An Empirical Study of the Robustness of Google Play Store Applications Using Fuzzing Techniques

Harrison Fernandez[1], Aleksey Kravtsov[1], Mevin Thomas[1], and Kiran Chandrapaul[1]

[1] John Jay College of Criminal Justice (CUNY), New York, NY.
[1] firstname.lastname@jjay.cuny.edu

Abstract. Mobile applications are heavily trusted with personal information, made to assist everyday life for the purpose of convenience, i.e. banking and computing, though mobile app developers are not required to actively seek out software vulnerabilities. A major foundation of application security is its robustness, i.e. how the application handles random or unexpected input. If the application crashes after unexpected input, it is not robust. The Android operating system is the most widely used in mobile devices, including smartphones, tablets, and wearable technology. In this paper, we perform an empirical study of the robustness of Google Play Store applications using fuzzing techniques. We evaluate the top 100 apps on the Play Store's 'top charts' section, emulating the Android operating system using the NOX emulator, while utilizing the standalone fuzz testing Android application, DoApp. DoApp is an all-in-one intent fuzzer that is able to push the boundaries of applications using null, random, and semi-valid intent fuzzing; intents simply being messages that transfer data from one component (Activity, BroadcastReceiver, ContentProvider, and Service) in an Android application to another in the same application. We find that seventeen out of sixty-one tested applications return unfavorable outcomes, including crashes and unauthorized access to content that is otherwise blocked by a login screen that may ask for an email or telephone number credentials. These applications are all rated 4.0 or higher on the Google Play Store, and range from one million to five hundred million downloads or installs.

Keywords: Fuzz testing, android, robustness, vulnerability discovery.

1 Introduction

Smartphone use is continuously growing across the world, with market researcher Newzoo expecting 3.8 billion people will own a smartphone by 2021 [16, 17]. Smartphones, including Android [2] or iOS? [6] devices, store and handle most of our personal data with use of the correct smartphone applications (or 'apps'), i.e. banking, computing, web browsing, and email. The security of our smartphones is of the utmost importance, yet it is arguably the device most susceptible to threats. Of the possible threat vectors, including but not limited to wifi, bluetooth, and spoofed application signatures, the smartphone's security relies heavily on the applications that it runs. The main security feature of an application is its robustness. In computer

science, robustness refers to the ability of the application to handle errors during execution, potentially caused by invalid or unexpected inputs. If an application is not robust, why should we continue to use it? If a malicious actor were able to gain access to our smartphone, they could utilize an invalid or unexpected input and cause our applications to shut down or worse, gain unauthorized access to the application.

In 1989 at the University of Wisconsin Madison, a team of students led by Professor Barton Miller developed fuzz testing after using UNIX command line utilities over a noisey dial-up line [7]. As a result to the noise, false signal characters caused the command line utilities to crash. This revealed that software can be manipulated by false inputs, ultimately leading to security-relevant weaknesses. Mainly aimed at command-line and user-interface fuzzing, the original goal was to demonstrate that modern operating systems are vulnerable to those simple implementations of the attack.

Android has become the most used smartphone operating system, powering more than eighty percent of devices worldwide [28]. Android's founders Rich Miner, Nick Sears, Chris White, and Andy Rubin created the operating system designed initially for digital cameras in 2003 and then mobile devices in 2004 in Palo Alto, California. Later in 2005, Google bought Android and released the operating system for commercial mobile devices. Google produced releasing the first-ever Android smartphone in 2008, the G1 [21]. Several years and Android versions later, Google has developed the operating system to support smartphones, tablets, and Internet of Things devices alike.

The Google Play Store initially released in 2008 and served as the official application store for the Android operating system ever since [11]. This digital distribution service allows users to browse and download applications developed with the Android software development kit (SDK), approved by Google. The Google Play store houses more than 3.3 million applications as of 2018, and more than 2 billion active users each month [20]. Data storage practices, malware, sideloading, lack of encryption, and lack of updates all contribute to mobile application vulnerabilities.

The Google Play store and Android present an excellent target for fuzzing techniques. Android is an open source project, meaning it receives a huge amount of attention from researchers and active developers. With the Android OS supporting more than 80 percent of mobile devices worldwide [28], any type of software vulnerability discovery is a major concern. Fuzz testing creates a major attack vector for exploiting vulnerabilities, and application developers should write code that guards against fuzz testing. With such a huge market share and open source developer community, we would expect the smartphone to be robust and secure.

The purpose of this project is to perform an empirical study of the robustness of Google Play Store applications. Robustness in this context refers to the ability and extent of handling unexpected input. To do so, we consider the use of current fuzzing techniques on mobile devices [5, 8, 14, 19, 22, 23, 24, 26, 27, 29, 30, 31, 35, 36, 37, 38, 40, 41]. For our experimentation, we utilize the Android emulator, NOX [18], with the smartphone fuzzing application, DoApp [8]. The Android emulator is rooted (root access is already attained upon installation of the emulator), the iPhone equivalent of jailbreaking, which is a requirement for use of the fuzzing application.

Additionally, the NOX emulator has USB debugging enabled, for use with the Android Debug Bridge, also known as ADB [1]. ADB is used to run Logcat to record errors made by DoApp. The 100 Google Play Store applications that were evaluated in this experiment were chosen from the Top 100 Apps list as of October 9th, 2018. The applications on this list range from having 1 million to 500 million downloads, and on average are rated as 4.7, with lowest being 2.7 and the highest being 4.8. Appendix A contains the full list of the applications. Our results show vulnerabilities or bugs that cause a number of applications to crash, display an 'Application Not Responding' message, and allow us unauthorized access. We find that applications mainly focused on video and audio content do not check for null intents, and thus allow us unauthorized access to content we would have to log in for. By finding these bugs, we should hold application developers accountable for the software they publish and the content that they sell. If the application is not robust, it should not be trusted for use whether it be banking, web browsing, computing, etc.

The paper is organized as follows: Section 2 details work related to this study, including different types of fuzzing applications and techniques available for use on mobile devices, specifically the Android operating system. Section 3 discusses our approach to fuzz testing, including more detail regarding our experimental setup and the applications that were evaluated, including the NOX emulator, the DoApp fuzz tester, and ADB and Logcat for analysis. Section 4 exhibits the results of our experiment, showing common results, including examples of crashes and unauthorized access. Section 5 explains the significance of our findings, including our interpretation and shortcomings. Finally, section 6 declares our plans for future work as well as the ideal experiment setup that would overcome the weaknesses in this study.

2 Related Works

There are many existing fuzz tools for the Android OS. They aim to assist application developers in testing of their application before deployment. There are many tools available because there are many different strategies when it comes to generating inputs. Inputs may take the form of user interactions (UI), which may include clicks and scrolls, or classic data input, or system events, which may take the form of system notifications such as low battery or SMS notices. Additionally, tools can approach the application to be tested as a black box [14], where application code structure is unknown, or a white box [30], where application code structure is known. A gray box approach [27] is also possible, where only a high-level structure of the code is known. Lastly, fuzzing tools may also follow a random [14] or systematic [29] generation of input. To allow for systematic generation of input, dynamic traces are used, usually by the fuzz tester evaluating the target application. For example, a fuzzer that employs random inputs may use the same strategy across all applications that are tested, while fuzzers aimed at using systematic inputs may analyze execution paths in the application before testing to reveal properties or acceptable inputs that may generate unwanted behaviors. Google is known to review any application that is submitted for

placement on the Google Play store by emulation, a framework known as Google Bouncer, now rebranded as Google Play Protect [9, 10]. However, while the proprietary framework is used to scan applications for threats, specifically malware and spyware, it is unknown whether or not the scans evaluate the application's robustness. The following review is a compilation of various mobile fuzz testers for use on Android applications, including applications from the Google Play store.

Intent Fuzzing:

Android applications typically have four types of components that interact with each other, named Activity, BroadcastReceiver, ContentProvider, and Service. To trigger an interaction, or inter-component communication (ICC), between two or more of these components, an Intent must be called. In simplest terms, an intent describes the intention of the application, i.e. a message that is used to transfer data from one Activity to another. Random input intent fuzzers are known to generate invalid intents, thereby testing the robustness of an application. Random input intent fuzzers are also useful for the discovery of security bugs, vulnerabilities, and denial-of-service. The fuzzing approaches listed here perform intent fuzzing but may differ among other functions, including black/white/gray box approach, UI or system event fuzzing, and/or random or systematic generation of intents.

Monkey [14] is the most common tool used to test Android applications. It is part of the Android developer toolkit, and very simple to use. However, it is the most basic of the fuzz testers, considering the target application as a black box. It uses a pseudorandom flow of user events, including clicks, touches, and gestures.

Dynodroid [29] is a black box fuzz tool that generates random inputs based on checking the application through monitoring of the application prior to testing. It is more efficient than other basic random input fuzzers, such as Monkey [14], as it is able to generate system events, and not only user events.

Null-Intent Fuzzer [19] is a basic intent fuzzer that aims to crash applications that do not check the input of intents. Though it is a tool with a very specific purpose, it is still widely used. Null intents are a major factor in crashing applications, yet it is still a common flaw. Null-Intent Fuzzer would be categorized as a black box fuzz tester as well.

Intent Fuzzer [37] is a basic fuzzer that utilizes randomized data inputs, specifically focusing on how one application interacts with another application that is installed on the same device. It is also able to recognize the structure of intents that the application would generate, so that Intent Fuzzer is able to generate random intents.

ICCfuzzer [38] is an intent fuzzing framework, developed to find inter-component communication (ICC) vulnerabilities in the Android operating system and Android applications. It utilizes interprocedural control flow graph (ICFG) analysis of the target application to find all possible components that can accept Intents to be fuzzed, and generative data fuzzing which generates random data fields to be passed through an Intent. Four types of vulnerabilities may be found in Intent fuzzing: (1) null pointer exceptions that will cause an application to crash, (2) Intent spoofing, or when a malicious actor is able to create and launch false Intents, (3) Intent hijacking, a malicious actor may gain access to an Intent that contains

sensitive data through a data leak, and (4) data leak, a vulnerability that leaks sensitive data without user authorization.

Droidfuzzer [23] is an automated Intent fuzzer to find bugs in Android applications. It is focused on fuzzing in terms of data input. The tool extracts information from Activity components to find what is 'normal' data that can be accepted. It uses this as a seed to generate random data. Then, the random data is injected into the application, and execution is monitored for crashes. Several video and audio players were tested, including MX Player, Baidu, and TTPod. These apps mostly contained bugs causing crashes, consumption of resources, application not responding (ANR) screens.

PUMA [26] is a black box fuzz tool that acts similar to Monkey, in that it utilizes random data input. However, it builds upon Monkey in that it automates user interactions as well as acts as a framework for the extension of other dynamic analyses. PUMA is open source, and can be implemented for analysis on Android applications.

GUIRipper [22] is a fuzzer that is able to build a GUI modeling of the application to be tested and generate input specifically for the target application. GUIRipper specializes in generating user interaction events. It creates the GUI model by testing of the target application, while also keeping track of what possible events can be triggered by a user at any given step of the model.

CONTEST [23] is an automated concolic testing algorithm for Android applications that is used to test input, known as symbolic execution, along specific paths (concrete execution). Concolic testing is able to give directed inputs to discover bugs during execution, while fuzz testing is used to test through a wider breadth of possible inputs. CONTEST is tested on open source Android applications such as Random Music Player, Sprite, Translate, Timer, and Ringdroid, and shows that it is more efficient than existing concolic execution algorithms.

3 Approach

The approach is split into two components: artifact collection and experimental setup.

The artifacts we collected include the top 100 applications on the Google Play app store. Specific reasons about how applications are ranked are not publicly disclosed by Google Play, as it is a proprietary ranking algorithm, but one factor besides those listed, which include number of downloads and ratings, is the retention rate of an application, or how long an application remains on a device. For example, a simple bot can be created to generate downloads for an application, as a way to boost the number of downloads. However, if the application is being deleted immediately after, or the 'phone' disappears after, these downloads can be noted as invalid.

Following the above survey of existing fuzz testing approaches, we designed our own approach which utilizes a state-of-the-art Android application fuzzer, DoApp (Denial of App). We choose DoApp over existing work as it is an intent fuzzer that tests using null, random, and semi-valid intents. It is also important as it is an

standalone Android application, and does not require an external setup.

Artifact Collection

The artifact collection phase of this project consists of downloading and annotating Google Play Store applications. This includes free applications from the top 100 list across many categories, including but not limited to, games, utilities, social, and news. The applications were all running on their latest updated versions which was no later than 2 months before initial fuzz testing begun. This gave us the opportunity to test the developer's latest structure and security implementations that users are actually using currently for relevant results. Appendix A lists the entire list of the top 100 apps on the Google Play Store as of October 9th, 2018. These are the targeted applications for fuzz testing. The image below describes how annotations are made for each application and includes category, date of publication, latest update, current version, permissions, developer, number of installs, size, and which version of Android is required. The applications on this list as of October 9th, 2018 have an average rating of 4.7. The highest rated application (4.8 out of 5), Free Antivirus 2019, is number 38 on our list. The lowest rated application (2.7 out of 5), Video Player, is number 82 on the list.

Name - Top Free Apps in Play	Developer	Category	Rating	# of Installs	Last Updated
TikTok	musical.ly	Social	4.4	100,000,000+	October 9, 2018
Messenger - Text and Video Ch	Facebook	Communication	4.1	1,000,000,000+	October 9, 2018
McDonald's	McDonalds USA, LLC	food and drink	3.6	10,000,000+	September 26, 2018
Wish - Shopping Made Fun	Wish Inc.	Shopping	4.5	100,000,000+	October 9, 2018

Experimental Setup

The experimental setup phase of our project consists of multiple components: the emulation environment, the fuzzing application, and the tools necessary to document and archive results. For each of these components, we utilize the NOX Android emulator [18], DoApp fuzz application [8], and Android Debug Bridge [1] and Logcat [13], respectively.

The NOX Android emulator is an application for Windows and Mac OS that allows users to install and emulate Android applications. There are many ways to emulate the Android operating system [3], but NOX is useful for its speed, motion controls, and ability to become 'rooted'. In order to have unbiased results while using multiple machines, the environment we used needed to have the same technical settings on all of them. This allows the team to fuzz multiple applications at the same time, while being efficient. For the experiment, we used the Nox Android Emulator using two gigabytes of memory. We also enabled root user to the emulator as it is requested by the DoApp fuzzer. Rooting is another term for obtaining the highest privileged access and control over various components of the operating system, the Apple equivalent to jailbreaking. We utilize NOX on Mac Mojave (10.14.1), NOX version 1.2.1.0. The NOX emulator uses Android version 4.4.2 and Linux kernel version 3.4. By using NOX emulator, we are able to run Android without the use or purchase of a cell phone and download applications from the Google Play Store. In

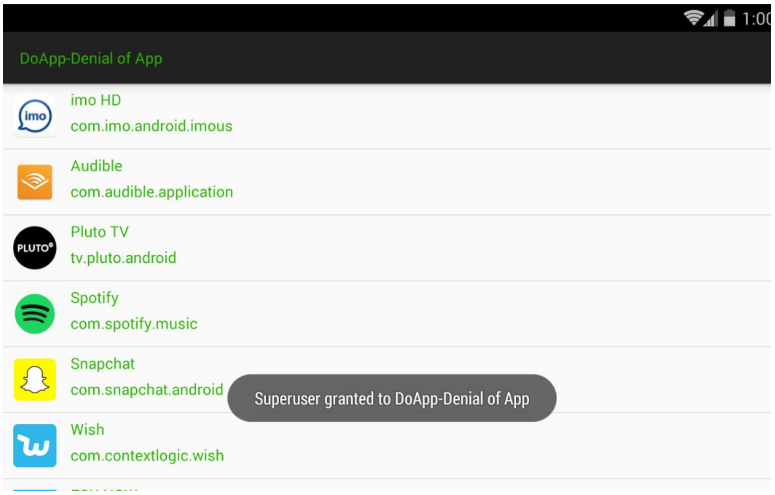
addition, we use DoApp [8], which automatically finds paths to generate intents and executed outputs based on the paths.

Below is an example of the NOX emulator, running Android 4.4.2:

It is a basic Android emulator, with a simple switch to give superuser privileges and root the device, both of which DoApp requires to perform proper testing.

The DoApp fuzzing application runs on the Android OS as an application, officially named DoApp - Denial of Application [8]. The apk file was downloaded from the github project's webpage and installed onto the Android emulator. Once installed, the application will display on the user interface. As we enable the DoApp application, it asks us to choose what application we would like begin fuzzing. From there, the interface switches between the targeted app and DoApp, as DoApp inputs and retrieves data. DoApp was been created to function using the following five steps: (1) DoApp receives information on all the installed applications on the system. This process uses Android Package Manager, and only non-system apps are considered, i.e. applications that are installed by the user. A list of installed apps is then displayed on DoApp so that the user can choose what they want to fuzz. (2) DoApp analyzes the application the user chooses and extracts all the data fields representing inputs authorized by the application. (3) Specific intents are generated for these different components of the application. For example, Activities allow the user to interact with the app while keeping track of user inputs, Services keeps the application running in the background if needed to perform long-running operations or remote processes, Content Provider manages access to data stored by the application and also allow sharing data with other apps, and Broadcast Receiver allows applications to register for system events and receive broadcasts from the system when a specific event happens, i.e. text messages. (4) Randomly generated intents are sent to these different components. Each time a malicious intent is sent to the component and the application responds, the service kills the app being tested to perform a new test and ensure independent outcomes. (5) Logcat is used to analyze any crashes and generate a report.

Below is a screenshot of the DoApp fuzzer running on the NOX emulator:



You can see that the application requires superuser permission in order to properly test the applications. The user interface of the application is also very beneficial, as one would simply tap on any of the applications currently installed on their device to begin testing.

In order to use Logcat to generate these reports, we first need Android Debug Bridge (ADB). ADB is a command line tool that helps the user communicate with an Android device. ADB is usually used by testers for installing and debugging applications on a device plugged into a computer or to debug applications on an emulator. ADB also provides access to the device's UNIX shell which can run a variety of commands. With the help of ADB, we are able to quickly install applications on to the device and run Logcat to monitor and record fuzzing results, whether they proved to contain software bugs or not.

Logcat is a command line tool within the Android device's UNIX shell (accessible via ADB) that outputs a log of system messages. Using Logcat, we collect application and system logs that originated during the fuzzing process using DoApp. In order to connect ADB Logcat to the Nox Android Emulator, the emulator has to be modified. Without modification, the emulator will block the connection. Bypassing the block requires enabling 'Developer Mode' on Nox.

Below is an example of a Logcat output:

```
D/su      ( 8964): sending code
D/su      ( 8964): child exited
D/su      ( 8962): client exited 127
I/DoAppLOG( 2134): Application PID:
I/DoAppLOG( 2134): Kill app
D/su      ( 8969): su invoked.
D/su      ( 8969): starting daemon client 10039 10039
D/su      ( 8971): remote pid: 8969
D/su      ( 8971): remote pts_slave:
D/su      ( 8971): waiting for child exit
D/su      ( 8973): su invoked.
D/su      ( 8973): db allowed
D/su      ( 8973): 10039 /system/xbin/su executing 0 killall -9 com.microsoft.office.word using binary
/system/bin/sh : sh -c killall -9 com.microsoft.office.word
D/su      ( 8971): sending code
D/su      ( 8971): child exited
D/su      ( 8969): client exited 127
D/AndroidRuntime( 8974):
```

The above example shows the Logcat output after a fuzz test through DoApp on Microsoft Word. In this Logcat output, we actually see that the Microsoft Word crashes with the passing of a null intent.

4 Results

Fuzzing a wide array of applications should allow one to discover multiple types of vulnerabilities that lead to either a) crashes, b) displays of an “Application Not Responding” message, or c) consuming of system resources for an indefinite period of time, or d) giving access to a screen that only authorized users should have access to.

Given our experimental setup, we were able to test 61 out of the top 100 applications. 39 of 100 were omitted because they were not compatible with the NOX emulator, as it runs Android version 4.4.2. The 39 omitted applications required a higher version of Android. Of the 61 tested applications, 17 were found to have software bugs that caused them to either crash or allow unauthorized access to the rest of the application. Of the 17 vulnerable applications, 12 crashed due to errors in handling random input and the remaining 5 applications were able to be exploited, bypassing an authentication screen requiring some form of identification, i.e. email, Facebook credentials, or phone number. Surprisingly, these applications are all rated 4.0 or higher, with 1 to 500 million installs. All of the vulnerable applications vary by category, ranging from communication and social, to shopping and entertainment.

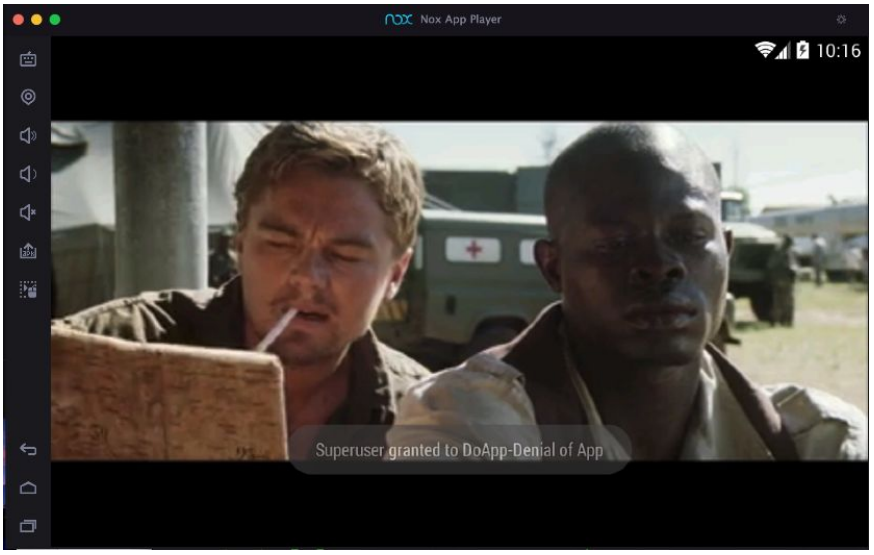
There were a few developers that created more than one application on the top 100 list such as Google and Amazon Mobile. As a result, we found that applications created by related developers had similar frameworks and and security protocols vulnerabilities associated with one another. These large developers have invested immensely to protect against from fuzzing tools and other attacks. Steps to protect their data include enforcing secure communication, applying signature-based permissions, using SSL traffic, using intents to defer permissions, and securing private data on internal storage. Although this may be convenient for developers to maintain multiple apps efficiently, it also means that one vulnerability can be exploited in all the applications shared by that related developer.

Almost ninety percent of the the applications tested required the user to login with some type of credentials. The credentials consist of a user generated password,

email, name, and possible billing information. In some cases OAuth, Facebook Login, and other authentication solutions were available. Although this seems like a great way to prevent unauthorized access, fuzzing tools just like DoApp try to challenge this theory by injecting large amounts of malicious data continuously to specific components for multiple rounds. If DoApp is able to generate a test case random enough to confuse the application, a user can skip the login screen and go straight to the following screen.

Results varied from app to app. In most cases, the applications that dealt with video tended to appear in our result lists as either crashing or lacking proper intent structure. Our most prominent fuzzing attempts allowed us to bypass a login screen and move to screens beyond the authentication screen. In 5 of our applications, intents were not built properly to where we could watch movies, change settings, and send messages without authentication. Improper handling of intents could allow us to craft exploits that would allow attackers to access the app in ways they should not be allowed to.

We were able to watch a movie on PlutoTV, “Blood Diamond”, without authenticating or authorizing the service. This shows that the intents were not properly formed because the application allowed us to call an intent that brings us straight to the video player.



The above image shows the PlutoTV application giving us access to watch Blood Diamond.

The following details the Logcat output, showing us what occurred after DoApp passed a null intent to play the movie:

I/DoAppLOG(14214): Type: null - Action:null - Extra Text: null - Extra Stream: null - Data: http://dev.pluto.tv/7Wu58pTEDO

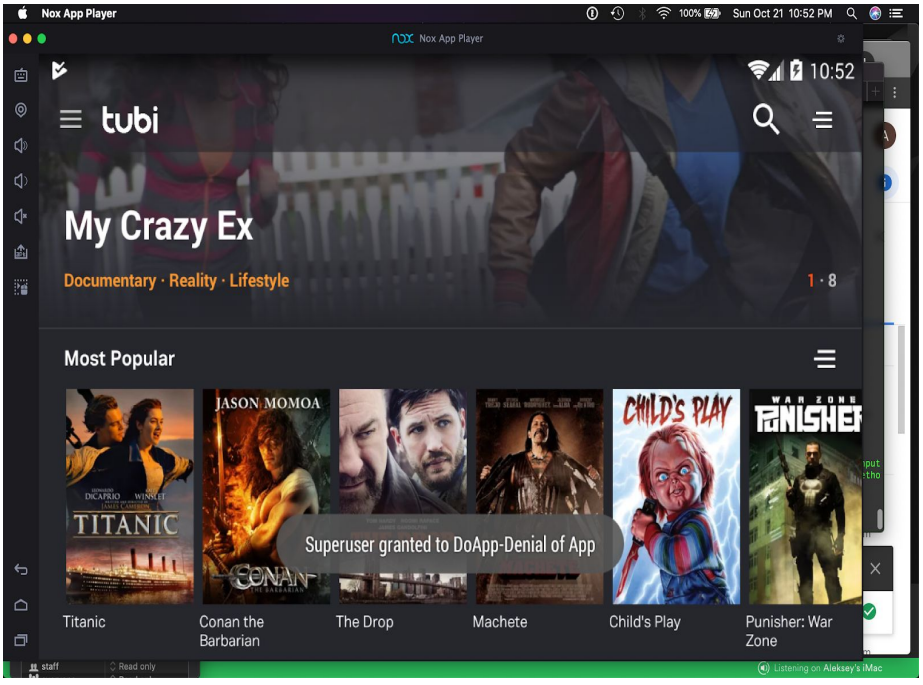
I/ActivityManager(567): START u0 {dat=http://dev.pluto.tv/7Wu58pTEDO flg=0x10000000 cmp=tv.pluto.android/.controller.MainActivity} from pid 14214

Type: null - Action:null - Extra Text: null - Extra Stream: null - Data: http://dev.pluto.tv/watch/jWnA4ObRlg

I/ActivityManager(567): START u0 {dat=http://dev.pluto.tv/watch/jWnA4ObRlg flg=0x10000000 cmp=tv.pluto.android/.controller.MainActivity} from pid 14214

We can see in the above text that the null intent was passed through Activity to start PlutoTV in an attempt to give us unauthorized access.

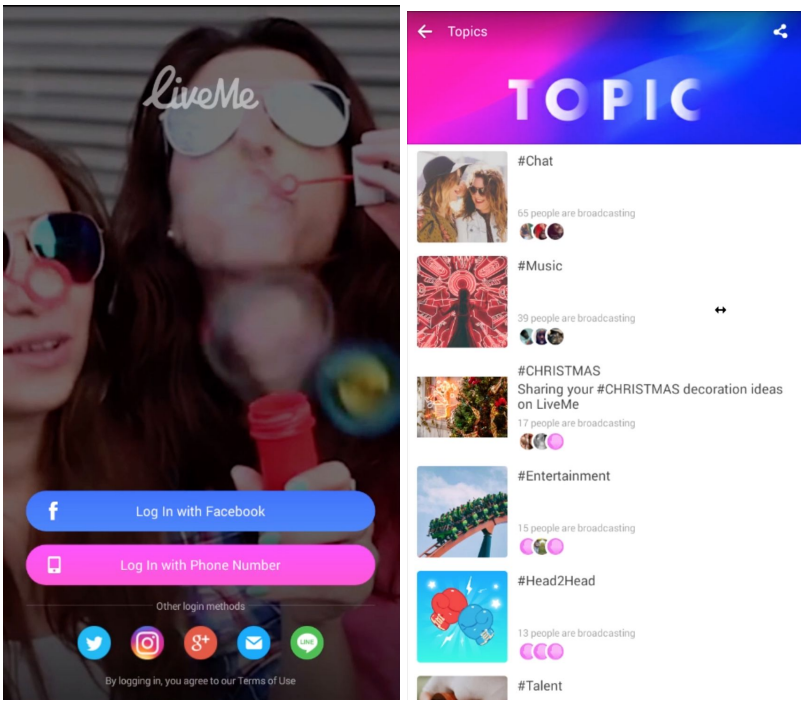
In the following image we also see that DoApp allows us unauthorized access to the tubi application, another video player. To gain access to the following screen, one must sign in with some type of user credentials, however with a null intent, we are able to bypass the login screen.



However, there were more than simply null intents passed that were able to show us that a few applications are weak when it comes to robustness. In the following image, we can see the DoApp output in Logcat that caused the Bitmoji application to crash. We see in the last line of the following image that the crash was caused by an SSL error:

```
E/ImageLoader(  867):         at
com.nostra13.universalimageloader.core.decode.BaseImageDecoder.decode(SourceFile:74)
E/ImageLoader(  867):         at it.a(SourceFile:265)
E/ImageLoader(  867):         at it.d(SourceFile:238)
E/ImageLoader(  867):         at it.run(SourceFile:136)
E/ImageLoader(  867):         at
java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1112)
E/ImageLoader(  867):         at
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:587)
E/ImageLoader(  867):         at java.lang.Thread.run(Thread.java:841)
E/ImageLoader(  867): Caused by: javax.net.ssl.SSLProtocolException: SSL handshake
aborted: ssl=0xb8ede670: Failure in SSL library, usually a protocol error
```

In LiveMe, a user is presented with a login screen, as seen on the left. After dozens of forced intents, including null, random, and semi-valid, we were able to get past the authentication screen, and into the section “topics”. Had we manually crafted the intent to bring us to the screen, we would have been able to start moving about the app as an authenticated user.



In Snapchat, a crash is caused by an ActivityNotFoundException. Below we can see the Logcat output as well.

```
android.content.ActivityNotFoundException: Unable to find explicit activity class
{com.snapchat.android/com.snap.mushroom.MainActivity}; have you declared this activity in
your AndroidManifest.xml?
```

Results from other applications varied with different types of exceptions being thrown such as SSL handshake errors and permission errors. One of the other prominent exceptions that were discovered that stood out is a class casting exception which can allow a cloak-and-dagger type of attack in the Snapchat application. Such an attack can allow an attacker to distribute a fake version of Snapchat that can mimic an Activity and steal credentials.

5 Discussion

For many applications, the fuzzing test was not able to penetrate them leading us with no vulnerability results. However with the apps that did crash, Logcat was able to provide a report of what happened during testing along with possible causes.

Based on our findings, we discovered that even popular applications such as Whatsapp and Snapchat were vulnerable to fuzzing attacks. With both applications exceeding 500 million users, even the smallest vulnerabilities can be cause for

concern. Those high tier applications did not allow us into their inner workings, but they did crash. If a crash is manipulated in the correct manner, with the correct inputs, those can be approaches on the plane of attack. However, with Pluto TV we saw major flaws where attackers could access the application's premium content without proper user authentication and authorization.

10 out of the 17 apps that crashed may contain code that will affect many versions of android. A specially crafted intent with crafted parameters could potentially affect hundreds of thousands if not millions of users. As these apps are some of the most used and downloaded apps available, a single exploit in an app that supports many versions of android can affect thousands of people.

Out of the apps that crashed, most contain these vulnerabilities and can carry them through various versions as they are built only once in the economy of time. 10 out of the 17 applications that crashed actually had versions that supported multiple versions of android. This can be extremely dangerous as finding a single exploit on a single version of android can affect all the other versions, and supported phones.

6 Future Work

As for future work, we are curious to find further correlation between robustness and other variables. A variable we would like to analyze would be the developer's access to funding, which may include their net worth or access to capital (for example, a startup may receive investments from capitalists to get off the ground). We would also have liked to fuzz test applications regarding banking or financial information, including but not limited to Bank of America and Chase, as these types of applications handle very sensitive data, and should be robust most of all.

The team would like to experiment and review various fuzzing applications to determine accuracy and reliability of testing across state-of-the-art fuzzers. So far, we were able to successfully implement the DoApp fuzzer on the NOX emulator. In doing so, the fuzzer demonstrated vulnerability traces in a few applications. However, it may be worthwhile to fuzz test the same applications across fuzz testers, and gather whether the results differ. Our next steps include the continuation of fuzz testing various applications listed on our Top 100 Chart list using different fuzzing techniques.

Another next step may include a survey of Android emulation environments, as there are various emulators for Android, including Bluestacks, NOX, Android x86, and Android Studio. It is not commonly known which is the best, accurate, or most true to life emulator of existing approaches. One measure may include whether or not the same applications crash or allow unauthorized access with the same fuzz testing techniques on different emulators. While the NOX emulator has many positives, as previously discussed, however, it does not support the most recent Android OS. We would like to discover which is the best emulator, which may require testing and comparison with a physical device.

In addition, if we could repeat the experiment, we would consider creating an automated method of running our experiments, which may include scripting multiple

instances of the Android OS to run at the same time. Furthermore, we would find a method of emulation and experimentation that allows us to fuzz test as much of the Google Play Store top charts as possible. Our experimentation allowed us to only test 61 out of 100 applications, due to the emulation environment that only allowed for Android 4.4.2. The Android OS is currently at version 9. It would be vital to repeat the experiment at the latest Android version used in real life. We would aim to test 100 or more applications, including the top charts across categories.

Lastly, we would also like to explore concolic execution on Android applications, as we may find more sophisticated software bugs that fuzz testers cannot achieve. These tools will be the key to finding the required paths in order to find where we can input data. Based on our findings, we can have a greater discussion on how the effects of fuzz testing can affect consumers, developers, stakeholders, and the privacy of user data.

7 References

1. Android Debug Bridge: https://developer.android.com/studio/command-line/adb
2. Android website: https://www.android.com/.
3. Android x86 ISO: https://www.osboxes.org/android-x86/.
4. Android American Fuzzy Lop: https://github.com/ele7enxxh/android-afl.
5. Android LibFuzzer: https://source.android.com/devices/tech/debug/libfuzzer.
6. Apple website: https://www.apple.com/
7. Barton Miller's biography: http://pages.cs.wisc.edu/~bart/includes/bio.html.
8. DoApp Android Fuzzer: https://github.com/lmartire/DoApp.
9. Google Bouncer: http://googlemobile.blogspot.com/2012/02/android-and-security.html
10. Google Play Protect: https://www.android.com/play-protect/.
11. Google Play Store: https://play.google.com/store/.
12. Google Play Store Distribution Agreement: https://play.google.com/intl/ALL_us/about/developer-distribution-agreement.html.
13. Logcat: https://developer.android.com/studio/command-line/logcat
14. Monkey Fuzz Tester: https://developer.android.com/studio/test/monkey.
15. National Vulnerability Database: https://nvd.nist.gov/.
16. Newzoo 2018 smartphone report: https://venturebeat.com/2018/09/11/newzoo-smartphone-users-will-top-3-billion-in-2018-hit-3-8-billion-by-2021/.
17. Newzoo 2018 users by country report: https://newzoo.com/insights/rankings/top-50-countries-by-smartphone-penetration-and-users/.
18. Nox Android Emulator: https://downloadnox.onl/.
19. Null Intent Fuzzer: https://www.nccgroup.trust/us/our-research/intent-fuzzer/.
20. Statistica: Google Play Apps and Users: https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/
21. T-Mobile G1: https://www.gsmarena.com/t_mobile_g1-2533.php
22. D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using GUI Ripping for Automated Testing of Android Applications," In Proc. of 27th IEEE/ACM International Conference on Automated Software Engineering, 2012.
23. S. Anand, M.Naik, H. Yang, M. Harrold, "Automated Concolic Testing of Smartphone Apps", In proc. of 2012 ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, 2012.
24. B. Chi, Y. Ni, and Y. Fu, "ADDFuzzer: A New Fuzzing Framework of Android Device Drivers," In Proc. of 2015 International Conference on Broadband and Wireless Computing, Communication and Applications, 2015.
25. J. E. Forrester and B. P. Miller, "An empirical study of the robustness of windows nt applications using random testing," in Proceedings of the 4th conference on USENIX Windows Systems Symposium - Volume 4. Berkeley, CA, USA: USENIX Association, 2000.
26. S. Hao, B Liu, S Nath, W. Halfond, R Govindan, "PUMA: Programmable UI-Automation for Large-Scale Dynamic Analysis of Mobile Apps", In Proc. of The International Conference on Mobile Systems, Applications, and Services (MobiSys) , 2014.
27. A. K. Iannillo, R. Natella, D. Cotroneo, and C. Nita-Rotaru, "Chizpurfle: A Gray-Box Android Fuzzer for Vendor Service Customizations," In Proc. of 2017 IEEE International Symposium on Software Reliability Engineering, 2017.
28. S. Kovach, "How Android Grew to be More Popular Than the Iphone," Business Insider, 2013.
29. A. MacHiry, R. Tahiliani, and M. Naik, "Dynodroid: An Input Generation System for Android Apps," In Proc. of 9th Joint Meeting on Foundations of Software Engineering, 2013.
30. R. Mahmood, N. Esfahani, T. Kacem, N. Mirzaei, S. Malek, and A. Stavrou, "A Whitebox Approach for Automated Security Testing of Android Applications on the Cloud," In Proc. of 2nd International Workshop on Automation of Software Testing, 2012.

31. A. K. Maji, F. A. Arshad, S. Bagchi, and J. S. Rellermeyer, “An Empirical Study of the Robustness of Inter-component Communication in Android,” In Proc. of 2012 IEEE/IFIP International Conference on Dependable Systems and networks, 2012.
32. B. P. Miller, G. Cooksey, and F. Moore, “An empirical study of the robustness of macos applications using random testing,” SIGOPS Oper. Syst. Rev., vol. 41, pp. 78 – 86, January 2007.
33. B. P. Miller, L. Fredriksen, and B. So, “An empirical study of the reliability of unix utilities,” Commun. ACM, vol. 33, pp. 32 – 44, December 1990.
34. B. P. Miller, D. Koski, C. Pheow, L. V. Maganty, R. Murthy, A. Natarajan, and J. Steidl, “Fuzz revisited: A re-examination of the reliability of unix utilities and services,” University of Wisconsin Madison, 1995.
35. C. Mulliner and C. Miller, “Fuzzing the Phone in your Phone,” Presented at BlackHat USA 2009 Conference, 2009.
36. A. Pietschker, I. Schieferdecker, J. Grossmann, M. Schneider, “Online Model-Based Behavioral Fuzzing”, IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops, 2013.
37. R. Sasnauskas and J. Regehr, “Intent Fuzzer: Crafting Intents of Death,” In Proc. of 2014 Joint International Workshop on Dynamic Analysis and Software System Performance Testing, Debugging, and Analytics, 2014.
38. W. Tianjun and Y. Yuexiang, “Crafting Intents to Detect ICC Vulnerabilities of Android Apps,” In Proc. of 12th International Conference on Computational Intelligence and Security, 2016.
39. J. Wu, Y. Wu, M. Yang, Z. Wu, and Y. Wang, “Vulnerability Detection of Android System in Fuzzing Cloud,” In Proc. of 2013 International Conference on Cloud Computing, pp. 954-955, 2013.
40. H. Ye, S. Cheng, L. Zhang, and F. Jiang, “DroidFuzzer: Fuzzing Android Apps with Intent-Filter Tag,” In Proc. of 2013 International Conference on Advances in Mobile Computing and Multimedia, 2013.
41. E. B. Yi, A. Maji, and S. Bagchi, “How Reliable is My Wearable: A Fuzz-Testing Based Study,” In Proc. of 2018 IEEE/IFIP International Conference on Dependable Systems and Networks, 2018.

8. Appendix A: Top 100 Play Store Apps as of 10/09/18

TikTok
Messenger - Text and Video Chat for Free
McDonald's
Wish - Shopping Made Fun
imo free HD video calls and chat
Youtube Music
Snapchat
Instagram
Spotify Music
Pandora Music
Facebook
Google Play Games
Hulu
Bitmoji
Uber
The Ace Family
News Break
TextNow
Pluto TV
Amazon Prime Video
Cash App
Messenger Lite:
Drum Pad Machine
Walmart
Lyft
Horoscope Pro - Free Zodiac Sign Reading
IN Launcher - Love Emojis & GIFS, Themes
Waze - GPS, Maps, Traffic Alerts & Live Navigation
Pinterest
Grubhub: Local Food Delivery
The Weather Channel: Fall Forecast & Live Alerts
Uber Eats: Local Food Delivery
ZEDGE Ringtones & Wallpapers
Venmo: Send & Receive Money
SoundCloud - Music and Audio
Tubi
Free Antivirus 2019 - Scan and Remove Virus, Cleaner
Microsoft Outlook
Google Translate
Facebook Lite
Discord Chat
VPN Free - Betternet Hotspot VPN & Private Browser
Twitter
FOX NOW: Live & On Demand TV

NFL
DoorDash
Ebay
Reddit
Tinder
LiveMe
Wayfair
Messenger for SMS
Facemoji Emoji Keyboard: GIF, Emoji, Keyboard Theme
Audiobooks from Audible
Horoscope Secret - Crystal Ball Horoscope App
Bird - Enjoy the Ride
CBS - Full Episodes
Google Home
Yahoo Mail
Target
NFL Football 2018 Live Streaming
POF Free Dating App
Step Tracker - Pedometer, Daily Walking Tracker
IHeartRadio - Free Music, Radio and Podcasts
Roku
Free TV Shows App:News, TV series, Episode, Movies
Family Dollar
ESPN
Google Duo
Relax Music
MyRadar NOAA Weather Radar
Google Docs
SmartNews: Breaking News Headlines
Twitch: Livestream Multiplayer Games
Glute Workout
Messages, Text and Video Chat for Messenger
SmartThings
Marco Polo Video Walkie Talkie
Indeed Job Search
Capital One Mobile
Video Player
Amazon Music
Poshmark - Buy and Sell Fashion
Akinator
Power Clean - Antivirus and Phone Cleaner App
Peel Universal Smart TV Remote Control
Groupon - Shop Deals
Credit Karma
Tophatter: Shopping Deals, Fun Discounts and Savings
GoNoodle - Kid Movement and Mindfulness Videos
Wendy's
Chick-fil-A

Yelp
Hotspot Shield Free VPN Proxy and Wi-Fi Security
Microsoft Word
Duolingo
Zillow: Find Houses for Sale
Free Music - Online and Offline Music
Walmart Grocery
Draw.ai - Learn to Draw and Coloring