Security Weaknesses of the Android Advertising Ecosystem

Jeremy R. Tate

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

T. Charles Clancy, Chair
David Levy
Wenjing Lou

December 4, 2014
Arlington, Virginia

Keywords: Android, Cyber Security, Advertising Network

Security Weaknesses of the Android Advertising Ecosystem

Jeremy R. Tate

ABSTRACT

Mobile device security is becoming increasingly important as the number of devices that are used continues to grow and has surpassed one billion active devices globally. In this thesis, we will investigate the security of Android ad supported apps, security vulnerabilities that have been identified in the way those ads are delivered to the device and improvements that can be made to protect the privacy of the end user. To do this, we will discuss the Android architecture and the ecosystems of apps and ads on those devices. To better understand the threats to mobile devices, a threat analysis will be conducted, investigating the different attack vectors that devices are susceptible to. This will also include a survey of existing work that has been conducted within the realm of Android security and web based exploits. The specific attacks that are detailed in this research are `addJavascriptInterface` attacks against a WebView used to display an ad and information leakage from the ad URL request. These attack vectors are discussed in detail with applicability and feasibility studies conducted. The results of these attacks will be analyzed with a discussion of the methodology used to obtain them. In order to combat such attacks, there will also be discussion of potential solutions to mitigate the threats of attack from a variety of angles, to include steps that users can take to protect themselves as well as changes that should be made to the Android operating system itself.

Acknowledgments

I would like to express my gratitude and appreciation to my adviser and committee chair, Dr. Charles Clancy for his guidance and leadership through the entire course of my Masters degree. His insightful comments and questions greatly directed the research that I conducted as well as enhanced my experience with research.

I would like to thank the members of my committee, Drs. David Levy and Wenjing Lou for the frequent and frantic last minute emails and form signing that you had to do in order for me to graduate. Thank you also to Dr. Lou for the class that she taught as it was one of the highlights of my experience and opened my eyes to the wide ranging research that has been conducted in wireless security.

I would also like to thank my parents, Ralph and Carolyn Tate, for their long distance support and encouraging emails. I am extremely grateful to my Dad for not letting me quit when the work was difficult and the nights long. Thank you also for the model of how to be a diligent student and to have the persistence to not give up.

And most importantly, I would also like to thank my wife, Luice, for trusting the leading that God had given my to quit my day job and to pursue my degree full time. Thank you for understanding all the late nights and the "Can't go, have homework" excuses that you got. Your support has enabled me to complete this degree.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

There are many reasons why mobile phone security is important. In this thesis, we will explore the vulnerabilities that are found with Android applications, specifically in the way ads are displayed within applications as well as how they communicate over the internet. This research has identified two attack vectors, `addJavascriptInterface` attacks against a WebView used to display an ad and information leakage from the ad URL request. We document the results of the attacks against a variety of different Android cell phones as well as detail the requirements to carry out each of these attacks. More importantly, we include a discussion of steps that can be taken by the user to protect themselves from such attacks as well as recommendations for app developers in terms of software best practices. We also consider changes that Google could make to the Android operating system that would make it a more secure platform for ad delivery.

1.1 Problem Description

Mobile devices are becoming a greater and greater part of our individual and daily lives. As a result, we trust them with more and more personal details; everything from contact information for our friends and family, medical and health information to banking and financial details. Practically everything about us is stored on our mobile devices. Additionally, they are also with us most of the time, allowing us access to our email and games at a moment's notice. However, there is also a downside to having ever present mobile devices and that is the potential for malicious users to exploit different vulnerabilities for their gain. In order to support the vast number of mobile applications, or apps, that are available in numerous app stores, mobile advertising is frequently used. While there has been significant research in other areas of the Android ecosystem and operating system, the implications of widespread ad use within apps has not been investigated. This thesis will address that shortcoming and investigate two different attacks against ads being displayed in apps. Ads provide a unique avenue through which it is possible to gain access to mobile devices and their contents through active as well as passive means. Ads are most commonly delivered to mobile devices through simple HTTP connections and are displayed as HTML pages, frequently with Javascript to enable analytics tracking. These vectors provide opportunities to potential attacks to gain access to the device and to compromise it in some way.

Ads are delivered to an app through an ad library which is a separate application that a developer integrates with their own work. These ad libraries then interact with the advertising network to retrieve the ad that will be displayed to the user.

1.2 Motivation

Android is the most popular mobile operating system in the world with more than one billion active devices. The most popular digital market, or app store, for Android is the Google Play Store, where there are more than 1.3 million apps with 1.1 million of those being free. Of those free apps, more than 700,000 or 63% are supported by advertising [1, 2]. These numbers have been continually on the rise ever since the introduction of the mobile platform. With ad supported apps being such a significant portion, it is important to understand the security implications of how the libraries that support the advertising impact the security of the app as well as the mobile device that it is being run on.

Additionally, due to the nature of mobile devices and the societal norm that has developed in carrying such devices with us at all times, exploits that target such devices have more potential for harm than do other computing devices as mobile devices are able to track us using cellular network and GPS coordinates. We investigate these possibilities and seek to limit their exposure to attackers and to make the market place safer for users.

1.3 Approach

The approach taken for this research was to develop a standalone app that was used in the analysis of code execution via WebViews and the `addJavascriptInterface` that exposes Java methods to Javascript. This same app was also used as a data collection point in analyzing the personal information that two different ad libraries sent as part of their request for ads to be displayed: MoPub and AdMob. From this point, the lessons and approaches learned were applied to commercial apps that used the same ad libraries to determine the applicability of such an attack in the real world.

1.4 Significance of Research

Ad supported apps are here to stay and this research adds to the knowledge base of how to keep consumers safe and their personal information out of the hands of potential miscreants. To that end, we show vulnerabilities that are present in Android's WebView using the `addJavascriptInterface`. We also detail information that is leaked from two different ad libraries as a result of requesting an ad and finally, we propose solutions to the above vulnerabilities in order to mitigate their risks and provide recommendations that can be taken by both users as well as developers to minimize the loss of personal data as well as progress the work towards a more secure advertising ecosystem.

The major work that has been done in this area has been from Grace et al. in [16] who were the first to describe the unsafe behaviour of ad networks and the risks they expose the users of applications to. The work done by Pearce in [29] and Shekhar in [33] investigate the possibility of separating an ad library from the app that it is packaged with.

This work builds on that done by these and other researchers by examining in detail the information that is leaked specifically by the ad request URL. The other works built static analysis tools to examine ad libraries to find potential paths between private data and network sinks, indicating the possibility of personal information loss to the ad network; no particular attention was paid to the information that was sent during the ad request process. This work assumes that the ad network is benign with adversaries in the vicinity of the target device who are able to eavesdrop on the request.

This research also builds on the knowledge of the `addJavascriptInterface` attack by detailing its applicability to the WebViews used to display ads within applications.

1.5 Organization of Research

This research is organized in the following way. Chapter 2 discusses the background to include the Android OS architecture, the app ecosystem and the ad ecosystem. From there, we move on to Chapter 3 to consider the different threat surfaces that exist for Android and discuss prior work in this area. Chapter 4 discusses in detail the two different attacks that were identified as part of this research with the results given in Chapter 5. With the vulnerabilities described, recommendations on how to mitigate those attack vectors are described in Chapter 6. Finally, concluding thoughts are presented in Chapter 7. Full examples of the ad request URLs are given in Appendix A.

Chapter 2

Background

This chapter familiarizes the reader with the basis of the technologies that are used in both the Android operating system as well as those that are used for the ad networks and app ecosystems.

2.1 Android OS Architecture

Figure 2.1 shows the complete Android architecture being broken down into its five main components. The components are the Linux kernel, Libraries and the Android Runtime, the application framework and finally, applications.

2.1.1 Linux Kernel

The kernel forms the basis for the operating system, allowing all higher level processes to run. It is at this layer that the responsibility for managing the hardware/software interactions is controlled. Commands issued by applications running on the device are translated into

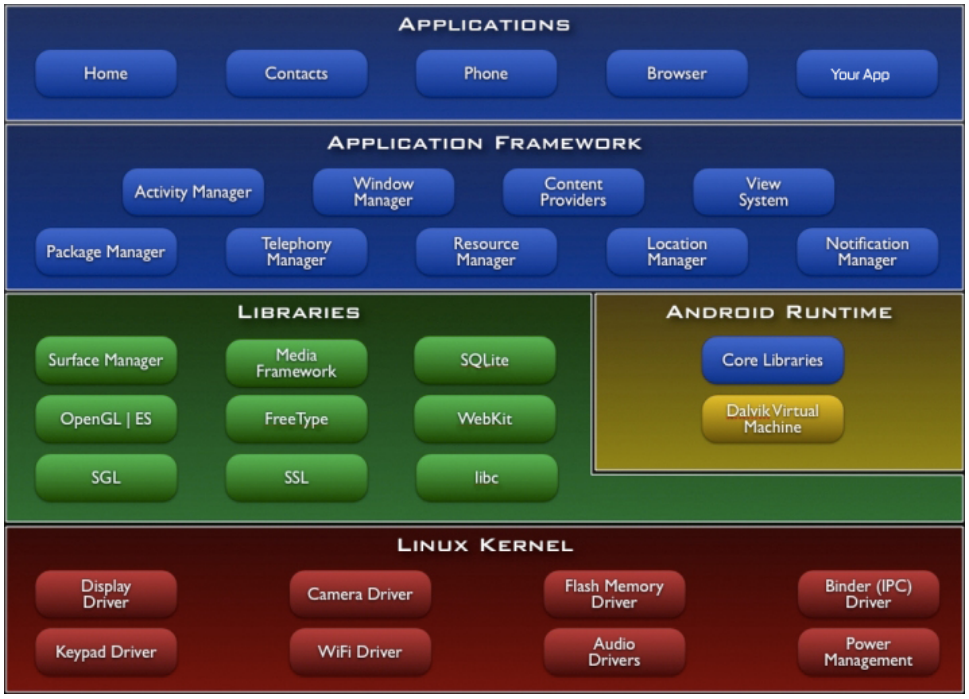


Figure 2.1: Android Architecture [13]. (Google. *Android, the worlds most popular mobile platform.* 2014. URL: `http://developer.android.com/about/index.html` (visited on 10/30/2014). Used under fair use, 2014.)

control signals that are sent to the physical components that make up the device. Due to this critical nature, the kernel is loaded into and operates from within a protected place in memory, where it has more control than other programs and applications on the device.

Kernels are also responsible for managing system resources that may be requested by different processes. This functionality provides an abstraction of the details of how to control the hardware to simplify development. Android was originally based on Linux Kernel v2.6, but has since updated to more recent versions; the specific kernel version is dependent upon the target Android device and chipset, using versions as recent as v3.4.

Linux is a monolithic kernel, whereby the entire operating system functions in the elevated privileges of kernel space. Kernel space is strictly reserved for operating elevated privileged code which is needed to interact with device hardware. This is contrasted with user space which is where applications operate from. The separation is created to provide security for the operating system. The monolithic architecture differs from other operating system architectures by providing a single interface through which higher level applications access the hardware. All operating system service such as process and memory management, and execution order are controlled by the kernel instead of at higher levels in other models.

The Linux kernel was initially developed by Linus Torvalds in 1991 and has since been expanded to support a wide variety of computer architectures, being deployed to conventional computer systems as well as embedded devices such as routers and mobile devices. While initial development was undertaken solely by Torvalds, it has since been expanded by a global community of volunteers and contributors through the release of the code under a public license.

Using the desktop version of the kernel, Google made specific modification to increase battery life while maintaining performance on resource limited mobile devices. In addition to these

changes, device manufacturers and operators have additional software requirements in order to interface with the specific baseband chipsets used in their devices.

The Android file system is mounted to a number of different locations, separating user data from the operating system with the operating system files are marked as read only. Unlike desktop versions of Linux, Android users are not given root access to the device and must resort to exploiting security flaws in order to obtain access.

2.1.2 Libraries

Forming the basis for higher level software are the libraries. Libraries are a collection of functions or methods written in a specific programming language that are defined by an interface, that is, the specifics of the functionality that is available and how to use it are static and well defined. The interface also determines what type of data is being sent back and forth between components of the system. Libraries commonly provide a low level functionality of the operating system to the rest of the system as a whole, such as utilizing SSL or SQLite, or playing an audio file. This allows higher programming languages to utilize these common features without having to implement them again

Another benefit of interfaces is that the details of the underlying implementation requirements for a specific platform have no bearing on higher level languages. This separation of software layers from the hardware allows specific library implementations to change, but the software that depends on them to remain constant across hardware platforms and devices. This leads to a modular code base which eases code distribution and maintenance.

2.1.3 Android Runtime

As all Android apps are written in Java, they are compiled into Java bytecode for a Java virtual machine. The Java bytecode is then translated and stored in a Dalvik executable file (.dex), designed for systems such as mobile devices that are constrained on processing speed and memory. It does this by performing compilation of Java as the application is launch, a process known as "just-in-time". There are also optimizations that take place to increase performance. This provides an improvement in performance compared to using the standard Java virtual machine [23].

The Android Runtime (ART) module has replaced the Dalvik virtual machine in the latest version of Android, version 5.0 "Lollipop". With it come a number of performance enhancements including ahead-of-time compilation of application code as well as the potential for improved optimizations of code. This is done when the app is first installed and requires additional processing and additional space. The results of these enhancements have shown initial results to be up to twice as fast as compared to a Dalvik based device [9].

2.1.4 Application Framework

This layer forms the basis of the Android Software Development Kit, or SDK, that developers use to create their own apps. The Android SDK is maintained and published by Google, with each version being released to the public under an open source license. Once the latest version of Android has been released, device manufacturers and network operators work to add additional features to their devices as well as include the components that are needed to allow the device to work on their cellular network as well as the baseband that has been included in each specific model. The open source nature of Android has also contributed significantly to the body of research that has been conducted for this platform which in turn

has significantly improved the overall security.

Using the components of the SDK, individual developers are able to piece together their own application. The SDK builds on the library functionality and provides an additional layer of abstraction. Specifically, the details for how to play a video, reading the file from memory, decoding it, and displaying it on the screen, are all taken care of by the SDK. This allows developers to use common pieces of functionality to develop their own apps more quickly.

By default, an app runs within a single process, but specific Activities and Services can be specified to run in different processes [11]. This is done to keep user data from each app from interacting and interfering with one another, making use of Linux user separation of data. All apps run by default on the main UI thread on the device. An exception to this policy is for asynchronous tasks, which must be placed in their own thread so as to not interfere with UI events.

Messages between different apps are handled with Interprocess Communication, IPC, calls using the `IBinder` interface. One way in which IPC calls are used is to utilize functionality that is contained within a different app, such as a systems app that controls user location settings. Other IPC calls contain messages sent by one app to another to indicate that some event has happened and that further processing is required to complete the user request.

Messages between components of the same app can use any number of light weight communication options as the need for persistence is not required. These can include primitive data types as well as non-persistent objects. These messages are passed using Intents, a bundle data object which is effectively a hash map using a known key and the data. Components on the receiving side index into the hash map with the key that has been passed to them to retrieve the data object.

2.1.5 Applications

The software running at the Application layer is everything that the user interacts with while on their mobile device. This includes everything from the phone dialer to the contacts to the Facebook and Twitter app. Also running at this layer are the portions of apps that have no user interaction. These include services and asynchronous tasks. Services are components of apps that are always running in the background, performing tasks that the user has no direct control over, such as a mail client listening and waiting for new mail to arrive. Asynchronous tasks are those that have a specific and well defined start and end, such as uploading a video to Facebook or downloading a music album that has just been purchased.

Applications are composed of one or more Activities, which are the basic user interface component, combined with other system level functionality. An Activity is the only element that a user sees of a specific app, and contains all information that is conveyed to the user. A sample Activity from the app used as part of this research can be seen in Figure 2.2. As seen in the figure, there are a collection of buttons and text. Elements such as graphics and images are also common fixtures for Activities.

Apps are written in the Java programming language with some additional functionality for C/C++ code through the use of the Native Development Kit (NDK). Google strongly discourages developing apps primarily using the NDK and instead encourages developers to port their C and C++ code to Java if at all possible. This is due to the increased complexity that results from having native code portions in an application while not significantly improving performance. Using the NDK makes sense for components that are highly CPU intensive, such as game engines and signal processing.

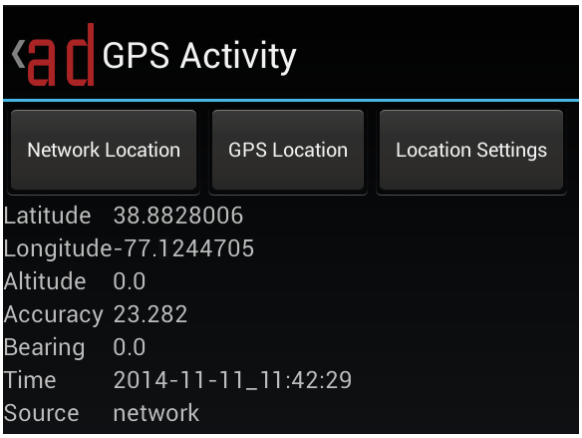


Figure 2.2: Example of an Activity showing the details of the location of the device based on the cellular network. Also visible are buttons that display the GPS and Network location as well as link to the device Location Settings to configure which features are available to apps.

2.2 Android App Ecosystem

Apps are made available to mobile devices through digital markets. The official market place for Android is the Google Play Store. From the Google Play Store, users are able to download and install a variety of apps that are related to everything from mobile banking and entertainment to games and social networking. Google maintains the Play Store by scanning submitted apps for malicious content and behaviour as well as inspecting apps that have been flagged by other users as potentially violating a portion of the Google Play Terms of Service. Intentionally misleading or malicious behaviour by an app is prohibited within the store. Enhancements to these procedures over the years have decreased the prevalence of malicious apps within the Play Store, boosting customer confidence in the safety of the content.

There are 26 different categories of applications with an additional 18 categories just for games. Within the Play Store, are 1.3 million apps with 1.1 million of those apps are free [2]. Of the free apps, approximately 700,000 are supported by advertising [1]. An additional

source of revenue available to app developers is by offering in-app purchases that enable additional content.

Apps are also made available through third party market places that are hosted by a variety of other vendors. Depending on the terms of these third party markets, they often contain apps that have been denied from the Google Play Store and are also likely to contain apps with malicious content [41].

2.3 Android Advertising Ecosystem

There are two major components to the ad ecosystem. The first is the underlying architecture that actually transmits the ad request from the user's device to the advertiser who wants to show them a specific ad and response that includes that ad. The other component is the ad library that is packaged alongside the app that users download and install. This set of files communicates with the ad network, transmitting ad requests and is responsible for displaying the ads when they are received.

2.3.1 Architecture

When a user opens an ad supported app they downloaded from the digital market, there are several steps that are first taken before the ad is displayed for the user in the designated space for the ad. Advertisers work with ad brokering companies, also called ad networks, such as the two investigated in this research: Google's AdMob and Twitter's MoPub. App developers register their app with the ad network and are given a unique identifier for each ad space they want to add to their app. These can be of various sizes, from small banner ads up to full screen, and can host a variety of content, such as videos and images. The unique

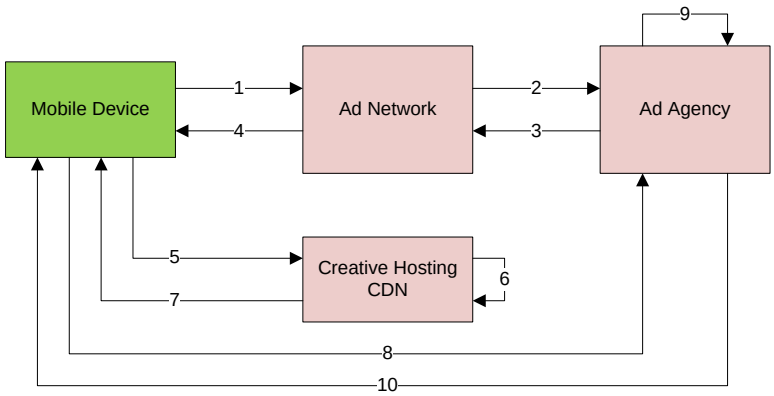


Figure 2.3: The architecture of a classical ad distribution network. Steps 1 and 2 are the ad being requested by the app. Steps 3 and 4 are the response of the advertisers with an address to a specific ad which is potentially hosted on a different network. The ad is then retrieved in Steps 5-7 before being displayed for the user. Steps 8-10 optionally retrieve additional analytics and tracking information from the advertiser.

ad id is used for tracking purposes by the ad network to count impressions that are made as there may be times when an ad is requested and no impression is made and to ensure proper payment to the app developer. An impression is a successful rendering of an ad within the app. When an impression is made, the advertiser pays a fee to the ad network to display the ad and the developer receives a portion of that fee. Developers also receive a fee when users click on an ad. The unique ad identifiers also enable the ad networks to create profiles for specific mobile devices and users in order to tailor the types and content of ads that are delivered to them with the desire being to increase the effectiveness of ads in turning a viewer impression into a sale or action. These are called conversions and are the result of an action that the user takes on the website that is indicated in an ad. For some sites, a conversion doesn't mean the user purchased something, but instead they viewed some specific content or did a specific action.

The ad distribution process is illustrated in Figure 2.3. The process starts when the ad supported app makes an ad request to the ad network (Step 1). The ad network then selects

an ad that will be sent to the user. The ad network may instead use a Real Time Bidding system that dynamically adjust the prices that advertisers pay in order for users to see their ads, but also allows them an opportunity to have access to higher value users with specific demographics (Steps 2-3). Once the ad has been selected, the ad network sends the address of the ad to the user app (Step 4). The app then resolves this address with the Content Distribution Network or a different portion of the ad network (Step 5), which counts this as an impression (Step 6). Impressions are measurements by which developers and ad networks are paid by the advertiser. The ad is then sent to the user where it is displayed within the app (Step 7). Depending on the architecture of the ad network and the analytics that have been enabled, there may be additional steps that are required to contact the advertiser to complete the ad request (Steps 8-10) [20, 32].

From the application level, the architecture of including an ad library can be seen in Figure 2.4. The app is created by a team of developers by using the Android SDK through a series of API calls. These API calls are made from Activities, basic units of the user interface. When ads are added to the app, the developers download and package an additional library alongside the rest of their code, this is the ad library. From the Activity, calls are made to the ad library to request and load ads. When a response is returned from the ad network, a secondary request is made to load the actual ad which is then displayed for the user to see.

2.3.2 Libraries

There are different ad libraries available to developers for inclusion in their apps. The most popular are listed in Table 2.1. It should also be noted that it is possible for an app to have multiple ad libraries, with [26] showing some apps containing as many as 28 different ad

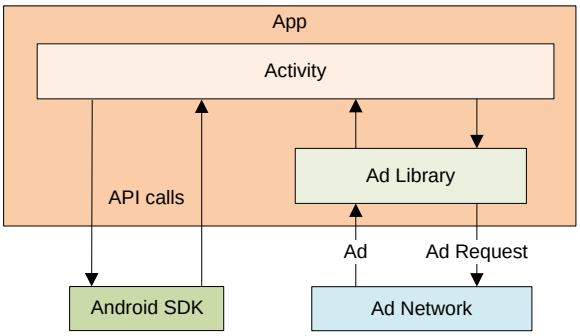


Figure 2.4: Current ad architecture with the ad library packaged with the app.

Table 2.1: The ten most popular ad networks for Android according to [1]. The ad libraries that have been investigated in this work, AdMob and MoPub, have been marked in bold. (AppBrain. *Android Ad networks.* 2014. URL: `http://www.appbrain.com/stats/libraries/ad` (visited on 10/17/2014). Used under fair use, 2014.)

Name	**% of Apps**	**% of Installs**
AdMob	**37.78**	**39.56**
Millennial Media	3.3	11.04
Charboost	2.7	9.95
InMobi	3.09	9.84
TapJoy	1.55	8.24
MoPub	**1.39**	**7.33**
AdColony	1.07	5.47
Vungle	0.61	4.02
MobileAppTracker	0.47	3.47
Amazon Mobile	0.64	3.27

libraries.

Two ad libraries were investigated as part of this thesis: AdMob and MoPub. According to the listing from Table 2.1, AdMob is the most popular ad network while MoPub is less common and comes in at number six. AdMob is owned by Google and is closed source, while MoPub is open source and is owned by Twitter. MoPub is much less common in the ad network space by having an open source library. Most of the other ad libraries are closed source. Part of the reason for choosing to analyze MoPub is due in part because of its open

```
View tempView = findViewById(R.id.bannerAdView_1);
moPubBannerView = (MoPubView) tempView;

moPubBannerView.setAdUnitId(AD_UNIT_ID);
moPubBannerView.loadAd();
```

Figure 2.5: Code sample from test app that displays a banner ad using the MoPub library. The `AD_UNIT_ID` value is given by MoPub when the ad unit is registered with their service and is unique to each ad unit.

source nature.

In order to access the ad network, app developers must use the specific Java library provided by the network. This allows them to receive ads and display them for the users. While there is some ability to communicate with ad networks that did not also create the ad library, there is no common approach to creating and displaying ads within an app on Android. This sub-content architecture is generally configured such that the less popular ad network communicates and retrieves ads from the more popular ad network. As such, a configuration that is possible within the MoPub dashboard available to developers is a setting to use the AdMob network to deliver ads.

In Figure 2.5 is a code sample from the MoPub library for displaying a banner ad within the app. The `AD_UNIT_ID` was provided by MoPub when the ad space was registered. In Figure 2.6 is the similar ad creation mechanism with the ad library from AdMob. For this ad as well, the `AD_UNIT_ID` was provided when the ad was registered with the network. Notice the different structure and the use of a `Builder` mechanism.

2.3.3 Protocols

In analyzing each of these libraries, packet capture samples were taken from the test app for each of the libraries. The ads were separated so as to be studied independently. Upon

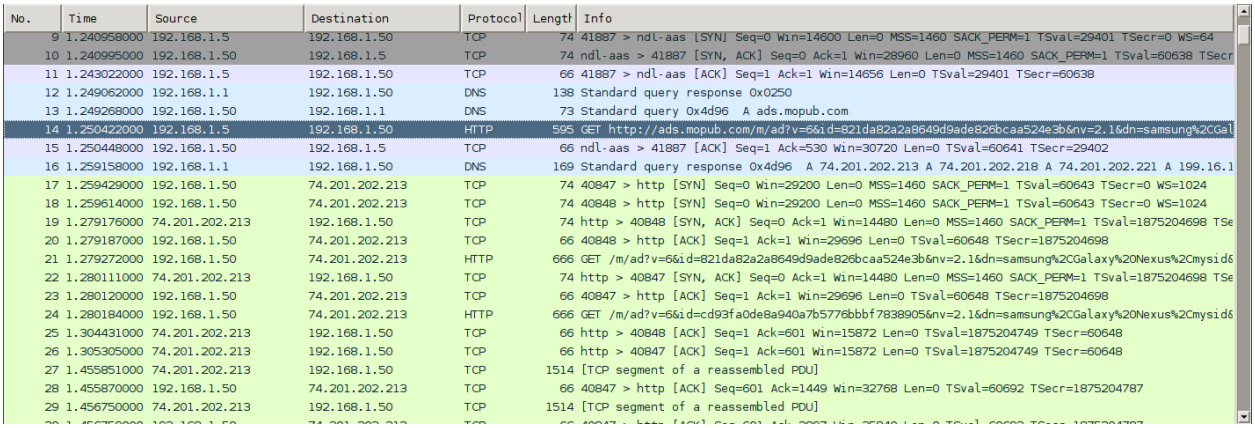
```
AdView adView;
adView = new AdView(this);
adView.setAdSize(AdSize.BANNER);
adView.setAdUnitId(AD_UNIT_ID);

// Create an ad request.
AdRequest adRequest = new AdRequest.Builder().build();

// Start loading the ad in the background.
adView.loadAd(adRequest);
```

Figure 2.6: Code sample from test app that displays a banner ad using the AdMob library. The `AD_UNIT_ID` value is given by AdMob when the ad unit is registered with their service and is unique to each ad unit.

analyzing the communications of the ad libraries, it was determined that MoPub used primarily HTTP and TCP protocols in communicating with their servers. AdMob used the same protocols in addition to encrypting their traffic with TLSv1. Both libraries followed standard DNS query-response paths as part of initializing the ad request. More details are available in Section 5.5.

In the set of Figures 2.7, 2.8, 2.9 and 2.10, we see sample packet capture traces from both ad libraries. Figures 2.7 and 2.8 show excerpts from the MoPub library. It should be noted in these figures that there is no use of encryption. In Figure 2.7, it is possible to see the DNS request in packet 12 and the response in 16.

Figures 2.9 and 2.10 detail the AdMob library packet trace. In these figures, the use of encryption can be seen by the use of TLSv1 packets. After the initial ad request, all subsequent packets were encrypted, to include the ad response that was sent back from the ad network.

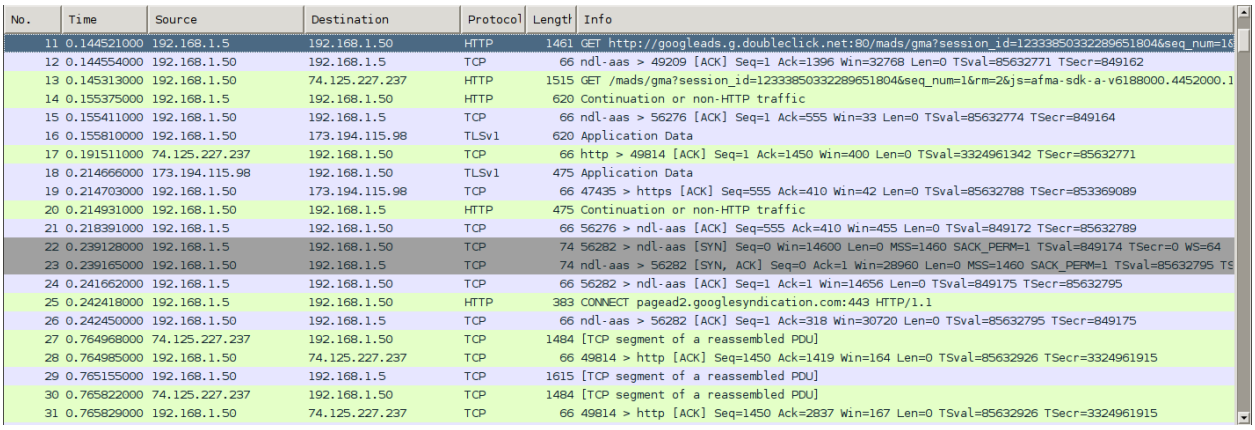


No.	Time	Source	Destination	Protocol	Length	Info
9	1.240958000	192.168.1.5	192.168.1.50	TCP	74	41887 > ndl-aas [SYN] Seq=0 Win=14600 Len=0 MSS=1460 SACK_PERM=1 TSval=29401 TSecr=0 WS=64
10	1.240995000	192.168.1.50	192.168.1.5	TCP	74	ndl-aas > 41887 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TSval=60638 TSecr
11	1.243022000	192.168.1.5	192.168.1.50	TCP	66	41887 > ndl-aas [ACK] Seq=1 Ack=1 Win=14656 Len=0 TSval=29401 TSecr=60638
12	1.249062000	192.168.1.1	192.168.1.50	DNS	138	Standard query response 0x0250
13	1.249268000	192.168.1.50	192.168.1.1	DNS	73	Standard query 0x4d96 A ads.mopub.com
14	1.250422000	192.168.1.5	192.168.1.50	HTTP	595	GET http://ads.mopub.com/m/ad?v=6&id=821da82a2a8649d9ade826bcaa524e3b&nv=2.1&dn=samsung%2CGal
15	1.250448000	192.168.1.50	192.168.1.5	TCP	66	ndl-aas > 41887 [ACK] Seq=1 Ack=530 Win=30720 Len=0 TSval=60641 TSecr=29402
16	1.259158000	192.168.1.1	192.168.1.50	DNS	169	Standard query response 0x4d96 A 74.201.202.213 A 74.201.202.218 A 74.201.202.221 A 199.16.1
17	1.259429000	192.168.1.50	74.201.202.213	TCP	74	40847 > http [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=60643 TSecr=0 WS=1024
18	1.259614000	192.168.1.50	74.201.202.213	TCP	74	40848 > http [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=60643 TSecr=0 WS=1024
19	1.279176000	74.201.202.213	192.168.1.50	TCP	74	http > 40848 [SYN, ACK] Seq=0 Ack=1 Win=14480 Len=0 MSS=1460 SACK_PERM=1 TSval=1875204698 TSe
20	1.279187000	192.168.1.50	74.201.202.213	TCP	66	40848 > http [ACK] Seq=1 Ack=1 Win=29696 Len=0 TSval=60648 TSecr=1875204698
21	1.279272000	192.168.1.50	74.201.202.213	HTTP	666	GET /m/ad?v=6&id=821da82a2a8649d9ade826bcaa524e3b&nv=2.1&dn=samsung%2CGalaxy%20Nexus%2Cmysid&
22	1.280111000	74.201.202.213	192.168.1.50	TCP	74	http > 40847 [SYN, ACK] Seq=0 Ack=1 Win=14480 Len=0 MSS=1460 SACK_PERM=1 TSval=1875204698 TSe
23	1.280120000	192.168.1.50	74.201.202.213	TCP	66	40847 > http [ACK] Seq=1 Ack=1 Win=29696 Len=0 TSval=60648 TSecr=1875204698
24	1.280184000	192.168.1.50	74.201.202.213	HTTP	666	GET /m/ad?v=6&id=cd93fa0de8a940a7b5776bbbf7838905&nv=2.1&dn=samsung%2CGalaxy%20Nexus%2Cmysid&
25	1.304431000	74.201.202.213	192.168.1.50	TCP	66	http > 40848 [ACK] Seq=1 Ack=601 Win=15872 Len=0 TSval=1875204749 TSecr=60648
26	1.305305000	74.201.202.213	192.168.1.50	TCP	66	http > 40847 [ACK] Seq=1 Ack=601 Win=15872 Len=0 TSval=1875204749 TSecr=60648
27	1.455851000	74.201.202.213	192.168.1.50	TCP	1514	[TCP segment of a reassembled PDU]
28	1.455870000	192.168.1.50	74.201.202.213	TCP	66	40847 > http [ACK] Seq=601 Ack=1449 Win=32768 Len=0 TSval=60692 TSecr=1875204787
29	1.456750000	74.201.202.213	192.168.1.50	TCP	1514	[TCP segment of a reassembled PDU]

Figure 2.7: A packet trace following the communication of a MoPub ad request. The highlighted packet (number 14) is the ad request being sent off. Notice the lack of secure communication on the link, which would be indicated by the presence of TLS packets.

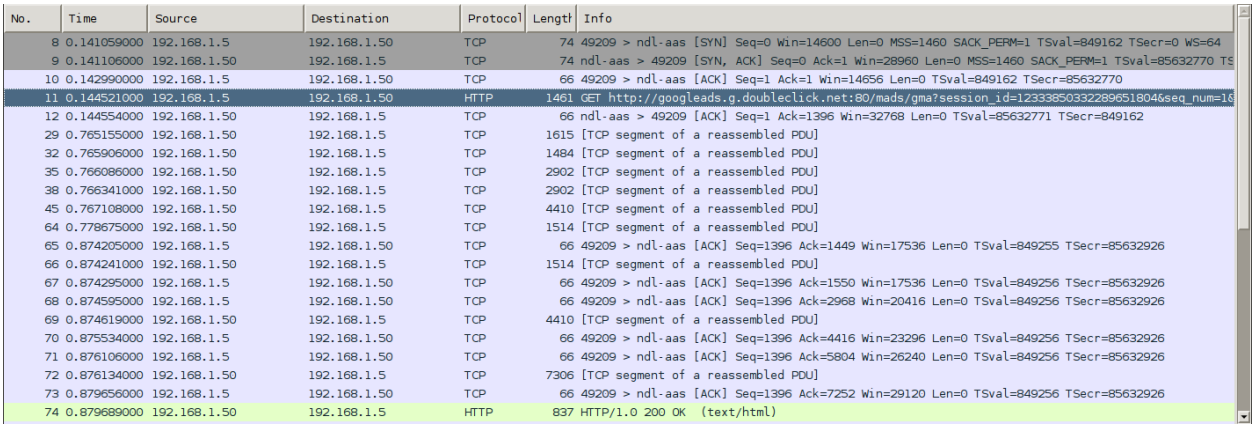
No.	Time	Source	Destination	Protocol	Length	Info
9	1.240958000	192.168.1.5	192.168.1.50	TCP	74	41887 > ndl-aas [SYN] Seq=0 Win=14600 Len=0 MSS=1460 SACK_PERM=1 TSval=29401 TSecr=0 WS=64
10	1.240995000	192.168.1.50	192.168.1.5	TCP	74	ndl-aas > 41887 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TSval=60638 TSecr=2
11	1.243022000	192.168.1.5	192.168.1.50	TCP	66	41887 > ndl-aas [ACK] Seq=1 Ack=1 Win=14656 Len=0 TSval=29401 TSecr=60638
14	1.250422000	192.168.1.5	192.168.1.50	HTTP	595	GET http://ads.mopub.com/m/ad?v=6&id=821da82a2a8649d9ade826bcaa524e3b&nv=2.1&dn=samsung%2CGalax
15	1.250448000	192.168.1.50	192.168.1.5	TCP	66	ndl-aas > 41887 [ACK] Seq=1 Ack=530 Win=30720 Len=0 TSval=60641 TSecr=29402
64	1.588119000	192.168.1.50	192.168.1.5	TCP	4293	[TCP segment of a reassembled PDU]
65	1.588367000	192.168.1.50	192.168.1.5	TCP	1762	[TCP segment of a reassembled PDU]
66	1.591777000	192.168.1.5	192.168.1.50	TCP	66	41887 > ndl-aas [ACK] Seq=530 Ack=1449 Win=17536 Len=0 TSval=29446 TSecr=60725
67	1.593235000	192.168.1.5	192.168.1.50	TCP	66	41887 > ndl-aas [ACK] Seq=530 Ack=2897 Win=20416 Len=0 TSval=29446 TSecr=60725
68	1.593435000	192.168.1.5	192.168.1.50	TCP	66	41887 > ndl-aas [ACK] Seq=530 Ack=4228 Win=23296 Len=0 TSval=29446 TSecr=60725
69	1.593448000	192.168.1.5	192.168.1.50	TCP	66	41887 > ndl-aas [ACK] Seq=530 Ack=5676 Win=26240 Len=0 TSval=29446 TSecr=60725
70	1.593715000	192.168.1.5	192.168.1.50	TCP	66	41887 > ndl-aas [ACK] Seq=530 Ack=5924 Win=29120 Len=0 TSval=29446 TSecr=60725
274	2.163429000	192.168.1.50	192.168.1.5	HTTP	4036	HTTP/1.0 200 OK (text/html)
275	2.166622000	192.168.1.5	192.168.1.50	TCP	66	41887 > ndl-aas [ACK] Seq=530 Ack=7372 Win=32000 Len=0 TSval=29519 TSecr=60869
276	2.166981000	192.168.1.5	192.168.1.50	TCP	66	41887 > ndl-aas [ACK] Seq=530 Ack=8820 Win=34880 Len=0 TSval=29519 TSecr=60869
277	2.167002000	192.168.1.5	192.168.1.50	TCP	66	41887 > ndl-aas [ACK] Seq=530 Ack=9894 Win=37824 Len=0 TSval=29519 TSecr=60869
278	2.227188000	192.168.1.5	192.168.1.50	TCP	66	41887 > ndl-aas [FIN, ACK] Seq=530 Ack=9894 Win=37824 Len=0 TSval=29527 TSecr=60869
279	2.227445000	192.168.1.50	192.168.1.5	TCP	66	ndl-aas > 41887 [FIN, ACK] Seq=9894 Ack=531 Win=30720 Len=0 TSval=60885 TSecr=29527
280	2.229323000	192.168.1.5	192.168.1.50	TCP	66	41887 > ndl-aas [ACK] Seq=531 Ack=9895 Win=37824 Len=0 TSval=29527 TSecr=60885

Figure 2.8: A packet trace following the communication of a MoPub ad request. The highlighted packet (number 14) is the ad request being sent off. The response can be seen in as packet 274.



No.	Time	Source	Destination	Protocol	Length	Info
11	0.144521000	192.168.1.5	192.168.1.50	HTTP	1461	GET http://googleads.g.doubleclick.net:80/mads/gma?session_id=1233385033228965180 4&seq_num=1&
12	0.144554000	192.168.1.50	192.168.1.5	TCP	66	ndl-aas > 49209 [ACK] Seq=1 Ack=1396 Win=32768 Len=0 TSval=85632771 TSecr=849162
13	0.145313000	192.168.1.50	74.125.227.237	HTTP	1515	GET /mads/gma?session_id=1233385033228965180 4&seq_num=1&rm=2&js=afma-sdk-a-v6188000.4452000.1
14	0.155375000	192.168.1.5	192.168.1.50	HTTP	620	Continuation or non-HTTP traffic
15	0.155411000	192.168.1.50	192.168.1.5	TCP	66	ndl-aas > 56276 [ACK] Seq=1 Ack=555 Win=33 Len=0 TSval=85632774 TSecr=849164
16	0.155810000	192.168.1.50	173.194.115.98	TLSv1	620	Application Data
17	0.191511000	74.125.227.237	192.168.1.50	TCP	66	http > 49814 [ACK] Seq=1 Ack=1450 Win=400 Len=0 TSval=3324961342 TSecr=85632771
18	0.214666000	173.194.115.98	192.168.1.50	TLSv1	475	Application Data
19	0.214703000	192.168.1.50	173.194.115.98	TCP	66	47435 > https [ACK] Seq=555 Ack=410 Win=42 Len=0 TSval=85632788 TSecr=853369089
20	0.214931000	192.168.1.50	192.168.1.5	HTTP	475	Continuation or non-HTTP traffic
21	0.218391000	192.168.1.5	192.168.1.50	TCP	66	56276 > ndl-aas [ACK] Seq=555 Ack=410 Win=455 Len=0 TSval=849172 TSecr=85632789
22	0.239128000	192.168.1.5	192.168.1.50	TCP	74	56282 > ndl-aas [SYN] Seq=0 Win=14600 Len=0 MSS=1460 SACK_PERM=1 TSval=849174 TSecr=0 WS=64
23	0.239165000	192.168.1.50	192.168.1.5	TCP	74	ndl-aas > 56282 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TSval=85632795 TS
24	0.241662000	192.168.1.5	192.168.1.50	TCP	66	56282 > ndl-aas [ACK] Seq=1 Ack=1 Win=14656 Len=0 TSval=849175 TSecr=85632795
25	0.242418000	192.168.1.5	192.168.1.50	HTTP	383	CONNECT pagead2.googlesyndication.com:443 HTTP/1.1
26	0.242450000	192.168.1.50	192.168.1.5	TCP	66	ndl-aas > 56282 [ACK] Seq=1 Ack=318 Win=30720 Len=0 TSval=85632795 TSecr=849175
27	0.764968000	74.125.227.237	192.168.1.50	TCP	1484	[TCP segment of a reassembled PDU]
28	0.764985000	192.168.1.50	74.125.227.237	TCP	66	49814 > http [ACK] Seq=1450 Ack=1419 Win=164 Len=0 TSval=85632926 TSecr=3324961915
29	0.765155000	192.168.1.50	192.168.1.5	TCP	1615	[TCP segment of a reassembled PDU]
30	0.765822000	74.125.227.237	192.168.1.50	TCP	1484	[TCP segment of a reassembled PDU]
31	0.765829000	192.168.1.50	74.125.227.237	TCP	66	49814 > http [ACK] Seq=1450 Ack=2837 Win=167 Len=0 TSval=85632926 TSecr=3324961915

Figure 2.9: A packet trace following the communication of an AdMob ad request. The highlighted packet (number 11) is the ad request being sent off. Notice the use of HTTPS as indicated by the TLSv1 packets that are sent shortly after the ad request, packets 16 and 18.



No.	Time	Source	Destination	Protocol	Length	Info
8	0.141059000	192.168.1.5	192.168.1.50	TCP	74	49209 > ndl-aas [SYN] Seq=0 Win=14600 Len=0 MSS=1460 SACK_PERM=1 TSval=849162 TSecr=0 WS=64
9	0.141106000	192.168.1.50	192.168.1.5	TCP	74	ndl-aas > 49209 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TSval=85632770 TS
10	0.142990000	192.168.1.5	192.168.1.50	TCP	66	49209 > ndl-aas [ACK] Seq=1 Ack=1 Win=14656 Len=0 TSval=849162 TSecr=85632770
11	0.144521000	192.168.1.5	192.168.1.50	HTTP	1461	GET http://googleads.g.doubleclick.net:80/mads/gma?session_id=1233385033228965180 4&seq_num=1&
12	0.144554000	192.168.1.50	192.168.1.5	TCP	66	ndl-aas > 49209 [ACK] Seq=1 Ack=1396 Win=32768 Len=0 TSval=85632771 TSecr=849162
29	0.765155000	192.168.1.50	192.168.1.5	TCP	1615	[TCP segment of a reassembled PDU]
32	0.765906000	192.168.1.50	192.168.1.5	TCP	1484	[TCP segment of a reassembled PDU]
35	0.766086000	192.168.1.50	192.168.1.5	TCP	2902	[TCP segment of a reassembled PDU]
38	0.766341000	192.168.1.50	192.168.1.5	TCP	2902	[TCP segment of a reassembled PDU]
45	0.767108000	192.168.1.50	192.168.1.5	TCP	4410	[TCP segment of a reassembled PDU]
64	0.778675000	192.168.1.50	192.168.1.5	TCP	1514	[TCP segment of a reassembled PDU]
65	0.874205000	192.168.1.5	192.168.1.50	TCP	66	49209 > ndl-aas [ACK] Seq=1396 Ack=1449 Win=17536 Len=0 TSval=849255 TSecr=85632926
66	0.874241000	192.168.1.50	192.168.1.5	TCP	1514	[TCP segment of a reassembled PDU]
67	0.874295000	192.168.1.5	192.168.1.50	TCP	66	49209 > ndl-aas [ACK] Seq=1396 Ack=1550 Win=17536 Len=0 TSval=849256 TSecr=85632926
68	0.874595000	192.168.1.5	192.168.1.50	TCP	66	49209 > ndl-aas [ACK] Seq=1396 Ack=2968 Win=20416 Len=0 TSval=849256 TSecr=85632926
69	0.874619000	192.168.1.50	192.168.1.5	TCP	4410	[TCP segment of a reassembled PDU]
70	0.875534000	192.168.1.5	192.168.1.50	TCP	66	49209 > ndl-aas [ACK] Seq=1396 Ack=4416 Win=23296 Len=0 TSval=849256 TSecr=85632926
71	0.876106000	192.168.1.5	192.168.1.50	TCP	66	49209 > ndl-aas [ACK] Seq=1396 Ack=5804 Win=26240 Len=0 TSval=849256 TSecr=85632926
72	0.876134000	192.168.1.50	192.168.1.5	TCP	7306	[TCP segment of a reassembled PDU]
73	0.879656000	192.168.1.5	192.168.1.50	TCP	66	49209 > ndl-aas [ACK] Seq=1396 Ack=7252 Win=29120 Len=0 TSval=849256 TSecr=85632926
74	0.879689000	192.168.1.50	192.168.1.5	HTTP	837	HTTP/1.0 200 OK (text/html)

Figure 2.10: A packet trace following the communication of a MoPub ad request. The highlighted packet (number 11) is the ad request being sent off. The response can be seen in as packet 74.

Chapter 3

Android Threat Surface

This chapter investigates the different threats that Android is vulnerable to and the ease of exploiting those threats.

3.1 Attacker Motivation

There are many reasons why an attacker wants to gain access to a mobile device owned by somebody else. Those could include financial incentives in spreading malware or hoping to collect banking or credit card information. It is also possible now that there are more and more health apps, that the attacker wants access to this highly prized data. It could also be desirable to obtain private and personal information in order to sell to another party. Because mobile devices are frequently brought to work and connected to corporate networks, these could be means of infecting corporations and stealing trade secrets. It was recently demonstrated that a cell phone was able to detect and intercept signal data being sent to computer displays as a means of capturing passwords [17]. This was demonstrated using an air-gapped computer with the wireless networks of the cell phone disabled.

3.2 Attacker Access

An attacker can gain access to a mobile device in a number of ways. They can include repackaging legitimate apps to carry a malicious payload. While the primary online stores for apps keep these types of apps to a minimum, they are more common in unofficial app stores. Attackers can also gain access through enticing users to click on links or images that direct the users to malicious webpages that then download a package to the users device, so called "drive-by downloads". It is also possible for access to be granted by attackers exploiting vulnerabilities within the mobile device. This is often associated with drive-by downloads. But attacks are not always active, they can also be passive. Attackers can follow specific devices and people around, watching for their specific device through a wireless sensor networks or by monitoring web traffic that is hosted in public places or on unsecured WiFi networks.

3.3 Android Segmentation

One of the most defining characteristics of Android is the split ownership between Google, the developer of the Android operating system, and the device manufacturers, who customize Android to work on specific networks and add non-essential UI enhancements and proprietary apps in collaboration with the network operator. As a result of these modifications, it is ultimately the responsibility of the device manufacturers and network operators to update handsets to the latest version released by Google. For each release of Android from Google, the operators must update their software to ensure it continues to work as expected and then make those updates available to customers, resulting in a delay between the release of the software from Google and final availability to end users. As a result of these

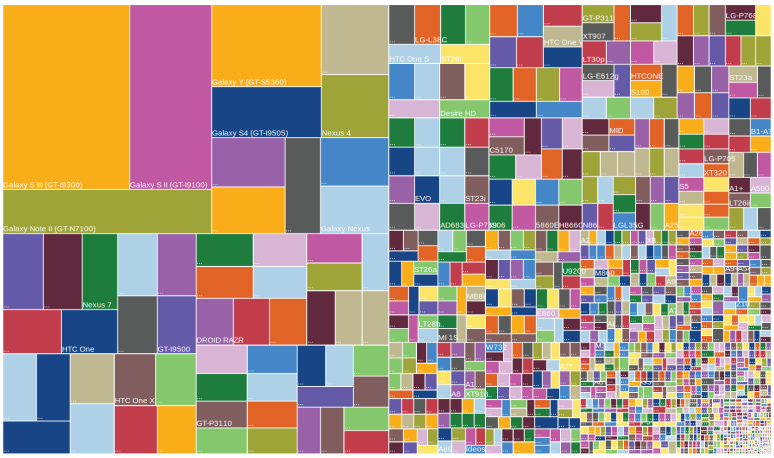


Figure 3.1: The popularity of specific Android devices in July 2013 according to [28]. (OpenSignal. *Android Fragmentation Visualized.* 2013. URL: `http://opensignal.com/reports/fragmentation-2013/` (visited on 11/20/2014). Used under fair use, 2014.)

manufacturer-operator relationships, and the resulting delays in updating devices to recent versions of Android, the market has becoming increasingly segmented between network operators and even among devices on the same network. It should also be noted that part of the segmentation is the result of manufacturer-operators only updating their current and previous flagship devices to the latest version. This leaves those who have devices that are several years old out of the update loop, exposing them to more recent vulnerabilities. The argument made by the manufacturer-operators is primarily based on financial incentives but also partly on only wanting to continue to support the latest in a rapidly changing hardware realm. This segmentation in device popularity can be seen in Figure 3.1.

Apple's iOS ecosystem does not suffer from the same fate as all of the hardware is controlled by Apple and so when a new version of iOS is released, Apple has already done all the work that is needed to ensure that it works on all supported devices.

Due to the popularity of Android as well as its open source nature, it has been thoroughly studied by researchers who publish security vulnerabilities after notifying Google so that corrections can be made. But this only helps those who are able to receive those updates

and leaves the rest of the market open to potential attack. These researchers often detail the vulnerability very extensively and in some cases, even publish tools to help exploit that vulnerability [21].

3.4 Data Theft Attacks

In an ever increasing digital world, people are defined by their electronic records. Those records can take many forms: medical, financial, to the more mundane social network accounts and Netflix preferences. However, the more data that is stored in any one location, the more an attacker can learn about an individual, their preferences and potentially even their real identity. This is all done by gaining data on potential victims and compiling it together.

3.4.1 Types of Data

There are many types of information that are stored on mobile devices. If they are personal, they may contain or leak the following information:

- device information to include device id, Android version, certain installed apps, network carrier
- Location (GPS coordinates and accuracy)
- Audio
- Video
- Text messages

- Phone call log
- Banking information and credentials
- Medical information
- Personal habits such as work and home location
- Social networks
- Email which can contain
  - Home address
  - Birthday
  - Full or partial SSN
  - Password reset emails for other accounts

In a corporate setting, whether the device is owned by the individual or the company, these devices may contain:

- Merger and acquisition details
- Proprietary information and trade secrets
- Strategic vision
- Employee personal data (SSN, Home address, employment information)

3.4.2 Access Approaches

There are a multitude of ways that an attacker can gain access to the data and information that they desire, so many so, that there is an entire website dedicated to the collection

of exploits for vulnerable applications and languages. The website exploit-db.com hosts a collection of more than 30,000 exploits affecting everything from PHP to Windows to iOS and Android [31].

Due to the scope of potential vulnerabilities that affect the Android platform, it is often helpful to use other sources to determine their applicability as well as severity. To assist with this, NIST hosts the National Vulnerability Database [34] which provides a description as well as a severity score for all CVE entries [25]. Additionally, private security research firms have also published their findings regarding specific vulnerabilities and they often have assessments of their impact. One such example is found in [18].

The most obvious is for some sort of malicious app to be downloaded to the phone. This could be done through repackaging a legitimate app and placing it on a less common app store bundled with malware. This would result in the complete exploitation of the target device and any information that was present on it. Depending on the device and the malware, it might also be possible to gain root privileges on the device for even more control. However, this would depend on the device itself and a vulnerability with one might not work for another. However, given the high value of success, many attackers continue to work towards this goal even though their success rate might be less than through other means.

It is also possible for an attacker to trick a user into installing a malicious app or by installing an app in the background without the users' permission or by having the user click on a link that has been disguised as something else. This is known as a drive by download and has been common on web platforms for many years. The difficulty with this attack is the need to trick the user into clicking the link or installing the package. While users are wary of such tactics on their desktop or laptop computers, anecdotal evidence suggests that such behaviours seem to be less prevalent with mobile devices, potentially due to their age relative to standard computers. The success of these attacks depends on primarily two factors: the

allure of the link and the payload loaded on the site. The more attractive or more interesting a link or the text in a link is, the more likely people will be to click on that link. This is especially true if they already trust the site they are visiting. If the trust can be extended, users are more susceptible. The other component to this attack is the success of actually loading the payload onto the target. Users may have devices that are immune to certain attacks or may be running software versions that have been patched against the attack that is being exploited.

Depending on the type of information that an attacker wants to get, there are also a number of side channel attacks that have been demonstrated to include using the FM radio to install malware [8] as well as using both the WiFi and cellular connections simultaneously as in [5]. These attacks are more complicated due to the hardware requirements to successfully launch the attack and the need to have the target within range of the equipment.

3.4.3 Exfiltration Approaches

When it comes time for the attacker to offload the information they have gathered from the target handset, there are a number of different approaches they can use to exfiltrate the data.

NFC attacks are difficult to achieve due to the requirement of being in close physical proximity to the target. However, recent developments may make this a more attractive attack vector. Many commuters in metropolitan areas use their mobile device to pass the time during their commute which would allow an attacker in the same vicinity a significant window as well as physical proximity in order to mount their attack. This attack also requires that users have enabled the NFC radio on their device, a condition of unknown quantity. However, it should be noted that some metro areas are starting to roll out NFC enabled

fare gates to allow passengers to avoid carrying a specialized fare card with them [38] and to instead pay the fare with their mobile device. Continuing in our attack scenario, this would heighten the probability of susceptible devices and a successful attack.

Following a successful attack to gain access to a device, an attacker can direct the device to transfer content to a remote server. This has a high probability of success once the initial attack has been successful. As mobile devices have almost constant connection to the internet, such a data transfer is quite feasible. One challenge that attacks face is deciding what information should be transferred. While an internet connection is quite common, the speed of that connection has a high degree of variability depending on whether older cellular technologies or home based WiFi are being used. Defending against such attacks can be done at the network layer by tracking incoming and outgoing connections. If a sudden increase in mobile data usage occurs, this could indicate such an attack.

There has also been recent work done by Hayashi et al. to read key presses from a mobile device based on the EM emanations from device screens [19]. While the attack range is relatively short at two meters, all devices are potential targets, making this a promising attack vector. The authors do provide a working countermeasure against their attack by adding a transparent conductive film to the surface of the device

3.5 Prior Work on Android Advertisement Security

There are generally two different camps of work that have been done with regards to investigating Android ad libraries. The first of these is to look at the source code for potential information leakage about either the user or the device to the ad network or the advertiser themselves. The work done by [16, 7, 6, 27] belong to this group. The second camp are those researchers who have provided recommendations for mitigating the privacy leakage by

the ad network by proposing solutions to separate the ad library from the base application. The work done by [22, 29, 33, 4] falls into this groups.

3.5.1 Extraneous Permissions

Within the realm of malicious Android apps, there has been a significant body of work both to uncover new vulnerabilities [39] as well as to help classify malicious apps [35, 30, 40].

The work done by Grace et al. in [16] provides the most foundational work to this thesis by being the first to describe the unsafe behaviour of ad networks and the risks they expose the users of applications to. The work they did was to statically analyze the first 100 unique ad networks they encountered while decompiling the most popular free apps on the Google Play store. Once decompiled, the authors looked for personally identifiable information that was traceable to a network sink. They learned that most uses of potentially privacy violating code were used for identification within the advertising network. They also concluded that the more popular advertising networks, such as AdMob, behaved nicely while the smaller ad libraries more commonly used potentially privacy violating practices.

The paper by Felt et. al [7], analyzes the permissions requested by different Android apps and the percentages of those that requested permissions that were not needed by the code. The authors state that many instances of over-permission can be attributed to developer confusion about what permissions are needed to perform what actions and to use what methods in the Android API. The authors created a tool that generated a set of permissions that were required based on the bytecode of a decompiled Android app. This set was then compared to the set that the app included in the manifest file to determine if there was a case of over-permission. Part of their analysis included determining that 323 of the 900 apps they tested included unnecessary permissions. The top five most commonly requested and

Table 3.1: The six most common unnecessary permissions and the percentage of over-privileged applications that requested them as reported by [7]. (Adrienne Porter Felt et al. "Android Permissions Demystified". In: *Proceedings of the 18th ACM Conference on Computer and Communications Security.* CCS 11. Chicago, Illinois, USA: ACM, 2011, pp. 627–638)

Permission	**Usage**
ACCESS_NETWORK_STATE	16%
READ_PHONE_STATE	13%
ACCESS_WIFI_STATE	8%
WRITE_EXTERNAL_STORAGE	7%
CALL_PHONE	6%
ACCESS_COARSE_LOCATION	6%

unused permissions are listed in Table 3.1

The work done by Enck in [6] views the security of Android devices from a high level. By developing an Android Dalvik decompiler which translates Dalvik bytecode into Java bytecode, the authors were able to statically analyze the 1100 top free apps to determine which personally identifying information was being transmitted by the app off the device. They found that numerous device specific identifiers were being used, such as IMSI (subscriber identifier), IMEI (device identifier), phone number and ICC-ID (SIM card serial number). They also found that some of these were used as part of the registration process for using the app and were used to identify the device with both advertising networks as well as for their own internal purposes. They found that around 20% of the apps studied used these identifiers in some way, most of them to track individual users. They also found that these were often done in plain text over HTTP.

Narayanan et al. [27] have contributed to the static analysis landscape of apps by increasing the state of the art in detection of ad libraries and the associated classes in an app. They then use to detect over privileges that an app has requested that are not needed. Their solution was particularly effective against common practices such as obfuscation and had a

95% success rate.

3.5.2 Separating App and Advertisement Permissions

In order to protect the privacy of users and to keep permissions to the minimum set required, the authors of [22] propose SanAdBox, a sandbox utility that separately installs an application and its ad library. By doing so, the authors claim to be able to protect the user from privacy leakage by the ad library. The separate installation is done via a custom installer that keeps track of interactions between the base app and the ad library, including interactions during app use. The prototype installer added a latency of 20ms to ad loading time.

An idea that works at a lower level is offered in [29] whereby Pearce et al. propose the introduction of a new Android service that would handle all the advertising needs of all apps installed on a device. This would separate ads from the apps they support to protect user privacy. The proposed idea would be to integrate the ad library with the Android system API have apps interact with those to deliver and display ads. User security would be enhanced by limiting the app permissions which limits the attack surface of apps. Privacy would also be enhanced by allowing for more detailed monitoring of personal information being sent to ad libraries. In support of their idea, two new permissions are added that indicate that an app has advertising and needs access to the new Android API.

Shekhar et al. in [33], propose a visible separation of apps from their ad libraries. In order to achieve this, there are two activities that the user sees. The top most one is the default app and it has transparent regions for ads which are displayed from the ad activity directly underneath the default app. This also allows advertising libraries to be run as services instead and supply the ads for all the apps that use that particular ad library across all apps

installed on the device. This would require modifications in the way that Android controls the life cycle of apps.

The work done by Do in [4] took a very different approach to the issue of Android permissions and went about reverse engineering and repackaging social networking applications to selectively remove an apps permission to unnecessary resources such as contact lists and phone state. The authors then tested each app to see if there were any side affects due to the changes that they made.

Of particular issue among all of these papers was the usage and identification of dynamic class loading and reflection. Both of these allow the source code to be determined at run time, as opposed to at compile time. As was the case with all of these analyses, statically inspecting the code did not allow for a complete picture of all that was happening within the ad library.

3.5.3 Other works

The work done by Kuzuno and Tonami in [24] tries to counteract the leakage of privacy information from Android devices. The authors accomplish this by clustering HTTP packets and scanned them for sensitive information. They were able to do this without any special permissions or changes to the Android OS. Their work is directly related to this research.

The work completed by Book et al. in [3] investigates how ad libraries interact with the underlying application. The authors work towards developing an understanding of which ad libraries are most commonly used when accessing privacy related APIs. [10] describes the impact of Android applications that have been cloned, repackaged and redistributed by other parties. Some may do so for malicious purposes, such as adding a payload to a popular game, but the authors also consider other, more subtle intention such as redirecting

advertising revenue. Seneviratne et al. in [32] investigated the possibility of having privacy preserving ads be delivered to mobile devices as well as doing so in a bandwidth efficient manner.

Some additional interesting work was done by Ruiz in [26] which looked at the possibility that the number of ad libraries an app had might affect its user rating negatively; their findings did not support their hypothesis. However, while conducting the work, they surveyed a number of apps and found that a majority of them had five or fewer ad libraries with some apps including as many as 28 different ad libraries.

The work done by Wei et al. in [37] is very similar to the work done here. They analyze different ad networks for information leakage as well as explore the possibility of attacking vulnerable WebViews with malicious Javascript.

3.5.4 Contributions

This research makes the following specific contributions to the work that has been previously completed. This work contributes to the knowledge of feasible attacks that have been conducted against vulnerable WebViews using the `addJavascriptInterface` framework for Android apps. While the exploitation of WebViews has been well studied by both private and academic researchers, their application within ad supported apps has not been. Through analyzing the ad requests by two ad libraries, MoPub and AdMob, we determine that this attack is indeed feasible against particular combinations of ad networks and vulnerable devices.

Additionally, this research provides a detailed analysis of the specific information that is present in an ad request URL from the studied ad networks. Determining what is included in such requests is the first step in addressing security vulnerabilities. We also analyze

the impact that changing user settings on the mobile device has on the contents of the ad requests, in particular showing that user location preferences are reflected in MoPub requests while having no noticeable affect for AdMob requests.

Finally, a series of recommendations are made for app developers, users and ad networks to make use of current encryption technologies such as HTTPS and temporarily disabling location settings to protect user security. Users are also encouraged to keep their devices updated with the latest version of Android available. Recommendations to developers include such ideas as limiting the types of advertising that are displayed for older devices that are vulnerable to these attacks and auditing their code for the purposes of uncovering vulnerabilities. We also look into the future and provide recommendations for changes that could be made by Google to the Android platform and ad networks to the way they transmit data that would enable a more secure advertising ecosystem. These are changes that would take some time to implement but would be beneficial to the user community in the long run.

Chapter 4

Advertisement Vulnerability Assessment

In this chapter, we are going to discuss two different attacks that are possible against ad supported apps.

4.1 `addJavascriptInterface` Attack

In an effort to create a flexible framework that minimizes development time and allows for quick migration of apps to the Android platform, Google created a Javascript bridge that would allow apps to load specific HTML content and display that within the app without having to load the web browser on the device. However, this bridge is vulnerable to a number of different attacks that are not present on other platforms.

```
function showAndroidToast(message)
{
    Android.showToast(message);
}
```

Figure 4.1: The following Javascript code calls the Java method to display a Toast to the user with the specified message

```
webView.addJavascriptInterface(new JavascriptInterface(this), "Android");
```

Figure 4.2: Initialization required in the Java code to establish a Javascript Interface. The `Android` keyword is used in the Javascript to access the Java methods.

4.1.1 Description

Ads are frequently displayed in an app using a WebView, which essentially displays a small webpage. This has a number of advantages from the perspective of the advertisers in that they do not need to create additional advertising materials in order to reach the mobile market aside from changing the size of the image that is displayed. For the creators of the ad, they are able to continue to use well known web based protocols and tricks, such as displaying animation. The distribution doesn't change as the ad is still requested via the web, regardless of whether the destination is a mobile device or a desktop or laptop. It is also possible for advertisers to add specially crafted Javascript to the ad in order to create animations or control custom analytics.

However, there is a known vulnerability in Android JellyBean 4.1 and older in the way WebViews handles the Javascript. From the app code, this functionality is added using the `addJavascriptInterface`. Once added, Java methods can be called and executed from Javascript as seen in Figures 4.1, 4.2, 4.3.

It is possible to call any public Java method through the Javascript sent to the WebView. Additionally, through the means of Java reflection, it is also possible to call methods that

```
public void showToast(String message)
{
    Toast.makeText(mContext, message, Toast.LENGTH_SHORT).show();
}
```

Figure 4.3: Java code called from Javascript to display a message to the user

have been marked as private. As an attack vector, it would possible to intercept the ad that is sent back to the device, replace any Javascript with custom code and then return the custom, malicious code to the user. This Javascript does not need to be activated in any way, such as requiring a user to click on the ad. Instead, it is possible to craft the Javascript to run as soon as the ad loads, requiring no user interaction. The user will be completely unaware that anything has happened to their device.

4.1.2 Applicability

As seen in Figure 4.4, Google reported that 46.8% of Android devices active for the seven day period ending 9 September 2014 were running Android 4.1 or older, which is vulnerable to the addJavascriptInterface attack as described in this research [12]. These versions are denoted in green. This figure could potentially be higher if the window for activity that Google used in their figures was extended beyond seven days. According to [36], there are more than one billion active Android devices worldwide. This means there were at least 468 million active Android devices were vulnerable to these attacks in September.

Google reported updated numbers for the seven day period ending on 3 November 2014, which are seen in Figure 4.5. The more recent numbers indicate that the percentages of vulnerable devices has decreased in the interval between the two reports, from 46.8% to 41.7%; a 10.9% decrease. Because of the multitude of factors playing into the adoption of newer versions of Android, it is difficult to predict the rate of decay for old versions as users

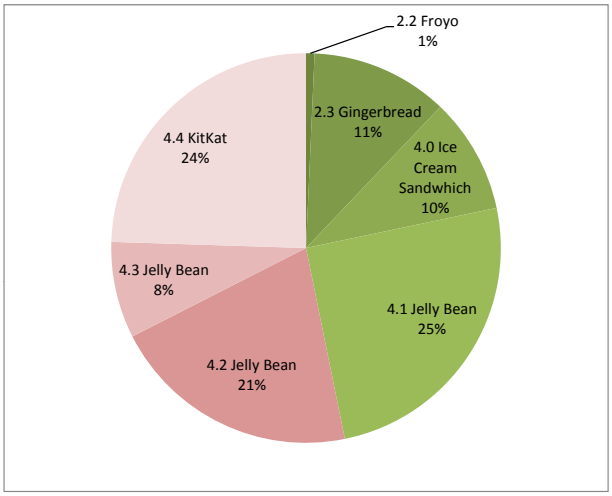


Figure 4.4: Breakdown of Android versions as of 9 Sep 2014 [12]. The sections marked in green are vulnerable to the `addJavascriptInterface` attack which are versions 4.1 and older. (Google. *Android Dashboards.* 2014. URL: `http://developer.android.com/about/dashboards/index.html` (visited on 10/17/2014). Used under fair use, 2014.)

migrate to newer devices or upgrade existing devices with the latest release of Android. If we inspect even older versions of Android, we see that Froyo and Gingerbread, versions 2.2 and 2.3, were released 20 May 2010 and 6 December 2010, respectively and still have a sizable active user base more than four years later of more than 11 million users. Considering the age of these versions of Android, it is quite reasonable to assume that an active user base of millions of users will continue to be present globally for many years to come that will be using these vulnerable versions of Android. The continued presence of vulnerable versions of Android make this a viable attack well into the future.

To address this issue, a change was made to Android 4.2 that requires any Java method that is accessed via Javascript have an annotation. Annotations are a form of metadata on the Java code that are used to identify certain functionality to the Java compiler. This change limits the number of methods that are accessible from Javascript by default. However,

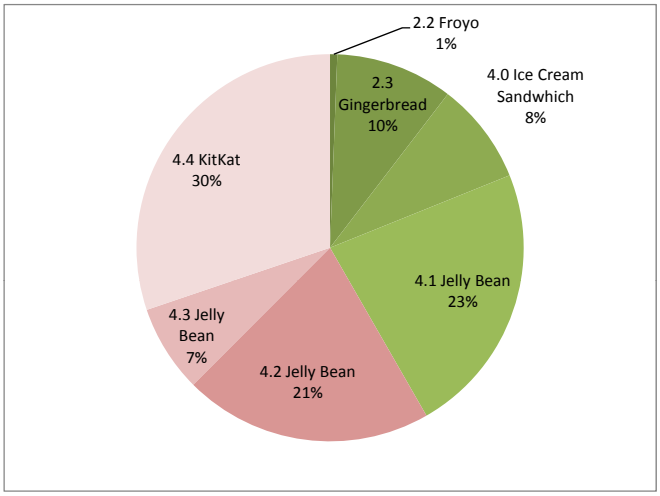


Figure 4.5: Breakdown of Android versions as of 3 Nov 2014 [12]. The sections marked in green are vulnerable to the `addJavascriptInterface` attack which are versions 4.1 and older. (Google. *Android Dashboards.* 2014. URL: `http://developer.android.com/about/dashboards/index.html` (visited on 11/10/2014). Used under fair use, 2014.)

according to the work done by [37], developers are still adding the annotation to their own custom methods which in turn access sensitive information on the phone, such as the list of contacts, call log and camera. This reduces the risk of attack in newer devices, but does not eliminate it.

4.1.3 Feasibility

The requirements to undertake this attack are malicious Javascript and a position from which to inject this Javascript into an ad request. The opportunity to gain a position from which to interject the attack Javascript could be as simple as setting up in the local coffee shop and waiting for patrons to access the unsecured WiFi that is frequently offered. An attacker could spoof the DNS request sent by the ad and respond with their own content, injecting the malicious content in the response. The Javascript that is required is quite straight forward, following standard Java reflection patterns.

4.2 Information Leakage Attack

People and their personal information are increasingly being put into electronic forms. To take advantage of all this personal information, advertisers have capitalized on the possibility of delivering specifically tailored ads against a target audience. While the advertisers' objective is to sell users some sort of product, if that same personal information were obtained by a malicious actor, their intentions cannot be as easily dismissed.

4.2.1 Description

The information leakage attack is determining information from an information source without being the intended destination. Colloquially, this is known as eavesdropping. In the context of ad networks and mobile devices, information about the device and the user are sent to various ad networks as part of the request to show ads. Information such as a unique device ID, location and carrier are sent as part of the request. This attack is when a malicious party is able to either intercept the ad request or monitor the request as it is sent over wireless links.

For the purpose of this investigation, we assume that the attacker does not modify the ad request in any way and is simply interested in skimming information that has already been included as part of the request by the ad library. We also assume that the attacker is not able to decrypt traffic that has been encrypted by standard means, such as HTTPS.

In an attacker is able to determine such information as location and correlate that to the time of the request over a period of time, an attacker can determine locations such as place of work and home. Based on the types and the names of an app, an attacker might also be able to determine who in a company the device belongs to. And because requests include unique device IDs, it is possible to separate different targets captured as part of the data set.

4.2.2 Applicability

As described in Section **??**, in order for an ad to be displayed on a device, a request for that ad must first take place. The objective of an ad is to appeal to an audience, convince them to see their content and then do something with that content. Seeing advertising content is called an Impression while doing something with that content, often making a purchase, is

called a Conversion. In order to increase the likelihood of impressions and conversions, ads must be appealing to specific market segments, and this requires that specific information about the user and the device be sent with the request. While additional analytics are gathered by the ad networks, personal information is required to add to the user profile, allowing the attacker to obtain specific information about the target.

While the amount of personal information that is sent as part of this request varies from ad library to ad library and even depends on user settings, there is a lot that can potentially be learned. It should also be noted that Ruiz in [26] and Shekhar in [33] showed that while it was the most common for an app to have only one ad library, it was not uncommon for many ad libraries to be included in the same app which is done by app developers do to increase their revenue. Combining these data sources allows for even more personal information to be gathered.

4.2.3 Feasibility

All that is required for this attack to happen is to be able to observe the ad request on a network. This can happen over unsecured WiFi networks which are common in places of business, such as coffee shops and at airports. Also feasible using specialized equipment and software is the ability to monitor cellular networks. This allows an attacker complete freedom in choosing which technology to focus on. They are also limited in attack range only by the range of their equipment and the range of the wireless technology of their choosing.

4.2.4 Permissions

As reported in [22], the most common permissions used by ad libraries are show in Table 4.1. What this shows is that ad advertisers are keenly interested in knowing the state of the phone

Table 4.1: The ten most common permissions used by ad libraries in the 100 most popular free apps

Permission	**Ad Networks requiring Permission**
INTERNET	63
ACCESS_NETWORK_STATE	42
READ_PHONE_STATE	38
ACCESS_FINE_LOCATION	32
ACCESS_WIFI_STATE	9
ACCESS_COARSE_LOCATION	8
VIBRATE	7
WAKE_LOCK	5
CAMERA	4
READ_CONTACTS	3

(whether it is in a call or not), access to what kind of network the device is connected to and the specific (GPS) location of the device. As the number of apps that are supported by ad libraries continues to increase, users have effectively decided that they are willing to allow this level of information be transmitted to advertisers.

Chapter 5

Experimental Results

This chapter details the specific results that were obtained during testing of the sample app and ad libraries.

5.1 Goals

The primary goal of the experimental results was to determine the feasibility of the two different types of attacks as well as to determine the effects that user settings have in preventing portions of the attacks.

5.2 Methodology

The system that was used to conduct the tests was the following. A virtual machine (VM) running Kali Linux 1.0.9 was used as a proxy server to sniff packets that were sent from a custom app that featured the two different ad networks running in separate Activities,

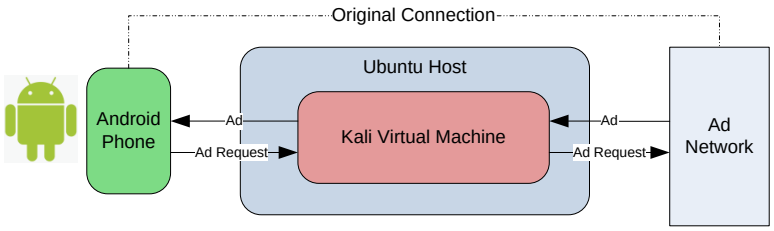


Figure 5.1: Testing environment architecture showing the Man in the Middle that the Kali Virtual Machine. Orignal web traffic is routed from the device through the VM to enable packet capture and analysis. For the `addJavascriptInterface` attacks, the ad requests were not forwarded and instead, a prepackaged HTML page containing malicious Javascript was returned.

or user views within the app, separating their functionality. The test devices, as noted in Table 5.1, were configured to use the Kali virtual machine as a proxy server. The proxy server was using Squid3 and DNS requests were intercepted using Bind9. The VM was hosted on a Ubuntu Linux 14.04 host with an Intel Core i5 processor and 8GB of RAM. The VM was also used as the DNS resolver to be able to inject malicious Javascript into the ad request WebViews. The architecture for this can be seen in Figure 5.1.

In order to analyze the information within the ad request, the ad Activities were loaded while capturing the packets for the request. These captures were then analyzed to determine what personal information was being included. Tests were also conducted by changing the user location preferences as well as whether the device was in air plane mode or not. Additionally, packet captures were also analyzed from a sample set of commercial apps that utilized the same ad libraries to ensure that there were no effects of running the ad libraries in isolation. It was determined that the testing environment had no impact on the results.

In order to test the `addJavascriptInterface` attack, the first step was to create a custom app that featured a WebView which was used to determine viable malicious Javascript and exploit the weaknesses in the WebView. Once the attack code had been perfected, it was then introduced into a test webpage posing as a advertisement. This injection was done by using

Table 5.1: Mobile devices used during testing and their respective versions of Android

Device	Android Version
Samsung Galaxy Nexus	4.2.2
Samsung Galaxy 2	4.1
HTC One X	4.1.2
Samsung Galaxy Ace	2.3

the VM as the DNS resolver for the target device. When a DNS request was detected for one of the ad network URLs, a prepackaged HTML page containing the malicious Javascript was returned instead of the ad. In a real attack, this could be disguised to look like a real ad or could simply replace the Javascript that is contained within the ad response. This work was then extended to attack the WebViews used to display ads in commercial apps obtained from the Google Play Store that also used the same ad library.

5.3 Software Framework

To test the various aspects of this research, three different versions of Android were used: 2.3, 4.1 and 4.2. These three different versions were represented by four different handsets, the details of which are listed in Table 5.1. Due to limitations in the way that proxy settings were configured in the Android 2.3 device, the results from that handset were limited in useful analysis.

5.4 Results for `addJavascriptInterface` Attack

With this attack, we were able to showcase a number of different attacks that can be carried out. The most straight forward attack was to inject custom written Java into the Javascript displayed for a WebView that sent a text message to a predefined number. The code for this

```
var sms = Android.getClass().forName("android.telephony.SmsManager")
    .getMethod("getDefault", null).invoke(null, null);
sms.sendTextMessage(phoneNumber, null, message, null, null);
```

Figure 5.2: Malicious Javascript code that sends a text message to the designated number.

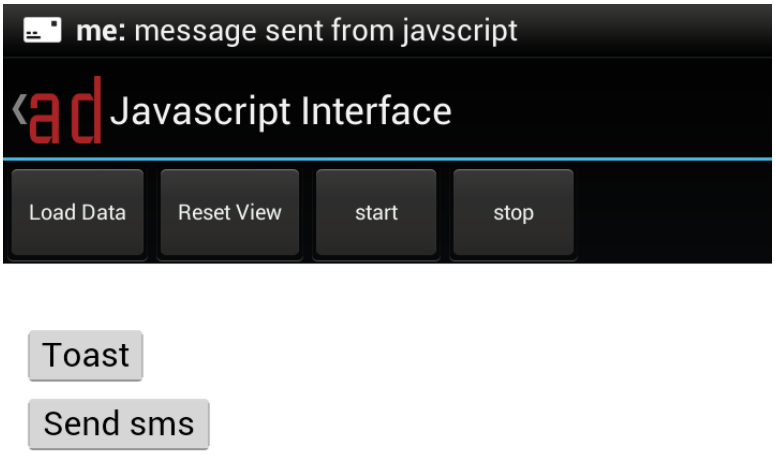


Figure 5.3: The results of the SMS attack against the target device. The SMS message has been received on the device and is displayed for preview in the notifications bar at the top of the figure. The sender "me" indicates that the device sent the message to itself.

attack can be seen in Figure 5.2 with the results as seen on the test device in Figure 5.3. The attack was then attempted to be extended to achieve another attack vector, such as enabling a remote shell or initializing an audio recording. However, these attacks were met with some technical difficulty and were not able to be fully implemented.

It was also attempted to inject the malicious Javascript into an ad request, but due to configuration settings in use by the ad libraries, the infected response would not load in the ad frame and the code would not execute. The ad frames were also protected against replay attacks using the same HTML content. It was determined that additional parameters accompanied the real ad responses but these could not be replicated.

As noted in Section 4.1, it is possible to call any public Java method as well as any private method through reflection. Due to reflection, the possibilities for attack vectors are only

limited by the attackers' imagination and hardware constraints. Due to the vast variety of possible attacks, these vectors were not exhaustively explored, with only a representative attack being undertaken to demonstrate the capabilities of such an attack. It is also noted in [37] that some mobile devices are vulnerable to certain attacks, such as remote video recording due to the lack of state validation that the hardware performs. This variability within the Android marketplace allows for creative attacks on a wide variety of devices, but the vulnerabilities that each device has are dependent on the hardware.

5.5 Results for Information Leakage Attack

Here, we detail the different parameters of interest that are sent out as part of the ad request URL by the two different ad libraries and highlight results that would allow attackers to potentially identify individuals. The complete listing of parameters for both ad networks are listed in Appendix A.

5.5.1 MoPub

In order to better understand the information that is being leaked by the MoPub ad request URL, the more interesting parameters are analyzed in detail. The complete listing of parameters as well as their analysis can be found in Table A.1 and Table A.2, the latter of which includes parameters that were not observed as part of the test data.

GPS Coordinates GPS coordinates for the device are a location specific value; tests were conducted to determine which of the different location settings were involved. Android has three location specific settings that allow apps to use different means to determine a user's

geographic location and all of these can be changed by the user. The three settings allow apps to access 1) GPS satellites to determine position, 2) use WiFi and mobile network services to determine position and 3) use location to determine what results are shown in Google products. Initial analysis led to the hypothesis that the GPS location setting was the sole control over the ad network sending coordinates as part of the request. However, after additional testing and further analysis, it was determined that if a user has specified to use only WiFi and mobile network based location, this not only reports coordinates, but can also result in a potentially more accurate location. Of particular interest in this is the fact that this does not require an active cellular or WiFi connection. The location that the network reports is based on the latest known position, with no reference to age. If a device has airplane mode turned on, the current position will not be updated until the connection is reestablished. The only way for a device to report no cellular network position is for the device to be started in airplane mode.

The accuracy of the geolocation using either GPS or WiFi was observed to be accurate down to 15 meters for GPS and 20 meters for Cellular/WiFi. According to the documentation for determining location, the accuracy that is reported is the 68% confidence circle around the users' actual location[14]. As seen in the table, there are no references to which technology the position is derived from. Additionally, when the GPS is used, a system level notification appears to the user indicating its use. This can be seen in Figure 5.4. This same notification is not used when cellular positioning is used, allowing the ad network to report specific geospatial locations without notifying the user.

When the app does not have the fine location permission, which is required for GPS locations, the geoposition reported by the network has an accuracy of 2000 meters. The documentation specifies that with only the coarse location permission, "[the app] will still return location results, but the update rate will be throttled and the exact location will be obfuscated to a

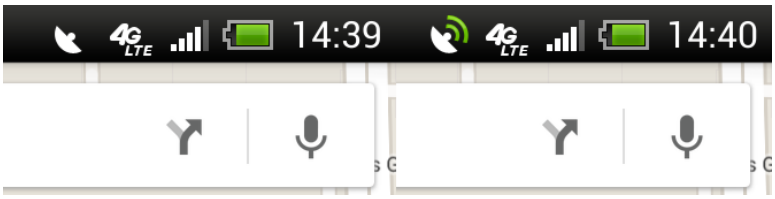


Figure 5.4: Different indicators that are shown in the notification area alerting the user to the use of GPS signals to identify their location. The indicator pulses until their position has been identified at which point the indicator becomes solid.

coarse level of accuracy" [15].

In order to achieve the increased GPS accuracy, Google Maps was first brought up and GPS was given an opportunity to pinpoint the location of the device. Being indoors, a "lock" was never achieved. The test app was then started and the ad request was watched for increased accuracy.

Device Name The specific make and model of the device was included in plain text as part of the request. In conjunction with an `addJavascriptAttack`, an attacker could specifically target the payload to exploit vulnerabilities of the device the user has.

Cellular Carrier One of the more surprising details that are transmitted as part of the MoPub ad requests are the cellular network that provides service to the device. This is indicated in a number of ways. First, some requests have a string value for the *cn* field, but the *mcc* and the *mnc* fields can also be used to identify the network carrier as a result of the unique cellular bands they have licensed.

Connection Type Additionally, the ad request URL also included information regarding the type of internet connection that the device currently had, such as whether that was cellular or WiFi. This information could be used in conjunction with the location to determine

if the attacker was inside their workplace or home.

Installed Apps The last parameter in the ad request informs the ad network whether or not the Twitter app is installed. While this makes some sense as MoPub is owned by Twitter, this behaviour was not observed with the AdMob requests, even though AdMob is owned by Google and there is a high likelihood that some Google apps have also been installed. With the ability of Twitter updates to include a position, it would be possible for an attacker to determine which device corresponds to specific ad requests.

5.5.2 AdMob

In analyzing the URL request strings sent from the AdMob library, the following list highlights the more interesting parameters. A complete listing of the request parameters and example values are contained in Appendix A.

Application Name While the MoPub requests did not include the app name, the responses from the ad network did. As these were not part of the analysis, it was interesting to observe this in the requests from AdMob. By having the app name, an attacker can correlate personal preferences that would otherwise only be available to the ad network. This information can then be used to identify the individual device.

MS A significant portion of the request URL was taken up by a parameter *ms* that was followed by a 342 alphanumeric string, most likely encrypted. It would seem that this is where the majority of the tracking information about each user and request is stored. As it is encrypted, this also protects the users' security and privacy from the eavesdropping attack.

Timing Information Sepcific timing information such as request time *treq*, time to fetch the ad $tfetch$, response time *tresponse*, load time *tload*, delay time to load ad *dload* and others were observed in only some of the requests. How some of these metrics were reported as part of the request is unknown. There was a correlation between the values of these and the time before the ad was displayed. Requests that had a low *tload* value displayed their ads much faster than those with a higher value. It is likely that there is a delay on the ad and it wasn't displayed until after *dload* or *tload* milliseconds had elapsed since the response back from the ad network. If these values are too large, an attacker could abort an attack as the likelihood for success probably decreases over time.

The *currts* of the first request URL in a sequence is also the value used for the *basets*. This value is then continued to be used for the *basets* for the rest of the requests in the same sequence.

Location After investigating the MoPub requests and their inclusion of latitude and longitude coordinates when location settings have been enabled by the user, the AdMob requests were similarly investigated. However, unlike MoPub, there are no clear text coordinates. Additionally, it was theorized that the length of the request may change when location settings have been enabled. However, again there was no discernible difference between requests that included location and those that did not. Therefore, it is not possible to determine if an AdMob user has enabled location settings simply based on the request URL.

Content Type The type of content that is to be delivered is mentioned. Potentially with an ad space larger than banner ad, it would be possible for videos to be displayed, but the sample app only included a banner ad size. Attackers could use to know the format the response is expected to be in and only launch an attack when the user will see an HTML ad.

Session Of other interesting note is the fact that requests made temporally close to one another are associated with the same *session_id*. Subsequent ad requests increment the *seq_num* counter by one. This seems to be another way in which the user is tracked, but it could also be used to show a limited number of ads to decrease load time by caching the ads in the ad network.

Carrier Service provider information was also indicated as one of the fields. Again, this was only available when the device has cellular service. The amount of information about the carrier was less than the MoPub requests, with only a single value being reported instead of three.

URL Length For all the requests, the last parameter is the URL length, which is probably used as a form of error correction. However, this location is susceptible to attack if it is also intended as a security measure. An attacker can modify the request stream prior to the end, adding, removing or modifying parameters and values as needed and then recalculate the URL length as a result. This will enable them to make any changes that are needed for their attack and the URL length will not reflect the modified parameters.

5.6 Ads as a Service

While the previous two attacks are possible in commercial apps, lending themselves to a broader attack vector, there is also an additional way to utilize ads in a more clandestine way. Both of the inspected ad libraries were capable of running in the background as a service, out of sight of the user, each submitting an ad request and successfully loading the ad. This functionality could potentially be used by a malicious app developer through which

the vulnerabilities in the ad framework described here could be used to exploit the device. While this is not the most direct path to exploiting a device, it is one that would escape static analysis tools that look for malicious behaviour.

Chapter 6

Secure Advertisement Library Design Principles

The solution for protecting user privacy and mobile device security belongs to many different parties, from end users to the ad networks, each having a share of the responsibility. In this chapter, we will discuss steps that the different actors can take that will protect users and enable a more secure environment.

6.1 Addressing Vulnerabilities

In this section, we discuss a variety of facets through which mobile device security can be improved, from both the perspective of the app developer as well as the end user.

6.1.1 Extraneous Permissions

Building from the work of [6, 22, 29], we provide several recommendations on how to limit extraneous permissions in order to limit the exposure of user data and information to potential attack. The first suggestion would be for developers to have their code audited, either internally as part of a code review process, or externally by another trusted organization. The aforementioned papers indicated developer confusion as the most likely source of which permissions were needed and which were not. More careful consideration of permission granting would limit the attack surface that apps have. Code audits would also have the additional benefit of helping to discover other potential vulnerabilities, security risks and coding defects before they reached the production environment. While there would be the increased initial cost to audit the code, it has long been known that a defect is much more expensive to correct after it has made it into the release, allowing the organization to make up the lost funds over time. The tools created by the aforementioned researchers would also be a profitable means to limit extraneous permissions from entering the application, by providing static analysis of the code and the permissions that are needed to execute each module.

6.1.2 Weak Command and Control

For developers to better limit the amount of information that can be observed from packet inspection, it is recommended that HTTPS be used to encrypt all web traffic. Additionally, it should be after the secure connection is established that the request for an ad is made instead of the current approach of sending the request before the connection is secured. Such an approach would significantly reduce the information that is leaked over the insecure channel. While it would require an additional HTTP request/response pair by waiting until

the secured link was established, this is minimal overhead compared to the security benefits that it would provide to users.

If HTTPS is not possible for some reason, additional security could be used such as encrypting the parameter values with a pre-shared secret between the ad library company and the app developers, which could be established during registration of the app with the ad library. This secret key could be shared in the same way that the unique identifiers for each ad are shared with the developers.

It should also be noted that with the increasing performance capabilities of most modern mobile devices, the computational and resource overhead required to implement HTTPS is negligible.

6.1.3 Ecosystem Changes

In contrast to the segmentation that is present in the Android ecosystem, the reader is pointed to the long running support that Microsoft had for their once popular operating system XP. Microsoft released Windows XP in August of 2001 and continued to release updates through April of 2014, nearly 13 years later. While desktops and laptops are a different market from mobile devices, this is evidence that it is possible to support multiple platforms and device capabilities for an extended period of time.

This type of support could be achieved by making security updates available much as they are for the desktop platforms in small pieces. The Android architecture could be changed such that smaller pieces of the operating system could be downloaded from a trusted update server, the files themselves being signed by a private key owned by Google, that would address vulnerabilities as they are made known and fixed by Google. This would allow for older devices to continue benefiting from updates while not being forced to update to the

latest version of Android which may not be available for their device.

6.1.4 Code Injection

Due to the technical differences between Android 4.1 and 4.2, possible solutions to mitigate the code injection attack are platform dependent.

Android 4.1 and Older

Android 4.1 and older continues to be a popular platform with close to 50% of the market share[12]. In order to protect these users, there are a number of recommendations that can be made. The first is to limit the type of advertising that is available to these devices. As this specific attack requires a WebView, any ad that was not displayed in a WebView would be a safer alternative such as video ads. It would also be possible for the developers to remove the advertising functionality on these devices to safeguard their users. It would be a decision that the developer would have to make to balance between the security of their users and the loss of advertising revenue. Another solution would be to completely eliminate these devices from the supported list. App developers must specify the minimum SDK version that is needed to support their apps, and setting this to a version of Android newer than 4.1 would significantly limit such an attack.

Ideally, disabling Javascript for a WebView would provide the protection required while providing the maximal functionality, but this is not possible, even in more recent versions of Android.

As a balanced approach that will work without significant changes to the underlying app or Android itself, app developers can create a hard coded white list of acceptable resources that are to be loaded in the WebView. By limiting these to the ads that are expected, the

possibility of loading a foreign and malicious resource is averted.

Android 4.2 and Newer

With the changes that were implemented in Android 4.2 that require annotations to be added to Java methods that are called from Javascript, this has significantly reduced that attack surface of newer devices. As a result of these changes, it is highly recommended that app developers carefully consider any functionality that they might add the annotation to and determine if the functionality might be available through other means. If not, a careful consideration should be made to determine if the risk to users is worth the gain. If left with no alternatives, careful coding practices should be utilized to minimize the risk for exploitation. App developers could also utilize some of the same practices as listed in Section 6.1.1 and utilize a form of code analysis.

6.1.5 User Recommendations

In addition to the changes suggested for developers, users also have significant control in limiting their exposure to malicious attacks. In order to limit geolocation information from being transmitted, location settings can be turned off when not in use. This makes certain functionality less convenient to use, but the tradeoff here is between convenience on one side and privacy and security on the other.

Users should also only install apps from trusted source, such as the Google Play store and the device should disallow app installation from unknown sources. Google has recently taken some additional steps to limit the ability of websites and other untrusted sources from installing apps to a device.

Limiting the use of public or unsecured WiFi hotspots reduces the chance that another

entity will be able to intercept transmissions that would allow them to conduct the attacks as outlined in this thesis.

It is also recommended that users be aware of any software updates that are available for their specific device and apply them as soon as possible. However, this is made more difficult by the segmentation and the inconsistent update cycle for all handsets as described in Section 3.3.

6.2 Architecture for Secure Advertising Library

In order to make ad delivery a secure and sustainable venture for users, network operators, app developers, ad networks and advertisers, there need to be changes in order to better protect the security of mobile devices as well as users. Towards that end, we provide several suggestions of varying complexity and security enhancements for consideration. In addition to these suggestions, we also describe the security vulnerabilities that they address.

The first suggestion would be to implement one of the many different ad library-app permission separation schemes as was discussed in Section 3.5. One example of this is seen in Figure 6.1, which has been proposed by Pearce et al. in [29]. Their solution separates the ad communication mechanism from the app and consolidates all ad communication in a new operating system service. If an app wishes to make a request for an ad, a signal would be sent to the AdDroid service which would have its own set of permission checks that correspond to each app. If the app has permission to request an ad, the service would securely communicate with the ad network, transmitting any needed information to fulfill the request. Once the ad request has been fulfilled, the service sends the completed ad request back to the app where the ad is displayed for the user.

A more straight forward solution would be for Google to add a common ad library interface

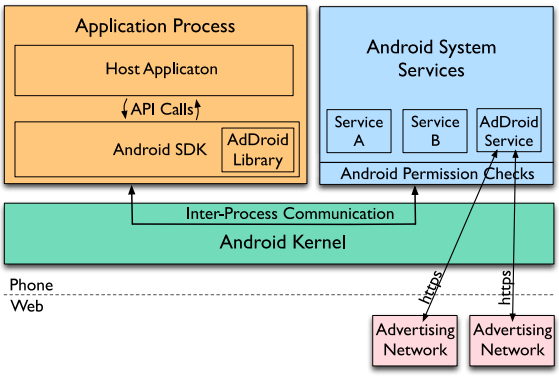


Figure 6.1: Architecture suggestion from [29] which adds a new service through which all apps need to communicate in order to receive ads. (Paul Pearce et al. "AdDroid: Privilege Separation for Applications and Advertisers in Android". In: *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*. ASIACCS 12. Seoul, Korea: ACM, 2012, pp. 7172. Used under fair use, 2014.)

to the Android SDK. Such an interface would provide a uniform workflow for app developers to request ads from the ad network, regardless of their choice of ad network. The architecture for the proposed system is seen in Figure 6.2.

This unified architecture would be able to protect users from a variety of threats. The fact that it would be included with the standard Android SDK would indicate a high standard of quality, something that is of questionable status for other ad libraries. By including this in the Android SDK, there would also be the benefit of being open source so that developers and security professionals outside of Google would be able to examine the code being used and identify security vulnerabilities that might have been missed before being released. The open source nature of the ad interface would also allow ad networks to be able to configure their networks to work with the change in format. Software best practices are certain to be used which will make use of secure command and control channels utilizing HTTPS to encrypt traffic between the device and the ad network. As security vulnerabilities are identified, they would be fixed and released along with updates to the SDK. As has been discussed

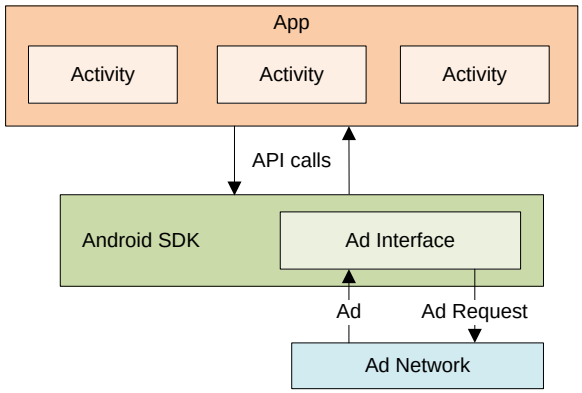


Figure 6.2: Proposed solution for secure ad communication between ad network and apps. Here, the ad library has been replaced by an ad interface as a component of the Android SDK.

previously, the release cycle and segmentation for Android results in delayed updates for certain devices, but the uniformity of this approach outweighs these detractors.

This common interface would also have the benefit of allowing very simple replacements of ad libraries from a coding perspective. If it was determined that one ad library wasn't meeting the needs of the app developer, a replacement could be found and immediately dropped in as a replacement with little to no changes required due to the common approach in displaying ads between all ad libraries. As one of the hallmarks of good coding, clean and reusable interfaces are also a coding best practice.

If a user's private data is never sent over the air, then it can't be intercepted by an eavesdropper. The previous enhancements have focused on stopping vulnerabilities at the source, and the same can be done for personal information that is available to the ad network in the first place. Pearce and Shekhar have proposed that new permissions be added to address advertising access to personal information. Following from the proposal to move advertising functionality to the Android SDK, this proposal continues to have a convenient location as part of the comprehensive solution. As proposed in [29] and [33], users would be presented with additional permissions indicating what personal information was being requested by

the ad component. This would continue to be the case.

The other attack that was identified, the exploitation of the `addJavascriptInterface` of WebViews, has largely already been addressed from an architectural perspective through an update to the Android SDK. This move already demonstrates a willingness on the part of Google to address significant issues by updating the SDK. Due to the nature of this vulnerability, the recommendations that have been made in Sections 6.1.2 and 6.1.4 have more bearing than any changes to the architecture.

6.3 Alternatives

As an alternative to displaying ads within the app, the app developer could instead offer to sell their app for a fee or offer in app purchases. Such in app purchases are frequently used to unlock additional content or allow for more flexibility in the ways the app is used, such as allowing full HD recording for a movie app. By removing ads, the app would have a much smaller attack surface that the developer would be in complete control of and one that is subject to much more scrutiny as it would be the exchange of money for goods. Some of the most popular and most profitable apps on the Google play store implement this revenue model.

Chapter 7

Conclusion and Future Work

While many advances have been made through this work, there is still room for further advances to be made. Some ideas for advances are presented here.

7.1 Future Work

This research has only considered two different attack vectors against ads within Android applications. While this has yielded significant findings, this is only a sample of potential vulnerabilities that are exposed to the most common way application developers fund their apps. To further understand the groundwork for what is even possible, a survey of additional security vulnerabilities as well as an assessment of their feasibility and applicability would go a long way towards protecting users from malicious attackers.

Additionally, this work has only considered two ad libraries, AdMob and MoPub. Due to the open nature of Android and the free market that exists in the global economy, there are scores more ad libraries that are operating in the Android ecosystem. While [1] gives the

listing of the most popular ad libraries, there might be some that are extremely popular for a specific market segment, such as in China or other Asian countries, that are not captured on a broad survey as presented there. A more detailed analysis of additional ad libraries is essential to understanding the security landscape that is present in the marketplace. This additional analysis would also be able to determine if the security vulnerabilities identified here are isolated cases or are endemic of the entire advertising ecosystem.

There is also significant potential for a system that would meet the needs of advertisers but would request information in a way that limits the exposure of private data from eavesdropping as well as increasing the privacy of users. Other interesting future work could be done to determine if there is a limit to what type of data is exposed via `addJavascriptInterface` attacks to include such items as reading arbitrary sections of memory or calling native methods to access hardware controls. Android has the ability to call native methods that have been written in C/C++; the feasibility of calling such functions is left as an open problem. Depending on the device, there might be additional features that could be accessible, such as audio recording, picture and movie taking. An analysis of which devices are most susceptible would be quite beneficial.

7.2 Conclusion

With the wide variety of personal and private data that is present on mobile devices and the fact that they follow us around wherever we go, they make ideal targets for potential attackers. This attractiveness increases when we also consider the potential for mobile devices to hold business information such as emails and company reports. Whatever the motivation for attackers, whether that be for financial gain, personal exploits or bragging, mobile devices present a number of opportunities for attackers to pilfer information from unknowing users.

Following along on the heels of the attractiveness of mobile devices is the fact that users download and use a wide variety of free applications that are supported by advertising revenue. The majority of free apps in the Google Play store are supported by advertising and the number of such apps is steadily increasing. In order to display these ads, special ad libraries have been packaged along with the app which contain vulnerabilities that can be exploited to gain information from the device. When users see an ad within the app they are using, there is no conscience acknowledgment of the potential for a security lapse to have taken place. This lack of user awareness as well as vulnerabilities with the way that ads are displayed makes this a prime target for exploiting mobile devices.

In this work we explored in detail two of these different avenues through which mobile devices can be exploited. The first one makes use of the display mechanism that shows small pieces of HTML which are known as WebViews. Because these behave very similar to a standard web browser, there is also the ability for WebViews to handle Javascript that has also been included. In order to make WebViews for appealing to developers, Android has a bridging mechanism, `addJavascriptInterface`, which allows Java based Android method calls to be made from the Javascript that the WebView displays. If an attacker loads their own malicious Javascript instead of that intended by the ad network, full exploitation of the Android device is possible.

The second attack that was investigated was the information about the user and the device as the ad library made a request to display an ad through the ad request URL. A high amount of personal information about users is contained within this request, which can also include their exact GPS location.

While the attacks are noteworthy, the real advance has been the discussion towards developing effective countermeasures to combat these and other attacks. These included such ideas as separating the permissions of ad libraries from the apps they support, creating a

unified advertising interface as part of the Android SDK to provide a secure communication layer between and modifying the apps on older devices to protect them from such grievous vulnerabilities.

Mobile devices will continue to shape the future landscape of the computing environment. To that end, the privacy of users and the security of the information that is on their device need to be maintained. The research that has been presented here works towards achieving both of these objectives, especially in the context of providing secure ad delivery to apps in the Android ecosystem.

Appendix A

URL Request Parameters

Table A.1 shows a sample of a MoPub ad request URL as well as the meaning for each of the parameters. For this request, the device was in Airplane mode and the location settings were such that only WiFi networks and the cellular connection could be used to assist in geolocating the device. There are additional parameters that are listed in Table A.2 that were not observed in any of the test ad requests.

Table A.1: Ad request URL from 2014-10-10_1232 MoPub activity with airplane mode turned on and location only able to be determined using WiFi and cellular networks. The complete IDs for the ad unit and *udid* have been obfuscated for privacy. The GPS coordinates refer to a location near the Virginia Tech Arlington Research Center where the tests were conducted.

Parameter	**Value**	**Definition**
id	cd93fa0dexxxxxxxxxxxxxxxxxxxxxxx	Ad identifier
nv	2.1	Ad Library SDK version
dn	samsung,Galaxy Nexus,mysid	Device Info
udid	ifa:A16281fb7-fec3-4786-b6ec-xxxxxxxxxxxx	ifa: Google Play Services ID sha: Android device ID
ll	38.88288288288288,-77.12444365826433	Latitude and Longitude
lla	2000	Accuracy (meters)
z	-400	Time Zone offset
o	p	Orientation ((p)ortrait, (l)andscape, (s)quare)
sc_a	2	Density
mr	1	Mraid Flag (1=present)
mcc	311	Mobile Country code
mnc	480	Mobile Network code
cn	Verizon Wireless	Carrier Name
ct	2	Network Type (WiFi=2, Mobile=3)
av	1	Application version
Android_ perms_ ext_ storage	1	External Storage Permission Granted
ts	1	Twitter app installed (1=Yes)

Table A.2: URL parameters that were not observed in sample web traffic as well as their definitions.

Parameter	**Definition**
dnt	Do Not Track
v	API Version
udid	Device ID (similar to ifa or sha)
q	Query String (Keywords)
iso	ISO country code

Table A.3 shows a sample of the AdMob query string that was captured. The *ms* field has been truncated for the sake of brevity. Some definitions have been left blank for lack of evidence to support their functionality.

Table A.3: AdMob request URL parameters and definitions

Parameter	**Value**	**Definition**
session_id	15361362647424100000	Ad session ID
seq_num	4	Ad displayed within the same session
rm	1	
js	afma-sdk-a-v6188000.4452000.1	
hl	en	Device language
gnt	13	
ma	0	
carrier	311480	Cellular carrier (Verizon)

Parameter	**Value**	**Definition**
u_sd	2	
sp	0	
cnt	1	Ad count
muv	3	
riv	0	
ms	VVhmtyop5PkoCf87p43plcod...	Encrypted ad and device information
mv	80300031.com.android.vending	Ad library name
format	320x50_mb	Ad size
coh	1	
gl	US	Country Code
am	0	Is morning? (1=yes)
u_w	360	Screen width
u_h	640	Screen height
msid	edu.vt.AdMobTest	App package name
app_name	1.android.edu.vt.AdMobTest	App package name
an	1.android.edu.vt.AdMobTest	App package name
net	wi	Type of connection (wi=WiFi)
u_audio	3	Type of audio (3=speakers, 2=headphones)
u_so	p	Screen orientation ((p)ortrait, (l)andscape)
preqs	3	

Parameter	Value	Definition
support_ transparent_ background	FALSE	App supports transparent background
pimp	3	
currts	70776591	Current time
pclick	0	
basets	68709284	Base Time
treq	69421467	Time of the request
tfetch	69421751	Time of the ad retrieval
tresponse	69433386	Time of the ad response
tload	69433928	Time of the ad loading on the ad network
dload	12461	Delay to load ad (ms)
timp	69433928	Time of the impression
pcc	0	
ismediation	FALSE	Is a mediation (intermediate) ad network being used
bisch	FALSE	
blev	0.6100000143	
cans	5	
canm	FALSE	Can the app use a mediation network
output	html	Type of ad to be displayed
region	mobile_app	Target device

Parameter	**Value**	**Definition**
u_tz	-300	Time zone offset
client_sdk	1	App version
ex	1	
client	ca-app-pub-3550757145636180	Advertiser
slotname	6984458958	Advertiser campaign ID
askip	3	Allow the ad to be skipped? (1=Yes, 2=No, 3=after 5 seconds)
gsb	wi	Network type (wi=WiFi)
caps	inlineVideo_ interactiveVideo_ mraid1_ th_ autoplay_ mediation_ av_ sdkAdmobApiForAds_di	
eid	46621044	
jsv	130	

Bibliography

[1] AppBrain. *Android Ad networks*. 2014. URL: http://www.appbrain.com/stats/libraries/ad (visited on 10/17/2014).

[2] AppBrain. *Distribution of free vs. paid Android apps*. 2014. URL: http://www.appbrain.com/stats/free-and-paid-android-applications (visited on 10/22/2014).

[3] Theodore Book and Dan S. Wallach. "A Case of Collusion: A Study of the Interface Between Ad Libraries and Their Apps". In: *Proceedings of the Third ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*. SPSM '13. Berlin, Germany: ACM, 2013, pp. 79–86.

[4] Quang Do, B. Martini, and K.-K.R. Choo. "Enhancing User Privacy on Android Mobile Devices via Permissions Removal". In: *System Sciences (HICSS), 2014 47th Hawaii International Conference on*. 2014, pp. 5070–5079.

[5] S. Dolev et al. "Exploiting simultaneous usage of different wireless interfaces for security and mobility". In: *Future Generation Communication Technology (FGCT), 2013 Second International Conference on*. 2013, pp. 21–26.

[6] William Enck et al. "A Study of Android Application Security". In: *Proceedings of the 20th USENIX Conference on Security*. SEC'11. San Francisco, CA: USENIX Association, 2011, pp. 21–21.

[7] Adrienne Porter Felt et al. "Android Permissions Demystified". In: *Proceedings of the 18th ACM Conference on Computer and Communications Security*. CCS '11. Chicago, Illinois, USA: ACM, 2011, pp. 627–638.

[8] E. Fernandes, B. Crispo, and M. Conti. "FM 99.9, Radio Virus: Exploiting FM Radio Broadcasts for Malware Deployment". In: *Information Forensics and Security, IEEE Transactions on* 8.6 (2013), pp. 1027–1037.

[9] Andrei Frumusanu. *A Closer Look at Android RunTime (ART) in Android L*. 2014. URL: http://anandtech.com/show/8231/a-closer-look-at-android-runtime-art-in-android-l (visited on 11/11/2014).

[10] Clint Gibler et al. “AdRob: Examining the Landscape and Impact of Android Application Plagiarism”. In: *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*. MobiSys ’13. Taipei, Taiwan: ACM, 2013, pp. 431–444.

[11] Google. *Android Application Framework FAQ*. 2014. URL: http://developer.android.com/guide/faq/framework.html (visited on 11/11/2014).

[12] Google. *Android Dashboards*. 2014. URL: http://developer.android.com/about/dashboards/index.html (visited on 10/17/2014).

[13] Google. *Android, the world’s most popular mobile platform*. 2014. URL: http://developer.android.com/about/index.html (visited on 10/30/2014).

[14] Google. *Location — Android Developers*. 2014. URL: https://developer.android.com/reference/android/location/Location.html (visited on 10/30/2014).

[15] Google. *LocationManager — Android Developers*. 2014. URL: https://developer.android.com/reference/android/location/LocationManager.html (visited on 10/30/2014).

[16] Michael Grace et al. “Unsafe Exposure Analysis of Mobile In-App Advertisements”. In: *Security and Privacy in Wireless and Mobile Networks (WISEC ’12), Proceedings of the fifth ACM conference on*. 2012, pp. 101–112.

[17] Mordechai Guri et al. “AirHopper: Bridging the Air–Gap between Isolated Networks and Mobile Phones using Radio Frequencies”. In: *Malicious and Unwanted Software (MALCON 2014), Proceedings of the 9th IEEE International Conference on*. 2014.

[18] D. Hartley. *WebView addJavascriptInterface Remote Code Execution*. https://labs.mwrinfosecurity.com/advisories/2013/09/24/webview-addjavascriptinterface-remote-code-execution/. (visited on 2014/11/19).

[19] Yuichi Hayashi et al. “A Threat for Tablet PCs in Public Space: Remote Visualization of Screen Images Using EM Emanation”. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’14. Scottsdale, Arizona, USA: ACM, 2014, pp. 954–965. URL: http://doi.acm.org/10.1145/2660267.2660292.

[20] IAB. *Interactive Advertising Bureau: Mobile Web Advertising Measurement Guidelines*. 2011. URL: http://www.iab.net/media/file/MobileWebMeasurementGuidelines_final.pdf (visited on 10/30/2014).

[21] jduck and joev. *Android Browser and WebView addJavascriptInterface Code Execution*. 2013. URL: http://www.rapid7.com/db/modules/exploit/android/browser/webview_addjavascriptinterface/ (visited on 10/21/2014).

[22] H. Kawabata et al. “SanAdBox: Sandboxing third party advertising libraries in a mobile application”. In: *Communications (ICC), 2013 IEEE International Conference on*. 2013, pp. 2150–2154.

[23] Neeraj Kumar. *What is dalvik virtual machine in android*. 2014. URL: http://www.87android.com/what-is-dalvik-virtual-machine-in-android/ (visited on 11/18/2014).

[24] H. Kuzuno and S. Tonami. "Signature generation for sensitive information leakage in android applications". In: *Data Engineering Workshops (ICDEW), 2013 IEEE 29th International Conference on*. 2013, pp. 112–119.

[25] Mitre. *Common Vulnerabilities and Exposures*. 2013. URL: https://cve.mitre.org (visited on 11/18/2014).

[26] I Mojica Ruiz et al. "On the Relationship between the Number of Ad Libraries in an Android App and its Rating". Accepted for publication in IEEE Software. 2014.

[27] A Narayanan, Lihui Chen, and Chee Keong Chan. "AdDetect: Automated detection of Android ad libraries using semantic analysis". In: *Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2014 IEEE Ninth International Conference on*. 2014, pp. 1–6.

[28] OpenSignal. *Android Fragmentation Visualized*. 2013. URL: http://opensignal.com/reports/fragmentation-2013/ (visited on 11/20/2014).

[29] Paul Pearce et al. "AdDroid: Privilege Separation for Applications and Advertisers in Android". In: *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*. ASIACCS '12. Seoul, Korea: ACM, 2012, pp. 71–72.

[30] Xiong Ping et al. "Android malware detection with contrasting permission patterns". In: *Communications, China* 11.8 (2014), pp. 1–14.

[31] Offensive Security. *Exploit Database*. 2014. URL: http://www.exploit-db.com/ (visited on 11/15/2014).

[32] A. Seneviratne et al. "Reconciling bitter rivals: Towards privacy-aware and bandwidth efficient mobile Ads delivery networks". In: *Communication Systems and Networks (COMSNETS), 2013 Fifth International Conference on*. 2013, pp. 1–10.

[33] Shashi Shekhar, Michael Dietz, and Dan S. Wallach. "AdSplit: Separating Smartphone Advertising from Applications". In: *Proceedings of the 21st USENIX Conference on Security Symposium*. Security'12. Bellevue, WA: USENIX Association, 2012, pp. 28–28.

[34] National Institute for Standards and Technology. *National Vulnerability Database*. 2014. URL: https://web.nvd.nist.gov (visited on 11/18/2014).

[35] G. Suarez-Tangil et al. "Evolution, Detection and Analysis of Malware for Smart Devices". In: *Communications Surveys Tutorials, IEEE* 16.2 (2014), pp. 961–987.

[36] Christopher Trout. *Android still the dominant mobile OS with 1 billion active users*. 2014. URL: http://www.engadget.com/2014/06/25/google-io-2014-by-the-numbers/ (visited on 10/17/2014).

[37] Tao Wei et al. “Sidewinder Targeted Attack against Android in The Golden Age of Ad Libraries”. In: *Black Hat USA 2014*. 2014.

[38] WMATA. *Pay for Metro with your smartphone or watch? Testing starts soon.* 2014. URL: `http://www.wmata.com/about_metro/news/PressReleaseDetail.cfm?ReleaseID=5778` (visited on 11/15/2014).

[39] Luyi Xing et al. “Upgrading Your Android, Elevating My Malware: Privilege Escalation Through Mobile OS Updating”. Accepted for publication in proceedings of SP. 2014.

[40] S.Y. Yerima, S. Sezer, and G. McWilliams. “Analysis of Bayesian classification-based approaches for Android malware detection”. In: *Information Security, IET* 8.1 (2014), pp. 25–36.

[41] Yajin Zhou et al. “Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets”. In: *Network and Distributed System Security Symposium (NDSS 2012), Proceedings of the 19th*. 2012, pp. 1–13.