size then
30         return ERROR_MALFORMED;
31     end
32     ...
```

Tabelle 4.2: Die ersten Bytes des ESDS-Atoms, welche wichtig für den Integer-Underflow sind. [52]

Feld	Größe in Bytes	Kommentar
Atom-Größe	4	
Atom-Typ	4	ësds"
Version	1	
Flags	3	
Tag	1	
ESDS-Größe	Variabel	können mehrere Bytes sein, wobei das letzte Byte das High-Bit gesetzt hat
ES_ID	2	
streamDependenceFlag	1 Bit	mit & 0x80 zu lesen, teilt sich das Byte mit den anderen Flags
URL_Flag	1 Bit	mit & 0x40 zu lesen, teilt sich das Byte mit den anderen Flags
OCRStreamFlag	1 Bit	mit & 0x20 zu lesen, teilt sich das Byte mit den anderen Flags
dependsOn_ES_ID	2	existiert nur wenn streamDependenceFlag gesetzt ist
URLlength	1	existiert nur wenn URL_Flag gesetzt ist
URLstring	1	existiert nur wenn URL_Flag gesetzt ist
OCR_ES_ID	2	existiert nur wenn OCRStreamFlag gesetzt ist

Tabelle 4.3: Kombination der ESDS-Flags mit der jeweiligen Mindesgröße für size um einen Integer-Underflow zu verhindern

streamDependenceFlag	URL_Flag	OCRstreamFlag	Mindesgröße für size
x	o	o	5
x	x	o	5 + URLlength
x	x	x	7 + URLlength
o	x	o	3 + URLlength
o	x	x	5 + URLlength
o	o	x	5
x	o	x	7

drei sein muss. In Tabelle 4.3 sind die einzelnen Kombinationen der drei Flags und die resultierende Mindestgröße für eine fehlerfreie Ausführung von parseESDescriptor angeführt.

Damit eine korrupte MP4-Datei erfolgreich erkannt wird, muss die ESDS-Größe abhängig von den gesetzten Flags mit den Werten *0x05* oder *0x07* verglichen werden. Ist zusätzlich zu einem der anderen Flags das Bit URL_Flag gesetzt, muss zuerst die URL-Länge

kalkuliert und anschließend die ESDS-Größe mit *3 + URLlength*, *5 + URLlength* oder *7 + URLlength* verglichen werden.

CVE-2015-3824

Die Sicherheitslücke CVE-2015-3824 betrifft die Verarbeitung der TX3G-Atome in der Stagefright Library und wurde als Zahlenüberlauf kategorisiert. NorthBit[53] hat Ende 2015 den Stagefright-Exploit *Metaphor* für das TX3G-Atom veröffentlicht, welcher den ASLR-Schutzmechanismus von Android umgeht und infolgedessen auch auf neueren Androidversionen lauffähig ist. Grundsätzlich kann eine MP4-Datei ein oder mehrere TX3G-Atome besitzen, welche die Untertitel des Videos beinhalten. Die Menge an vorhandenen TX3G-Atome ist für den Erfolg eines Angriffs mit anschließender Ausführung des eigenen Codes wesentlich, da eine Zahlenüberlauf zumindest zwei TX3G-Atome voraussetzt. Der allgemeine Ablauf beim Einlesen der TX3G-Atome ist abhängig von der Anzahl an vorher eingelesenen TX3G-Atomen. Ist das aktuelle TX3G-Atom das erste seiner Art, wird ein Speicherbereich in der Größe seiner Daten alloziert und anschließend mit den Daten befüllt. Wurden schon ein oder mehrere TX3G-Atome eingelesen, wird ein Speicherbereich in der Größe der vorherigen TX3G-Atome plus der Größe des aktuellen Atoms alloziert und die Daten in der Reihenfolge des Einlesens der Atome in den Puffer geschrieben. Die entsprechende Stelle im Quelltext ist in Skript 4.4 ersichtlich.

Algorithm 4.4: Ausschnitt aus der Methode parseChunk in der Klasse MPEG4Extractor.cpp [50]

```
1  uint64_t chunk_size;
2  size_t size;
3  const void *data;
4  ...
   /* size und data werden mit den Daten des letzten
      TX3G-Atoms befüllt                                  */
5  ...
6  uint8_t *buffer = new (std::nothrow) uint8_t [size + chunk_size];
7  if buffer == NULL then
8  |  return ERROR_MALFORMED;
9  end
10 if size > 0 then
11 |  memcpy(buffer, data, size);
12 end
13 if (size_t)(mDataSource-> readAt(*offset, buffer + size, chunk_size) <
     chunk_size then
14 |  ...
15 |  return ERROR_IO;
16 end
```

Aufällig ist in Zeile 6 das Allozieren des neuen Speicherbereichs, da es bei der Addition von *size* und *chunk_size* zum Zahlenüberlauf kommen kann. Folglich wird ein unterdimensionierter Speicher reserviert, welcher dann entweder in Zeile 11 oder 13 weit über seine Grenzen hinaus befüllt wird. Zu beachten ist, dass die Summe der Größen einen Zahlenüberlauf sowie das Kopieren der Daten der vorherigen Atome den Speicherüberlauf lediglich dann verursachen, wenn bereits ein oder mehrere TX3G-Atome eingelesen wurden.

Die automatische Erkennung eines entsprechenden Angriffs ist abweichend zu jenen der bisherigen MP4-Atomen. Im Gegensatz zu den restliche Atomen ist ein Vergleich der Größe eines TX3G-Atoms mit einem vorher berechneten Wert unzureichend. Viel mehr muss über alle TX3G-Atome iteriert und deren Größen summiert werden, um den Zahlenüberlauf zu simulieren. Wird beim Iterieren der Atome der Grenzwert von *SIZE_MAX* überschritten, kann von einer modifizierten MP4-Datei ausgegangen werden.

CVE-2015-3826 / CVE-2015-3828

Die beiden Sicherheitslücken *CVE-2015-3826* und *CVE-2015-3828* treten bei der Verarbeitung der Atome TITL, PERF, AUTH, GNRE und ALBM auf, wobei *CVE-2015-3826* ein Überlesen des Speichers und *CVE-2015-3828* einen Zahlenunterlauf mit anschließendem Speicherüberlauf darstellt. Die genannten Kürzel stehen für die Einträge title, performer, author, genre und album einer MP4-Datei und enthalten Text, welcher eventuell zum Überlesen des Puffers führt. In Skript 4.5 ist der entsprechende Quelltext ersichtlich, wobei nur die relevanten Passagen angeführt sind.

Beim Parsen einer der betroffenen Atome, wird zuerst ein Speicher, *buffer*, in der Größe von den zu erwartenden Daten reserviert und anschließend mit den Daten aus *offset* befüllt. Danach wird *metdadataKey* mit dem zugehörigen Schlüssel des jeweilige Atoms befüllt. Der Inhalt der if-Abfrage von Zeile 5 wird nur ausgeführt, wenn *metadataKey* mit einer Zahl größer 0 befüllt ist. Diese Bedingung trifft auf die oben genannten Atome, aber nicht auf YRRC, zu.

In Zeile 19 wird dann *setCString* auf *mFileMetdaData* mit dem aktuellen Key und einem Zeiger auf den Puffer aufgerufen. Gleichzeitig kann es in der Methode *setCString* zum Überlesen des Puffers. Zuerst wird mit *strlen* die Länge des Textes im Puffer bestimmt, welche dann in *setData* die Anzahl der zu kopierenden Zeichen darstellt. Ist der String in *buffer* nun nicht Null-Terminiert, liest *strlen* solange weiter bis das Zeichen *0x00* erreicht wird. Infolgedessen wird zwar ein Speicherbereich mit ausreichender Größe alloziert und beschrieben, allerdings werden über die Grenzen von *buffer* hinaus, unkontrollierte Adressbereiche gelesen.

Mittels einem Vergleich des Bytes an der Stelle Atom-Start + Atom-Größe mit *0x00*, kann ein Angriff auf die Sicherheitslücke *CVE-2015-3826* erkannt werden. Ist der Wert des Bytes an dieser Stelle abweichend zu *0x00*, wurden die 3GPP-Metadaten der MP4-Datei möglicherweise manipuliert und ein Überlesen des Puffers ist möglich.

Algorithm 4.5: Ausschnitt aus der Methoden parse3GPPMetaData der Klasse MPEG4Extractor.cpp und setCString aus MetaData.cpp[50]

```
1  MPEG4Extractor::parse3GPPMetaData off64_t offset, size_t size, int depth
2      ...
3      uint8_t *buffer = new (std::nothrow) uint8_t [size];
       /* buffer wird mit den Daten die in offset stehen befüllt
          */
4      ...
5      if metadataKey then
6          bool isUTF8 = true;
7          char16_t *framedata = NULL;
8          int len16 = 0;
9          if size - 6 >= 4 then
10             len16 = ((size - 6) / 2) - 1;
11             framedata = (char16_t *)(buffer + 6);
12             if 0xfffe ==* framedata then
13                 for i = 0 to len16 do
14                     framedata[i] = bswap_16(framedata[i]);
15                 end
16             end
17         end
18         if isUTF8 then
               /* mFileMetaData ist vom Typ MetaData                */
19             mFileMetaData→setCString(metadataKey, (const char *)buffer +
                6);
20         end
21         ...
22     end
23     ...
24     return OK;
25 ...
26 MetaData::setCString uint32_t key, const char *value
27     return setData(key, TYPE_C_STRING, value, strlen(value))
```

Bezüglich des Fehlers von *CVE-2015-3828* ist die Daten-Größe des Atoms maßgeblich. Ist der Wert kleiner als sechs, kommt es in Zeile 10 zum Zahlenunterlauf und infolgedessen enhält *len16* eine größere Zahl, als *buffer* eigentlich groß ist. Darüber hinaus kommt es in Zeile 14 zum Speicherüberlauf, wenn die 16-Bit Bytefolge an der Stelle *buffer + 6* der Zahl *0xFFFE* entspricht.

Nachdem Atom-Größe = 8 + *size* und *size* kleiner 6 gilt, muss die Atom-Größe mit *0x0E* verglichen werden, um eine manipulierte MP4-Datei zu erkennen. Damit beschädigte MP4-Dateien nicht fälschlicherweise als Angriff erkannt werden, muss die Atom-Größe zusätzlich größer als *0x0A* sein.

CVE-2015-3827 / CVE-2015-3829

Bei dem Fehler *CVE-2015-3827* handelt es sich um einen Integer-Underflow, während *CVE-2015-3829* einen Integer-Overflow kategorisiert, wobei beide bei der Verarbeitung eines COVR-Atoms einer MP4-Datei auftreten können. In Skript 4.6 ist der verantwortliche Quelltext zu sehen.

Algorithm 4.6: Ausschnitt aus der Methode parseChunk in der Klasse MPEG4Extractor.cpp für das COVR-Atom [50]

```
1 uint64_t chunk_size;
2 off64_t chunk_data_size;
3 ...
4 sp<ABuffer > buffer = new ABuffer (chunk_data_size + 1);
5 if mDataSource→readAt(data_offset, buffer→data(), chunk_data_size) !=
  (ssize_t)chunk_data_size then
6 |  return ERROR_IO;
7 end
8 const int kSkipBytesOfDataBox = 16;
9 mFileMetaData→setData(kKeyAlbumArt, MetaData ::
  TYPE_NONE, buffer → data() +
  kSkipBytesOfDataBox, chunk_data_size − kSkipBytesOfDataBox);
```

Die Variablen *chunk_size* und *chunk_data_size* beeinhalten die Atom-Größe sowie die Daten-Größe. Der Zahlenüberlauf für *CVE-2015-3829* kann in Zeile 4 auftreten, falls *chunk_data_size >= 0xFFFFFFFF* gilt. Da auch *chunk_data_size == chunk_size - 8* gilt, muss für eine automatische Erkennung eines Angriffs auf das COVR-Atom, die Atom-Größe mit *0x07* verglichen werden. Bezüglich *CVE-2015-3827* passiert der Zahlenunterlauf in Zeile 9 bei der Subtraktion von chunk_data_size und der Zahl 16, wenn *chunk_data_size <= 0xF* gilt. Um einen Angriff auf *CVE-2015-3827* automatisch erkennen zu können, muss die Atom-Größe, also chunk_size, mit *0x17* verglichen werden, da nach wie vor *chunk_data_size == chunk_size - 8* gilt.

4.1.2 Android Libutils - CVE-2016-3861

In den Android Libutils werden verschiedene Hilfsfunktionen für Entwickler zur Verfügung gestellt, unter anderem Unicode.cpp. Diese Klasse liefert unterschiedliche Methoden, wie Konvertierungen oder Längenberechnungen, um Text im UTF-Format zu verarbeiten.

Bei der Schwachstelle CVE-2016-3861 handelt sich um einen Heap-Puffer-Überlauf bei der Konvertierung von Text im UTF-16 zu Text im UTF-8-Format. Entdeckt wurde sie im Juni 2016 von Mark Brand [54], welcher die Androidversionen 4.x bis exklusive 4.4.4, 5.0.x bis exklusive 5.0.2, 5.1.x bis exklusive 5.1.1, 6.x und 7.0 bis 01.09.2016 [55] verwundbar nennt.

Der Ursprung der Sicherheitslücke liegt in der Berechnung der benötigten Größe des Zielspeichers, welcher die konvertierten UTF-16-Zeichen erhalten soll. Zuerst wird die Funktion utf16_to_utf8_length aufgerufen, welche die Größe des resultierenden UTF-8-Textes berechnet. Das Resultat der Methode wird zur Allozierung eines Puffers verwendet, welcher beim Aufruf der Funktion utf16_to_utf8 mit den konvertierten Zeichen befüllt wird. Die Methode utf16_to_utf8 übernimmt dabei die eigentliche Konvertierung von Zeichen im UTF-16-Format zu Zeichen im UTF-8-Format. Die Differenz der beiden Funktionen liegt in der Verarbeitung des Strings, der Gültigkeit der gelesenen Byte-Paare und der berechneten Länge des Zieltexts. Ob ein Byte-Paar gültig ist, hängt von seiner Position im Surrogate Pair ab. Der High-Surrogate, das erste Byte-Paar, ist gültig wenn ein Bitweises-Und mit *0xFC00* dem Wert *0xD800* entspricht. Der Low-Surrogate, das letzte Byte-Paar, ist gültig wenn die analoge Rechenoperation dem Wert *0xDC00* entspricht. Demnach ist ein Surrogate Pair gültig, wenn sowohl der High- als auch der Low-Suorrogate gültig sind.

Algorithm 4.7: utf16_to_utf8_length in Unicode.cpp bei Version 4.2.2_r1 [56]

```
1  size_t ret = 0;
2  ...
3  while src < end do
4      if (*src & 0xFC00) == 0xD800 && src + 1 < end && (*++src & 0xFC00)
        == 0xDC00 then
5          ret += 4;
6          src++;
7      else
8          ret += utf32_codepoint_utf8_length((char16_t) *src++);
9      end
10 end
11 return ret;
```

utf16_to_utf8_length, ersichtlich in Skript 4.7, verarbeitet den UTF-16-Text, abhängig von der Gültigkeit des High-Surrogate, in Vier-Byte oder in Zwei-Byte-Schritten. Ist der High-Surrogate gültig, erfolgt ein Vier-Byte-Schritt zum nächsten Surrogate Pair,

unabhängig von der Gültigkeit des Low-Surrogate. Ist der High-Surrogate jedoch ungültig, erfolgt ein Zwei-Byte-Schritt und der aktuelle Low-Surrogate wird zum neuen High-Surrogate.

Algorithm 4.8: utf16_to_utf8 in Unicode.cpp bei Version 4.2.2_r1 [56]

```
1  ... const char16_t * cur_utf16 = src;
2  const char16_t * const end_utf16 = src + src_len;
3  char *cur = dst;
4  while cur_utf16 < end_utf16 do
5      char32_t utf32;
6      if (*cur_utf16 & 0xFC00) == 0xD800 && cur_utf16 + 1 < end_utf16 &&
        (*(cur_utf16 + 1) & 0xFC00) == 0xDC00 then
7          utf32 = (*cur_utf16++ - 0xD800) « 10;
8          utf32 |= *cur_utf16++ - 0xDC00;
9          utf32 += 0x10000;
10     else
11         utf32 = (char32_t) *cur_utf16++;
12     end
13     const size_t len = utf32_codepoint_utf8_length(utf32);
14     utf32_codepoint_utf8((uint8_t*) cur, utf32, len);
15     cur += len;
16 end
17 *cur = '\0';
```

Bei utf16_to_utf8, ersichtlich in Skript 4.8, ist die Größe der Byte-Schritte abhängig von der Gültigkeit des gesamten Surrogate Pair. Ist das überprüfte Paar gültig, wird daraus ein UTF-8-Zeichen kalkuliert und in den allozierten Speicher geschrieben. Gleichzeitig wird der aktuelle Zeiger um vier Byte erhöht. Ist das Surrogate Pair jedoch ungültig, erfolgt ein Zwei-Byte-Schritt und der aktuelle Low-Surrogate wird, analog zu utf16_to_utf8_length, als neuer High-Surrogate verwendet.

Wird eine speziell präparierte UTF-16-Zeichenfolge an die beiden Methoden übergeben, liest utf16_to_utf8 ein gültiges Surrogate Pair an einer Stelle, welche von utf16_to_utf8_length nicht beachtet wird. Zur Veranschaulichung des Problems dient die Zeichenfolge

0x01 0xd8 0x02 0xd8 0x03 0xdc 0x00 0x00

, entnommen aus dem Proof-of-Concept[57] von Mark Brand. Beim Verarbeiten des manipulierten Texts, liest utf16_to_utf8_length die Surrogate Pairs *0x01d8 0x02d8*, *0x03dc* und *0x0000*. Die Methode erkennt folglich kein gültiges Paar und retourniert die Zahl 0 als benötigte Größe für den Zielspeicher. utf16_to_utf8 hingegen liest die Surrogate Pairs *0x01d8 0x02d8*, *0x02d8 0x03dc* und *0x0000*. Gleichzeitig wird das gültige

Surrogate Pair, *0x02d8 0x03dc*, erkannt und infolgedessen ein Überlauf des Heap-Speichers produziert, da vier Bytes in einen Speicher der Größe 0 geschrieben werden.

Laut Sicherheitsforscher Mark Brand[54] sind die Services system_server, keystore und mediaserver, neben vielen anderen, verwundbar. Anfang September 2016 stellte er außerdem ein Proof-of-Concept[57] zur Verfüfung, welches die Sicherheitslücke in Verbindung mit ID3-Tags ausnutzt. Mithilfe der generierten MP4-Datei, eingebettet in eine Webseite, wird anschließend eingeschleuster Schadcode auf dem missbrauchten Gerät ausgeführt. Um ein verwundbares System zu kompromittieren ist, analog zu den Stagefright-Schwachstellen, keine Benutzerinteraktion erforderlich.

Die automatische Erkennung dieser Sicherheitslücke ist etwas aufwändiger, als bei Stagefright, da nicht mit einem vordefinierten Wert verglichen werden kann. Stattdessen müssen die beiden Methoden nachgebaut und leicht modifiziert werden, sodass ein möglicher Angriff erkannt wird. Während utf16_to_utf8_length unverändert bleibt, muss utf16_to_utf8 dahingehend abgewandelt werden, dass die Länge der vermeintlich geschriebenen Bytes zurückgegeben wird. Ist nun der Rückgabewert von utf16_to_utf8_length kleiner, als der von utf16_to_utf8, kommt es zu einem Heap-Pufferüberlauf und ein Angriff kann nicht ausgeschlossen werden. Zusammen kann man die Methoden verwenden, um kompromittierte ID3-Tags oder andere UTF16-Strings zu erkennen.

4.1.3 Android Exif Interface - CVE-2016-3862

Das Android Exif Interface wird zum Verarbeiten von Bild-Metadaten benutzt. Es wird standardmäßig in zahlreichen Apps, wie Gmail und Gchat verwendet, um die Exifinformationen von Bildern automatisch zu lesen. Darüber hinaus stellt Android dieses Interface auch Drittapplikationen, zum Auslesen von Foto-Metadaten, zur Verfügung.

Anfang September 2016 wurde eine Schwachstelle, *CVE-2016-3862*, im Android Exif Interface vom Sicherheitsforscher Tim Strazzer entdeckt, welche es Angreifern ermöglicht schädlichen Code auf einem verwundbaren Gerät auszuführen. Betroffene Androidversionen sind 4.x bis exklusive 4.4.4, 5.0.x bis exklusive 5.0.2, 5.1.x bis exklusive 5.1.1 und 6.x bis 01.09.2016. Dabei ist es unerheblich, ob der Benutzer das Bild selbst öffnet, da bereits das Empfangen eines kompromittierten Bildes über Gmail oder eine Social-Media-App ausreicht, dass Exif-Daten gelesen und die Sicherheitslücke ausgenutzt wird.

In Skript 4.9 ist der für *CVE-2016-3862* verantwortliche Zahlenüberlauf ersichtlich. Der Funktion ProcessExifDir wird der Start-Offset der Directory-Einträge des Exifs übergeben, welche wiederum über alle Exif-Directories in einer Schleife iteriert. Dabei wird der Zeiger zum aktuellen Eintrag in der Variable *DirEntry* gespeichert. Danach wird die Größe der Daten und das Format des Exif-Eintrags gelesen, um anschließend die Anzahl an benötigten Bytes in *ByteCount* zu speichern. *OffsetVal* wiederum, wird mit dem relativen Offset vom Beginn des Exif-Headers zu den eigentlichen Exif-Entry-Daten befüllt.

Da *DirEntry* komplett durch den Angreifer kontrolliert wird, kann es in Zeile 11, bei der Summe von *OffsetVal* und *ByteCount*, zu einem Zahlenüberlauf kommen und der

Algorithm 4.9: Integer Overflow in exif.c in der Funktion ProcessExifDir bei Version 4.0.3_r1 [58]

```
1  int Tag, Format, Components;
2  int ByteCount;
3  ...
4  Format = get16u(DirEntry+2);
5  Components = get32u(DirEntry+4);
6  ...
7  ByteCount = Components * BytesPerFormat[Format];
   /* Wenn ByteCount <= 4 ist, steht der Wert in den nächsten
      4 Bytes, ansonsten befüllt mit dem Offset zu den Daten.
      */
8  if ByteCount > 4 then
9      unsigned OffsetVal;
10     OffsetVal = get32u(DirEntry+8);
11     if OffsetVal + ByteCount > ExifLength then
12         ...
13         continue;
14     end
15 end
16 ...
```

resultierende Wert kleiner als *ExifLength* sein. *ExifLength* selber ist die Längen der gesamten Exif-Daten und gleichzeitig die Größe des allozierten Speichers, in den die Exif-Daten kopiert werden sollen.

Anhand der Iteration über alle Exif-Einträge und der Verifizierung der Abfrage in Zeile 11 für jeden Eintrag, kann ein Angriff auf diese Sicherheitslücke automatisch erkannt werden. Ist die Summe von *OffsetVal* und *ByteCount* größer als *0xFFFFFFFF*, sind die Exif-Daten möglicherweise mutwillig manipuliert worden.

4.1.4 Android WebView

Android WebView ist eine Komponente des Android-Betriebssystems, welche es Entwicklern erlaubt, Webinhalte in ihre Applikationen einzufügen. Grundsätzlich bestehen zwei Möglichkeiten, wie man Webseiten oder Teile davon unter Android konsumieren kann. Mithilfe eines normalen Browsers oder aber anhand einer Android-App mit integriertem WebView. Letzteres ermöglicht Entwicklern ihre Applikationen mit einer Browserfunktionalität zu erweitern.

Die Funktion WebView.addJavascriptInterface bietet Javascript die Möglichkeit, Methoden auf Java-Objekten auszuführen und folglich eine beschränkte Kontrolle über das Androidsystem zu erlangen. Darüber hinaus ist auch der umgekehrte Weg, von Java zu

Javascript, durch die Funktion WebView.loadUrl möglich. Dies ermöglicht es Android-Apps, Javascript im Kontext der aktuellen Seite auszuführen. Beide Funktionalitäten zusammen, ermöglichen App-Entwicklern, ein nahtloses Zusammenspiel von Webseiten und Android-Apps.

CVE-2012-6636 - addJavascriptInterface

Die WebView-Schnittstelle *addJavascriptInterface* schafft die Voraussetzung, dass von Entwicklern ausgewählte Java-Objekte, Webseiten und deren Javascript zur Verfügung stehen. In Skript 4.10 ist die Benutzung der Schnittstelle ersichtlich.

Algorithm 4.10: HelloWorldObject wird für die Benutzung in Javascript bereitgestellt

```
1 WebView webView = (WebView) findViewById(R.id.webView);
2 webview.getSettings().setJavaScriptEnabled(True);
3 webView.addJavascriptInterface(new HelloWorldObject(), "helloWorld");
4 webView.loadUrl("http://mySite.com");
```

HelloWorldObject ist eine Java-Klasse die ausschließlich die Funktion *sayHello* zur Verfügung stellt. In Zeile 1 wird eine Instanz von WebView erzeugt, welcher anschließend die Ausführung von Javascript erlaubt wird. Die eigentliche Bereitstellung von HelloWorldObject, passiert mit dem Aufruf addJavascriptInterface in Zeile 3. Dabei wird ein Objekt des Typs HelloWorldObject zur Schnittstelle hinzugefügt und über die Javascript-Variable *helloWorld* verfügbar gemacht. Danach wird die Seite *mySite.com* aufgerufen, welche nun via Javascript Zugriff auf das Java-Objekt erlangt.

In Skript 4.11 ist der entsprechende Aufruf von *sayHello* durch ein Javascript-Skript ersichtlich.

Algorithm 4.11: Aufruf der Methode sayHello durch Javascript

```
1 <script>
2 | helloWorld.sayHello();
3 </script>
```

Der Aufruf sayHello auf dem Objekt helloWorld, stellt eine sehr marginale Nutzung der WebView-Schnittstelle dar, allerdings zeigt es ungeachtet dessen wie einfach, aber auch mächtig diese Funktionalität ist. Für einen Angreifer ist die Bereitstellung eines Java-Objekts, wie durch Skript 4.10 ausreichend, um einen Angriff zu starten. In Skript 4.12 ist eine entsprechende Attacke angeführt, wobei davon ausgegangen wird, dass der Angreifer den Namen des bereitgestellten Java-Objekts, helloWorld, kennt.

Durch die Javascript-Variable *helloWorld* steht nun das komplette Java-Objekt zur Verfügung, sodass via Java-Reflection ein beliebiges Java-Objekt oder eine Java-Klasse und deren Methoden aufgerufen werden können. Diese Funktionalität ist in Skript 4.12

Algorithm 4.12: runShellCommand.js führt einen Shell-Befehl aus

```
  /* Führt Argument über Reflection-Chain aus                    */
1 Function execute(cmdArgs):
2 |  return helloWorld.getClass().forName("java.lang.Runtime").
  |   getMethod("getRuntime", null).invoke(null,null).exec(cmdArgs);
3 end
  /* Kopiert Daten von /sdcard/source nach /sdcard/target       */
4 var p = execute(["/system/bin/sh",c","mv /sdcard/source /sdcard/target"]);
```

in der Funktion *execute*, welche mithilfe von Java-Reflection in der Androidumgebund ein beliebiges Systemkommando ausführt, ersichtlich.

In Zeile 4, wird die Funktion *execute* benutzt, um Dateien von einem, in einen anderen Ordner zu kopieren. Um auf die SD-Karte zugreifen zu können, muss die missbrauchte Android-App die entsprechenden Berechtigungen besitzen. An dieser Stelle könnten allerdings auch andere Aktionen, wie das Starten eines Lokalen-Exploits, um Root-Rechte zu erhalten, ausgeführt werden.

Diese Art von Exploit ist allerdings nur bis exklusive Android JellyBean 4.2, Android API Level 17, möglich. Ab Android JellyBean MR1 ist die Annotation *@JavascriptInterface* für jede Methode eines Java-Objekts nötig, welche durch Javascript erreichbar sein soll [59]. Dadurch werden Angriffe via Java-Reflection verhindert, da Methoden wie *getClass* oder *invoke* nicht annotiert sind.

Die automatische Erkennung von Angriffen auf addJavascriptInterface ist schwierig, da der Exploit-Code stark variieren kann. Essentiell für einen erfolgreichen Angriff auf WebView sind die veröffentlichten Java-Objekte. Manche Frameworks, wie Apache Cordova oder Phonegap, veröffentlichen standardmäßig deren Kern-Objekte, als Javascript-Objekte mit den Namen *cordova* und *phonegap*. Der Zugriff durch eine Webseite auf ein freigestelltes Java-Objekt stellt an sich noch keine Gefahr dar. Erst der Aufruf der Methode *getClass* auf einem Java-Objekt, lässt ein böswilliges Verhalten vermuten. Nachdem einem Angreifer die Namen der veröffentlichten Java-Objekte unbekannt sind, muss er jedes Javascript-Objekt auf das Vorhandensein der Methode *getClass* überprüfen. In Skript 4.13 ist ein Test auf *getClass* für alle definierten Objekte im Browser ersichtlich.

Hooked man nun die Methode *getClass* für jedes Javascript-Objekt, erkennt man einen Zugriff darauf. Geht man davon aus, dass ein Angreifer das erste Objekt verwendet, welches eine *getClass* Methode zur Verfügung stellt, ist es ausreichend nur ein entsprechendes Objekt zu erstellen und Aufrufe darauf festzuhalten. Da ein Aufruf von *getClass* während der Interaktion mit einer Webseite selten stattfindet - ab Android 4.2 wird der Zugriff standardmäßig verhindert - kann im positiven Fall von einem Angriff ausgegangen werden.

Algorithm 4.13: Test in Javascript, ob ein Java-Objekt via addJavascriptInterface veröffentlicht wurde

```
1  var foundJavaObject = null;
2  for var index in top do
3      try
4          top[index].getClass().forName("java.lang.Runtime");
5          foundJavaObject = top[index];
6      catch
           /* Wird nur gecatcht damit der For-Loop nicht abbricht
              */
7      endtry
8  end
9  if foundJavaObject != null then
       /* Java-Objekt steht zur Verfüfung und kann für einen
          Angriff verwendet werden                             */
10 end
```

loadUrl - Frame Confusion

Die WebView-Methode *loadUrl* lädt eine Webseite anhand der als Parameter übergebenen Url und zeigt diese anschließend dem Benutzer an. Die Anwendung von *loadUrl* beim Laden der Webseite *http://mySite.com* ist in Skript 4.10 in Zeile 2 veranschaulicht. Neben dem Laden und Navigieren zu Seiten, kann *loadUrl*, in Java definiertes Javascript, im aktuellen Webseitenkontext ausführen. Dazu muss der übergebene String-Parameter mit *javascript:* beginnen. Das nachfolgende Javascript wird von der WebView API in der aktuell geladenen Seite ausgeführt. In Skript 4.14 wird *loadUrl* benutzt, um mithilfe von Java, einen vorher definierten Text in den DOM-Baum der Webseite einzufügen.

Algorithm 4.14: loadUrl wird in Java genutzt, um Javascript auszuführen

```
1  String str = "Hallo Welt";
2  webView.loadUrl("javascript:document.appendChild("+ str + ")");
```

Erweitert man das Java-Objekt *HelloWorldObjekt* um eine Methode *navigateToLink*, welche einen String-Parameter entgegen nimmt und diesen zum Aufruf von *loadUrl* verwendet, kann ein Angreifer diese Methode nun benutzen, um den Seiteninhalt zu verändern. In dieser Konstellation wäre die Funktionalität von *loadUrl* noch kein Problem, da der Seiteninhaber ohnehin den Seiteninhalt verändern kann. Allerdings macht die fehlende Same-Origin-Policy von WebView bezüglich der loadUrl-Methode die Situation gefährlicher. Die Same-Origin-Policy in Browsern isoliert normalerweise iframes mit unterschiedlichem Ursprung vom restlichen Teil der Seite, sodass das Javascript in iframes keinen Zugriff auf den DOM-Baum der Seite und vice versa erhält. Dadurch soll eine Missbrauch von in iframes eingebettetem Javascript verhindert werden. Dieser

Schutzmechanismus fehlt allerdings beim Aufruf von loadUrl gänzlich, wodurch ein Angreifer aus einem iframe, Zugriff auf den eigentlichen Seiteninhalt bekommt. In Skript 4.15 ist der Missbrauch von *navigateToLink* durch das Javascript eines iframes zu sehen.

Algorithm 4.15: Missbrauch von *navigateToLink* durch ein iframe, um den Seiteninhalt zu ändern

```
1 <iframe>
2     <script>
3         helloWorld.navigateToLink("javascript:document.appendChild('Hello
           World')");
4     </script>
5 </iframe>
```

Einen entsprechenden Angriff zu erkennen ist schwierig, da die Methode des veröffentlichten Java-Objekts bekannt sein muss. Eine Möglichkeit wäre einen Javascript-Hook auf jeder Methode jedes Javascript-Objekts, welche einen String als Parameter entgegen nimmt, zu registrieren. Enthält der übergebene String den Text *javascript*, könnte man von einem Angriff ausgehen. Allerdings kann es dabei zu erheblichen Performance-Problemen kommen, weil viele Objekte eine entsprechende Methode besitzen.

Eine andere Möglichkeit wäre das Registrieren eines eigenen Objekts, ähnlich zu einem Honeypot, welches sich wie ein vergleichbares Objekt, hinzugefügt über addJavascriptInterface, verhält. Da jedes veröffentlichte Java-Objekt von java.lang.Object ableitet, muss unser Javascript-Objekt alle entsprechenden Methoden hooken, damit es nicht als Honeypot enttarnt wird. Infolgedessen können die nötigen Hooks auf ein Objekt beschränkt und das Performance-Problem gelöst werden.

4.1.5 Automatische Erkennung

Die automatische Erkennung von Angriffen auf die beschriebenen Sicherheitslücken, wurde in den Kapiteln 4.1.1 bis 4.1.4 im Detail analysiert. Die nachstehende Tabelle 4.1 stellt demnach eine Zusammenfassung der gesammelten Informationen dar. Darin sind die einzelnen CVE-Nummern in Spalte *Sicherheitslücke*, sowie der Angriffsvektor, über den die Schwachstelle ausgenutzt werden kann, ersichtlich. Darüber hinaus wird eine kurze Beschreibung des Typs, beispielweise Integer-Overflow, gegeben und das betroffene Objekt definiert. Zuletzt sind in der Spalte *Erkennung* die Vergleichswerte, sowie die durchzuführenden Aktionen für eine erfolgreiche Erkennung ersichtlich. Dabei entspricht *numEntries* dem Wert der vier Bytes 12 bis 16 nach dem Mp4-Chunk-Typ und der Version, inklusive der Flags. *chunkSize* bezieht sich auf die vier Bytes vor dem Mp4-Chunk-Typ, während *dataSize* die Anzahl der Bytes nach dem Chunk-Typ, also *chunkSize - 8* widerspiegelt.

Abbildung 4.1: Zusammenfassung der Erkennung von Remote-Exploits

Sicherheitslücke	Angriffsvektor	Beschreibung	Erkennung
CVE-2015-1538#1	MP4-Datei	Integer-Overflow bei STSC	numEntries >0x15555555
CVE-2015-1538#2	MP4-Datei	Integer-Overflow bei STSS	numEntries >= 0x40000000
CVE-2015-1538#3	MP4-Datei	Integer-Overflow bei CTTS	numEntries >= 0x20000000
CVE-2015-1538#4	MP4-Datei	Integer-Overflow bei STTS	numEntries >= 0x20000000
CVE-2015-1539	MP4-Datei	Integer-Underflow bei ESDS	In Tabelle 4.3
CVE-2015-3824	MP4-Datei	Integer-Overflow bei TX3G	$\sum_{1stTX3G}^{Last} chunkSize$ >0xFFFFFFFF
CVE-2015-3826	MP4-Datei	Buffer-Overread bei TITL, PERF, AUTH, GNRE, ALBM	Data doesn't end with 0x00
CVE-2015-3828	MP4-Datei	Integer-Underflow bei TITL, PERF, AUTH, GNRE, ALBM	0x0A <chunkSize <0x0E
CVE-2015-3827	MP4-Datei	Integer-Underflow bei COVR	dataSize <= 0xF
CVE-2015-3829	MP4-Datei	Integer-Overflow bei COVR	dataSize >= 0xFFFFFFFF
CVE-2016-3861	MP4-Datei	Buffer-Overflow beim Lesen eines UTF16-Strings	wenn Länge von utf16ToUtf8 größer ist
CVE-2016-3862	JPG-Datei	Integer-Overflow bei allen Tags	ByteCount + Offset >0xFFFFFFFF
CVE-2012-6636	Java-Objekt	Java-Reflection bei jedem Objekt möglich	Javascript-Hook auf die Methode getClass

4.2 Android-Emulatoren

Die Erkennung von Emulatoren spielt für Malwareschreiber und Sicherheitsforscher eine entscheidende Rolle, da Erstere versuchen Emulatoren zu erkennen, um Exploits nicht an Sicherheitsforscher auszuliefern, während Letztere Emulatoren verstecken, um Angriffe zu sichten und Exploits zu erhalten. Gemeinsam ist beiden die Identifizierung von Browser- oder Emulatorenmerkmalen zur eindeutigen Bestimmung oder Vortäuschung eines bestimmten Gerätes. Bevor Überlegungen bezüglich gewisser Differenzierungsparameter gemacht werden, müssen zunächst sämtliche verfügbaren Daten gesammelt werden, um anschließend aussagekräftige Merkmale zur Identifizierung eines Geräts zu selektieren. Da eine rein webbasierte Erkennung von Geräten bei Drive-By-Downloads erforderlich ist, wurde eine Website erstellt, welche Informationen über den Browser des Besuchers, das Betriebssystem und sonstige auslesbare Daten speichert.

Die Implementierung dazu besteht aus drei Komponenten: dem HTTP-Server sowie zwei Javascript-Dateien, *deviceDetectionScriptHead* und *deviceDetectionScriptBody*, welche das Auslesen der Daten übernehmen.

4.2.1 HTTP-Server

Der HTTP-Server wurde mithilfe der Skriptsprache Python 3 erstellt, welcher auf Basis des *BaseHTTPServer* des Python-Frameworks agiert. Da die eigentliche Identifizierung von den beiden bereits erwähnten Skripten vorgenommen wird, besitzt der Server diesbezüglich keine relevante Logik, sodass er ausschließlich zum Speichern von Daten und Liefern des HTML-Inhalts verwendet wird. Ungeachtet dessen enthält er benutzerdefinierte Implementierungen für die HTTP-Anfragemethoden GET und POST, welche unterschiedliche Aufgaben übernehmen.

Die POST-Methode nimmt Daten entgegen, welche beim Auslesen eines Geräts entstehen und speichert diese in einem vorher definierten Ordner. Dazu wird beim Start des HTTP-Servers ein Pfad zu einem Ordner angegeben, welcher als Basis für die zu erstellenden Dateien dient. In diesem Ordner wird ein Unterordner *store* erstellt, sodass alle zukünftig eintreffenden Informationen darin gespeichert werden. Da unterschiedlichste Werte der einzelnen Webskripte an die POST-Methode übergeben werden, wird anhand des URL-Pfads entschieden, in welche Datei und welchen Unterordner die empfangenen Daten geschrieben werden sollen. Wird beispielsweise die URL *http://localhost/Nexus5x/GPSDaten* mithilfe eines POST-Aufrufs angefragt, wird zuerst der Ordner *Nexus5x*, falls er nicht bereits existiert, als Kind von *store* angelegt. In diesem Fall entspricht *Nexus5x* dem bereits ausgelesenen Gerätetyp, sodass einlangende Informationen unterschiedlicher Geräte streng danach gespeichert und nicht miteinander vermischt werden. Anschließend wird an diesem Ort - falls nötig - die Datei *GPSDaten* erstellt, welche dann mit den eigentlichen als Post-Parameter übergebenen Daten, in diesem Fall GPS-Daten, befüllt wird. Da einige Webskripte, wie Event-Handler, öfters Informationen an die selbe URL senden, werden zuvor gespeicherte Dateien nicht über-

schrieben, sondern mit den neu erhaltenen Daten ergänzt. Infolgedessen werden eventuell doppelte Einträge erzeugt, welche anschließend händisch aussortiert werden müssen.

Die GET-Methode liefert den HTML-Code mit den beiden eingebetteten Javascript-Dateien *deviceDetectionScriptHead* und *deviceDetectionScriptBody* zurück. Werden Ressourcen wie Bild-, CSS- oder Flashdateien angefragt, wird statt dem präparierten HTML-Code, die entsprechende Datei mit dem passendem MIME-Typ zurückgeliefert.

4.2.2 Erkennungsskripte

Das Einlesen der Daten von Websitebesuchern, wurde in zwei Skripte unterteilt, wobei ein Skript in den Head und das verbleibende in den Body des HTML-Codes der Seite injiziert wird. Da einerseits Javascript-Event-Handler registriert werden mussten, andererseits aber manche Daten erst nach dem Laden der kompletten Seite gesammelt werden konnten, war die Aufteilung in zwei Skripts erforderlich. Die angesprochenen Event-Handler könnten zweifellos auch erst nach dem Laden der gesamten Seite eingetragen werden, allerdings wäre infolgedessen der Verlust einiger wichtiger, bereits gefeuerter Events denkbar. Beim Auslesen einiger Geräteinformationen wurden die Javascript Bibliotheken Fingerprintjs2 [60] und ClientJs[61] zu Hilfe genommen.

Head - deviceDetectionScriptHead

In den Kopf der Webseite wird die Javascript-Datei *deviceDetectionScript.js* eingebettet und dient überwiegend als Sammlung von Funktionen und Hilfsfunktionen, welche teilweise von diesem, als auch vom zweiten Skript verwendet werden. Zusätzlich registriert das Skript zu Beginn einige Event-Handler und weitere Javascript-Objekte, welche für die korrekte Arbeit einzelner Methoden erforderlich sind. Dies beinhaltet die Listen *iteratedObjects*, *sortList*, *sameReferences*, *notIterableElems* und *notIterableTypes*, welche bei den verwendeten Passagen genauer beschrieben werden.

Nachfolgend werden die Implementierungen der einzelnen Funktionen kurz erklärt, um einen groben Überblick über die Funktionalität dieses Skripts zu vermitteln.

Enthaltene Hilfsfunktionen werden geblockt erwähnt, jedoch nicht im Detail beschrieben, da sie nur Aufgaben, wie das Hinzufügen von Text zum HTML-Body oder das Durchsuchen einer Liste nach einem übergebenen Objekt, erfüllen. Zusätzlich zu den angesprochenen Aufgaben, welche in den Funktionen *appendToBody* und *includes* implementiert sind, übernimmt die Funktion *post* die Übermittlung der gesammelten Daten an den Python-Server.

registerEventHandler wird am Beginn der Seite aufgerufen, sodass die Javascript-Event-Handler noch vor dem Aufbau der Webseite registriert sind. Bei seiner Ausführung, wird über alle Eigenschaften des Toplevel-Objekts *top* iteriert und für jede Eigenschaft der jeweilige Name für einen Vergleich herangezogen. Enthält der Name das Schlüsselwort *on* genau einmal und befindet es sich zusätzlich am Beginn, wird der Eigenschaft die

Funktion *eventHandler* zugewiesen und somit ein Event-Handler registriert. Kommt es später zu einem Event wie *ondevicemotion*, wird automatisch die Funktion *eventHandler* mit dem eigentilchen Event als Argument aufgerufen.

eventHandler stellt eine Methode zum Verarbeiten aller geworfener Javascript-Events dar und nimmt ein entsprechendes Objekt als Parameter entgegen. Aus dem übergebenen Objekt werden dann der Typ und der Wert des Events gewonnen, welche mithilfe der Funktion *post* an den Server übermittelt werden.

getBrowserVars iteriert mithilfe von *iterateObject* rekursive über alle Objekte in *top* und übermittelt anschließend die sortierte Liste *sortList* mit den gesammelten Daten an den Server. Vor dem Aufruf an *iterateObject* wird mit dem Javascript-Object-Prototype die Methode *includes* in jedem Javascript-Objekt verfügbar gemacht, damit sie später für Vergleiche zur Verfügung steht.

iterateObject übernimmt ein Objekt und den aktuellen Pfad du diesem Objekt als Parameter, fügt es zu der Liste der bereits verarbeiteten Objekte *iteratedObjects* hinzu und iteriert über seine Kinder. Für jedes Kind wird der Name, der Typ und der Wert bestimmt, welche anschließend in Kombination mit dem Pfad des übergeordneten Objekts, der Liste *sortList* hinzugefügt werden. Weiters verwendet es *sameReferences*, beinhaltet Objekte mit der gleichen Referenz, *notIterableTypes*, beinhaltet Datentypen, welche keine Kinder besitzen, und *notIterableElems*, beinhaltet bekannte Objekte wie *sortList*, um Kinder von der rekursiven Abarbeitung auszuschließen. Befindet sich das aktuelle Kindobjekt in keiner der Listen, wird der übergebene Pfad mit dem Namen des Kindes ergänzt und neben dem eigentlichen Objekt als Argument für den Aufruf an *iterateObject* benutzt.

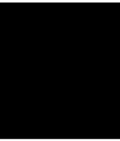
getFonts versucht eine Liste aller verfügbarer Schriftarten zu erstellen und leitet diese bei Erfolg an den HTTP-Server weiter. Da grundsätzlich keine Möglichkeit zum Auslesen der Schriftarten mit Javascript besteht, wurde auf die Bibliothek Fingerprintjs2[60] zurückgegriffen. Fingerprintjs2 benutzt, falls vom Gerät unterstützt, eine Flash-Applikation, um die installierten Schriftarten zu erhalten. Falls, wie bei allen Androidgeräten mit einer Version ab 4.1[62], Flash im Browser nicht verfügbar ist, wird auf eine reine Javascript basierte Erkennung zurückgegriffen. Dieser Modus, eigentlich ein Workaround, ändert mehrmals die Schriftart eines vordefinierten Textes von einer mit bekanntem Platzverbrauch, zu einer die erkannt werden soll. Besteht eine Differenz zwischen den verbrauchten Längen nach dem Wechsel der Schriftart, wurde die gewählte Schriftart vermutlich geladen und steht somit im System zur Verfügung.

ClientJs-Aufrufe Mithilfe von ClienJs[61] werden zusätzlich einige Bildschirmparameter, installierte Plugins, verfügbare MIME-Types sowie der Useragent ausgelesen und durch einen Aufruf auf *post* an den Python-Server zum Speichern gesendet. Die

angesprochenen Funktionalitäten wurden in einzelne, gleichnamige Funktionen gekapselt, damit sie leichter verwendbar sind.

Body - deviceDetectionScriptBody

In den Body des HTML-Codes der Webseite wird die Javascript-Datei *deviceDetectionScriptBody* eingefügt, welche vergleichsweise wenig Logik beinhaltet. Sobald die Seite komplett geladen wurde, startet sie die Abarbeitung vordefinierter Punkte, bei denen sie einzelne Methoden des Skripts *deviceDetectionScriptHead* verwendet. Zu Beginn werden die Browser-Objekte mithilfe von *getBrowserVars* ausgelesen, gefolgt von einem Aufruf auf *getFonts*, um die installierten Schriftarten zu verarbeiten. Zum Abschluss werden die gekapselten Methoden der ClientJs Bibliothek zum Extrahieren der Bildschirmauflösung, Plugins, MIME-Types und Useragent verwendet.

KAPITEL 5 

Resultate

Malvertising-Kampagnen und Drive-By-Downloads für Android sind zurzeit kompliziert zu erkennen. Während herkömmliche Antivirensoftware eine akute Infektion oder einzelne bösartige Programme problemlos identifizieren, verlieren sie bedauerlicherweise den Missing-Link zum Urheber oder Verteiler des eigentlichen Angriffs. Infolgedessen wird der User zwar vor Infektionen geschützt, aber Angreifer bleiben unentdeckt und können andere, ungeschützte Benutzer ungestört attackieren.

Um dieses Problem zu beheben, wird ein Webcrawler, DryCrawl, implementiert, welcher Drive-By-Downloads erkennt, sie Werbenetzwerken oder Seiten zuordnet und daraus eine Liste an gefährlichen Webpages erstellt. Sein Design, die geplanten Module, sowie seine Implementierung sind in den Kapiteln 5.2, 5.3 und 5.4 mit einigen Quelltextbeispielen festgehalten. Davor ist in Kapitel 5.1 die Analyse und Auswertung der Emulatorendaten ersichtlich, welche für die Implementierung von DryCrawl und seinen späteren Einsatz von Bedeutung sind.

5.1 Emulatoren

Nachdem die Implementierung aus Kapitel 4.2 erfolgreich fertiggestellt und getestet wurde, mussten geeignete Kandidaten für aussagekräftige Tests gefunden werden. Dazu wurden einige der in Kapitel 2.3 beschriebenen Emulatoren und Imitatoren als Testobjekte herangezogen und den Versionen von echten Smartphones als Referenzobjekte angeglichen. Da das Ziel der Vergleiche die Feststellung von grundsätzlichen, unveränderten Unterschieden zwischen Emulatoren und echten Geräten ist, wurden neben der Verwendung der sich deckenden Versionen keine Modifizierungen vorgenommen, um ein echtes Gerät bestmöglich zu imitieren. Das Endresultat der Tests soll genau diese Unterschiede aufzeigen, damit die entsprechenden Emulatoren dahingehend konfiguriert werden können, dass Differenzen zwischen den Test- und Referenzobjekten nicht, oder nur schwer, erkennbar sind.

5.1.1 Testumgebungen

Da für die Benutzung der Emulatoren AVD-Emulator und Genymotion keine spezielle Hardware notwendig ist, standen die ersten beiden Testkandidaten kurzerhand fest. Die verwendeten Androidversionen waren 5.0.2 und 7.0, sodass jeder der Betriebssystememulatoren einen Testlauf mit beiden Versionen durchführen musste.

Für die Kategorie der Browser-Emulatoren gingen PhantomJS in Version 2.1.1, SlimerJS 0.10.2, Google Chrome in Version 56.0.2924.87 mit aktivierten Entwicklerwerkzeugen, kurz Chrome-Dev, und Ripple 0.9.15 an den Start, wobei alle unter Ubuntu 16.10 ausgeführt wurden. Um ausreichend Referenzwerte gegenüber den Emulatoren zu erzeugen, mussten zwei geeignete Smartphones mit unterschiedlichen Versionen gefunden werden.

Das erste Gerät meiner Wahl, war mein produktives Alltagsgerät LG Nexus 5x mit der Version 7.0 und vorinstalliertem Chrome Browser, welches als Referenz für neuere Systeme dienen sollte. Da Google in Koorperation mit LG dieses Gerät produziert, läuft darauf das native Androidbetriebssystem ohne jeglichen herstellerspezifischen Modifikationen. Das zweite Smartphone war das Samsung Galaxy A3 mit der Androidversion 5.0.2, welches einige Betriebssystemmodifikationen, wie Samsungs Benutzeroberfläche TouchWiz sowie eigene Nachrichten-Apps, enthält. Zusätzlich wurde darauf noch der Chrome Browser installiert, um Differenzen zwischen ihm und dem nativen Androidbrowser aufzudecken.

Der nächste Schritt zum Start der Emulatorentests war die Auswahl der zu testenden Kontexte. Um im Internet surfende Smartphonebenutzer zu simulieren, wurde auf Google Chrome Mobile, für Geräte mit Version 7.0, sowie den vorinstallierten Android Browser, für Testkandidaten mit Androidversion 5.0.2, zurückgegriffen. Da die Android Webview Schnittstelle das Navigieren im Internet und auf Webseiten erlaubt und einige Android Apps wie Gmail, bei einem Klick auf einen per Email erhaltenen Link wie ein Browser agieren, wurde dieser Teil der Seitenbesucher mit der Webview-Testapplikation aus Kapitel 4.1.4 abgedeckt. Zusätzlich zur Simulation von Webseitenbesuchern, decken die gewählten Kontexte die Angriffsvektoren und teilweise verwundbaren Androidversionen, beschrieben in Kapitel 4.1, ab. Die Androidversionen kleiner 4.2, nötig für die Webview-Sicherheitslücke in Kapitel 4.1.4, konnten nicht getestet werden, da kein echtes Smartphone als Referenzgerät mit einer der betroffenen Versionen zur Verfügung stand.

5.1.2 Testresultate

Nachdem die erstellte Webseite mit allen Geräten und Emulatoren erfolgreich besucht wurde, begann die Auswertung der ausgelesenen System- und Browserdaten. Da die gewonnen Daten sehr umfangreich und teilweise mehr Referenzgeräte für eindeutigere Ergebnisse bezüglich einzelner Parameter nötig wären, wurde die Analyse auf einige vielversprechende Merkmale reduziert. Die Wahl der Unterscheidungsmerkmale fiel auf die offensichtlichsten und gröbsten Unterschiede aller getesteten Geräte und ist folgendermaßen definiert.

- Gefeuerte Javascript-Events

- Javascript-Objekte
- Font-Families
- Bildschirmauflösung
- Useragent-String
- Plugins
- MimeTypes

Jedes der Android-Smartphones wurde mit den Emulatoren mit der zugehörigen Version, sowie den kopflosen Browsern und den Chrome-Extensions verglichen, um eine Emulatorkonfiguration zu finden, welche einem echten Gerät bestmöglich ähnelt.

Events

Um ausreichend Events zu erzeugen, wurde auf allen Systemen, falls möglich, ein Text markiert, die Seite in alle Richtungen gescrollt, sowie hinein und heraus gezoomt. Anschließend wurden die gefeuerten Events der Emulatoren mit denen der Referenzgeräte verglichen und bei einer beidseitigen Übereinstimmung auf Unterschiede der gelieferten Werte geprüft. Dabei stachen einige geworfene Events sofort ins Auge, welche nachfolgend angeführt und beschrieben werden.

ondevicemotion Das Event *ondevicemotion* wird gefeuert, wenn der Beschleunigungssensor des betroffenen Geräts neue Werte liefert, was bereits direkt nach dem Laden, ohne eine eigentliche Bewegung des Smartphones, passiert. Das Galaxy A3 operierte nach dem Laden der Webseite kurios, da die ondevicemotion-Events erst nach dem Öffnen und Schließen der Browsereinstellungen geworfen wurden. Der Beschleunigungssensor des Nexus 5X hingegen lieferte für *ondevicemotion* wie erwartet regelmäßig neue Werte. Kongruent warfen die Emulatoren Genymotion und AVD in beiden Versionen und beiden Kontexten das Event gleichmäßig. Gleichzeitig feuert Genymotion zwar regelmäßig *ondevicemotion* mit unterschiedlichen Werten, allerdings ist der Wert der X-Koordinate immer genau 0, was bei einer 15-stelligen Zahl unwahrscheinlich scheint. Während Chrome-Dev und Ripple das Event zwar wiederkehrend werfen, aber die gelieferten Werte immer *Null* sind, verzichten PhantomJS und SlimerJS komplett darauf.

ontouch* Die Events *ontouch** betreffen alle Situationen, in denen eine Berührung des Gerätebildschirms stattfindet, weshalb sie bei der Benutzung der echten Smartphones regelmäßig beobachtet wurde. Auch die Betriebsystememulatoren Genymotion und AVD, sowie Chrome-Dev liefern Werte beim Scrollen und Markieren von Text. Die beiden kopflosen Browser sowie der Ripple Emulator verzichten jedoch gänzlich auf diese Events, weshalb sie deswegen über Javascript als Geräte ohne Androidbetriebssystem erkennbar wären.

onorientationchange Dieses Event wird bei Wechsel vom Landscape- in den Portrait-Modus eines Geräts geworfen und ist bei den Smartphones, den Betriebssystememulatoren und bei Chrome-Dev vorhanden. Ripple, SlimerJS und PhantomJS besitzen dieses Event nicht, weshalb sie anhand dessen ausgeschlossen werden können.

Javascript-Objekte und Funktionen

Welche Javascript-Objekte verfügbar sind, ist abhängig von der verwendeten Web-Engine und deren Version. Alleine durch die Benutzung zweier unterschiedlicher Web-Engines mit unterschiedlicher Version, kann eine enorme Differenz an verfügbaren Objekte ergeben, weshalb nur auf die auffälligsten Unterschiede eingegangen wird.

performance.memory Das Objekt *performance.memory* beinhaltet Daten über den aktuellen Speicherzustand und die entsprechende Größe am Heap. Zu beachten ist allerdings, dass es, aufgrund von Bedenken bezüglich Side-Channel-Attacken von Seiten der Webkit-Entwickler[63], keine Echtzeitdaten liefert, sondern nur alle zwanzig Minuten einen neuen Wert retourniert. Es existiert ausschließlich in WebKit basierenden Browsern, weshalb PhantomJS und SlimerJS diese Eigenschaft nicht besitzen. Nachdem sowohl Chrome, auf Mobil- und Desktopgeräten, und der Androidbrowser auf der Basis der WebKit-Engine arbeiten, lassen sich mit der Existenz dieses Objekts die beiden kopflosen Browser ausschließen.

Ein weiterer Unterschied liegt im gespeicherten Wert hinter den Variablen *memory.jsHeapSizeLimit*, *memory.totalJSHeapSize* und *memory.usedJSHeapSize*. Während unter den beiden echten Smartphones Nexus 5X und Galaxy A3 als auch dem Chrome-Browser reale Werte in den Feldern gespeichert sind, liefern die Emulatoren Genymotion und AVD immer die Zahl 0 bei einer Abfrage der Eigenschaften.

_phantom und callPhantom Die beiden Objekte _phantom und callPhantom sind Teil der PhantomJS- und SlimerJS-Javascriptumgebung, wobei _phantom ausschließlich in PhantomJS existiert. Durch sie können Entwickler mithilfe von Javascript den Browser steuern und neue Aktionen wie klicken und scrollen durchführen. Da weder _phantom noch callPhantom Teil der WebKit-Engine sind, stehen sie in einer Androidumgebung nicht zur Verfügung, weshalb sie zur Identifizierung von PhatomJS und SlimerJS verwendet werden können. Darüber hinaus liefert ein Aufruf auf *_phantom.toString()* das Resultat *QtRuntimeObject*, wodurch eine PhantomJS-Instanz erkannt werden kann.

speechSynthesis Das Objekt *speechSynthesis* liefert die Kontrolle über die Web Speech API, welche zum Vorlesen von Texten verwendet wird. Es steht bei den beiden Smartphones als auch bei Genymotion, AVD, Ripple und Chome-Dev zur Verfügung, fehlt allerdings bei den beiden kopflosen Browsern.

webkitRequestFileSystem und webkitResolveLocalFileSystemURL Die Funktionen webkitRequestFileSystem und webkitResolveLocalFileSystemURL sind Chrome-

Zusätze und bieten Zugriff auf das lokale Dateisystem in einer Sandbox. Die kopflosen Browser PhantomJS und SlimerJS besitzen diese Funktionalität allerdings nicht, weshalb sie deswegen aussortiert werden können.

navigator.getBattery() Die Funktion *getBattery()* liefert ein Objekt vom Typ BatteryManager zurück, welches für diverse Statusabfragen bezüglich des Gerätebatteriestands verwendet werden kann. Das Objekt steht in den beiden Smartphones, SlimerJS, Chrome-Dev, Ripple, sowie AVD und Genymotion mit Androidversion 7.0 zur Verfügung, kann aber in letzteren beiden Fällen mit der Androidversion 5.0.2 und PhantomJS nicht verwendet werden.

screen.orientation Das Objekt *screen.orientation* liefert Informationen über den Typ und den entsprechenden Winkel der aktuellen Bildschirmorientierung und steht in beiden Smartphones, SlimerJS, Ripple und Chrome-Dev gänzlich, aber in PhantomJS garnicht zur Verfügung. AVD und Genymotion hingegen weisen dieses Objekt nur bei Android 7.0 als Eigenschaft auf, weshalb sie von einem echten Androidgerät zu unterscheiden sind.

SlimerJS-Objekte und Events Zusätzlich zu den bisher genannten Objekten, existieren weitere Javascript-Objekte welche in der Gecko-Engine von SlimerJS's Browser Firefox, aber nicht in der WebKit-Engine von Chrome und der QtWebkitEngine von PhantomJS vorhanden sind. Grundsätzlich sind jegliche *moz** Objekte und Eigenschaften nur in Gecko implementiert, weshalb sie sich zwar als Identifikationsmerkmale von SlimerJS anbieten, aber nur als Ausschlusskriterium verwendet werden können. Weiters sind die Javascript-Events *ontouch**, *onorientationchange*, *oncancel*, *onclose*, *onsearch*, *onwebkit** und *ontransitionend*, im Objekt *top* nicht existent, weshalb auch sie zum Ausschluss von SlimerJS dienen können. Neben den fehlenden existieren zusätzlich die Events *ondevicelight*, *ondeviceproximity*, *onmoz** und *onuserproximity*, welche ihren Ursprung in der Gecko-Engine haben.

PhantomJS-Objecte und Events Bei PhantomJS hingegen stehen einige Events, welche auf den beiden Referenzgeräten mit den Versionen 5.0.2 und 7.0 existieren, auf dem Objekt *top* nicht zur Verfügung. Dazu zählen *onanimation**, *oncuechange*, *ondevice**, *onlanguagechange*, *onshow*, *ontoggle*, *onwheel*, *onorientationchange*, *onclose* und *oncancel*. Dieser Umstand macht einen echten Webseitenbesucher von PhantomJS leicht unterscheidbar, sodass er zwar nicht zur eindeutigen Identifizierung, aber zur Ausschließung eines Webcrawlers benutzt werden kann.

Schriftarten

In den Browsern und Testapplikationen der Smartphones und der Emulatoren Genymotion und AVD, sind, unabhängig von der verwendeten Androidversion, die Schriftarten Arial, Courier, Courier New, Georgia, Helvetica, Monaco, Palatino, Tahoma, Times, Times New Roman, Verdana und Baskerville laut FingerprintJS2 verfügbar. Bei PhantomJS und

SlimerJS hingegen, liefert die Bibliothek die Schriftarten des aktuellen Betriebssystems zurück, sodass eine Abfrage in unterschiedlichen Ergebnissen zu Android resultiert. Unter Chrome-Dev und Ripple wiederum liefert FingerprintJS2 eine Mischung aus Schriftarten, welche sowohl in Android als auch in PhantomJS und SlimerJS vorkommen.

MIME-Typen

Das Array *mimetypes* ist ein Kind des Objekts *navigator* und spiegelt die unterstützten MIME-Typen des Browser wieder. Es kann verwendet werden, um festzustellen, welche Datentypen der Browser eines Benutzers unterstützt. Unter beiden getesteten Androidversionen, sowohl auf den Referenzgeräten als auch auf den Betriebssystemeemulatoren, ist dieses Objekt zwar Vorhanden, aber nicht mit MIME-Typ-Einträgen befüllt. Chrome unter Linux, mit aktivierten Entwicklerwerkzeugen, passt das Navigator-Objekt des Browsers bestmöglich an das von Android an, weshalb auch dort das MIME-Type-Array leer bleibt. Da PhantomJS, SlimerJS und Ripple hingegen einige MIME-Typen auflisten, können sie als Besucher ohne Androidsystem erkannt werden.

Plugins

Das Plugins-Objekt unter *navigator* stellt eine Liste an installierten Plugins des Browsers dar. Ähnlich zu den Ergebnissen der MIME-Typen, weisen nur PhantomJS, SlimerJS und Ripple eine Auswahl an Plugins auf, während alle anderen getesteten Umgebungen ein leeres Objekt besitzen.

Useragent

Der Vergleich der einzelnen Useragent-Strings gestaltet sich etwas komplexer, da bereits auf ein und dem selben Gerät, Unterschiede zwischen dem Browser und der Webview-Testapplikation zu sehen sind. Grundsätzlich ist der Useragent sehr variable und stark abhängig von der Systemumgebung, dem Browser, der Browserversion und der Systemversion. Die Konstellation eines gültigen Useragents von Chrome und Android ist in [64] definiert und folgt dem Schema

> Mozilla/5.0 (Linux; <Android Version>; <Build Tag etc.>) AppleWebKit/<WebKit Rev> (KHTML, like Gecko) Chrome/<Chrome Rev> Mobile Safari/<WebKit Rev>

Im Nachfolgenden wird, bis auf eine Ausnahme, nur der Bereich *(Linux; <Android Version>; <Build Tag etc.>)* beachtet, da die restlichen Einträge von Applikationsversionen abhängen. Hinter *<Build Tag etc.>* können, durch einen Strichpunkt getrennt, noch weitere Eigenschaften, wie *wv* für Webview, aufgelistet werden.

Nimmt man das Galaxy A3 und betrachtet den Useragent-String des Browsers,

> Mozilla/5.0 (Linux; Android 5.0.2; SAMSUNG SM-A300FU Build/LRX22G) AppleWebKit/537.36 (KHTML, like Gecko) SamsungBrowser/3.3 Chrome/38.0.2125.102 Mobile Safari/537.36

und den der Webview-Applikation

> Mozilla/5.0 (Linux; Android 5.0.2; SM-A300FU Build/LRX22G; wv) AppleWebKit/537.36 (KHTML, like Gecko) Version/4.0 Chrome/51.0.2704.81 Mobile Safari/537.36

stechen einige Unterschiede beim Build-Tag als auch bei den verwendeten Versionen, in diesem Fall farblich hinterlegt, schnell ins Auge.

Der erste Unterschied, *SM-A300FU*, betrifft den Build-Tag des Androidsystems und soll die enorme Vielfalt des übermittelten Useragent, trotz eines Standards[65], widerspiegeln. Obwohl beide Strings aus dem selben System kommen, weist der des Browsers den zusätzlichen Text *Samsung* als Teil des Build-Tags auf, sodass dadurch kein eindeutiger Rückschluss auf das Ursprungsgerät gezogen werden kann. Startet man allerdings Chrome Mobile anstelle des Standardandroidbrowsers, verschwindet diese Differenz und es bleibt nur der übereinstimmende Text *SM-A300FU Build/LRX22G* zurück. Analog verändert sich nach dem Wechsel zu Chrome Mobile der Versionsunterschied des Tags *Chrome/38.0.2125.102* zu *Chrome/51.0.2704.81*

Die nächste Differenz ist die Ergänzung *wv*, welche den Kontext einer Webview-Applikation identifiziert und auch auf dem zweiten Referenzgerät Nexus 5x vorhanden ist. Ab Androidversion 5.0[64] wird dieses Kürzel als Kennzeichen für das Android Webview Interface verwendet und kann daher verlässlich zur Erkennung dieser verwendet werden.

Der letzte Unterschied, *Version/4.0*, betrifft das aktuelle Produkt und seine Version und ist auch beim Nexus 5x zu sehen. Die Dokumentation, von Seiten Chrome und Android, ist diesbezüglich sehr dürftig, weshalb nur Vermutungen angestellt werden können. Naheliegend ist, dass Android Webview und Teile seines Useragents aus der ursprünglichen Webkit-Implementierung übernommen werden und, im Gegensatz zu Chrome, nur teilweise verändert wurden.

Bei den weiteren Vergleichen der einzelnen Umgebungen, werden Versionsunterschiede, wie *Chrome/51.0.2704.81* oder *Chrome/38.0.2125.102*, ignoriert, da sie großen Variationen unterliegen und ohnehin nicht für eine Identifizierung herangezogen werden können.

AVD-Emulatoren Die AVD-Emulatoren der Androidversionen 5.0.2 und 7.0 weisen standardmäßig den Text *Android SDK built for x86_64* gefolgt von der Android-Build-Nummer als Build-Tag auf und können daher daran erkannt werden. Allerdings lässt sich in einer Webview-Applikation der Useragent via *setUserAgentString* auf der Webview-Instanz setzen, wodurch etwaige Unterschiede zu anderen Geräten eliminiert werden können. Interessanterweise fehlt das Webview-Kürzel *wv* unter Android 5.0.2 in der Webview-Testapplikation obwohl es standardmäßig ab Android 5.0 eingefügt werden sollte.

Genymotion Die getesteten Genymotion-Emulatoren folgen dem Schema *<Geräte-Name>* - *<Android-Version>* - *<SDK-Version>* - *<Auflösung>* für den Build-Tag,

welches sich aus dem emulierten Smartphone, der genauen Androidversion und der Android SDK-Version, gefolgt von der verfügbaren Bildschirmauflösung zusammen setzt. Ein Beispiel dafür ist der Useragent-String *Google Nexus 5X - 7.0.0 - API 24 - 1080x1920 Build/NRD90M*, welcher jedoch im Kontext einer Webview-Applikation überschrieben werden kann.

PhantomJS PhantomJS verwendet die Prozessorarchitektur des unterliegenden Betriebsystem als Build-Tag und enthält darüber hinaus standardmäßig den Text *PhantomJS/<Version>* an Stelle von *Chrome/<Version>*. Allerdings ist über die Variable *page.settings.userAgent* der Useragent noch vor dem Besuch einer Webseite konfigurierbar, sodass nur uninteressierte oder unerfahrene PhantomJS-Benutzer darüber erkannt werden können.

SlimerJS Wie auch PhantomJS setzt SlimerJS den Text *SlimerJS/<Version>*, mit dem entscheidenden Unterschied, dass der Useragent nach dem Mozilla-Schema generiert wird und der SlimerJS-Identifier sich deshalb am Ende befindet. Auch hier kann leicht ein alternativer Useragent mithilfe der Variable *webpage.settings.userAgent* oder der Kommandozeilenoption *–user-agent* definiert werden.

Chrome-Dev Bei der Auswahl eines Smartphones in Chrome mit aktivierten Entwicklerwerkzeugen, wird sofort der entsprechende Useragent, welcher einem echten Gerät entspricht, gesetzt. Der definierte Text lässt keine Zweifel aufkommen, dass es sich nicht um ein Androidhandy handeln könnte.

Ripple Ripple wiederum verwendet standardmäßig den in Chrome definierten Useragent, welcher mit Prozessorarchitektur und Systeminformation des Hosts bestückt, aber abänderbar ist.

Bildschirmauflösung

Die Bildschirmauflösung ist wie der Useragent vom aktuellen Androidgerät und dem verwendeten Kontext abhängig. Während unter dem Samsung Galaxy A3 die zurückgegebene Auflösung der Browser *360x640* beträgt, liefert die Webview-Applikation einen Wert von *540x960*. Diese Differenz zwischen Browser und Webview ist bei den Smartphones als auch bei AVD und Genymotion zu finden und liefert teils enorme Unterschiede. Die gelieferten Werte des Nexus 5x, von *412x732* bei Chrome zu *1082x1922* bei Webview, sind ein besonders extremes Beispiel der möglichen Schwankungsbreite. Darüber hinaus liefern die beiden Eigenschaften *innerWidth* und *innerHeight* des Top-Objekts die Größe des aktuellen Bildausschnitts exklusive ScrollBar und Toolbar, während *outerWidth* und *outerHeight* diese inkludieren. Paradoxerweise sind die von *inner** zurückgegebenen Werte der Smartphones und Betriebssystemulatoren größer, als die der Eigenschaften *outer**. Im Gegensatz dazu sind bei Chrome ohne aktivierten Entwicklerwerkzeugen die

Dimensionen von *outer** größer als die von *inner**. Schaltet man jedoch auf die Entwicklerwerkzeuge um oder verwendet Ripple, weisen die Variablen die selben paradoxen Werte auf, wie die des Nexus 5X und Galaxy A3. Als Sonderfall gilt SlimerJS, welcher für beide Dimensionen die gleichen Werte hinterlegt hat, wohingegen PhantomJS die Zahl 0 bei *outerWidth* und *outerHeight* zurückliefert.

Grundsätzlich kann aber durch die Bildschirmauflösung alleine kaum ein Rückschluss auf das Gerät des Webseitenbesuchers gezogen werden. Vielmehr muss der Useragent in Kombination mit der Auflösung analysiert und mit einer Datenbank für Bildschirmauflösungen, wie *viewportsizes*, und einer für Useragent-String, wie *useragentstring*, verglichen werden, um zwischen einem emulierten und einem echten Gerät unterscheiden zu können.

5.1.3 Zusammenfassung der Differenzen

In der nachfolgenden Tabelle 5.1 sind einige ausgewählte Differenzen der Emulatoren gegenüber den echten Smartphones ersichtlich. Als Basis dienen die ausschließlich in beiden Smartphones Samsung Galaxy A3 und LG Nexus 5x, vorkommenden Einstellungen, Objekte, Funktionen und Events. Eine Ausnahme stellt das Event *ondevicemotion*, kurz *odm*, dar, welches beim Galaxy A3 nur über Umwege geworfen wurde, aber trotzdem in der Tabelle angeführt ist. Die Überschriften *Gef. Events* und *Vorh. Events*, spiegeln von Emulatoren gefeuerte und in ihnen verfügbare Events wider. Die Einträge welche mit der Zahl *1* markiert sind, können nur teilweise zum Ausschluss des angeführten Emulators beitragen, da sie zwar bei Android 7.0, aber nicht bei Android 5.0.2 in den Emulatoren AVD und Genymotion beobachtet wurden. Die Zeichen $+$, $\sim$ und $-$ sind gleichbedeutend mit *zusätzlich zu*, *unterschiedlich zu* und *fehlt auf Android*. Das Zeichen *, dient als Wildcard-Character und schließt alle Objekte, Eigenschaften und Events mit ein, welche mit dem Ausdruck davor beginnen, allerdings danach noch weitere Zeichen besitzen. Die Einträge **HeapSize** beziehen sich auf das Javascript-Objekt *performance.memory* und die darunterliegenden Eigenschaften *jsHeapSizeLimit*, *totalJSHeapSize* und *usedJSHeapSize*. Die Abkürzungen *webkitRFS* und *webkitRLFSU* sind Platzhalter für die Funktionen *webkitRequestFileSystem* und *webkitResolveLocalFileSystemURL*.

Gut zu erkennen ist, dass beide Betriebssystememulatoren kaum von den echten Smartphones zu unterscheiden sind, weshalb sie infolgedessen eine ausgezeichnete Grundlagen für einen Crawler bilden. Die Differenzen der kopflosen Browser gegenüber den Referenzgeräten sind geradezu enorm, weshalb der Konfigurationsaufwand zur nahtlosen Imitation eines Androidsystems größer ist. Da SlimerJS und PhantomJS allerdings zum automatisierten Testen und Besuchen von Webseiten entwickelt wurden, kompensieren sie damit ihre Konfigurationsnachteile fast komplett. Chrome-Dev und Ripple weisen hingegen einige Unterschiede bezüglich der Schriftarten, Plugins, Mimetypes und geworfenen Events auf, weshalb der Aufwand zum Verstecken dieser Eigenschaften etwas größer, aber durch die einfache Erweiterbarkeit von Chrome kompensierbar ist.

Abbildung 5.1: Auswahl an Differenzen der getesteten Emulatoren im Vergleich zu den Referenzgeräten

	Gef. Events	Vorh. Events	JS-Objekte	Schriftarten	Plugins	MimeTypes
AVD			∼ *HeapSize* = 0 - screen.orientation[1]			
Gm	∼ odm.x = 0		∼ *HeapSize* = 0 - screen.orientation[1]			
Chrome-Dev	∼ odm = null			+ Ubuntu		
Ripple	- ontouch* ∼ odm = null	- ontouch* - onorientationch.		∼ Ubuntu	+ Chrome	+ Chrome
SlimerJS	- ondevice* - ontouch*	+ ondevicelight + ondeviceproximity + onmoz* + onuserproximity - oncancel - onsearch - onclose - onorientationch. - onwebkit* - ontransitionend	+ callPhantom + moz* - *HeapSize* - speechSynthesis - webkitRFS - webkitRLFSU	+ Ubuntu	+ Firefox	+ Firefox
PhantomJS	- ondevice* - ontouch*	- onanimation* - oncancel - onclose - oncuechange - ondevice* - onlanguagech. - onorientationch. - onshow - ontoggle - onwheel	+ __phantom + callPhantom - *HeapSize* - speechSynthesis - webkitRFS - webkitRLFSU - getBattery - screen.orientation	∼ Ubuntu	+ QtWebkit	+ QtWebkit

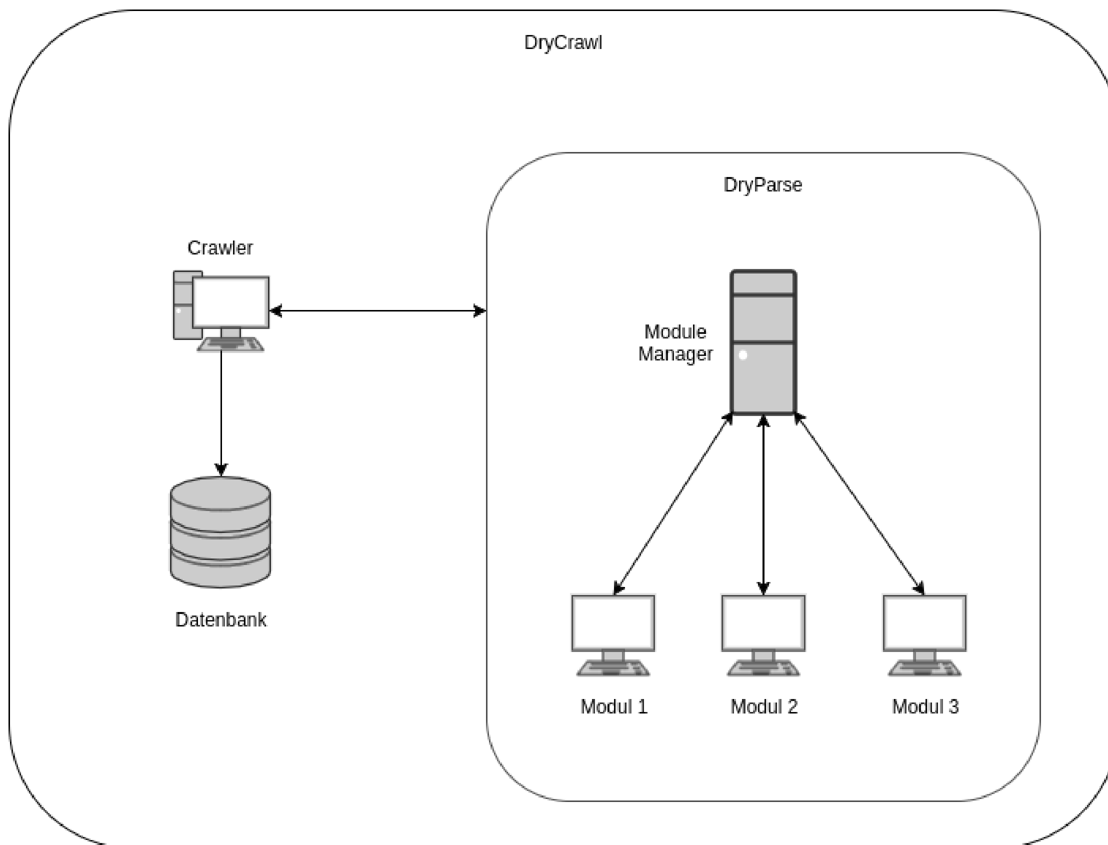


Abbildung 5.2: Aufbau von DryCrawl

5.2 DryCrawl - Design

DryCrawl besteht aus zwei Hauptkomponenten: einem Crawler und der Applikation DryParse. In Bild 5.2 ist die Interaktion der einzelnen Komponenten von DryCrawl ersichtlich.

Der Crawler kommuniziert ausschließlich mit DryParse und ist für die Durchforstung des Internets zuständig. Jeglichen Inhalt den er findet, leitet er an die Applikation DryParse weiter, welche entscheidet ob der Inhalt genauer überprüft oder verworfen wird.

5.2.1 DryCrawl - Crawler

Der Crawler speichert besuchte Webseiten und gefundene Malware in einer Datenbank ab Er durchsucht das Internet und klickt Links, Banner, Bilder und Ähnliches, um bei Bedarf Javascript auszulösen und auf böswillige Seiten weitergeleitet zu werden. Egal ob Bild, Video oder anderer Datentyp: der gefunden Inhalt wird an DryParse weitergeleitet und das Durchforsten fortgesetzt. Bewertet DryParse eine Überprüfte Datei als böswillig,

speichert der Crawler dieses Ergebnis zu der zugehörigen Webseite, um den Missing Link zwischen Malware und Malware-Verteiler zu schließen.

Da vor der eigentlichen Infizierung oft eine Geräteidentifizierung vonseiten des Angreifers passiert, muss der Crawler sich lückenlos als verwundbares Androidsystem tarnen, um schädlichen Inhalt angeboten zu bekommen. Andernfalls würden versierte Malware-Entwickler und Malvertising-Netzwerke ihre Angriffe verstecken, um ihre Exploits nicht an falsche Geräte oder Emulatoren zu schicken und eine Analyse durch Sicherheitsforscher oder Ähnliche zu riskieren.

5.2.2 DryParse

Die Applikation DryParse besteht aus mehreren Komponenten: dem Modulmanager und einer Vielzahl an Modulen. Die einzelnen Module können unterschiedliche Aufgaben bewältigen und kommunizieren ausschließlich mit dem Modulmanager, welcher die erhaltenen Resultate interpretiert.

Modulmanager

Der Modulmanager ist für die interne Organisation in DryParse verantwortlich. Er nimmt Daten wie Bilder, Videos oder Text entgegen, entscheidet abhängig vom Datentyp ob er ein passendens Modul, welches den Inhalt verarbeiten kann, geladen hat und leitet die Daten im positiven Fall an das entsprechende Modul weiter.

Das Modul verarbeitet den Inhalt und meldet dem Modulmanager ob der gesendete Content böswillig oder betrügerisch ist. Der Modulmanager speichert die Daten vom Crawler und vom Modul in einem Logfile, gibt den Rückgabewert des Moduls an den Crawler weiter und wartet anschließenden auf den nächsten Auftrag.

5.3 DryCrawl - Module

Die Module bilden das Herzstück von DryCrawl, da sie für die Erkennung von Angriffen, Täuschungen oder Geräteidentifizierung verantwortlich sind. Folgende Standardmodule soll DryCrawl beinhalten:

- StagefrightScanner
- Android LibutilsScanner
- Android Exif InterfaceScanner
- Webview Detector
- Social Engineering Detector
- Emulator Evasion Detector

Grundsätzlich können beliebig viele Module zu DryCrawl hinzugefügt werden, um bisher unbekannte Angriffe oder Techniken zu erkennen.

Module müssen sich entweder beim Modulmanager von DryParse für ein oder mehrere Datentypen oder Dateierweiterungen registrieren oder sie werden direkt im Crawler für eine Echtzeitanalyse von beispielsweise statischem Javascript integriert. Jeder Datentyp oder Dateierweiterung kann nicht exklusiv angefordert werden, sodass mehrere Module die gleichen Daten erhalten können.

5.3.1 DryParse - StagefrightScanner

Das Standardmodul *StagefrightScanner* ist für die Erkennung von Angriffen auf die Stagefright-Sicherheitslücken verantwortlich und ist teil von DryParse. Der StagefrightScanner registriert sich beim Modulmanager für die Datentypen MPEG und MP4 , um entsprechende Videodateien zu erhalten. Die übermittelten Dateien werden in ihre Einzelteile zerlegt, um anschließend die Atome stsc, ctts, stts, stss, esds, tx3g und covr zu analysieren und etwaige Angriffe zu entdecken.

Demnach werden die Sicherheitslücken mit den Nummern CVE-2015-1538, -1539, -3824, -3827 und -3829 erkannt.

5.3.2 DryParse - Android LibutilsScanner

Dieses Modul erkennt Angriffe auf die gemeldete Sicherheitslücke CVE-2016-3861, welche bei der Konvertierung von UTF-16 zu UTF-8 Text auftritt. Nachdem die Libutils-Schwachstelle viele unterschiedliche Angriffsvektoren besitzt, wird hier nur die Erkennung von einem unterstützt. Passend zum veröffentlichten Proof-of-Concept von Mark Brand[57], werden kompromitierte ID3-Tags in MP4-Dateien erkannt. Dazu registriert sich Android LibutilsScanner beim Modulmanager für die Dateierweiterung MP4.

Das Modul kann zu einem späteren Zeitpunkt um andere Angriffsvektoren wie ID3-Tags in MP3-Dateien erweitert werden.

5.3.3 DryParse - Android Exif InterfaceScanner

Der Android Exif InterfaceScanner registriert sich für die Datentypen JPG, TIFF und WAV, wobei nur der Angriffsvektor via JPG-Dateien erkannt wird. Für die Formate TIFF und WAV kann das Modul später erweitert werden, um Angriffe auf die Exif-Sicherheitslücke komplett abzudecken. Zur Erkennung von CVE-2016-3862 werden die Exif-Daten von Bilddateien ausgelesen und nach korrupten oder bösartigen Exif-Einträgen gesucht.

5.3.4 Crawler - Webview Detector

Dieses Modul erkennt Angriff auf das Webviewinterface von Android und muss direkt im Crawler integriert werden. Infolgedessen können Echtzeitanalysen durchgeführt und

Javascript-Hooks registrierte werden, um verdächtige Aufrufe auf veröffentlichte Webview-Objekte zu erkennen.

5.3.5 Crawler - Social Engineering Detector

Der Social Engineering Detector durchsucht HTML und XHTML-Dateien nach Social Engineering Tricks, welche den Benutzer zum Download von böswilliger Software oder zum Klick auf Werbung verleiten soll. Dabei sollen Techniken von TrueClick [46] und DART [45] analysiert und teilweise übernommen werden.

5.3.6 Crawler - Emulator Evasion Detector

Viele Exploits versuchen Geräte zu identifizieren, um einer Analyse durch Sicherheitsforscher zu umgehen. Deshalb werden einige Techniken von Malware-Entwicklern angewandt, um virtuelle Maschinen oder Emulatoren zu erkennen. Das Modul Emulator Evasion Detector soll genau diese Techniken erkennen, um Seiten oder Skripte zu identifizieren, welche auf virtuellen und realen Geräten unterschiedlich agieren.

5.4 DryCrawl- Implementierung

Bei der Implementierung von DryCrawl wurde beachtet, dass das Projekt leicht anpassbar, erweiterbar und auf andere Systeme portierbar ist, um den Einsatz bei der Suche nach Malvertising-Kampagnen möglichst flexibel zu gestalten. Damit zukünftige Sicherheitslücken und die daraus resultierenden DryCrawl-Module höchstmögliche Flexibilität genießen, ist eine Abhängigkeit auf ein spezielles Betriebssystem oder bestimmte Prozessorarchitektur kontraproduktiv. Diese Kriterien müssen bei der Wahl der Verwendeten Komponenten berücksichtigt werden, um eine Einschränkung von DryCrawl vorab zu verhindern.

5.4.1 Verwendete Technologien

Programmiersprache

Die nächstliegende Wahl der Programmiersprache fällt auf die systemnahe Sprache C++, da die fehlerhaften Androidsystemteile allesamt darin geschrieben wurden. Infolgedessen bleibt der Entwicklungsaufwand der Funktionen, welche einen möglichen Angriff auf die jeweilige Sicherheitslücke analysieren, relativ überschaubar, da der bestehende Androidquelltext weiterverwendet werden und nur eine Überprüfung der Parameter an den sicherheitskritischen Stellen stattfinden kann. Allerdings ist die Portabilität von C++-Applikationen im Gegensatz zu anderen Programmiersprachen, wie Java, gering, da sie für jedes Betriebssystem, welches unterstützt werden soll, einen anderen, vom Betriebssystem abhängigen Compiler zum Erstellen der Applikation benötigen. Dieser Umstand und mangelnde Entwicklungserfahrung in C++ meinerseits, welche potenzielle

neue Sicherheitslücken provozieren würde, schließt die Sprache von einem möglichen Einsatz aus.

Um von Betriebssystem und Prozessor unabhängig agieren zu können, ist die Wahl einer Hochsprache wie Java oder C#, welche ihren kompilierten Bytecode in einer virtuellen Maschine zur Ausführung bringen, naheliegend. C# ist im .NET-Framework lauffähig, dessen Quelltext 2008 von Microsoft offen gelegt und infolgedessen der Weg für eine Benutzung außerhalb des Microsoft-Ökosystems geebnet wurde. Eine Portierung des Frameworks auf Linux läuft unter dem Namen Mono, welches einigen .NET-Programmen die Ausführung außerhalb von Windows erlaubt. Wie auch das .NET-Framework, erfüllt Java die gleichen Voraussetzungen der Betriebssystem- und Prozessorunabhängigkeit und macht Applikationen unter Windows, Linux und Mac OS X lauffähig. Obwohl Java im Gegensatz zu C# keine vorzeichenlosen Datentypen wie *unsigned int* oder *unsigned long*, welche beim Überprüfen diverser Sicherheitslücken von Vorteil wären, besitzt, fällt die Wahl der Programmiersprache auf Java. Dieser Umstand ist auf meine Entwicklungserfahrung als Java-Softwareentwickler zurückzuführen, welche die Probleme bezüglich vorzeichenloser Datentypen in den Hintergrund rücken lässt.

Für die Interpretation, Verarbeitung und Vergleiche von vorzeichenlosen Datentypen aus C++, welche kleiner als ein *Long* sind, kann bedenkenlos der nächstgrößere Datentyp in Java verwendet werden. Für den C++-Typ *unsigned short* muss dementsprechend der Java-Typ *int* und für *unsigned int* der Typ *long* verwendet werden. Der C++-Typ *unsigned long* ist unter Java nur mit dem Datentyp *long* verwendbar. Demnach müssen Vergleiche von zwei vermeintlich vorzeichenlosen Zahlen vom Typ *long*, statt mit dem Größer- oder Kleiner-Operator durch eine Subtraktion und einem anschließenden Vergleich mit Null ersetzt werden.

Um die Lesbarkeit des Quelltexts zu verbessern und die Unterstützung der neuesten Funktionalitäten zu erhalten, fällt die Wahl auf Java in der Version 8, welche neben Default-Methoden und Optional-Values, die Abarbeitung von Listen via Lambda-Ausdrücken und das automatische Schließen von Ressourcen via try-with-resource Statements lesbarer gestaltet.

Build- und Dependency-Management

Für das Build- und Dependency-Management wird auf Maven[66] zurückgegriffen, welches einen modularen Projektaufbau und eine einfache Verwaltung von Abhängigkeiten erlaubt. Durch die einfache Benutzung und die große Zahl an verfügbaren, externen Bibliotheken im Maven-Repository, gestaltet sich die Suche nach und die Einbindung von diesen als trivial. Da keine lokalen Kopien von Bibliotheken von Drittanbietern nötig sind, bleibt das Projekt leicht reproduzier- und erstellbar, da externe Abhängigkeiten automatisch heruntergeladen und zur Verfügung gestellt werden. Darüber hinaus kann durch eigene Maven-Skripte der Build-Prozess an die eigenen Anforderungen angepasst und individualisiert werden, sodass projektinterne Abhängikeiten einzelner Module realisierbar sind.

Abhängigkeiten

Die Bibliothek Reflections[67] wird verwendet, um eine einfache und dynamische Verarbeitung von neuen Klassen und Packeten zu gewährleisten. Sie erweitert die Standard Java Reflection API um nützliche Funktionen wie die Suche nach Klassen, deren Namen einem bestimmten Regex-Muster entsprechen. Die Verwendung dieser Bibliothek wird in den abhängigen Modulen im Detail erklärt.

Entwicklungsumgebung

Als Entwicklungsumgebung wird IntelliJ IDEA von Jetbrains verwendet, welche Autovervollständigungen, Quelltextoptimierungen und Applikations-Debugging bietet. Durch IntelliJ's Quelltextoptimierungen, wird der geschriebene Quelltext übersichtlicher und leichter lesbar, wobei unnötige doppelte Textstellen markiert und zur automatischen Refaktorierung angeboten werden. Das Debuggen von Applikationen ermöglicht die Analyse der Abläufe der einzelnen DryCrawl-Module bei deren Verarbeitung und Interpretation einzelner, aus MP4- oder Bilddateien gelesener Bytes, welche für das Erkennen eines Angriffs entscheidend sind.

Webcrawler

Für das Durchforsten und Herunterladen von Dateien aus dem Internet, wird das Crawler-Framework Crawlium[68] verwendet, da eine entsprechende Implementierung außerhalb des Bereichs dieser Arbeit liegt. Der erwähnte Crawler basiert auf SlimerJS, übernimmt das Suchen von Seiten mit potenziellem Malvertising und imitiert Benutzerverhalten via Klicken, Scrollen und Texteingaben mithilfe einer Javascript-Api. Infolgedessen können typische an Javascript-Events geknüpfte Benutzerinteraktionen wie onClick, onDeviceMotion oder onKeyDown gefeuert werden, um die Verarbeitung von Skripten der Webseite oder integrierter Werbung anzustoßen. Die ausgeführten Skripte wiederum, liefern neue Seiten, Bilder oder andere Dateien, welche der Crawler herunterladen und an den DryParse übergeben kann. Darüber hinaus übernimmt der Webcrawler die Verwaltung der Datenbank und , mithilfe des Rückgabewerts von DryParseim Falle eines Angriffs, die Klassifizierung von Seiten und Werbungen.

Mit den Resultaten aus Kapitel 5.1 muss nun eine SlimerJS-Konfiguration erstellt werden, welche die nahtlose Imitation eines Androidsmartphones ermöglicht. Dazu werden die Werte des Samsung Galaxy A3 mit Android 5.0.2 herangezogen, um mögliche Stagefright-Angriffe zu erhalten. Um SlimerJS daran anzupassen, kann die Javascript-Umgebung in der *page.onInitialized* Funktion verändert werden. Darin können in der Funktion *page.evaluate*, noch vor dem Laden des eigentlichen Seiteninhalts, Modifikationen an den verfügbaren Javascript-Eigenschaften vorgenommen werden.

Dazu wird im ersten Schritt der Useragent von SlimerJS mithilfe von *webpage.settings.userAgent*, noch vor dem Aufruf von *page.onInitialized*, auf den Wert *Mozilla/5.0 (Linux; Android 5.0.2; SAMSUNG SM-A300FU Build/LRX22G) AppleWebKit/537.36 (KHTML, like Gecko) SamsungBrowser/3.3 Chrome/38.0.2125.102 Mobile Safari/537.36* gesetzt.

Im Anschluss sollten die Bildschirmauflösung und verfügbare Bildschirmgröße im Screen-Objekt auf *360x640*, sowie *innerWidth = 1392*, *innerHeight = 980*, *outerWidth = 360* und *outerHeight = 511* im Top-Objekt gesetzt werden.

Als Nächstes sollten die fehlenden Events, ersichtlich in Tabelle 5.1, im Top-Objekt hinzugefügt und regelmäßig mit falschen Daten beliefert werden, damit keine Zweifel bezüglich des agierenden Webseitenbesuchers aufkommen. Grundsätzlich können Event-Hanlder mit *top.addEventListener* und dem Namen sowie der Callback-Funktion als Argumente registriert werden, weshalb ein beliebiges Event, unabhängig von seiner Existenz, damit abgehört werden kann. Falls ein Skript über die in *top* registrierten Eigenschaften iteriert und auf Namen, beginnend mit *on*, vergleicht, müssen fehlende Events via *top.oncustomevent = null* erstellt werden. Dadurch wird ein Objekt mit dem Namen *oncustomevent* in *top* hinzugefügt, welches von einem tatsächlich verfügbaren Event nicht zu unterscheiden ist. Um das eben erstellte Event regelmäßig mit Werten zu beliefern, kann ein falsches Event-Objekt mithilfe *new Event('oncustomevent')* erstellt und anschließend durch *top.dispatchEvent* gefeuert werden. Damit so geworfene Events nicht als Fälschung entlarvt werden, sollte die Strutktur des zu imitierenden Events analysiert und das erstellte Objekt daran angepasst werden.

Gleichzeitig müssen mithilfe des Javascript-Befehls *delete* die überflüssigen Events *ondevicelight*, *ondeviceproximity*, *onuserproximity* und *onmoz** entfernt werden, sodass aus Sicht der Top-Events eine perfekte Imitation entsteht. Die Variable *callPhantom* sollte umbenannt oder mit *delete* gänzlich entfernt werden. Weiters müssen alle fehlenden Variablen in die Javascript-Umgebung von SlimerJS eingefügt werden, um eine Identifizierung anhand fehlender Objekte auszuschließen.

Anschließend müssen die verfügbaren MIME-Typen und Plugins entfernt werden, damit ein Abgleich darauf keinen Unterschied zu einem Androidsystem liefert. Dazu kann durch die Zeile

```
window.navigator.__defineGetter__('mimeTypes', <customFunction>);
```

der zurückgegebene Wert des MimeTypes- oder auch Plugins-Array manipuliert und eine leeres Objekt retourniert werden. Alternativ dazu kann mithilfe des Firefox-Profil-Managers ein Konfiguration erstellt werden, welche ausgewählte Plugins und MimeTypes deaktiviert, sodass die Objekte *mimeTypes* und *plugins* nicht mit Inhalten befüllt sind. Darüber hinaus kann damit die Unterstützung von Java und Flash gesteuert werden, um gleichzeitig das Aufzählen von Systemschriftarten zu verhindern.

5.4.2 Projektstruktur

Das Quelltext-Projekt ist in ein Hauptprojekt, den ExploitDetector, und mehrere Unterprojekte unterteilt, wobei das Hauptprojekt keine Funktionalität zur Verfügung stellt, sondern ausschließlich für organisatorische Zwecke dient. Jedes Unterprojekt ist ein eigenes Maven-Modul und kann als solches gesondert erstellt werden, sodass Abhängigkeiten auf Drittbibliotheken in jedem Modul direkt verwaltet werden. In der nachfolgenden

Auflistung sind die einzelnen Unterprojekte und deren Verwendung als ausführbares Modul innerhalb von DryCrawl, ersichtlich.

- Executor - implementiert den Einstiegspunkt von DryParse und den Modulmanager
- StagefrightScanner - implementiert das Modul StagefrightScanner
- ExifScanner - implementiert das Modul Android Exif InterfaceScanner
- LibUtilsScanner - implementiert das Modul Android LibutilsScanner

Die drei Projekte StagefrightScanner, ExifScanner und LibUtilsScanner, sind abhängig vom Projekt Executor, welches zusätzlich zu den Funktionalitäten als Commandline-Applikation DryParse und Modulmanager auch als Bibliothek für oft verwendete Methoden wie Konvertierungen und Byte-Vergleiche verwendet wird. Das vereinfacht die Wartung und Lesbarkeit der einzelnen Module, da der ständig wiederkehrende Quelltext oder die Standardfunktionalitäten an einem Ort zu finden sind. Darüber hinaus enthält der Executor ein Interface, welches für die Ausführung der einzelnen Module ausschlaggebend ist und infolgedessen als Abhängigkeit in jedem der Module existiert. Die Funktionsweise und Bedeutung des Interface wird später bei der Erklärung des Executor und der implementierten Module erläutert.

5.4.3 Executor - DryParse und Modulmanager

Die Implementierungen für DryParse und den Modulmanager wurden in einem Maven-Modul, dem Executor, zusammengefasst, wobei zusätzlich eine Utils-Klasse mit Standardfunktionalitäten für die anderen Module integriert ist.

DryParse

Die Realisierung von DryParse ist simpel und stellt eine Schnittstelle zur Kommunikation mit Drittapplikationen zur Verfügung. Um dies zu gewährleisten, beinhaltet das Maven-Modul *Executor* die Klasse *DryParse*, welche den Aufruf via Command-Line zulässt und einige Parameter als Konfiguration entgegen nimmt. Aufrufbar ist DryParse mit dem Kommando *dryParse <pathToFileOrDir> <pathToScanners>*, wobei zur Ausführung die zwei Argumente *pathToFileOrDir* und *pathToScanners* benötigt werden.

Der erste Parameter, *pathToFileOrDir*, beinhaltet den Pfad zu einer Datei oder einem Ordner mit Dateien, welche später von den einzelnen Scannern nach Angriffen auf bekannte Sicherheitslücken durchsucht werden.

Der zweite Parameter, *pathToScanners*, ist der Pfad zu einem Ordner, welcher die verfügbaren Scanner im Jar-Format beinhaltet.

Beide Parameter werden von DryParse an den Modulmanager weitergereicht, sodass DryParse nur eine Schnittstellenfunktion erfüllt. Diese Implementierung wurde bewusst

gewählt, sodass zukünftig weitere Schnittstellen, wie Webservices oder Rest-APIs, erstellt werden können, ohne die Geschäftslogik des Modulmanager zu duplizieren.

Modulmanager

Der Modulmanager beinhaltet die eigentliche Logik des Executors und verarbeitet die beiden Argumente *pathToFileOrDir* und *pathToScanners*. Seine Funktionalität wird in der Klasse *ModuleMangaer* zur Verfügung gestellt und unterteilt sich in drei Bereiche. Erstens in das Bestimmen des Typs von potenziell schädlichen Dateien sowie zweitens das Ermitteln und Starten der einzelnen Exploit-Scannern. Darüber hinaus beinhaltet er drittens ein Interface, welches als Vorlage für die auszuführenden Scanner verwendet werden muss. Bei seiner Instanziierung lädt er die Klassen der Jar-Dateien im Ordner *pathToScanners* in den aktuellen Classloader, damit sie später beim Ermitteln der FileScanner zur Verfügung stehen. Das dynamische Laden von externen Jar-Dateien erhöht die Erweiterbarkeit von DryCrawl, da ohne umständliche Quelltextänderungen zusätzliche Scanner registriert und verwendet werden können. Der stark vereinfachte Ablauf des Modulmanager ist in 5.1 zu sehen.

Algorithm 5.1: Vereinfachte Suche und Verarbeitung von Exploit-Scannern in der Methode checkAllFileScanners in der Klasse ModuleManager.java

```
1  Reflections reflections = new Reflections ();
2  Set <Class <? extends FileScanner > > subTypes =
     reflections.getSubTypesOf(FileScanner.class);
3  String fileType = null;
4  fileType = Files.probeContentType(new File(filePath).toPath());
5  for fileScannerClass in subTypes do
6  |   FileScanner fileScanner = fileScannerClass.newInstance();
7  |   if fileType != null && !fileScanner.isFileTypeSupported(fileType) then
8  |   |   continue;
9  |   end
10 |   boolean currentVuln = fileScanner.isVulnerable(filePath);
11 |   vulnerable |= currentVuln;
12 end
13 return vulnerable;
```

FileScanner-Interface Das Interface welches von den Scannern verpflichtend implementiert werden muss, ist in der Datei *FileScanner* definiert, und beinhaltet die Signaturen der Funktionen *isVulnerable* und *isFileTypeSupported*. *isVulnerable* übernimmt den Pfad zu der Datei welche analysiert werden soll und retourniert einen Boolean-Wert, ob die angegebene Datei schädlich ist oder nicht. *isFileTypeSupported* wiederum verlangt nach einem Datei-Typ im Textformat als Parameter, um einen Boolean-Wert zurückzugeben, welcher die Unterstützung des Typs durch den Scanner widerspiegelt.

Ermitteln der Scanner Die zur Verfügung stehende Scanner werden mithilfe der Bibliothek Reflections[67] geladen. Zu Beginn werden im aktuellen Java-Classloader alle geladenen Klassen gesucht, welche das Interface *FileScanner* implementieren. Jeder Exploit-Scanner, der von diesem Interface ableitet und im Jar-Format im Ordner *pathToScanners* liegt, wird bei diesem Schritt gefunden und für die spätere Benutzung gespeichert, was in Zeile 2 ersichtlich ist. Alle weiteren Jar-Dateien oder etwaige Exploit-Scanner, die diesen Kriterien nicht entsprechen, werden ignoriert und somit nicht für die Überprüfung von Dateien herangezogen.

Bestimmen des Typs Das Bestimmen des Typs der übergebenen Datei oder Dateien wird zentral vom Modulmanager übernommen, sodass die einzelnen Scanner nur dann gestartet werden, wenn sie den aktuellen Dateityp unterstützen. Dazu wird die Methode *probeContentType* der Java-API verwendet, ersichtlich in Zeile 4, um den MIME-Type zu ermitteln, welcher anschließend an *isFileTypeSupported* sämtlicher verfügbarer Scanner übermittelt wird. Retourniert einer der Scanner, dass er den angegeben MIME-Typ unterstützt, wird er mit der Methode *isVulnerable* gestartet.

Starten der Scanner Die Ausfürung der Scanner und Sammeln der Rückgabewerte ist in Zeile 10 ersichtlich. Der Pfad der potenziell schädlichen Datei wird an die einzelnen Scanner übergeben, welche anschließend deren Verarbeitung starten. Es wäre zwar ressourcenschonender, jede Datei nur einmalig einzulesen und anschließend ein Byte-Array mit den Daten an die Scanner weiterzuleiten, allerdings würde das die Arbeit der Scanner einschränken, da sie nur mit den Byte-Daten arbeiten können. Eine mögliche Ausführung einer potenziell schädlichen Datei durch einen Scanner, würde dadurch verhindert, was zulasten der Funktionalität und Verwendbarkeit von DryCrawl gehen würde.

Liefert einer der Exploit-Scanner ein positives Resultat zurück, wird dieses als Gesamtresultat an den Modulexekutor zurückgegeben. Für den Fall, dass genauere Informationen darüber, welcher Scanner einen Angriff erkannt hat, erforderlich sind, schreibt der Modulmanager für jedes gelieferte Resultat einen Log-Eintrag mit dem Pfad der Datei, dem Namen des Scanners und seinem Rückgabewert.

Utils-Klassen

Einige häufig benutzte Funktionen, wie das Einlesen der Bytes einer Datei oder das Überspringen des Chunk-Headers in einer MP4-Datei, wurden in die Klasse *Utils* ausgelagert, damit redundanter Quelltext vermieden werden kann. Einige Methoden, wie die Konvertierung eines ByteBuffers in einen vorzeichenlosen Integer, wurden teilweise weiterverwendet, abgewandelt oder zur Gänze von der Bibliothek Mp4Parser[69] übernommen.

Weiters wurde die Klasse Mp4Parser implementiert, welche eine Mp4-Datei und die darin enthaltenen Chunks liest. Dies war erforderlich, da bestehende Mp4Parser wie [69], modifizierte Mp4-Dateien nicht korrekt einlesen konnten, und so eine Analyse auf einen potenziellen Angriff nicht möglich war. Der Grund dafür ist, dass Mp4-Chunks

nacheinander in einer Datei gespeichert sind, sodass die Größe eines Mp4-Chunks den Offset zum nächsten Chunk darstellt. Wird nun eine Mp4-Datei mit korrupten Größen, wie denen der Stagefright-Exploits, eingelesen, kann es während der Verarbeitung der Chunks zu einem Fehler kommen, da der Parser an einer ungültigen Stelle zum Lesen des neuen Chunks beginnt. Daraufhin beendet der Parser die weitere Abarbeitung und die Datei wird für ungültig erklärt.

Der Quelltext der Klasse Mp4Parser, ist an den Parser [69] angelehnt, allerdings mit einer entscheidenden Änderung: Im Gegensatz zu [69], ist eine sequentielle Abarbeitung der Chunks nötig, wobei bei jedem gelesenen Chunk noch vor der weiteren Verarbeitung auf einen Angriff geprüft werden muss, da es möglicherweise beim Übergang zum nächsten Chunk zu einem Fehler kommt. Um eine möglichst generische Verarbeitung und Bereitstellung der Mp4-Chunks zu ermöglichen, wird nach dem Einlesen jedes Chunk-Headers, eine Callback-Funktion aufgerufen, welche der Aufrufer des Mp4Parsers definiert. Infolgedessen ist die Analyse einer korrupten oder böswilligen Mp4-Datei, noch vor dem Auftreten eines Fehlers bezüglich des ungültigen Offsets, realisierbar.

5.4.4 StagefrightScanner

Das Maven-Modul StagefrightScanner beinhaltet die Funktionalitäten des gleichnamigen DryCrawl-Moduls und gliedert sich in eine Hauptklasse, den StagefrightScanner, und die spezialisierten Nebenklassen COVRScanner, ESDescriptorScanner, SampleTableScanner, ThreeGPPScanner und Tx3gScanner. Während die Klasse StagefrightScanner das Modulmanager-Interface FileScanner implementiert, beinhaltet das Modul noch zusätzlich die abstrakte Klasse VulnerabilityScanner, welche von den einzelnen Nebenklassen erweitert wird.

Wie auch das Modul Executor, besitzt das StagefrightScanner-Modul eine Abhängigkeit auf die Bibliothek Refelections [67], damit Klassen zur Laufzeit dynamisch anhand ihres Supertyps gesucht und ausgeführt werden können. Im Gegensatz zum Modulmanager, werden hier allerdings keine externen Objekte geladen, sondern alle packetinternen Nebenklassen, welche von VulnerabilityScanner ableiten. In diesem Fall, übernimmt die Reflections-Api dementsprechend nicht die Funktion zum dynamischen Erweitern des StagefrightScanners, sondern eher eine kosmetische, um die gleichbleibenden Methodenaufrufe an die einzelnen VulnerabilityScanner übersichtlicher zu gestalten. Der Quelltext dazu ist ähnlich zu dem des Modulmanagers, weshalb nicht näher darauf eingegangen wird.

Damit die StagefrightScanner-Klasse den Richtlinien ihres Interfaces entspricht, implementiert sie die Methode *isFileTypeSupported*, sodass sie bei den MIME-Typen *video/mpeg*, *audio/mpeg*, *video/x-mpeg*, *audio/x-mpeg*, *video/mp4* und *video/mpeg4* ein positives Resultat retourniert.

Darüber hinaus implementiert sie die Methode *isVulnerable*, in der zu Beginn nach dem Laden der Stagefright-VulnerabilityScanner, ein Objekt der Klasse Mp4Parser des Executor-Moduls instanziert wird. Mithilfe des Parsers wird das Video, auf das der

übergebene Pfad zeigt, geladen, um anschließend nach Angriffen auf eine der Stagefright-Sicherheitslücken zu suchen. Ein Auszug aus *isVulnerable* ist in Skript 5.2 zu sehen, wobei die Variable *vulnerabilityScanners* ein Set, befüllt mit den verfügbaren Stagefright-VulnerabilityScannern, darstellt. In Zeile 3 wird die Methode parseMp4 des Mp4Parsers

Algorithm 5.2: Auszug aus StagefrightScanner.java bei der Verarbeitung einer Mp4-Datei

```
1  final boolean [] vuln = {false};
2  Mp4Parser mp4Parser = new Mp4Parser();
3  mp4Parser.parseMp4(fileName,
4  (FileChannel fileChannel, Long chunkSize, String chunkType, Long
     extendedChunkSize, Long chunkDataSize) ->
5      vulnerabilityScanners.forEach( vulnerabilityScanner ->
6          vuln[0] |= vulnerabilityScanner.isVulnerable(fileChannel, chunkSize,
             chunkType, extendedChunkSize, chunkDataSize);
7      )
8      return vuln[0];
9  );
10 return vuln[0];
```

mithilfe einer Java-Lambda-Expression aufgerufen, sodass die Zeilen 5 bis 7 bei jeder Iteration über die Chunks der Mp4-Datei, ausgeführt werden. Das hat zur Folge, dass jedes gefundene Chunk von jedem Stagefright-VulnerabilityScanner auf böswillige Veränderungen überprüft wird. Die Resultate der Scanner werden mit einer Adjunktion verknüpft, sodass bei einem positiven Resultat eines einzelnen Scanners bezüglich eines Chunks, das gesamte Ergebnis positiv und somit die komplette Datei als potenziell schädlich gewertet wird. Das vereinte Ergebnis wird nach dem Verarbeiten aller Chunks an den Aufrufer retourniert, wobei die Datei trotz eines vorzeitigen Fundes nach weiteren Angriffen durchsucht wird. Infolgedessen schreiben die jeweiligen Scanner bei einem positiven Resultat zusätzlich einen Log-Eintrag, sodass die StagefrightScanner-Klasse die Resultate nur kombinieren aber nicht verarbeiten muss.

Um die fehlerfreie Verarbeitung und korrekte Erkennung von manipulierten Mp4-Dateien zu gewährleisten, wurden die Proof-of-Concepts[70] der *Zimperium Handset Alliance (ZHA)* zum Testen verwendet. Das erwähnte Zip-Archiv enthält insgesamt zehn Proof-of-Concepts in Form von MP4-Dateien, welche jeweils einer oder mehreren Stagefright-Sicherheitslücken zugeordnet werden können, sodass sie in Summe alle gemeldeten Schwachstellen abdecken. Die manipulierten Videodateien enthalten allerdings keinen Schadcode, sondern weisen nur die nötigen Veränderungen der einzelnen Chunks, welche die Stagefright-Bibliothek zum Absturz bringen, auf.

COVRScanner

Der COVRScanner erkennt Angriffe auf die gemeldeten Sicherheitslücken *CVE-2015-3829* und *CVE-2015-3827* und ist in der Klasse COVRScanner.java implementiert. Da der COVRScanner für jeden Chunk einer MP4-Datei gestartet wird, muss am Anfang seiner Methode *isVulnerable* auf den korrekten Chunk-Typ geprüft werden. Ist der aktuelle Chunk nicht vom Typ *covr*, wird die weitere Verarbeitung abgebrochen und ein negatives Resultat retourniert. Im verbleibenden Fall werden die Unterroutinen *checkIntegerOverflow* und *chekIntegerUnderflow* aufgerufen, welche auf böswillige Manipulationen der Chunk-Data-Size überprüfen. Ist sie größer als 0xFFFFFFFF oder kleiner gleich 16, wird ein Log-Eintrag geschrieben und ein positives Resultat zurückgeliefert.

ESDescriptorScanner

Der ESDescriptorScanner analysiert Chunks des Typs *esds* bezüglich böswilligen Manipulationen hinsichtlich der Sicherheitslücke CVE-2015-1539 und ist in der Klasse ESDescriptorScanner.java implementiert. Wie der COVRScanner, bricht er die Verarbeitung beim Eintreffen falscher Chunk-Typen ab. Im Gegensatz zu seinem Vorreiter, ist die Analyse des Chunks allerdings erheblich komplexer, da die Struktur des eingebetteten Elementary Stream Descriptors, kurz ESDS oder ES-Descriptor, beachtet und eingelesen werden muss. Da - ähnlich zum Auslesen der Chunks einer Mp4-Datei - eine bestehende Bibliothek manipulierte ES-Descriptor eventuell nicht jedes mal fehlerfrei verarbeiten kann, werden die Felder des ES-Descriptor-Headers manuell eingelesen und auf Korrektheit geprüft. Dazu werden die einzelnen ESDS-Flags und die damit verbundenen Längen extrahiert und kombiniert, sodass anschließend ein Vergleich anhand der Tabelle 4.3 in Kapitel 7 erfolgen konnte. Widerspricht der kalkulierte Wert, abhängig der gesetzten Flags, einem Eintrag in der Tabelle, wurde der ESDS-Chunk vermutlich böswillig manipuliert und ein Log-Eintrag wird geschrieben. Anschließend wird ein positives Resultat an den Aufrufer retourniert und der Scanner beendet.

SampleTableScanner

Der SampleTableScanner durchsucht Chunks nach Angriffen auf eine der Schwachstellen aus CVE-2015-1538. Da CVE-2015-1538 aus vier Teilen besteht, wird im Gegensatz zu den anderen Scannern, die Ausführung statt bei einem Chunk-Typ, bei den vier Typen *stsc*, *stss*, *ctts* und *stts* fortgesetzt. Dazu wurden in der Klasse SampleTableScanner.java die Methoden *isVulnerable*, *isSTTS*, *isSTSS*, *isSTSC* und *isCTTS* implementiert, welche, bis auf *isVulnerable*, die Bytes an den Stellen 12-16 mit einem vordefinierten Wert vergleichen. *isVulnerable* hingegen delegiert die Verarbeitung an die einzelnen Methoden weiter und kombiniert anschließend deren Resultate mit einer Adjunktion. Die Grenzwerte für eine Einstufung als nicht böswilliger Chunk, sind in Kapitel 4.1.1 und seinen Unterkapiteln ersichtlich. Wird eine dieser Grenzen überschritten, wird ein Log-Eintrag geschrieben und ein positives Resultat retourniert.

ThreeGPPScanner

Der ThreeGPPScanner übernimmt die Suche nach böswilligen Manipulationen hinsichtlich der gemeldeten Sicherheitslücken CVE-2015-3826 sowie CVE-2015-3828 und ist in der gleichnamigen Javaklasse implementiert. Wird einer der Chunk-Typen *titl*, *perf*, *auth*, *genre* und *albm* an *isVulnerable* übergeben, startet der ThreeGPPScanner die Methoden *checkBufferOverread* und *checkIntegerUnderflow*, um einen potentiellen Angriff zu erkennen. *checkBufferOverread* überprüft das Chunk nach Angriffen auf CVE-2015-3826 und durchkämmt den Chunk-Speicher dementsprechend nach einem Null-Byte. Wird bis zum Ende des Speichers kein konformes Byte gefunden, wird ein positives Resultat retourniert. *checkIntegerUnderflow* hingegen verifiziert, passend zu CVE-2015-3828, die Ungleichung $3 < \text{Chunk-Data-Size} < 6$ und retourniert den Wahrheitsgehalt ihres Ergebnisses.

Tx3gScanner

In der gleichnamigen Klasse wurden Angriffe auf die Sicherheitslücke CVE-2015-3824 im Tx3g-Chunk untersucht. Im Gegensatz zu den anderen Scanneren, behält der Tx3gScanner die Summe der Größen der zuvor gelesenen Tx3g-Chunks in einer temporären Variable. Dies ist möglich, da alle VulnerabilityScanner pro Instanzierung des Stagefright-Moduls einmalig erstellt und somit erst nach dem kompletten Durchlaufen einer Mp4-Datei beendet werden. Da die Summe der vorangegangenen Tx3g-Chunks den positiven Wertebereich des Java-Datentyps Long, 0x7FFFFFFFFFFFFFFF, mithilfe der Extended Chunk Size überschreiten kann, muss der Grenzwertvergleich mit 0xFFFFFFFF anhand der Methode *Long.compareUnsigned(long a, long b)* erfolgen. Infolgedessen werden bei den beiden übergebenen Zahlen die Vorzeichen ignoriert und ein Vergleich von Zahlen mit gesetztem 64-igsten Bit ist problemlos möglich.

5.4.5 ExifScanner

Das Maven-Modul ExifScanner liefert die Implementierung zum gleichnamigen DryCrawl-Modul und ist in der Klasse ExifScanner.java gespeichert. Sie implementiert das Interface FileScanner und liefert beim Aufruf von *isFileTypeSupported* ein positives Resultat für die MIME-Typen *image/jpeg*, *image/jpg*, *image/pjpeg* und *video/x-motion-jpeg* zurück. Um einen Angriff auf das Android Exif Interface zu erkennen, werden, wie in Kapitel 4.1.3 skizziert, die Exif-Daten einer JPEG-Datei gelesen.

Da eine externe Bibliothek eine JPEG-Datei mit korrupten Exif-Einträge eventuell fehlerhaft verarbeitet, bleibt die Implementierung eines eigenen, abgespeckten JPEG-Parsers offen. Das Gewähr leisten einer korrekten Verarbeitung von manipulierten Exif-Einträgen verdeutlicht den Vorteil eines individuell implementierten Parsers, während der steigende Entwicklungsaufwand des Moduls diesen wieder schmälert. Ungeachtet dessen werden einige Exif-Parser von Drittanbietern getestet, um ein fehlerfreies Parsen der Exif-Daten nicht auszuschließen. Der reine Exif-Reader Exif[71], die multifunktionellen Bildverarbeitungsbibliotheken Metadata-Extractor[72] und Apache Commons Imaging[73], liefern bei der Verarbeitung eines korrupten Exif-Eintrags keine oder nur fehlerhafte Daten

zurück. Darüber hinaus ist es mithilfe dieser Bibliotheken nicht möglich auf die eigentlich ausschlaggebenden Daten, das Format und die Komponenten, eines Eintrags zuzugreifen. Infolgedessen rückt die ursprüngliche Idee eines eigenen Exif-Parsers erneut in den Fokus und eine abgespeckte Klasse wird kurzerhand implementiert. Um die Struktur des Exif-File-Formats korrekt zu verarbeiten, wird das Exif-File-Format mithilfe des Standards der *Japan Electronic and Information Technology Industries Association* analysiert.

Um den Richtlinien des Interfaces FileScanner zu entsprechen, wird der Exif-Parser in der Methode *isVulnerable* implementiert. Der Exif-Parser ist an die Exif-Implementierung von Android angelehnt, beinhaltet jedoch keine spezielle Handhabung der einzelnen Exif-Einträge, sodass GPS-Daten, Text und andere Bildmetadaten ignoriert und ausschließlich die Felder Tag, Format und Komponenten eines jeden Eintrags gelesen werden. Der Parser identifiziert den Beginn der gesamten Exif-Daten, den Exif-Header, anhand der magischen Bytes 0xFF und 0xE1, gefolgt von der Länge der Exif-Daten, bestehend aus zwei Bytes. Die Länge selbst wird jedoch nicht zur Identifizierung des Exif-Headers herbeigezogen, da der beinhaltete Wert variabel ist. Entsprechen die nachfolgenden sechs Bytes nicht dem Text *Exif*, gefolgt von zwei Null-Bytes, wird der Exif-Header als ungültig gewertet und die Verarbeitung abgebrochen. Wurden die genannten Kriterien erfüllt, werden die restlichen Daten des Exif-Headers, die Byte-Reihenfolge und der Offset zum ersten Exif-Eintrag, extrahiert. Wenn die Byte-Reihenfolge dem Wert MM entspricht, wird von der Motorola-Byte-Reihenfolge gesprochen und alle folgenden Bytes werden im Big-Endian-Format gelesen. Ansonsten wird das Little-Endian-Format benutzt. Stehen die Byte-Reihenfolge und der Offset fest, wird die Verarbeitung des ersten Exif-Eintrags gestartet. Da Exif-Einträge, ähnlich zu MP4-Chunks, ineinander verschachtelt sein können, wird eine rekursive Funktion, processExifDir, implementiert. Sie iteriert über alle Einträge der aktuellen Ebene und ruft sich gegebenenfalls mit Untereinträgen erneut auf. Zu Beginn jedes Eintrags wird sein Tag, das Format und seine Komponenten gelesen und anhand dieser Informationen die Größe des benötigten Puffers, *byteCount*, kalkuliert. Ein Ausschnitt aus der Methode processExifDir mit dem zugehörigen Quelltext, ist in Skript 5.3 zu sehen, wobei das rekursive Verarbeiten der Einträge vernachlässigt wurde. Darin werden zwei Hilfsfunktionen, get16u und get32u, zum Einlesen von vorzeichenlosen Zahlen aus Byte-Arrays der Länge zwei und vier verwendet. Die Variablen *tag* und *format* können eine Zahl der Größe eines Integer beinhalten, während *components* und *byteCount* vom Typ Long sind. Zum Zeitpunkt einer Iteration, ist *exifData* ein Byte-Array befüllt mit den Daten des aktuellen Exif-Eintrags, womit *dirEntry* einen Zeiger auf den Beginn des Arrays darstellt. Nach der Kalkulation der benötigten Speichergröße *byteCount* in Zeile 4, ist in Zeile 7 der entscheidende Vergleich, definiert in Kapitel 4.1.3, für die Erkennung eines manipulierten Exif-Eintrags ersichtlich. Übersteigt die Summe von *offsetVal* und *byteCount* die Zahl 0xFFFFFFFF, bricht der ExifScanner die restliche Verarbeitung ab und gibt ein positives Resultat zurück.

Im Gegensatz zu den Stagefright-Sicherheitslücken, wurde keine Proof-of-Concept veröffentlicht, weshalb der Sicherheitsforscher Tim Strazzere kontaktiert wurde. Bedauerlicherweise war die Kontaktaufnahme nicht erfolgreich, sodass kurzerhand individuelle

Algorithm 5.3: Auszug aus der Funktion processExifDir in der Klasse ExifScanner.java beim Auslesen und Berechnen von Tag, Format, Komponenten und ByteCount

```
1  tag = get16u(new byte []{exifData[dirEntry], exifData[dirEntry + 1]});
2  format = get16u(new byte []{exifData[dirEntry + 2], exifData[dirEntry + 3]});
3  components = get32u(new byte []{exifData[dirEntry + 4], exifData[dirEntry +
     5], exifData[dirEntry + 6], exifData[dirEntry + 7]});
4  byteCount = components * bytesPerFormat[format];
5  if byteCount > 4 then
6  |  long offsetVal; offsetVal = get32u(Arrays.copyOfRange(exifData, dirEntry
   |    + 8, dirEntry + 8 + 4));
7  |  if Long.compareUnsigned(offsetVal + byteCount, 0xffffffffL) > 0 then
8  |  |  return True;
9  |  end
10 end
```

Testbilder zum Provozieren des Zahlenüberlaufs erstellt wurden. Die generierten JPEG-Dateien, veränderte Screenshots meines Desktops, beinhalten jedoch keine richtigen Exploits, sondern weisen bloß eine Veränderung der Felder *components* und *offsetVal* eines einzelnen Exif-Eintrags auf. Mithilfe dieser manipulierten Bilddateien, konnte anschließend ein fehlerfreier Ablauf und eine korrekte Erkennung des ExifScanners garantiert werden.

5.4.6 LibUtilsScanner

Das Maven-Modul LibUtilsScanner liefert die Implementierung zum gleichnamigen DryCrawl-Modul und bildet seine Hauptfunktionalität in der Klasse LibUtilsScanner.java ab. Ähnlich zu den anderen Scanner leitet der LibUtilsScanner vom Interface FileScanner ab, um Angriffe durch Mp4-Dateien mit den MIME-Typen *video/mpeg*, *audio/mpeg*, *video/x-mpeg*, *audio/x-mpeg*, *video/mp4"* und *video/mpeg4* zu erkennen. Darüber hinaus ist eine Utils-Klasse, LibUtilsUtils.java, enhalten, welche zusätzliche Hilfsfunktionen implementiert.

Wie in Kapitel 4.1.2 bereits im Detail beschrieben, muss der LibUtilsScanner ID3-Tags einer Mp4-Datei analysieren, um Manipulationen oder Angriffe erfolgreich erkennen zu können. Diesbezüglich wurde vorab ein Testvideo mithilfe des Proof-of-Concept Skripts von Mark Brand [54] generiert. Um falsch-positive Erkennungen von Angriffen auszuschließen, musste im nächsten Schritt ein Referenzvideo mit gültigen ID3-Tags erstellt werden. Die dazu getesteten Programme EasyTag[74] und IDTE[75] setzten Mp4-Chunks wie *titl* und *albm* zwar erfolgreich, die Generierung von ID3-Tags blieb bedauerlicherweise jedoch ohne Erfolg. In der Hoffnung mit einer bestehenden Java-Bibliothek ID3-Tags in Mp4-Dateien erzeugen und lesen zu können, wurde das Internet

nach Mp4-ID3-Tag Parsern durchsucht. Trotz der großen Anzahl an ID3-Parsern für Audiodateien, konnte keine lauffähiger für Videodateien gefunden werden. Infolgedessen musste, wie bei den Scannern StagefrightScanner und ExifScanner zuvor, ein ID3-Parser für Mp4-Dateien geschrieben werden.

Anhand der Definition für ID3-Tags in einem ID32-Chunk [76], wurden die ID3-Daten des Chunks mithilfe des Mp4-Parsers, definiert in Kapitel 5.4.3, eingelesen. Anschließend wurde der ID3-Parser mithilfe des ID3-Standards der Version 2.2 [77] und dem Quelltext der Android Stagefright Bibliothek implementiert. Der ID3-Parser identifiziert einen gültigen ID3-Header anhand der magischen Bytes *id3* sowie den anschließenden Flags, zusammengefasst in einem Byte. Da die Flags laut Definition ausschließlich in den Bits sieben und acht gespeichert sind, kann mithilfe der restlichen Bits die Konformität des ID3-Headers geprüft werden. Ist das Byte konjugiert mit 0x3F verschieden zu 0x00, gilt der Header als ungültig und der Parser bricht die weitere Verarbeitung ab. Die nächsten drei Byte im ID3-Header, die Länge der gesamten ID3-Daten, markieren das Ende des ID3-Headers und sind für das korrekte Arbeiten des LibUtilsScanner belanglos, weshalb sie kurzerhand übersprungen werden. Unmittelbar danach, beginnt die Anordnung der einzelnen ID3-Tags, welche als Frame bezeichnet werden. Die einzelnen Frames liegen im Speicher direkt aufeinander folgend, sodass das Ende eines Frames den Beginn eines neuen oder das Ende der gesamten ID3-Daten bedeutet. Um sämtliche Frames rekursiv zu lesen, wurde eine Prozedur, welche über alle Frames und deren Header iteriert, implementiert. Neben dem Start der Verarbeitung von Kind-Tags, übernimmt die Funktion die immer gleichen Aufgaben des Einlesen des Frame-Typs, das Kalkulieren der Frame-Größe und das Verarbeiten der Codierung.

Wurden diese Daten erfolgreich extrahiert, wird die erste Voraussetzung, mit dem Vergleich der Codierung und den Werten 0x01 sowie 0x02, für einen erfolgreichen Angriff überprüft. Infolgedessen werden nur Frames beachtet, welche UTF-16 codierte Unicode-Strings mit und ohne Byte-Order-Mark sind. Entspricht die Codierung einem dieser Werte, wird im nächsten Schritt der Typ des aktuellen Frames überprüft. Da die LibUtils-Sicherheitslücke bei der Verarbeitung von UTF-Text auftritt, werden ausschließlich Text-Frames vom ID3-Parser berücksichtigt. Text-Frames können anhand ihres Typs, bestehend aus drei groß geschriebenen alphanumerischen Zeichen beginnend mit einem T, identifiziert werden und reichen von dem Typ *T00* bis *TZZ* exklusive *TXX*. Diesbezüglich wird die Funktion *getTags* implementiert, welche einmalig alle gültigen Text-Frame Typen generiert, sodass später ressourcenschonend ein Text-Frame erkannt werden kann. Ist der aktuelle Frame kein Text-Frame, wird erneut die Verarbeitung beim nächsten fortgesetzt.

Anschließend an den Frame-Header folgen die kodierten Frame-Daten, welche, im Android Quelltext, an die eigentlich mit Fehlern behafteten Methoden *utf16_to_utf8* und *utf16_to_utf8_length* weitergegeben werden. Analog zur Androidimplementierung, ruft der ID3-Parser die beiden teils modifizierten Methoden *utf16ToUtf8* und *utf16ToUtf8Lenth* auf. Die ursprüngliche Implementierung von *utf16_to_utf8_length*, beschrieben in Kapitel 4.1.2, wurde, bis auf einige C++-Sprachen typische Ausdrücke, gänzlich übernommen. Da die Differenz des allozierten und verbrauchten Puffers durch eine unterschiedliche

Handhabung von fehlerhaften Surrogate Pairs entsteht, wurde die Methode *utf16_to_utf8* dahingehend modifiziert, dass sie statt in den Puffer zu schreiben, die verbrauchten Bytes als Summe retourniert. Die veränderte Implementierung von *utf16_to_utf8* aus der Klasse LibUtilsUtils.java ist in Skript 5.4 ersichtlich. Während die Zeilen 3 und 4 lediglich

Algorithm 5.4: Auszug aus der Methode utf16ToUtf8 in der Klasse LibUtilsUtils.java bei der Kalkulation des Verbrauchten Puffers

```
1  int counter = 0;
2  while counter + 3< src_len do
3      long firstPair = Utils.getUnsignedInt(Arrays.copyOfRange(src, counter,
        counter + 2));
4      if (firstPair & 0xFC00) == 0xD800 && (counter + 3) < src_len &&
        (Utils.getUnsignedInt(Arrays.copyOfRange(src, counter + 2, counter +
        4)) & 0xFC00) == 0xDC00 then
5          counter += 4;
6      else
7          counter += 2;
8      end
9  end
10 return counter;
```

zu Java-Syntax transformiert wurden, ist in den Zeilen 5 und 7 das Inkrementieren des Zählers gut erkennbar. Darüber hinaus wurde nach dem If-Statement in Zeile 9, das Konvertieren des eingelesenen UTF-16-Zeichen und das Schreiben in den Puffer entfernt, da sie für die Erkennung eines Angriffs unerheblich sind.

Die retournierten Werte der Methoden *utf16ToUtf8* und *utf16ToUtf8Length* werden anschließend im ID3-Parser des LibUtilsScanners verglichen. Der zugehörige Quelltext der Klasse LibUtilsScanner.java ist in Skript 5.5 ersichtlich.

Algorithm 5.5: Auszug aus der Methode isVulnerable in der Klasse LibUtilsScanner.java bei der Kalkulation und dem Vergleich der Längen

```
1 byte [] tagContent = Arrays.copyOfRange(id3Bytes, id3Start, mFrameSize);
2 long calculatedLength = LibUtilsUtils.utf16ToUtf8Length(tagContent,
   tagContent.length);
3 long copyLength = LibUtilsUtils.utf16ToUtf8(tagContent, tagContent.length);
4 if copyLenght != calculatedLength then
5     return true;
6 end
```

Schlägt der Vergleich in Zeile 4 fehl, muss von einem böswillig manipulierten ID3-Tag ausgegangen werden und die Methode *isVulnerable* liefert ein positives Resultat an seinen Aufrufer zurück.

KAPITEL 6

Disskusion

Nach der ausführlichen Veranschaulichung der Implementierung des Webcrawlers und seiner Module zur Erkennung von Drive-by-Downloads, bildet dieses Kapitel die Zusammenfassung und Interpretation der gesammelten Informationen.

Android Remote Exploits

Die detaillierte Darstellung der Sicherheitslücken Stagefright, LibUtils und ExifInterface vermittelt einen Einblick in mögliche Remote-Exploits des Androidbetriebssystems. Die Schwachstelle Stagefright kann anhand von manipulierten MP4-Dateien als Angriffsvektor für die Infizierung und Kompromittierung von Smartphones missbraucht werden. Gleichzeitig führt bereits die gezielte Abwandlung individueller Bytes eines fehlerhaften MP4-Atoms zum Überlauf des allozierten Speichers und öffnet somit die Tore für die Ausführung von schädlichem Code. Ob die Videodatei per MMS, E-Mail oder über den Webbrowser empfangen wird, ist dabei unwesentlich, da diese Quellen allesamt die Verarbeitung durch den Stagefright-Medienserver anstoßen. Die ID3-Tags einer MP4-Datei bieten anlässlich des LibUtils-Fehlers zusätzliche Angriffsflächen für Malware-Schreiber, unerkannt Banking-Trojaner, Spyware oder andere böswillige Programme auf einem verwundbaren System zu platzieren. Der Ursprung von LibUtils liegt in der fehlerhaften Verarbeitung von UTF-kodierten Zeichen innerhalb eines ID3-Tags und resultiert aufs Neue mit der Option auf das Einschleusen von kompromittierendem Schadcode. Der Angriffsvektor via MP4-Datei ist jedoch nicht maßgeblich für eine erfolgreiche Infizierung eines Androidgeräts, da die verwundbaren Funktionen der LibUtils-Klasse in weiteren Androidservices, wie dem system_server und dem keystore, verwendet werden. Des Weiteren zeigen die Untersuchungen, dass Android, neben der problematischen Verarbeitung von MP4-Dateien, auch sicherheitskritische Fehler bei der Handhabung von Bilddateien aufweist. Fotos und Bilder im JPEG-Format werden nach dem Eintreffen im Androidsystem automatisch nach vorhandenen Bild-Metadaten, wie Exif-Daten, durchsucht, um zusätzliche Bildinformationen schnellstmöglich anzeigen zu können. Beim

Parsen der Exif-Daten wird der fehlerhafte Quelltext verwendet, welcher die Schwachstelle im Exif-Interface verursacht. Erneut erhalten Malware-Entwickler auf diese Weise einen ausgeklügelten Angriffsvektor, um eigenen Schadcode im Kontext eines fremden Androidgeräts auszuführen.

Im Bezug auf Pagehijacking-Angriffe ist das Sicherheitskonzept des Android Webview Interfaces maßgeblich. Durch die sorglose Verwendung der Methode *loadUrl* eines App-Entwicklers, kann böswilliges Javascript, eingebettet in eine iFrame, die Same-Origin-Policy der WebKit-Engine umgehen und im Kontext der Webseite schädlichen Code ausführen. Infolgedessen können Angreifer einen Webseitenbesucher durch Setzen der *window.location*-Variable auf eine böswillige Webseite weiterleiten, um mithilfe von Social Engineering Techniken den Download einer schädlichen Datei zu forcieren. Gleichzeitig ermöglicht die Veröffentlichung von Java-Objekten mittels der Funktion *addJavascriptInterface* den Zugriff auf die Java Reflection API. Die Verwendung von *getClass* erlaubt in weiterer Folge die Ausführung von beliebigen Methoden auf Java-Objekten, unter anderem Aufrufe auf das Java-Objekt Runtime, welche die Interaktion mit dem lokalen Dateisystem gestatten. Das Erfassen von böswilligem Code im Bezug auf die Schwachstelle muss im Kontext einer Webseite stattfinden, da mithilfe eines Javascript-Hooks alle registrierten Javascript-Objekte überwacht werden müssen. Der Javascript-Hook belauscht dabei Aufrufe auf die Methode *getClass*, welche als Einstiegspunkt für eine Kette von Java Reflection Befehlen dient. Hinsichtlich dieser Tatsachen muss ein Webcrawler bei der Durchforstung des Internets diese Aufgabe übernehmen, um Angriffe auf das Android Webview Interface zu erkennen.

Proof-of-Concept Implementierung

Darüber hinaus zeigen die Untersuchungen dieser Arbeit, dass Angriffe auf die beschriebenen Sicherheitslücken mithilfe der entwickelten Commandline-Applikation DryParse problemlos erkannt werden können. Um manipulierte MP4- und JPEG-Dateien von harmlosen Videos und Bildern zu unterscheiden, ist die Kontrolle einzelner Byte-Reihenfolgen ausreichend. Im Vordergrund stehen dabei vier-Byte-lange Werte, welche im fehlerhaften Quelltext der Schwachstellen, nach der Multiplikation oder Addition mit Konstanten, in einem Zahlenüber- oder Unterlauf resultieren. Im Gegensatz zu signaturbasierten Systemen erkennt DryParse deshalb auch zukünftige, noch unbekannte Angriffe, da sie unabhängig des eigentlichen Schadcodes ihre Validierungen vornimmt. Kombiniert mit einem Webcrawler, bildet das Programm das Framework DryCrawl, welches Drive-by-Downloads und Malvertising Kampagnen automatisch aufspürt. Dabei untersucht die Applikation sämtliche Bild- und Videodateien auf Angriffe, während der Webcrawler mithilfe der daraus gewonnenen Informationen den *Missing Link* zwischen Werbenetzwerken und schädlichen Dateien erstellt.

Android Emulatoren

Die Untersuchungen zeigen außerdem, dass die Wahl der zugrundeliegenden Technologie des Webcrawlers wohl überlegt sein muss, um bei Fingerprinting-Prozessen nicht vorzeitig

als Smartphone-Imitator identifiziert zu werden. Es existieren mehrere Emulatoren mit unterschiedlichem Funktionsumfang, welche zum Vortäuschen eines echten Smartphones verwendet werden können. Die Betriebssystememulatoren AVD und Genymotion bieten dabei die umfangreichsten Imitationen, da die komplette Funktionspalette von Android zur Verfügung steht. Gleichzeitig stellen PhantomJS, SlimerJS, Ripple und Chrome mit aktivierten Entwicklerwerkzeugen, einen ausreichenden Funktionsumfang zur Verwendung als Android-Webcrawler zur Verfügung. Webbasierte Tests, anhand von Javascript auslesbaren Eigenschaften, zeigen keine signifikanten Abweichungen der Betriebssystememulatoren im Vergleich zu echten Smartphones. Einzig das Javascript-Objekt *performance*, weist bei der Abfrage auf die Größe des verfügbaren Heap-Speichers, durchgehend den surreale Wert von 0 auf. Die kopflosen Browser SlimerJS und PhantomJS sowie Ripple und Chrome mit aktivierten Entwicklerwerkzeugen, weisen bei analogen Tests, teils erheblichen Differenzen zu den Referenzgeräten auf. Beginnend bei vorhandenen Javascript-Events der Smartphones, sind PhantomJS und SlimerJS eindeutig an der Vielzahl der fehlenden Ereignisse zu identifizieren. Um den Werten der Smartphones zu entsprechen, versteckt Chrome mit aktivierten Entwicklerwerkzeugen ausnahmslos alle installierten Plugins und MIME-Typen. Im Gegensatz dazu, sind die kopflosen Browsern und Ripple anhand dieser Testparameter zu erkennen, da deckungsgleiche Experimente die Existenz von Plugins und MIME-Typen beweisen. Weiters sind PhantomJS und SlimerJS, angesichts des bereitgestellten Javascript-Objekts *callPhantom* und der Abwesenheit einiger anderer Objekte, eindeutig von den Referenzgeräten zu unterscheiden. Schlussendlich ist ein unveränderter Useragent-String ein zweifelloses Erkennungsmerkmal für sämtliche getestete Emulatoren, weshalb dieser für eine einwandfreie Imitation ausnahmslos überschrieben werden muss.

Die Resultate der webbasierten Emulatorenerkennung zeigen, dass das auf Basis von SlimerJS verwendete Crawler-Framework Crawlium, eine umfangreiche Konfiguration benötigt, um bei Fingerprinting-Prozessen nicht vorzeitig eliminiert zu werden. Zwingend notwendig ist das künstliche Erzeugen der Events *ondevice** und *ontouch**, sowie das Deaktivieren und Entfernen der verfügbaren Plugins und MIME-Typen. Weiters müssen die fehlenden und überschüssigen Javascript-Objekte im *top*-Objekt, hinzugefügt beziehungsweise entfernt werden, um einer Identifizierung zu entgehen. Zu guter Letzt muss für eine nahezu ideale Imitation der Useragent und die Bildschirmauflösung an die Werte eines realen Smartphones angepasst werden.

6.1 Zukünftige Forschungen

Da die implementierte Applikation nicht selbstständig Angriffe auf neu entdeckte Sicherheitslücken erkennt, sollten regelmäßig neue Module für kritische Android-Schwachstellen hinzugefügt werden. Dabei sollten zusätzlich Drive-by-Downloads für unterschiedliche mobile Betriebssysteme wie iOS oder Windows Phone beachtet werden, um ein weitläufigeres Portfolio an erkennbaren Angriffen zu erlangen. Des Weiteren sollten die bereits erwähnten Module Social Engineering Detector und Emulator Evasion Detector implementiert werden, um Malvertising Kampagnen, unabhängig von Drive-by-Downloads, zu erkennen.

Während Social Engineering Kampagnen mithilfe der Forschungsergebnisse von TrueClick [46] und DART [45] lokalisiert werden könnten, wäre eine Adaptierung der Techniken des verwandten Projekts [20] für das Modul Emulator Evasion Detector denkbar. Darüber hinaus könnte Crawlium mit Erweiterungen für die Erkennung von Angriffen auf das Android Webview Interface sowie die Lokalisierung von Pagehijacking-Attacken ergänzt werden. Da die zu Grunde liegende Sicherheitslücke von LibUtils bei der Konvertierung von UTF kodierten Zeichen stattfindet und Mark Brand bereits mehrere verwundbare System Services genannt hat, sollte nach weiteren webbasierten Angriffsvektoren für diese Schwachstelle gesucht werden.

Der nächste Schritt wäre die Implementierung eines Webcrawlers auf Basis von Webview, um den Konfigurationsaufwand zu verringern und unabhängig von bisher unerkannten Ausschlusskriterien zu bleiben. Eine entsprechende Webcrawler-App könnte dann in der Amazon Web Services Device Farm auf echten Androidgeräten zum Einsatz kommen, um einen beträchtlichen Umfang an unterschiedlichen Geräteversionen und Konfigurationen zu erhalten. Gleichzeitig könnten dadurch Angriffe auf die Same-Origin-Policy von Webview sowie auf veröffentlichte Java-Objekte besser erkannt werden, da infolgedessen das Einrichten von Javascript-Hook obsolet wäre.

In weiterer Folge sollte eine Langzeitstudie mithilfe des Webcrawlers und der entwickelten Applikation realisiert werden. Der Crawler sollte zu Beginn die 500 populärsten Webseiten aus unterschiedlichen Kategorien der Alexa Liste besuchen, um potenziell beabsichtigte oder unbeabsichtigte Malvertising Kampagnen aufzuspüren.

KAPITEL 7

Zusammenfassung

In dieser Arbeit wurden Drive-by-Downloads im Kontext von Android untersucht. Android ist, analog zu anderen populären Betriebssystemen, anfällig für kritische Schwachstellen, wie die Analyse der Sicherheitslücken Stagefright, LibUtils und ExifInterface beweist. Das deren Angriffsvektoren die Platzierung von Malware ohne notwendige Benutzerinteraktion einschließt, verdeutlicht die potenzielle Bedrohung durch Malvertising Kampagnen. Um dem entgegen zu wirken, wurde ein Crawler kreiert, welcher Drive-by-Downloads und die dazugehörigen Werbekampagnen oder Webseiten aufspürt. Bei der Implementierung der zu Grunde liegenden Applikation, wurden denkbare Optionen zur automatischen Erkennung der vorher illustrierten Sicherheitslücken präsentiert. Weiters wurden potenzielle Emulatoren für die Implementierung eines Android-Webcrawlers anhand webbasierter Erkennungsmerkmale charakterisiert, um eine nahezu perfekte Illusion eines Smartphones beim Durchforsten des Internets zu erzeugen. Gleichzeitig wurden teils enorme Unterschiede zwischen den Erscheinungsbildern von Emulatoren und Referenzgeräten aufgezeigt, welche zum Ausschluss bei Fingerprinting-Prozessen durch Malvertising Kampagnen oder anderen böswilligen Skripten führen können.

Abbildungsverzeichnis

2.1 Infiziertes System durch Dogspectus[14] 9

4.1 Zusammenfassung der Erkennung von Remote-Exploits 44

5.1 Auswahl an Differenzen der getesteten Emulatoren im Vergleich zu den Referenzgeräten 58
5.2 Aufbau von DryCrawl 59

Tabellenverzeichnis

4.1 Kopf eines MP4-Atoms 26
4.2 Die ersten Bytes des ESDS-Atoms, welche wichtig für den Integer-Underflow sind. [52] 31
4.3 Kombination der ESDS-Flags mit der jeweiligen Mindesgröße für size um einen Integer-Underflow zu verhindern 31

List of Algorithms

2.1 Brwoserabhängige Verteilung von schädlichem Inhalt[17] 11

4.1 Integer Overflow in SampleTable.cpp in der Funktion setSyncSampleParams bei Version 5.0.0_r2 [50] 27

4.2 Ausschnitt aus der Methode setSampleToChunkParams in der Klasse SampleTable.cpp [50] 28

4.3 Ausschnitt aus der Methode parseESDescriptor in der Klasse ESDS.cpp [50] 30

4.4 Ausschnitt aus der Methode parseChunk in der Klasse MPEG4Extractor.cpp [50] 32

4.5 Ausschnitt aus der Methoden parse3GPPMetaData der Klasse MPEG4Extractor.cpp und setCString aus MetaData.cpp[50] 34

4.6 Ausschnitt aus der Methode parseChunk in der Klasse MPEG4Extractor.cpp für das COVR-Atom [50] 35

4.7 utf16_to_utf8_length in Unicode.cpp bei Version 4.2.2_r1 [56] 36

4.8 utf16_to_utf8 in Unicode.cpp bei Version 4.2.2_r1 [56] 37

4.9 Integer Overflow in exif.c in der Funktion ProcessExifDir bei Version 4.0.3_r1 [58] 39

4.10 HelloWorldObject wird für die Benutzung in Javascript bereitgestellt . . . 40

4.11 Aufruf der Methode sayHello durch Javascript 40

4.12 runShellCommand.js führt einen Shell-Befehl aus 41

4.13 Test in Javascript, ob ein Java-Objekt via addJavascriptInterface veröffentlicht wurde 42

4.14 loadUrl wird in Java genutzt, um Javascript auszuführen 42

4.15 Missbrauch von *navigateToLink* durch ein iframe, um den Seiteninhalt zu ändern 43

5.1 Vereinfachte Suche und Verarbeitung von Exploit-Scannern in der Methode checkAllFileScanners in der Klasse ModuleManager.java 67
5.2 Auszug aus StagefrightScanner.java bei der Verarbeitung einer Mp4-Datei . 70
5.3 Auszug aus der Funktion processExifDir in der Klasse ExifScanner.java beim Auslesen und Berechnen von Tag, Format, Komponenten und ByteCount 74
5.4 Auszug aus der Methode utf16ToUtf8 in der Klasse LibUtilsUtils.java bei der Kalkulation des Verbrauchten Puffers 76
5.5 Auszug aus der Methode isVulnerable in der Klasse LibUtilsScanner.java bei der Kalkulation und dem Vergleich der Längen 76

Literaturverzeichnis

[1] Statista.com, “Number of smartphone users worldwide from 2014 to 2019,” http://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/, accessed: 2016-09-10.

[2] R. Murtagh, “Mobile now exceeds pc: The biggest shift since the internet began,” https://searchenginewatch.com/sew/opinion/2353616/mobile-now-exceeds-pc-the-biggest-shift-since-the-internet-began, accessed: 2016-09-10.

[3] S. Online, “Sicherheitslücke bedroht angeblich 950 millionen android-geräte,” http://www.spiegel.de/netzwelt/gadgets/android-sicherheitsluecke-bedroht-angeblich-950-millionen-geraete-a-1045619.html, accessed: 2016-09-18.

[4] S. Melendez, “Hackers use google’s ad network to spread “fake login” malware,” https://www.fastcompany.com/3062867/overlay-malware-google-adsense, accessed: 2016-11-10.

[5] de.wikipedia.org, “Klassifizierung von malware,” https://de.wikipedia.org/wiki/Schadprogramm#Klassifizierung, accessed: 2016-11-09.

[6] ——, “Cih virus,” http://virus.wikia.com/wiki/CIH, accessed: 2016-11-09.

[7] www.symantec.com, “Sasser wurm,” https://www.symantec.com/security_response/writeup.jsp?docid=2004-050116-1831-99, accessed: 2016-11-09.

[8] de.wikipedia.org, “Backdoor,” https://de.wikipedia.org/wiki/Backdoor, accessed: 2016-11-09.

[9] Infoword.com, “Snowden: The nsa planted backdoors in cisco products,” http://www.infoworld.com/article/2608141/internet-privacy/snowden--the-nsa-planted-backdoors-in-cisco-products.html, accessed: 2016-11-09.

[10] S. Margaret Rouse, “Definition von spyware,” http://www.searchsecurity.de/definition/Spyware, accessed: 2016-11-09.

[11] B. Marczak and S. R. a. t. C. L. John Scott-Railton, “The million dollar dissident: Nso group’s iphone zero-days used against a uae human rights defender,” https://citizenlab.org/2016/08/million-dollar-dissident-iphone-zero-day-nso-group-uae/, accessed: 2016-11-09.

[12] n. s. b. S. Julian Bhardwaj, “Techniques in ransomware explained,” https://nakedsecurity.sophos.com/2012/09/14/new-technique-in-ransomware-explained/, accessed: 2016-11-10.

[13] S. Pontiroli, “Die ransomware-pest 2016,” https://blog.kaspersky.de/fighting-ransomware/9285/, accessed: 2016-11-10.

[14] B. C. LABS, “Android towelroot exploit used to deliver “dogspectus” ransomware,” https://www.bluecoat.com/security-blog/2016-04-25/android-exploit-delivers-dogspectus-ransomware, accessed: 2016-11-10.

[15] S. John Zorabedian, “How malware works: Anatomy of an attack in five stages (infographic),” https://blogs.sophos.com/2013/11/01/how-malware-works-anatomy-of-an-attack-in-five-stages-infographic/, accessed: 2016-11-10.

[16] S. I. Radha Gulati, “The threat of social engineering and your defense against it,” https://www.sans.org/reading-room/whitepapers/engineering/threat-social-engineering-defense-1232, accessed: 2016-11-10.

[17] X. G. P. K. Van Lam Le, Ian Welch, “Anatomy of drive-by download attack,” in *Proceedings of the Eleventh Australasian Information Security Conference (AISC 2013), Adelaide, Australia*, 2013, pp. 49–58. [Online]. Available: http://crpit.com/confpapers/CRPITV138Le.pdf

[18] Z. Li, K. Zhang, Y. Xie, F. Yu, and X. Wang, “Knowing your enemy: Understanding and detecting malicious web advertising,” in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS ’12. New York, NY, USA: ACM, 2012, pp. 674–686. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382267

[19] A. Zarras, A. Kapravelos, G. Stringhini, T. Holz, C. Kruegel, and G. Vigna, “The dark alleys of madison avenue: Understanding malicious advertisements,” in *Proceedings of the Internet Measurement Conference (IMC)*, 2014.

[20] G. Acar, M. Juarez, N. Nikiforakis, C. Diaz, S. Gürses, F. Piessens, and B. Preneel, “Fpdetective: Dusting the web for fingerprinters,” in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS ’13. New York, NY, USA: ACM, 2013, pp. 1129–1140. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516674

[21] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, “Cookieless monster: Exploring the ecosystem of web-based device fingerprinting,” in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, ser. SP ’13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 541–555. [Online]. Available: http://dx.doi.org/10.1109/SP.2013.43

[22] H. Bojinov, Y. Michalevsky, G. Nakibly, and D. Boneh, "Mobile device identification via sensor fingerprinting," *CoRR*, vol. abs/1408.1416, 2014.

[23] K. Mowery, D. Bogenreif, S. Yilek, and H. Shacham, "Fingerprinting information in JavaScript implementations," in *Proceedings of W2SP 2011*, H. Wang, Ed. IEEE Computer Society, May 2011.

[24] G. Ho, D. Boneh, L. Ballard, and N. Provos, "Tick tock: Building browser red pills from timing side channels," in *8th USENIX Workshop on Offensive Technologies (WOOT 14)*. San Diego, CA: USENIX Association, Aug. 2014. [Online]. Available: https://www.usenix.org/conference/woot14/workshop-program/presentation/ho

[25] M. A. Rajab, L. Ballard, N. Jagpal, P. Mavrommatis, D. Nojiri, N. Provos, and L. Schmidt, "Trends in circumventing web-malware detection," Tech. Rep., 2011.

[26] T. Nelms, R. Perdisci, M. Antonakakis, and M. Ahamad, "Webwitness: Investigating, categorizing, and mitigating malware paths," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 1025–1040. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/nelms

[27] M. Cova, C. Kruegel, and G. Vigna, "Detection and analysis of drive-by-download attacks and malicious javascript code," in *In WWW 2010*, 2010.

[28] A. Ikinci, T. Holz, and F. Freiling, "Monkey-spider: Detecting malicious websites with low-interaction honeyclients," in *In Proceedings of Sicherheit, Schutz und Zuverlässigkeit*, 2008.

[29] Y. min Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King, "Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities," in *In NDSS*, 2006.

[30] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose, "All your iframes point to us," in *Proceedings of the 17th Conference on Security Symposium*, ser. SS'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 1–15. [Online]. Available: http://dl.acm.org/citation.cfm?id=1496711.1496712

[31] A. Moshchuk, T. Bragin, D. Deville, S. D. Gribble, and H. M. Levy, "Spyproxy: Execution-based detection of malicious web content," in *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, ser. SS'07. Berkeley, CA, USA: USENIX Association, 2007, pp. 3:1–3:16. [Online]. Available: http://dl.acm.org/citation.cfm?id=1362903.1362906

[32] K. Rieck, T. Krueger, and A. Dewald, "Cujo: Efficient detection and prevention of drive-by-download attacks," in *Proceedings of the 26th Annual Computer Security Applications Conference*, ser. ACSAC '10. New York, NY, USA: ACM, 2010, pp. 31–39. [Online]. Available: http://doi.acm.org/10.1145/1920261.1920267

[33] Sujal and S. Verma, “Multimedia attacks on android devices using stagefright exploit,” in *Vol. 14 No. 9 SEPTEMBER 2016 International Journal of Computer Science and Information Security*. International Journal of Computer Science and Information Security, 2016, pp. 658–665.

[34] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin, “Attacks on webview in the android system,” in *Proceedings of the 27th Annual Computer Security Applications Conference*, ser. ACSAC ’11. New York, NY, USA: ACM, 2011, pp. 343–352. [Online]. Available: http://doi.acm.org/10.1145/2076732.2076781

[35] E. Chin and D. Wagner, “Bifocals: Analyzing webview vulnerabilities in android applications,” in *Revised Selected Papers of the 14th International Workshop on Information Security Applications - Volume 8267*, ser. WISA 2013. New York, NY, USA: Springer-Verlag New York, Inc., 2014, pp. 138–159. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-05149-9_9

[36] D. R. Thomas, A. R. Beresford, T. Coudray, T. Sutcliffe, and A. Taylor, “The lifetime of android api vulnerabilities: Case study on the javascript-to-java interface,” in *Security Protocols XXIII: 23rd International Workshop, Cambridge, UK, March 31 - April 2, 2015, Revised Selected Papers*, B. Christianson, P. Švenda, V. Matyáš, J. Malcolm, F. Stajano, and J. Anderson, Eds. Cham: Springer International Publishing, 2015, pp. 126–138. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-26096-9_13

[37] M. Georgiev, S. Jana, and V. Shmatikov, “Breaking and Fixing Origin-Based Access Control in Hybrid Web/Mobile Application Frameworks ,” in *2014 Network and Distributed System Security (NDSS ’14)*, San Diego, February 2014.

[38] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, “Drammer: Deterministic rowhammer attacks on mobile platforms,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’16. New York, NY, USA: ACM, 2016, pp. 1675–1689. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978406

[39] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu, “The ghost in the browser analysis of web-based malware,” in *Proceedings of the First Conference on First Workshop on Hot Topics in Understanding Botnets*, ser. HotBots’07. Berkeley, CA, USA: USENIX Association, 2007, pp. 4–4. [Online]. Available: http://dl.acm.org/citation.cfm?id=1323128.1323132

[40] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, “A survey of mobile malware in the wild,” in *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, ser. SPSM ’11. New York, NY, USA: ACM, 2011, pp. 3–14. [Online]. Available: http://doi.acm.org/10.1145/2046614.2046618

[41] S. Ford, M. Cova, C. Kruegel, and G. Vigna, "Analyzing and detecting malicious flash advertisements," in *Proceedings of the 2009 Annual Computer Security Applications Conference*, ser. ACSAC '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 363–372. [Online]. Available: http://dx.doi.org/10.1109/ACSAC.2009.41

[42] J. Ma, L. K. Saul, S. Savage, and G. M. Voelker, "Beyond blacklists: Learning to detect malicious web sites from suspicious urls," in *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '09, New York, NY, USA, 2009, pp. 1245–1254. [Online]. Available: http://doi.acm.org/10.1145/1557019.1557153

[43] V. Rastogi, R. Shao, Y. Chen, X. Pan, S. Zou, and R. Riley, "Are these ads safe: Detecting hidden attacks through the mobile app-web interfaces," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2016.

[44] T. Nelms, R. Perdisci, M. Antonakakis, and M. Ahamad, "Towards measuring and mitigating social engineering software attacks," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 773–789. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/nelms

[45] D. W. Kim, P. Yan, and J. Zhang, "Detecting fake anti-virus software distribution webpages," *Comput. Secur.*, vol. 49, no. C, pp. 95–106, Mar. 2015. [Online]. Available: http://dx.doi.org/10.1016/j.cose.2014.11.008

[46] S. Duman, K. Onarlioglu, A. O. Ulusoy, W. Robertson, and E. Kirda, "Trueclick: Automatically distinguishing trick banners from genuine download links," in *Proceedings of the 30th Annual Computer Security Applications Conference*, ser. ACSAC '14. New York, NY, USA: ACM, 2014, pp. 456–465. [Online]. Available: http://doi.acm.org/10.1145/2664243.2664279

[47] C. Fröhlich, http://www.stern.de/digital/smartphones/quadrooter-android-update-problem-samsung-google-7001118.html, accessed: 2016-12-01.

[48] C. S. T. LTD., http://blog.checkpoint.com/2016/08/07/quadrooter/, accessed: 2016-12-01.

[49] Android, https://source.android.com/security/bulletin/, accessed: 2016-12-01.

[50] androidxref.com, http://androidxref.com/5.0.0_r2/xref/frameworks/av/media/libstagefright, accessed: 2016-12-15.

[51] J. Drake, "Stagefright: Scary code in the heart of android," in *Researching Android Multimedia Framework Security*, ser. Black Hat USA 2015, 2015.

[52] blog.fortinet.com, https://blog.fortinet.com/2015/08/25/cryptogirl-on-stagefright-a-detailed-explanation, accessed: 2016-12-16.

[53] Researched and W. b. H. B. implemented by NorthBit, "Metaphor: A (real) reallife stagefright exploit," https://www.exploit-db.com/docs/39527.pdf, 2015, accessed: 2016-11-15.

[54] bugs.chromium.org, https://bugs.chromium.org/p/project-zero/issues/detail?id=840&q=, accessed: 2016-12-18.

[55] cve.mitre.org, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-3861, accessed: 2016-12-18.

[56] androidxref.com, http://androidxref.com/4.2.2_r1/xref/frameworks/native/libs/utils/Unicode.cpp, accessed: 2016-12-18.

[57] googleprojectzero.blogspot.co.at, https://googleprojectzero.blogspot.co.at/2016/09/return-to-libstagefright-exploiting.html, accessed: 2016-12-18.

[58] androidxref.com, http://androidxref.com/4.0.3_r1/xref/external/jhead/exif.c#536, accessed: 2016-12-17.

[59] developer.android.com, https://developer.android.com/reference/android/webkit/JavascriptInterface.html, accessed: 2016-12-10.

[60] https://github.com/Valve/fingerprintjs2, accessed: 2017-01-20.

[61] https://clientjs.org/, accessed: 2017-01-20.

[62] M. Karch, https://www.lifewire.com/android-and-flash-1616859, accessed: 2017-01-22.

[63] bugs.webkit.org, https://bugs.webkit.org/attachment.cgi?id=154876&action=prettypatch, accessed: 2017-03-09.

[64] developer.chrome.com, https://developer.chrome.com/multidevice/user-agent, accessed: 2017-03-07.

[65] tools.ietf.org, https://tools.ietf.org/html/rfc7231#page-46, accessed: 2017-03-07.

[66] https://maven.apache.org/, accessed: 2017-02-03.

[67] https://github.com/ronmamo/reflections, accessed: 2017-02-03.

[68] G. Merzdovnik, M. Huber, D. Buhov, N. Nikiforakis, S. Neuner, M. Schmiedecker, and E. Weippl, "Block me if you can: A large-scale study of tracker-blocking tools," in *2nd IEEE European Symposium on Security and Privacy (Euro S&P)*, 0 2017.

[69] https://github.com/sannies/mp4parser, accessed: 2017-02-03.

[70] https://s3.amazonaws.com/zhafiles/Zimperium-Handset-Alliance/ZHA-Crash-PoC.zip, accessed: 2017-02-07.

[71] https://github.com/reines/exif, accessed: 2017-02-05.

[72] https://github.com/drewnoakes/metadata-extractor, accessed: 2017-02-03.

[73] http://commons.apache.org/proper/commons-imaging/, accessed: 2017-02-05.

[74] EasyTAG, https://sourceforge.net/projects/easytag/, accessed: 2017-02-04.

[75] IDTE, https://sourceforge.net/projects/idteid3tagedito/, accessed: 2017-02-04.

[76] E. Ts, http://www.etsi.org/deliver/etsi_ts/126200_126299/126244/07.02.00_60/ts_126244v070200p.pdf, accessed: 2017-02-04.

[77] id3.org, http://id3.org/id3v2-00, accessed: 2017-02-04.