Android Fragmentation: Protecting the Gene Pool By Better Knowing It

Louis Ades
December 10, 2014
Mentor: Ming Chow, Tufts University

Abstract

Perhaps one of the most prominent security issues for Android, Google's mobile platform, revolves around fragmentation. A disconnect between Google's own updates to its operating system, and the carriers and hardware manufacturers who distribute it, creates lingering security vulnerabilities that cannot be patched as readily or as reactively as Google would like. This makes Android a much more open target to attack, as attackers are given much more time to create an exploit. Rather than focus on the general nature of fragmentation, however, this paper will instead focus on the preventative and protective side of the topic. The paper will begin with a recitation on the current state of Android fragmentation. I will then proceed to outline a few ways security teams are efficiently targeting key vulnerabilities resulting from fragmentation, with primary focus on using a DIY "Droid Army" to deploy and test Android software efficiently over a diverse range of devices. Finally, the paper will conclude by outlining ways developers, consumers, and Google have been (and can in the future) alleviate the dangers inherent in Android fragmentation.

To the Community

Android, in its many different flavors, currently has the largest worldwide smartphone OS market share, and by a longshot (according to the International Data Corporation, its share in unit shipments is up to 84.4%, versus iOS in second place at 11.7%). For its prevalence in the current technological world, it was a very attractive subject matter to approach, and since fragmentation on Android is still one of the biggest issues in security, I felt that it was an extremely relevant topic. However, for all the case studies and papers dedicated to fragmentation and the many risks it poses for Android users, I was very surprised to find that so few of them focuse on preventative measures, or ways in which security teams can overcome the issues that fragmentation gives them. Very few focus on the ways in which developers and users can better make themselves aware of potential vulnerabilities in their Android devices. This was surprising to me, as security is not supposed to be just about saying, “You’re going to get hacked!” but about also saying, “Here’s how you can stop yourself from getting hacked!” (Actually, regarding security in general, I wish more security papers spent more time focusing on the protective side of security, rather than simply including the obligatory 2-3 sentence blurb at the conclusion of each paper without much detail.) I wanted to write about fragmentation in order to better say the latter statement on the topic. A lot of this paper is about efficiently spotting the many vulnerabilities that arise from fragmentation, because in many ways, knowing one’s enemy is most of the work necessary to then deal with it in a proactive way.

Introduction

Google's Android mobile operating system's latest brand slogan is "Be Together, Not the Same." It is a statement that is intended to emphasize one of its key differences from its primary competitor--Apple's iOS--that every Android phone is different and offers a unique experience with unique hardware, yet that they all still run on the same operating system (OS). This creates an extensive level of diversity in Android devices unseen in iOS devices (since iOS devices are all manufactured by Apple and very few different models exist each year). While its diversity may be an advantage highlighted by Android's marketing team, it has also allowed the operating system to become enormously fragmented, a huge issue for any security teams working on the OS (and an advantage perpetually highlighted by any attackers on the OS). Normally in security, when a vulnerability exists, a "gate" is opened for an attacker, but a diligent security team can spot this vulnerability and close this gate before it is exploited. With Android's case, however, when phones are not upgraded in a timely manner because the phone manufacturer or carrier delays the upgrade, these gates are left open for extended periods of time, sometimes never to be closed [1].

Many academic papers and peer-reviewed articles focus on the nature of fragmentation, and the dangers to mobile security it poses. In this paper, however, after giving a brief overview of these dangers, I've resolved to instead focus on the protective and preventative side of fragmentation: since fragmentation is arguably one of the greatest core design issues the Android OS faces, how does Google deal with it? How do developers secure their apps amongst so many platforms? How can users protect themselves from known exploits on their phones that have yet to be patched or fixed? How might security researchers better spot exploits in Android, and how might they identify where in the "Android gene pool" they are prominent? These are the questions, particularly the latter one, that will be defined, expanded upon, and addressed in this paper.

1. The Current State of Fragmentation

In August 2014, OpenSignal published a visualization of the current state of Android fragmentation. It illustrates the enormous diversity of Android devices by model, hardware manufacturer (also known as original equipment manufacturer, or OEM), latest version of Android currently running on these devices, and other statistics. It then compares them with iOS's current state of OS fragmentation. As illustrated in this visualization, while iOS appears to strictly minimize their device diversity in order to inhibit fragmentation, Android appears to have an entire "gene pool" of Android devices, sharing certain traits over a wide spectrum of hardware while differing in other areas (throughout this paper, I will regularly use the term "Android gene pool" to describe this phenomenon) [2]. This is why Android fragmentation is such a key issue, and it is why it is imperative there be efficient ways of testing for security vulnerabilities over a wide diversity of Android devices. While patches for these vulnerabilities may not be distributed in a timely manner, it allows users to better protect themselves from vulnerabilities that they are aware exist.

Fortunately, CVE Details, an online database logging known security vulnerabilities on a variety of different platforms, holds data on many already known Android vulnerabilities. This data can be used to make users better aware of where their device may be vulnerable so that they can remain cautious where they may be attacked. This database also illustrates how seriously fragmentation can affect a wide range of devices, even when these vulnerabilities are known and the latest version of the software has been patched. For example, vulnerability CVE-2013-4787, aka the Android "Master Key" vulnerability, allows an attacker to execute arbitrary code in an application on a victim's phone, and it has a CVSS Score of 9.3/10 (very dangerous). This is a vulnerability that exists on Android versions 1.6 through 4.2--according to OpenSignal's data, a total of 71.2% of Android users (those not on versions 4.3 or 4.4) are susceptible to the Master Key vulnerability [3]. Many of these users may never have their phones patched with the latest version of Android once their phone is discontinued with the previous version. This is why powerful methods of vulnerability testing across wide ranges of devices is important: it allows for a quicker gauge on what *is* vulnerable, how severe it is, and how widely across the Android gene pool it exists.

2. Know Thy Enemy

a. Using Droid Armies for Cross-Device Research

One way to perform efficient testing over a huge sample size of the Android gene pool is with the help of a "Droid Army." There are many services online that allow for *app testing* over many flavors of Android, but none seem to focus on security research. Thanks to the work of Joshua J. Drake of Accuvant LABS, a set framework for DIY Android security research exists. A Droid Army, according to Drake, is a web of Android devices across the Android gene pool all prepared for immediate cross-device security research. In a presentation at the Black Hat USA 2014 security conference, he demonstrated how vulnerability testing over any range of Android devices (by version number, hardware, OEM user interface, etc.) [6]. This is similar to how, in the realm of medicine, diseases in humans are tested to see if they correlate with genetics by identifying how they react with different humans among the human gene pool.

There are two parts to his presentation: the physical "army" construction (assembling all of his Android devices, their hubs, power supplies, etc.), and the software base (building the software tools to allow for quick and widespread testing across devices using just a Linux terminal) [6]. As a college student, I don't have the resources (or the money) to build a web of Android devices and prepare them all, so I will instead focus on the software end, demonstrating how I performed testing on my old Samsung Galaxy Nexus developer phone (as if it was doing it across web of devices).

The first step to constructing a Droid Army is the hardware end. In his presentation, Drake comically illustrates all of the steps he took to reaching his final product--he found that, by connecting webs of hubs to each other (as a tree of hubs connecting others), he could realistically experiment on a maximum of 108 Android devices at once. Acquiring that many devices can be difficult, but so long as they are not all bought new and or on a mobile carrier contract, it may not have to be the most pricey experience (using eBay, asking around, garage sales, etc.) [6]. In order to replicate his experience, I took my Galaxy Nexus phone, and plugged it into my computer via microUSB--I don't have quite those physical resources.

The second step, the software end, is the more challenging part, and the one I will focus more on in this paper. Drake developed what he calls an "Android Cluster Toolkit", which works

with the Android SDK's *adb* (Android Debug Bridge) tool to perform widespread testing. To set up his framework, I first downloaded the Android SDK onto my Ubuntu machine, and opened up the SDK manager in order to install "platform-tools" (containing *adb*) via a simple GUI. In order to run the SDK manager (and the SDK in general), a Java runtime environment needed to also be installed on my computer. Once the SDK was installed and updated, in order for the Android Cluster Toolkit to work properly, *adb* (which is a bash script) needed to be accessed from any directory, so I edited the *bashrc* file to include the code in Figure 1 [7]. Once this was done, I used Git to clone the Android Cluster Toolkit from Drake's GitHub repository, and initialized it using the instructions in the Readme [8].

```
#AndroidDev PATH
export PATH=${PATH}:~/android-sdk-linux/tools
export PATH=${PATH}:~/android-sdk-linux/platform-tools
```

Figure 1

In order to use the toolkit to experiment with cross-device security research, I experimented with its relevant functionality. First, after I initialized (and it detected my Nexus phone), I accessed the /lib/devices-orig.rb file, and gave my device a name by changing the code for my device as shown in Figure 2. Because this toolkit allows for giving each device a different name, it allows for more proper and more readable results of different tests executed. I ran ./reconfig.rb (as shown in Figure 3), and it finished the setup of my environment.

```
{
  :name => 'My Test Nexus', # description
  :serial => '01467D561901001B',
},
```

Figure 2

```
lades01@lades01-G75VW:~/Desktop/comp116hubwork/android-cluster-toolkit$ ./reconf
ig.rb
[*] Loaded 1 device from 'devices-orig.rb'
[*] Found 1 device via 'adb devices'
[*] Matched 1 device!
[*] Missing 0 devices:
```

Figure 3

At this point, it was time for me to experiment with the two most relevant scripts included in the toolkit: ./mdo and ./mcmd. Executing ./mdo allows running any *adb* commands to any and all devices in the hub (selective execution by name is allowed, and a single period indicates sending it to all devices in the Droid Army). Executing ./mcmd allows running any shell commands on the devices in the same manner. *Adb* commands are also able to run shell scripts just by saying "shell", but running ./mcmd cuts out the middleman in a clean fashion. Without doing any full exploit testing, I used these scripts to probe my one device as if it was several devices in a cluster, modeled off a "checklist" in Drake's Black Hat Presentation [6]. I started by using a shell command to get the fingerprint for my "cluster", as shown in Figure 4. I then used the same process to get the Linux kernel version and Android SDK build (Figures 5 and 6, respectively). Using *adb* commands, I ran a log command on a single, specific device, and siphoned the output into a text file (Figure 7). (Figure 8 shows a snippet of the log when I was first outputting into the terminal instead of putting it into a separate file.) In the same vein, I used the *ps* shell command to show all current running processes, filtered (Figure 9). Finally, I tested distributing a test app to my cluster--I ran an *adb* command to install a small app I had developed over the summer onto my army of one (Figure 10). From here I'd be able to use further shell commands to run the app on all devices as well.

```
lades01@lades01-G75VW:~/Desktop/comp116hubwork/android-cluster-toolkit$ ./mcmd -
d . getprop ro.build.fingerprint
[*] My Test Nexus:
google/mysid/toro:4.2.2/JDQ39/573038:user/release-keys
```

Figure 4

```
lades01@lades01-G75VW:~/Desktop/comp116hubwork/android-cluster-toolkit$ ./mcmd -d . cat /
proc/version
[*] My Test Nexus:
Linux version 3.0.31-g9f818de (android-build@vpbs1.mtv.corp.google.com) (gcc version 4.6.
x-google 20120106 (prerelease) (GCC) ) #1 SMP PREEMPT Wed Nov 28 11:20:29 PST 2012
```

Figure 5

```
lades01@lades01-G75VW:~/Desktop/comp116hubwork/android-cluster-toolkit$ ./mcmd -
d . grep ro.build.version.sdk= system/build.prop
[*] My Test Nexus:
ro.build.version.sdk=17
```

Figure 6

```
^Clades01@lades01-G75VW:~/Desktop/comp116hubwork/android-cluster-toolkit$ ./mdo -d "My Te
 Nexus" logcat > logsample.txt
```

Figure 7

```
I/dalvikvm-heap(  611): Grow heap (frag case) to 17.013MB for 1644560-byte alloc
ation
D/dalvikvm(  611): GC_FOR_ALLOC freed 7K, 10% free 17393K/19148K, paused 18ms, t
otal 18ms
D/dalvikvm(  611): GC_FOR_ALLOC freed 678K, 11% free 18086K/20224K, paused 18ms,
 total 20ms
D/dalvikvm(  611): GC_FOR_ALLOC freed 2161K, 20% free 16781K/20740K, paused 18ms
, total 18ms
```

Figure 8

```
lades01@lades01-G75VW:~/Desktop/comp116hubwork/android-cluster-toolkit$ ./mcmd -
d . ps m.android.phone
[*] My Test Nexus:
USER     PID   PPID  VSIZE  RSS     WCHAN    PC         NAME
radio     569   127   485776 30652 ffffffff 00000000 S com.android.phone
```

Figure 9

```
lades01@lades01-G75VW:~/Desktop/comp116hubwork/android-cluster-toolkit$ ./mdo -d
 . install shapesync.apk
[*] My Test Nexus:
4142 KB/s (10841583 bytes in 2.556s)
        pkg: /data/local/tmp/shapesync.apk
Success
```

Figure 10

b. Applications of a Droid Army

While the adb and shell commands I used as part of my environment were relatively tame, they indicate the power behind this toolkit. Using the same tools, one may design an app designed to probe a vulnerability and exploit it, install this app on a mass of Android devices, and use their output responses as indicators as to which were affected (and if applicable, then how). As an example, a well-known vulnerability in the addJavaScriptinterface API allows for the injection of arbitrary and sandboxed Java code by an attacker, which is extremely dangerous. CVE-2014-1939 is an Android vulnerability opened because of this flaw in the described API [3]. Writing a simple app designed to exploit this vulnerability, pushing it to all devices in the Droid Army, then running it on all devices will allow a security research team to evaluate which members of the Droid Army have been crippled. This set of commands alone, along with the

expansion of this into sending intents, allows for testing for exploits in the core Android OS, exploits in apps, and exploits in hardware (if code were to be written to gain access to said hardware, which is simply a matter of setting permissions). Additionally, as a side note, testing of this scale also serves as a DIY cross-device app testing (as in, testing in general if an app being developed functions properly on a wide set of devices).

This type of testing works in tandem with the purpose of this paper--to demonstrate that one can best overcome fragmentation by first identifying what it affects, and how. From there, at least for developers and security teams, creating patches is the easier part.

3. Overcoming Fragmentation - Google, Developers, and You

For Google's development team, spotting and identifying vulnerabilities in Android is the hardest step--patching them becomes routine in most cases. That being said, Google as a whole has the larger hurdle of reducing both the extent and the effects of fragmentation on its OS--patches do not help if most of its user base has no access to them. Google seems to be aware of this hurdle, however, as they are continuously attempting to find new ways to control OEMs so that the OS can become less fragmented. One such stride was in February 2014, where Google issued approval windows for Google Mobile Services (GMS), where OEMs are required to adhere to certain deadlines and restrictions in order to run Google's applications and services on an Android device they sell [9]. According to these windows, an Android device cannot be approved to use GMS if it is sold with a version of Android nine months past the release of the latest version. This means, according to *Android Police*, "no OEMs can certify a device more than two versions behind the current Android release [9]." Strides such as this do not obliterate the problem, but it does help proactively reduce its growth in future years.

Android app developers, meanwhile, run into other issues regarding fragmentation--particularly, they not have to worry about potential security vulnerabilities within their apps, but be able to test for these vulnerabilities' potencies on an enormous diversity of devices (and, at the same time, ensure their application works and is polished at all on all those devices as well). For developers,there exists a wide variety of options for efficient cross-device testing. As mentioned in the previous section of this paper, one of the applications of a droid

army is efficient DIY cross-device testing for applications [6]. The caveat is that, being DIY, this may not be the most short-term cost effective solution, and it's difficult to gauge how diverse one's sample size of Android devices is compared to the diversity of devices that will eventually be running the app. There are other specialized services that developers can use as well. Testdroid is one such service that allows for testing and debugging in a cloud-based environment on virtually any model Android device [10]. Apkudo, also has a service dedicated to developers do perform cross-device and automated, dynamic testing of applications as well (in fact, Apkudo is a company dedicated to overcoming the hurdles of fragmentation in general--all of its services lend itself to the different parties that are required to deal with it) [11]. Both of these services are also cost effective in that they are specialized, efficient, and can perform app testing on a wider variety of devices than perhaps any DIY solution. The availability of these services makes overcoming fragmentation on Android much easier and simpler than if all developers were required to resort to a DIY solution. Additionally, like the purpose of droid armies, these services are dedicated to *better knowing the problems*, since for many developers, spotting the problems is much more difficult than actually patching them.

While ensuring that Android and the apps running on it overcome the dangers of fragmentation is mostly the job of Google and app developers, there are many several practices consumers can embrace in order to put their own security in their own hands as well. Veracode, a cloud-based application risk management service, offers several tips on how to better protect an Android device from hackers. Their website details what it believes are the three biggest hacking threats to one's device: a) data in transit, where hackers can use man-in-the-middle techniques to redirect data transferred over insecure connections; b) third party apps, since Google can't control the apps distributed in third party app stores; and c) SMS Trojans, since a Trojan app can obtain access to a victim's phone bill, and can also send out SMS text messages to contacts to get them to download the app and spread the worm. Veracode presents ways to mitigate these threats as well, respectively: a) SSL encryption, b) careful testing and examination of third party apps, and c) being wary of SMS Trojans by being wary of permissions granted and implementing controls for unauthorized access [12]. Technology news and review sites such as CNET also offer tips on protecting one's Android device. Expanding on Veracode's tips, CNET also warns

against pirating apps and promotes installing anti-virus, utilizing security settings built into later Android versions, and ensuring one's version of Android is the most recent that they can install [13].

Though many of these tips and tricks are very useful, not many of them help with directly overcoming the dangers of Android fragmentation. Perhaps the only true defense from letting one's phone always has the latest version of Android with the most recent patches to vulnerabilities is to ensure that one is regularly purchasing a newer model, which doesn't say a lot. One can do even better by purchasing one of Google's flagship Nexus models--these are always released with the latest version of Android, and receive updates very soon after Google releases them. Each is usually supported for at least two years as well. Aside from these tips, overcoming fragmentation is mostly in the hands of those creating the software for Android.

Conclusion

The prevalence of Android in the mobile marketplace only continues to grow. As it grows, it becomes a bigger target. As an open-source OS with completely outsourced hardware manufacturing, without proper guidance its fragmentation may remain just as big a problem as it always has been. Google is currently striving to either reduce fragmentation or at least alleviate the dangers brought about through fragmentation. Meanwhile, the best that security teams, developers, and consumers can do is know the threats they face, and know how to better protect themselves from these threats once they are brought into the light. This is why projects such as the Joshua Drake's Droid Army are so important--they better control that sense of awareness, and allow cross-device testing for Android to happen with better and faster results. In an open market, where there are so many strengths and opportunities and benefits to reap, there will be tradeoffs as well. With Android, it is not necessary to eliminate these tradeoffs, but to be made *aware* of them and address them, so that the boons (open source, open market, modability, flexibility and versatility, etc.) are not outdone by said tradeoffs.

References

[1] Jones, Margaret. "Android Fragmentation: More OS Versions, More Problems." *TechTarget*. Aug. 2013. Web. 10 Dec. 2014. <http://searchconsumerization.techtarget.com/feature/Android-fragmentation-More-OS-versions-more-problems>

[2] "Android Fragmentation Visualized." *OpenSignal*. Aug. 2014. Web. 10 Dec. 2014. <http://opensignal.com/reports/2014/android-fragmentation/>

[3] "Android: Security Vulnerabilities." *CVE Details*. 26 May 2009 (last updated 2 Jul. 2014). Web. 10 Dec. 2014. <http://www.cvedetails.com/vulnerability-list/vendor_id-1224/product_id-19997/Google-Android.html>

[4] "Vulnerability Details: CVE-2013-4787." *CVE Details*. 9 Jul. 2013 (last updated 11 Oct. 2013). Web. 10 Dec. 2014. <http://www.cvedetails.com/cve/CVE-2013-4787/>

[5] "Vulnerability Details: CVE-2013-1939." *CVE Details*. 2 Mar. 2014 (last updated 4 Mar. 2014). Web. 10 Dec. 2014. <http://www.cvedetails.com/cve/CVE-2014-1939/>

[6] Drake, Joshua J. "Researching Android Device Security With the Help of a Droid Army." *Accuvant, Inc*. 6 Aug. 2014. Web. 11 Dec. 2014. <https://www.blackhat.com/docs/us-14/materials/us-14-Drake-Researching-Android-Device-Security-With-The-Help-Of-A-Droid-Army.pdf>

[7] Uusitalo, Jani. "AndroidSDK." *Ubuntu*. Last updated 28 Jul. 2012. Web. 11 Dec. 2014. <https://help.ubuntu.com/community/AndroidSDK>

[8] Drake, Joshua J. "jduck / android-cluster-toolkit." *GitHub*. Last updated 11 Apr. 2014. Web. 11 Dec. 2014. <https://github.com/jduck/android-cluster-toolkit>

[9] Smith, Chris. "How Google Plans to Strike a Major Blow Against Android Fragmentation." *Boy Genius Report*. 11 Feb. 2014. Web. 10 Dec. 2014. <http://bgr.com/2014/02/11/android-fragmentation-google-plans/>

[10] "TestDroid." *Bitbar*. Web. 10 Dec. 2014. http://testdroid.com/

[11] "Apkudo for Developers." *Apkudo LLC*. Web. 10 Dec. 2014. <https://www.apkudo.com/developers.html>

[12] DuPaul, Neil. “Android Hacking.” *Veracode*. Web. 10 Dec. 2014. <http://www.veracode.com/products/mobile-application-security/android-hacking>

[13] Graziano, Dan. “Protect your Android Device from Malware.” *CNET. CBS Interactive*. 19 Dec. 2013 (last updated 25 Jun. 2014). Web. 10 Dec. 2014. <http://www.cnet.com/how-to/protect-your-android-device-from-malware/>