Mobile Malware Madness, and How to Cap the Mad Hatters

A Preliminary Look at Mitigating Mobile Malware

Gerry Eisenhaur, Michael N. Gagnon, Tufan Demir, Neil Daswani

Living in Smartphone Wonderland

Mobile devices have become an integral part of our lives. We game on them, surf on them, bank on them, and increasingly buy things on them. By early 2011, the two most popular app markets — Google Android and Apple iPhone — each hosted well over 300,000 applications with billions of downloads and thousands of new apps launched each month. While development on the proprietary iPhone platform has been a "walled garden" — with app developers subject to a formal review process, the Android marketplace is more open. Any unvetted developer can upload mobile applications to the Android Market, which allows for open innovation, but presents security challenges. Since the debut of the Android platform in late 2008, consumer acceptance has been stunning. In April 2011, *Computerworld* reported that Android overtook iPhone with north of 35 percent share of the smartphone market [TowerGroup].

Given today's critical mass of smartphone users, the future of commerce is also mobile. According to Juniper Research, the market for mobile payments will triple in value by 2015 reaching $670 billion, up from $240 billion this year [Juniper]. The stunning pace of growth in mobile apps, usage and purchasing is bound to attract cybercriminals, who predictably follow the money. Complicating the problem is consumer ignorance about the security risks with mobile computing. Mobile malware has been on the horizon since first emerging 2004, but the first six months of 2011 witnessed an explosion of zero-day attacks, with a new incident reported every few weeks. Far from "frozen at 6 o'clock," these Mad Hatters keep innovating at a furious pace.

This paper surveys recent mobile malware attacks that have infected hundreds of thousands of user devices. It explores how behavioral-based malware detection techniques can be used to identify and neutralize these nefarious programs before they can accomplish their ultimate aims of stealing user identity and interrupting mobile commerce. We also explore how web malware threats such as drive-bys and malvertising are now emerging on mobile networks.

A Shrunken History of Mobile Malware

A recent report from Juniper Networks found that Android malware jumped 400 percent in the first six months of 2011[Juniper]. The current crop of mobile malware is getting more sophisticated, and can operate in the background, completely invisible to the user, running executables and contacting botmasters for new instructions. The next wave of malware is expected to be even more advanced, with botnet tendencies to hijack and control devices.

How did we get here? To recognize the capabilities of today's malware and how best to detect it, a short history of past attacks can explain how malware went airborne:

Cabir (2004) – Believed to be the first computer worm capable of infecting mobile phones, Cabir targeted devices running Symbian OS. This attack simply hijacked the compromised phone's UI, but served as a hacker proof-of-concept to catch the world's attention. The worm attempted to exploit wireless Bluetooth signals to spread to other devices in the area using the "Object Push Profile" common to non-Symbian phones, desktop computers, and even printers. The worm payload is considered relatively harmless, short of running down the phone's battery from non-stop attempts to replicate itself. Still, even though Cabir never made it "big" in the wild, the tempest was out of the teapot. Within two years the number of viruses targeting smartphones soared from one to 200.

SMS/MMS Attacks (2006) – With names like CommWarrior, RedBrowser and FlexiSpy, the worms, trojans and spyware of the last decade relied on ubiquitous mobile messaging services to do their damage. Since users typically incurred a charge for every file or text sent, the cost of these attacks was often high… meaning expensive. Some sent a constant stream of surreptitious messages to international numbers, while others disabled the phone entirely. Still, these were just the warm up acts until the smartphone hit the headlines.

iKee (2009) – This simple yet effective worm targeted iPhones [iKeeSRI]. The worm compromised phones via default SSH passwords on jailbroken iPhones, and turned them into both bots and botmasters. It was a powerful demonstration of how easy it could be for an attacker to create a fairly large, functioning botnet using compromised mobile devices. Once on the device, iKee copied all of the phone's SMS messages and sent them off to a remote host. Like PC-based botnets, the worm assigned each infected iPhone a unique identifier so that the command and control (C&C) server could send specific new instructions to each individual device. The variant iKee.B later had the ability to query a remote C&C server periodically for new instructions, scan for new victims, and execute whatever other instructions the botmaster sent. iKee.B was set to run every five minutes to give the botmaster the option to send back new programming logic and execute any commands on the infected iPhone. Definitely a foreboding step up. However, iKee was not without a twisted sense of humor - it "Rick-rolled" the phone, changing the wallpaper to an image of 80s singer Rick Astley.

Zitmo (2010) – This attack which emerged on mobile devices running the Symbian operating system (SymbOS) was called "Zeus in the Mobile", or Zitmo for short. This trojan was geared to stealing two-factor authentication codes to aid the Zeus botmaster operators in stealing financial credentials. Zitmo duped users into installing it via social engineering, claiming to install a "security certificate" or "update" when in reality opening a malicious SIS file linked to Zbot. Thereafter Zitmo logged users SMS messages and two-factor authentication codes to a remote C&C number via an SMS message,

stealing credentials and sending unauthorized messages related to banking transactions among other activity.

DroidDream (2011) – Unlike previous malware available only in the wild, DroidDream appeared in the official Android Market. Packaged inside more than 50 seemingly legitimate applications, this virulent trojan tricked more than 250,000 users into downloading it before Google removed it from its marketplace and invoked a "kill switch" to remove the malicious applications from users' phones. DroidDream gave attackers root access to potentially hijack the entire device and its data, sending private user information to the attacker. The malware was configured to run overnight (hence its name), so the C&C could send new code new instructions to the phone when the phone's owner was sleeping. DroidDream joined iKee in laying the groundwork for a comprehensive system of remotely-controlled phones: a mobile botnet.

Plankton (2011) – This became the first Android infection to exploit Dalvik-class loading capability to dynamically extend its own functionality. Playing on the rapid popularity of the Rovio game "Angry Birds", Plankton was embedded in a cheat that loads a background service once fired up. It then scours the device for user data, including device ID code, and reports back to a remote server. The server parses the data and then sends a link back to the malware, which downloads an executable and runs nearly invisible in the background. The application then starts collecting more data, such as browser bookmarks, browser history, home page shortcuts, and runtime log information.

GGTracker (2011) – This Android-borne Trojan fools the victim by luring them with a malvertisement that, when clicked, sends them to a malicious app download page that spoofs the Android Market. GGTracker hopes that the user clicks to install a fake application (either a battery saver or adult app) that in fact signs the victim up to multiple legit premium SMS subscription services without their knowledge or consent. The back-end server component of GGTracker bypasses all of the new account controls of the premium services including replying to a PIN code received via SMS which can lead to unapproved charges to a victim's phone bill. GGTracker is one of the first malicious Android payloads distributed via a mobile malvertisement [Lookout].

The history of mobile malware continues to be written. After a slow start, the pace of attacks is accelerating, and it is possible that we should expect some "mobile malware madness" to occur in the near future, at the very least, if not longer. As such, it is an interesting time to think about how to "cap" the Mad Hatters not only in the near term, but also in the medium to long-term future.

Over the years, there have been many approaches used to detect malware on desktop PCs. Some approaches have worked well, while others have become less and less effective as the amount of malware targeting PCs has increased. For instance, one trend is the emergence of behavioral scanning. Traditional anti-virus clients have been based on signature checking to defend clients from known malicious programs. Behavioral scanning defends clients from previously unknown malware for which signatures do not yet exist. Of course, client-side software is often limited to using only the PC's local resources, including just its CPU(s), network bandwidth and so on. Since users typically get upset with waiting too long, only a limited number of checks can be done on an application or page needing to load and run within just a couple hundred milliseconds . Client-side PC anti-virus products have also started to leverage cloud computing to assist in their scanning.

By comparison, mobile phones have even less CPU processing power and network bandwidth than PCs. They will need to leverage server-side, cloud-based resources for anti-virus/anti- malware scanning, especially with "mobile malware madness" on the rise. Increased reliance on server-side resources makes possible a level of behavioral-based scanning not previously possible on a client. Behavioral scanning can be executed not only when a user downloads and attempts to run an application on a mobile phone, but also before an application is accepted into an app store. In the following section, we explore behavioral analysis of mobile applications.

Welcome to the Tea Party: Behavioral Analysis of Mobile Applications

Behavioral analysis (also sometimes called "dynamic analysis") involves running applications to see what they do. It differs from static analysis, in which program code is analyzed, but not run. For instance, many approaches to virus detection are based upon static analysis in which candidate virus code is not run, but is searched for pattern and signature matches. While static analysis does have the advantage of being very efficient, it typically cannot be relied upon to detect unknown malware for which no patterns or signatures exist.

Static analysis does not have access to the real-time data or control flow that happens in the application, which results in blindspots in the analyses. For instance, while static analysis may reveal that an application has the capability to send SMS messages, behavioral analysis can reveal how many messages an application sends, to whom, and in response to what input. So behavioral analysis can be of critical importance in identifying malware, and can do so with a very low false-positive rate. In another example, while static analysis may determine whether or not an application can access the network or address book, behavioral analysis can answer deeper queries such as:

- What domains and/or IPs are accessed via the network at run-time, and what is the access pattern?
- How many entries in the address book are accessed? Is each entry or some threshold number of entries accessed? Are all or some significant number of entries accessed serially, or is the access pattern random?
- Are the accessed address book entries sent over the network? Is sensitive data or PII leaked?

Today's malware authors are sophisticated. They create automated variants of their malware and run them through the most well known signature-based anti-virus checkers before deploying them. In such a threat environment, behavioral analysis is very effective at identifying malware based on "*what it does*" instead of "*what it looks like*" (i.e., what particular program code is used to accomplish its objectives). While behavioral analysis may be more resource intensive than static analysis, it can be sourced as a cloud-based service.

Results of a Behavioral Analysis Study

To address mobile malware madness - and to cap the Mad Hatters - one can take advantage of the strengths of both static and behavioral methods in a hybrid analysis model. Static analysis can be run on resource-constrained mobile clients to complete quick, efficient checks for known malware. Running heavyweight, server-side behavioral analysis can be used to identify as-yet- unknown malware. A hybrid approach can provide mobile device users with strong protections against malware threats. For example, an anti-virus client on a mobile phone can conduct static analysis checks as a first line of defense whenever new applications are downloaded and/or when signature updates are available. A second line of defense can send identifying information about an application (or the app itself) to a cloud-based malware detection service for analysis.

Dasient conducted a preliminary behavioral study of 10,000 Android applications from Google's Android Market to analyze their functionality. The applications included were chosen at random from 30 different categories of apps in the Android Market (listed in the table below). We observed several examples of applications leaking private information (namely, IMSI and IMEI).
We also ran analyses against Dasient's corpus of known malicious mobile applications.

Arcade & Action	Finance	Productivity
Books & Reference	Health & Fitness	Racing
Brain & Puzzle	Libraries & Demo	Shopping
Business	Lifestyle	Social
Cards & Casino	Media & Video	Sports
Casual	Medical	Sports Games
Comics	Music & Audio	Tools
Communication	News & Magazines	Transportation
Education	Personalization	Travel & Local
Entertainment	Photography	Weather

Such analysis is very helpful in developing models to determine if an application may be normal or may be exhibiting malicious behavior. Malicious behavior can include both privacy and security violations, among other types of violations.

In the study, we conducted behavioral analyses by running the target applications in an emulator for approximately 30 seconds, sending only a couple simulated user events to the applications. To-date, we have observed that minimal interaction with applications is sufficient to generate interesting

behavioral results. When we ran the analyses against our mobile malware corpus, this form of application stimulation was often sufficient to affect any malicious behavior, although we have also developed more sophisticated techniques to uncover malicious behavior for more advanced samples.

First, we present some preliminary findings from behavioral analysis of the set of 10,000 applications downloaded from the Android Market. We follow with an analysis of Android malware - delving into the details of how we use behavioral analysis to detect mobile malware.

Experimental Results from 10,000 Android-Market Applications

We discuss five results from our experiments: (1) privacy violations, (2) top domains accessed, (3) bandwidth usage, (4) text-message analysis, and (5) system-call analysis.

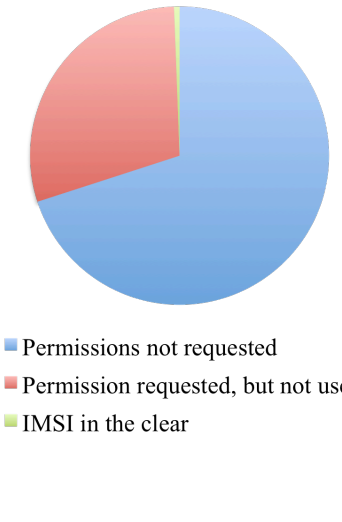
1) Privacy Violations

Apps often need a way to uniquely identify their users. To this end, apps use IMEI or IMSI numbers associated with the devices. The IMEI number of a phone identifies the device, while the IMSI identifies the subscriber. These numbers are private and apps are supposed to request permission to access them for that reason. The confidentiality of these numbers is important because they can be used for fraudulent purposes, such as cloning the SIM card.

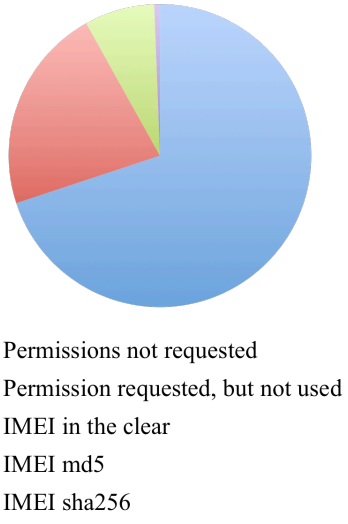
Recent studies using static analysis have shown that many applications request permissions to access private information such as IMEI and IMSI [LookoutFeb2011, SMobile2010]. However, because these analyses are static and not dynamic, they cannot report on *if and when this private information is actually leaked over the network*.

To contrast, the recent TaintDroid study used dynamic analysis [TaintDroid]. The TaintDroid study analyzed 30 popular Android apps and found that a majority of them leaked private information. We report a related result: we observed that from the apps that request permission to access the IMEI, at least 27 percent of the apps actually submit the IMEI (either in the clear or hashed with MD5) to a server. Our study differs from TaintDroid in two ways: (1) whereas TaintDroid used precise taint-tacking to observe leakage, we searched the network traffic for the signatures of our emulator's IMEI; and (2) whereas TaintDroid analyzed 30 apps, we analyzed 10,000 apps. A more detailed results of these findings follows:

IMEI Leakage Occurrences



IMSI Leakage Occurrences



Detailed findings:

- 29.9 percent of applications requested permission to access IMEI and IMSI
- Of these, 25 percent leaked the IMEI in the clear and 2 percent in MD5. There was only one occurrence where an IMEI was hashed with SHA, which was SHA256.
- IMSIs were leaked less often. Of the 29.9 percent of applications that requested permission to access IMSI, only 2 percent of these leaked the IMSI — all in the clear without hashing.
- Certain categories of applications are more likely to leak the IMEI (in the clear) than other categories
    - Brain & Puzzle, Personalization, Weather leaked 48 percent, 50 percent, 57 percent of the time respectively
    - Libraries & Demo only leaked the IMEI 2 percent of the time.

Hashing IMEI numbers does not protect privacy

As described above, IMEI numbers were sometimes hashed in an attempt to protect the privacy of users. However, hashed IMEI numbers are vulnerable to reversing via rainbow tables, due to the fundamental structure of IMEI numbers.

Rainbow tables are usually infeasible to build because there are so many possible entries. And at a first glance, it seems infeasible to build a useful rainbow table for IMEI hashes. There are 10^{15} (one quadrillion) possible IMEI values, seemingly necessitating 10^{15} entries. A rainbow table to store that many entries would need to be exceedingly large. As an example, for the SHA-1 hash function each entry would use at least 160 bits to store the hash value, requiring at least 17.8 petabytes.

However, IMEI numbers are not distributed uniformly at random. The first 8 digits of an IMEI represent the *Type Allocation Code* (TAC), which is determined by the model of the phone. Although

this is the most significant portion of an IMEI number, it is not private information; knowing the model of a phone (or guessing the model) is sufficient to guess most of an IMEI number.

After the 8-digit TAC there are 6 digits that uniquely identify the specific cellular device. These 6 digits are the only digits that are difficult for an attacker to guess. After those 6 digits the last digit is a Luhn-checksum digit, which is computed as a function of the first 14 digits. Thus, in a 15-digit IMEI number there is a relatively low amount of entropy.

With this knowledge in mind an adversary can build rainbow tables for only the most common TAC numbers. Since a relatively small number of mobile devices dominates the market, the attacker only needs to build the rainbow tables for the most popular TAC numbers.

Each rainbow table only needs to store one million hashes: about 19 megabytes.

To demonstrate the practicality of this attack we built 105 rainbow tables for 105 different iPhone TACs, using the SHA-1 hash function. Each table took up 55 megabytes of space, yielding 5.6 gigabytes in total (which is larger than the theoretical minimum since we stored data as ASCII). On an 8-core 2.26 GHz machine it took a simple Python script about six and a half minutes to build the iPhone rainbow tables.

To keep eavesdroppers from sniffing IMEIs we recommend that mobile apps refrain from sending hashes of IMEIs over the web. It is easy for attackers to generate IMEI numbers when given the hash values of IMEIs—even for cryptographically secure hash functions. Salting the hash function (adding random bits to the input) helps to obscure the IMEIs further. However, if the adversary knows the salt value and the model of the phone (or can guesses well), it is easy to rebuild custom rainbow tables.

2) Top Domains Accessed

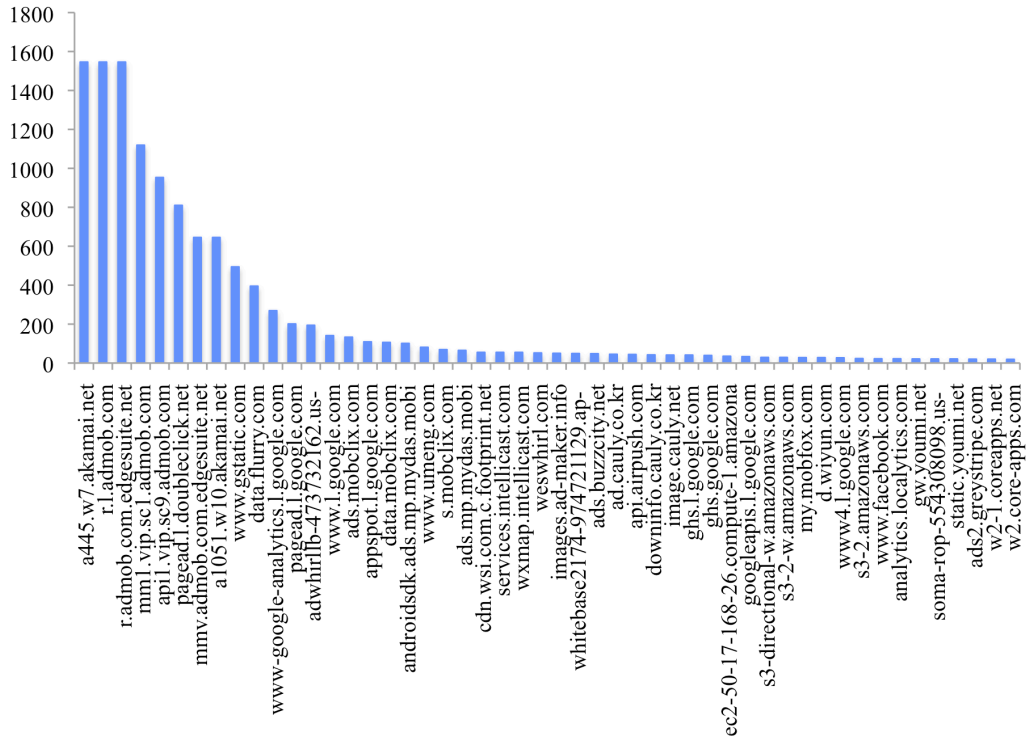
As a point of general interest we present the top 50 domains accessed by the apps in our sample. The most popular domains represent content delivery networks, advertising networks, and analytics providers.

List of Top 10 Domains:
1. a445.w7.akamai.net
2. r.l.admob.com
3. r.admob.com.edgesuite.net
4. mm1.vip.sc1.admob.com
5. api1.vip.sc9.admob.com
6. pagead.l.doubleclick.net
7. mmv.admob.com.edgesuite.net
8. a1051.w10.akamai.net
9. www.gstatic.com
10. data.flurry.com

Top Ad Networks:
1. Admob
2. Doubleclick
3. Google
4. Adwhirl
5. Mobclix

Top Analytics Providers:
1. Flurry
2. Google Analytics

Histogram of Top 50 Domains

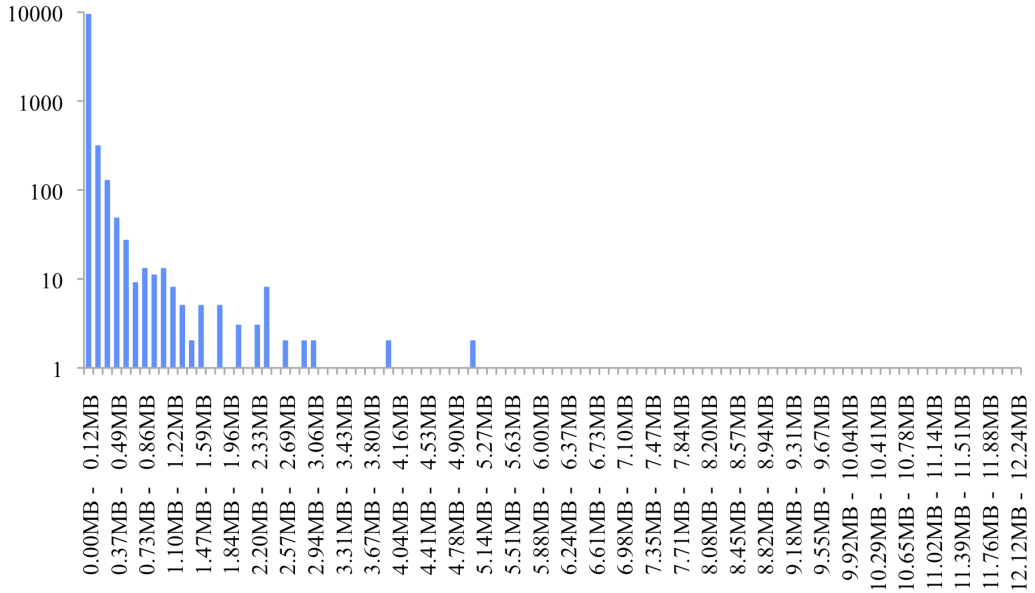


3) Bandwidth Usage

We measured bandwidth usage of apps and report those results here. The concern with bandwidth is that a malicious app can consume bandwidth in order to exceed the user's limit --- leading to surcharges. For example, Verizon charges $10 per month for every gigabyte over the allowance [Verizon Data Plans].

The log-scale histogram below illustrates the distribution of observed bandwidth usage. The graph shows that apps typically use only a small amount of bandwidth. Specifically we observed 90 percent of apps use less than 0.06 MB of bandwidth and 99 percent of apps use less than 0.81 MB. Recall, all these results are from running the app for only 30 seconds, with little simulated user interactions. Some applications could have had much more bandwidth usage with more user interaction. For instance, some applications could prompt the user before using bandwidth, but at least initially, we were interested in the bandwidth usage behavior that can occur with relatively little user interaction.

Histogram of Observed Bandwidth Usage



- The most bandwidth hungry application we observed was at.wienerstaedtische.events.apk. After installation and launch the app downloaded a 12 MB zip file containing PNG images, which is used to present users with geographic maps of music events.
- Apps from certain categories consumed more bandwidth than others. Below, we list the top bandwidth consumer from each category.

Size	Category	App name
12.12 MB	Tools	at.wienerstaedtische.events.apk
9.00 MB	Music	cocampo.music.mariskal.apk
5.07 MB	Arcade	com.atomicblaster.apk
4.23 MB	Sports	com.noticesoftware.AnyoneButDetroit.apk
2.89 MB	Brain	com.aamob.hiddenobjects.phone.englishD.NP.activities.apk
2.68 MB	Transportation	com.wistronits.easycard.mrt.apk
2.53 MB	News	com.cryosphere.apk
2.13 MB	Social	com.m2appl.Jeonbuk.apk
2.05 MB	Books	com.coreapps.android.followme.ibs2011.apk
1.90 MB	Libraries	com.syncables.captivate.apk
1.77 MB	Shopping	app.android.sgmalls.apk

Size	Category	App name
1.58 MB	Lifestyle	com.dreamstep.wfancy.apk
1.57 MB	Medical	com.cyberandsons.tcmaidtrial.apk
1.41 MB	Health	com.hivebrain.andrewjohnson.energyboostfree.apk
1.23 MB	Entertainment	cbb.product.entertain.bikinigirl1.apk
1.21 MB	Education	com.environmental.educator.apk
1.16 MB	Travel	com.ivanya.wififinder.apk
1.13 MB	Photography	com.apps.NYCPhotos.apk
1.10 MB	Personalization	com.accesslane.dxtop.theme.Camo.apk
1.09 MB	Media	com.palmedia.apk
1.08 MB	Business	com.appmakr.app147046.apk
0.98 MB	Communication	ackdev.com.AccessoryDroid.apk
0.81 MB	Sports	com.livedoor.android.soccer_journal.apk
0.79 MB	Games	com.aguacaliente.AguaCaliente.apk
0.61 MB	Cards	nl.ronsoft.foksukreader.apk
0.59 MB	Comics	com.chinatrust.mobilebank.apk
0.59 MB	Finance	ackdev.com.AllStoreDroid.apk
0.54 MB	Productivity	org.androworks.meteor.apk
0.43 MB	Weather Casual	biz.mtoy.hitball3.apk
0.43 MB	Racing	com.permeative.typingclass.apk

4) Text Message Analysis

Malicious apps have used text messages for various purposes, such as sending texts to premium numbers where the attacker receives the surcharge spoils, or spamming the contacts list.

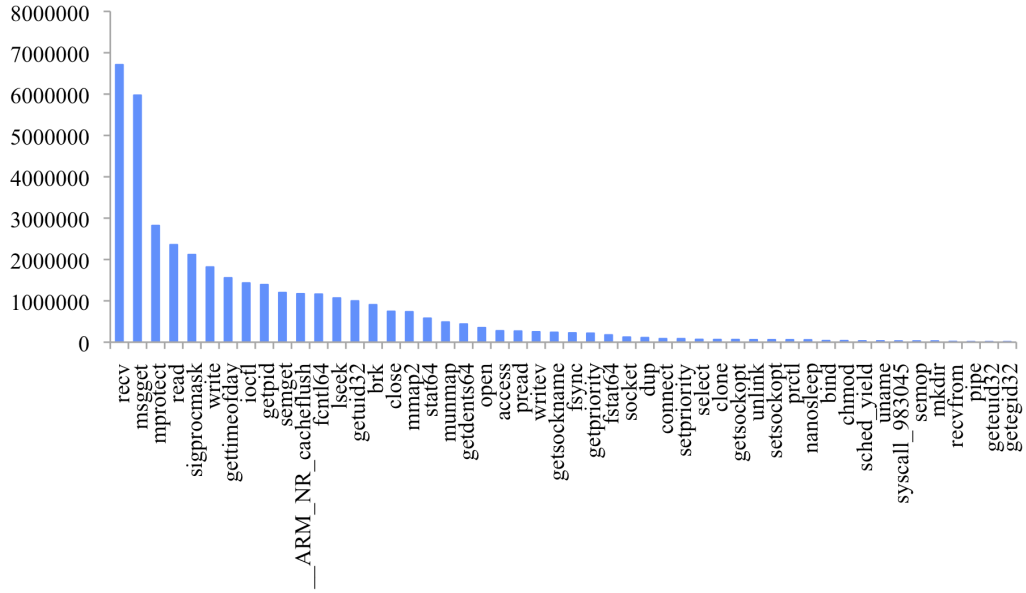
While we did not observe any outwardly malicious text messages in our sample of 10K apps from the Android Market, we did observe 11 applications that sent text messages that could be considered spam-like. In particular, the 11 applications sent text messages to the device itself that thanked the user for installing the app and suggested sharing the app with friends. These apps are generating SMS messages that are potentially unwanted by the user. Even though a user may give blanket permission to an application to use SMS, the user cannot specify what is acceptable use of SMS from a content or volume standpoint, as the permissions model on Android is coarse-grained.

5) System Call Analysis

On most systems, system-call analysis exposes significant information about an application's semantics. Application semantics are exposed because most application libraries (such as libc) often have a close correspondence to system calls. However, the Android OS is different because much of the interesting application-level functionality does not *directly* correspond to system calls. For example, Android does not have a get_imei() system call nor a send_sms() system call. Rather these operations are conducted using the Android application libraries, and may only result in "coarse-grained" system calls.

Our experimental results confirm that limited application semantics are available via coarse-grained system call tracing. The histogram below shows the frequency with which the most popular system calls are made.

Histogram of Popular System Calls



Using Behavioral Analysis to Detect Malware

To date, most Android malware is not very sophisticated and usually conducts its malicious behavior with little user interaction. On one hand, cybercriminals want users to trigger their malicious application functionality fairly quickly such that they see a high conversion rate of devices infected. On the other hand, Android malware will become much more sophisticated. To deal with this sophistication our analysis platform has the ability not only to simulate many user actions, but also to target those simulated actions based on intended activities within the application process.

While definitively determining whether or not an application is malware is a computationally intractable problem in theory, if a cybercriminal is to benefit from a malicious application in practice, prodding an application to exhibit its malicious behavior cannot be made too difficult. There are, of course, targeted attack and logic bomb / trojan cases in which malicious behavior can be disguised quite well and can be difficult to root out, but not investing in "good enough" heuristics due to theoretical impossibility results is a non-option for most businesses that have mobile assets to protect.

To provide an illustration of what behavioral analysis of mobile malware may reveal, we report on two very simple examples of detection signals that utilize the tracking of the number of processes that an application starts and SMS activity. Note that a full-fledged behavioral detection system for mobile malware utilizes hundreds of such signals, and we only include a couple here as a simple

illustration. In addition, each of these signals considered in isolation may not be sufficient to deem that an application is malware, but combinations of them when used as part of a machine learning classifier [Andromaly] can result in very high accuracy and extremely low false positives.

Example #1: Number of processes

DroidDream malware, as discussed in the earlier sections of this paper, used a variety of exploits including expoid and rageagainstthecage.

The goal of the rageagainstthecage exploit is to gain a root shell by getting adbd (android debugging bridge daemon) to run as root. The exploit first checks the maximum number of processes that can be run by looking at the NPROC environment variable. Then, it enters a loop forking as many new processes as possible. When the fork-process call fails, one process is killed and then adbd is restarted. When adbd starts, it has root privileges, which are to be dropped later using setuid. However, the setuid call fails when the number of processes running on the system is maxed out. The error returned by setuid is not checked by adbd hence adbd continues to run as root.

In our study, we have measured that 30 seconds after starting an application on a "fresh" emulator, there are on average 58.3 processes running on the phone with a standard deviation of 4.5 when using the set of 10,000 Android applications. When we ran the DroidDream samples, we measured that after 30 seconds, there were on average 660.3 processes running with a standard deviation of 238.8. This behavior is caused by the exploit creating many new processes with the name "rageagainstthecage". In general, very high number of new processes started by an application is a good indicator of malicious activity (as well as malfunctioning apps).

Example #2: Undesired SMS activity

One approach that cybercriminals currently use to monetize compromised smart phones is to have them send text messages to premium SMS numbers once they have socially engineered a user to download their Trojan application. Upon launching such a Trojan, the attacker's application automatically sends one or more texts to such numbers, and the cybercriminal gets paid for each such message sent. Example of Trojans that exhibit such behavior are shown in the table below.

Trojan Name (Package name)	MD5	Number of text messages sent	To number	Premium
Android/Fakeplayer.A (org.me.androidapplic ation1)	fdb84ff8125b3790011b8 3cc85adce16	3	3353, 3354	Yes
Android/Fakeplayer.B (org.me.androidapplic ation1)	5b087aef1247591b1efe7 8032476bde7	4	7132	Yes

Android/Fakeplayer.G EN (org.me.androidapplic ation1)	46a53f4a6637e2807d791 02a6a937c2e	4	7132	Yes
Android/Bgserv.A (com.android.vending. sectool.v1)	4e70abe0ae8a557f66239 95bef1d9ba7	1	10086	
Android/Twalktupi.A (com.incorporateapps. walktext.apk)	c3a0f5d584cc2c3221bbd 79486578208	# of contacts	All contacts	

When conducting behavioral analysis of such malware, observing SMS activity that occurs without any user action whatsoever (beyond the application requesting blanket permissions) is an interesting signal. However, attempting to use such a signal in and of itself can lead to false positives. For instance, out of the 10,000 Android applications from Google's market that we scanned via behavioral analysis, we found 11 of them sent SMS messages upon startup, but were not malware or malicious -- in particular, they sent SMS messages to the users phone thanking them for signing up and using the application. Of course, those SMS messages were sent to the user's phone number instead of a premium SMS number, but illustrate that looking only for SMS sends without also looking at what number a message is being sent to would be a very fragile approach. However, by looking at which numbers SMS messages are sent to, how many, and the content of the messages, a strong signal for when applications are abusing SMS can be derived.

We also note that mobile malware could detect whether or not it is being analyzed, as does much Windows malware does via red-pill-type techniques. Much mobile malware written to-date does not seem to employ such checks, but we fully expect that mobile malware will become more sophisticated, most likely on a time scale that is more accelerated compared to the sophistication of Windows/PC malware. That said, there are many techniques that can be employed as countermeasures to "red pills" up to and including conducting behavioral analysis on actual devices themselves.

Through the Looking Glass: Mobile Drive-bys

Most mobile malware attacks by trojans have relied on social engineering to encourage users to download them, but an emerging class of automated exploits resembles **drive-bys**, which don't require the user to do anything to get infected when visiting a web page. Let's examine the probable anatomy of a drive-by attack executed over a mobile network.

On the web, drive-by attacks have replaced email attachments as a preferred attack vector, because cybercriminals realize a higher rate of infected machines when the user is not required to click on anything to get infected. Simply by visiting an infected webpage and with no user interaction required, the victim is infected within a few hundred milliseconds without any signal that their device has been compromised. What's more, drive-by malware may start only one or two processes to keep a low footprint of activity. And as with any human endeavor, malware authors automate their manually intensive labors.

Drive-bys have plagued web sites for several years. While some of the first drive-by attacks surfaced around 2005 and papers on widespread use of them emanated in 2007 [Provos], widespread use of drive-by attacks on mobile platforms has yet to emerge -- although there indeed have been interesting efforts relating to drive-bys on mobile phones. For example, the JailbreakMe.com project allows users to jailbreak and unlock their iPhones via a drive-by method that exploits the way Apple's mobile operating system processes certain fonts [ZDNet].

There's a simple reason why mobile phones are an increasingly tempting target, namely due to the following technological shifts:

1. Smartphones are valuable. They are in fact mini-computers loaded with browsers, email, applications and media. In addition, they are increasingly being used for mobile commerce, including mobile banking and retail transactions. Because mobile phone store data valuable to criminals, such as credentials, contacts, preferences and other information on user identity.
2. Smart phones are vulnerable. Mobile web browsers are as robust as their desktop counterparts, with JavaScript interpreters and third-party plug in support, resulting in increased attack surface. Plug-ins such as Flash, which have been targeted on the desktop browser with shellcode and drive-bys because of their inherent vulnerabilities, are now becoming standard on many mobile phones.
3. Smart phones are using common software packages. The WebKit browser engine has been adopted by most major, growing mobile device platforms (both Android and iOS), and vulnerabilities that allow for remote code execution have been found in WebKit. Due to its large, complex code base, there will inevitably be additional vulnerabilities uncovered in WebKit.

A Prototype of an Android Drive-By Attack

There is little documentation of how mobile drive-bys occur. Can drive-bys occur on mobile web browsers just as easily as they can occur on desktop/PC-based browsers? Absolutely.

Let's walk through the sequence of steps that occurs in a mobile drive-by that we prototyped for the Android platform. In the attack below, we conduct a data ex-filtration attack against Skype via a drive-by in which we are able to steal the contents of a user's instant messaging conversations. Once the user simply loads an infected web page that takes advantage of a WebKit browser exploit, (steps 1 and 2) the attacker is able to get backdoor access to issue commands of his or her choosing to the device (step 3). From there the attacker targets Skype by accessing the username account name, traversing to the user's Skype data directory, and exports the user's private data files to his or her own machine (steps 4 - 8). Finally, once the data is locally cached, the attacker can run sqlite to query and view the uesr's data (steps 9 – 10).

Attack Overview

1. The user visits a malicious or infected website via the on-phone browser.
2. The website serves active content to exploit a vulnerability on the phone (Webkit vulnerability CVE-2010-1807)
3. The exploit payload connects back to the attacker via a TCP socket. *(The attacker now has local shell access to the phone)*
4. The attack then finds the Skype username for the device by extracting it from `/data/data/com.skype.raider/files/shared.xml` which is world readable:
   ```
   USERNAME=$(sed -n "s:.*<Default>\(.*\)</Default>:\1:gp"
   /data/data/com.skype.raider/files/shared.xml)
   ```
5. The attack then exploits a Skype vulnerability (CVE-2011-1717) that allows local users to read sensitive files including contacts, conversation transcripts, voicemail, and so on.
6. The attack can then traverse to the user's Skype data directory:
   ```
   $ cd /data/data/com.skype.raider/files/$USERNAME
   $ ls -l

   -rw-rw-rw- app_117  app_117               0 2011-07-12 11:20 config.lck
   -rw-rw-rw- app_117  app_117               0 2011-07-12 11:20 main.lock
   -rw-rw-rw- app_117  app_117          348160 2011-07-12 12:14 main.db
   drwxrwxrwx app_117  app_117            2011-07-12 11:20 voicemail
   -rw-rw-rw- app_117  app_117               0 2011-07-12 11:20 keyval.lock
   -rw-rw-rw- app_117  app_117           40960 2011-07-12 11:20 keyval.db
   -rw-rw-rw- app_117  app_117           12824 2011-07-12 11:20 keyval.db-journal
   -rw-rw-rw- app_117  app_117               0 2011-07-12 11:20 bistats.lock
   -rw-rw-rw- app_117  app_117           61440 2011-07-12 11:21 bistats.db
   -rw-rw-rw- app_117  app_117           33344 2011-07-12 11:21 bistats.db-journal
   -rw-rw-rw- app_117  app_117            3075 2011-07-12 12:51 config.xml
   drwxrwxrwx app_117  app_117                 2011-07-12 11:22 chatsync
   ```

7. In order to successfully exfiltrate these files the attack uses a mix of built-in commands on a stock Android device. First, the attack sets up a listener on the attacker's home machine that will receive and uncompress files from a network socket:

   ```
   $ nc -l 4000 | tar xvf -
   ```

8. Once the listener is active, the attack then sends the files to the attacker's home machine

```
$ tar -cf - *.db | busybox nc ATTACKER_HOST 4000
```

9. Now the attacker has the files on their machine and can perform further analysis.

```
$ nc -l 4000 | tar xvf -
x ./main.db
x ./keyval.db
x ./bistats.db
$ sqlite3 main.db
SQLite version 3.7.5
Enter ".help" for instructions
Enter SQL statements terminated with a ";"

 sqlite> .tables
 Accounts        ChatMembers     Conversations   Participants
 Voicemails
 Alerts          Chats           DbMeta          SMSes
 CallMembers     ContactGroups   LegacyMessages  Transfers
 Calls           Contacts        Messages        Videos

sqlite> select from_dispname, body_xml from Messages;
 Joe Smith|This is a secret message because skype uses the
 encryptions!
```

10. The attacker can read the user's conversations.

To date, there has not been a wide prevalence of a mobile drive-by attack. But if this kind of exploit follows a growth path similar to other malware that has migrated to phones, one can assume that an attack similar to what we detail here is not only practical, but probable. As mobile applications are often structured as a thin client application that serves as an interface for a web application back-end, cybercriminals are likely to become interested in infecting mobile web applications just as they do so for desktop web applications. In addition, as many mobile applications are monetized by third-party advertising, and we have seen malvertising on the web be increasingly used as a distribution vector for drive-by malware, it would not be surprising if more mobile malvertising were to occur.

In the GGTracker malware incident, for instance, attackers redirected users to a spoofed version of Google's Android Market to socially engineer users into downloading a malicious Android application. However, the last step in that attack could instead take place via a mobile drive-by, thereby increasing the conversion rate with which cybercriminals are able to infect users' phones.

Conclusion

In this paper, we have taken a preliminary look at mobile malware threats:

1. While mobile malware threats have had a slow ramp since 2004, there has been a significant growth in the prevalence of mobile malware incidents in the past two years.
2. Behavioral analysis can be used to conduct automated studies of application functionality, as well as a complement to static analysis to identify privacy and security violations conducted by mobile applications.
3. We conducted a preliminary, behavioral analysis of 10,000 Android applications, and also used behavioral signals to identify characteristics of mobile malware and applications that violate user privacy:

    - The top ad networks used by mobile applications are Admob, Doubleclick, Google,
    - Adwhirl, and Mobclix
    - The top analytics packages used by mobile applications are Flurry and Google Analytics
    - Over 800 applications leaked private information, such as IMEI and IMSIs
    - 11 of the applications we analyzed sent potentially unwanted SMSes to users
    - Mobile malware such as DroidDream and Fakeplayer can be successfully identified via behavioral analysis signals

4. Mobile drive-by attacks are possible on Android. We prototyped a drive-by data ex-filtration attack against Skype.

Acknowledgements

Special thanks to Mike Teeling, Ameet Ranadive, Shariq Rizvi, Ariana Beil, and Sriram Puthucode for their contributions to this paper.

REFERENCES

[Andromaly] http://andromaly.wordpress.com/documentation/

[Fortinet] http://blog.fortinet.com/airpush-pushes-the-envelope/

[iKeeSRI] An Analysis of the iKee.B (Duh) iPhone Botnet, http://mtc.sri.com/iPhone/

[Juniper] Juniper Research, "Mobile Payment Strategies: Opportunities & Markets 2011-2015"

[Lookout] The Lookout Blog, "Security Alert: Android Trojan GGTracker Charges Premium Rate SMS Messages", 20 June 2011

[LookoutFeb2011] MyLookout AppGenome Project February 2011 report, https://www.mylookout.com/appgenome/

[Provos] The Ghost in the Browser Analysis of Web-Based Malware, http://www.usenix.org/event/hotbots07/tech/full_papers/provos/provos.pdf

[SMobile2010] Threat Analysis of the Android Market, http://www.globalthreatcenter.com/wp-content/uploads/2010/06/Android-Market-Threat-Analysis-6-22-10-v1.pdf

[TaintDroid] TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones
William Enck, Peter Gilbert, Byung-gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. In Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI), October 2010

[TowerGroup] Tower Group, *Computerworld* 14 April 2011

[Verizon Data Plans] https://www.verizonwireless.com/b2c/mobilebroadband/?=plans[

[ZDNet] ZDNet, "iPhone hacked with zero-day font vulnerability", 6 July 2011