Exploitation and Detection of a Malicious Mobile Application

Thanh Nguyen
University of South Alabama
tnguye001@gmail.com

J. Todd McDonald
University of South Alabama
jtmcdonald@southalabama.edu

William Bradley Glisson
University of South Alabama
bglisson@southalabama.edu

Abstract

Mobile devices are increasingly being embraced by both organizations and individuals in today's society. Specifically, Android devices have been the prominent mobile device OS for several years. This continued amalgamation creates an environment that is an attractive attack target. The heightened integration of these devices prompts an investigation into the viability of maintaining non-compromised devices. Hence, this research presents a preliminary investigation into the effectiveness of current commercial anti-virus, static code analysis and dynamic code analysis engines in detecting unknown repackaged malware piggybacking on popular applications with excessive permissions. The contribution of this paper is two-fold. First, it provides an initial assessment of the effectiveness of anti-virus and analysis tools in detecting malicious applications and behavior in Android devices. Secondly, it provides process for inserting code injection attacks to stimulate a zero-day repackaged malware that can be used in future research efforts.

1. Introduction

Mobile devices are rapidly becoming the dominant mode for voice and data communications in today's globally networked environment. Market reports indicate smartphone sales reached 1.4 billion in 2015, they predict that the number of connected devices will expand to 6.4 billion in 2016 and that application downloads will hit 268 billion by 2017 [4, 11, 15]. The Android Operating System (OS) has been, by far, the dominant mobile device OS for several years making up 86.2% of the world mobile market in quarter 2 of 2016 [6]. Hence, it is reasonable that attackers are not only refocusing targeting efforts from computers to mobile devices, they are focusing specifically on the Android OS.

The proliferation of the Android OS makes it a natural target for the distribution of malicious code that has the potential to impact individuals along with public and private sector organizations. A number of recent articles highlight the fact that malicious code has successfully bypassed market vendor security [1-3]. According to one article, recent adware/malware, referenced as Android/Clicker.G, that was available on the Play Store, targeted Russian speakers, implemented a six-hour delay prior to behaving badly and then proceeded to bombard the user with requests every two minutes [2]. Another article discusses phishing applications that pose as interface applications for online payment systems that were available on the Play Store [3]. Coupling this type of activity with the growth of the Internet of Things (IoT) introduces new opportunities for remote code execution, Distributed Denial of Service (DDoS) attacks and acquisition of personal information [8]. The impact of an increased attack surface escalates reoccurrence issues by stifling malicious software detection and eradication efforts.

Complicating matters, increases in sophisticated stealth techniques such as code virtualization, encryption, and transformation have made it even harder to detect malware. As noted by Zhou and Jiang [24], a popular approach in the distribution of malware is the injection of seemingly innocuous code into trusted android applications. They go on to state that out of 1260 malware samples that they collected, 86% were repackaged applications. This indicates that repacking popular applications with malicious code and distributing them through market vendors is a viable attack vector. Hence, it is advantageous for both individuals and industry professionals alike to acquire an understanding of the effectiveness of anti-virus, static and dynamic software solutions in detecting repackaged applications that contain malicious code. This environment prompted the hypothesis that commercial and open source tools will not detect a repackaged application with excessive permissions that contain malicious code. In order to address this hypothesis the following research questions were identified:

1. Do software solutions detect repackaged applications?
2. Do analysis tools detect malicious code activity?

URI: http://hdl.handle.net/10125/41911
ISBN: 978-0-9981331-0-2

The contribution of our work is two-fold. First, we demonstrate that detecting repacked applications with malicious code is a significant issue with current commercial and open source software solutions that are currently available from the market. We developed new malware, inserted it into a reverse engineered application (app) that already contained extensive permissions and is currently available in the market. Once the malware had been successfully inserted into the application, the app was re-packaged and tested against 12 anti-virus solutions, three static analysis engines, two dynamic analysis engines and four engines that implement a combination of static and dynamic solutions. Second, we present a reverse engineering methodology that can be replicated to test detection solutions and used as a guide for future code investigations and research efforts.

The balance of the paper is structured in the following manner. Section two presents relevant back ground research. Section three presents the research methodology along with a detailed experimental design. Section four discusses the research results along with providing relevant analysis. Section five draws conclusions from the analysis and presents areas of future work.

2. Background

The continued amalgamation of mobile devices into businesses environments and personal activities raises concerns about risk [16, 27]. Coupling this concern with the growing impact that mobile device residual data appears to be having in legal environments escalates interest in understanding how to mitigate this risk [10, 18]. Hence, the popularity of the Android mobile platform has prompted increased research interest in detecting malicious Android applications. Existing research can be grouped into three broad categories: static analysis, dynamic analysis and Risk Analysis.

2.1 Static analysis

Static analysis, in the context of android operating systems, is typically based on source code, manifest, or binary analysis that searches for strings and patterns that may match known malicious behaviors. These techniques are not limited to analyzing manifest permission requests from applications, but also control flow, signature-based detection, and static taint-analysis.

Felt, et.al., [26] developed Stowaway, a static analysis tool, that extracts manifest files and detects over-privilege requests during install. Stowaway's main concern is detecting whether developers followed a least privilege guideline when setting up their permission requests. Stowaway analyzes the applications and determines the set of API calls that it uses. It then maps those API calls to permissions in order to detect overly privileged applications.

A severe limitation of current static analysis techniques is the reliance on permission-based requests in the Android manifest. Although permissions are a key factor in characterizing and detecting malware, the manifest contains much more information that might help detect malware. Feldman et al. [12] proposed Manilyzer, a tool that uses additional information found in the manifest file. It employs a machine learning algorithm that classifies an application as malicious or benign. Specifically, Manilyzer considers the following characteristics in the manifest file to be significant: 1) permission requests, 2) high priority receivers, 3) low version number, and 4) abused services. Based on profiling 617 applications, they reported that the accuracy detection rate of the tool was at 90% with a false positive and false negative rate of only 10%.

Sanz et al. [13] proposed a method for malware detection using extracted strings from application files as a way to detect anomalies. The researcher's methodology relies on readable string extraction from applications. The process involves the following steps: disassembly, tokenization of symbols, and term frequency. Disassembly involves using Android disassembler smali in order to extract the disassembly. The researchers search for const-string operational code within the disassembled code in order to obtain the strings. The tokenizer utilizes dot, comma, colon, semi-colon, and blank space in order to conform the string. Finally, the string is tested with a point in feature space for anomalies detection. The researchers, however, stated that their detection systems produced high error rates because of false positives, but they state that with more normalization the error rates will decrease. Furthermore, the authors state that this is only from a static stand-point and not a dynamic one where strings can be generated at run-time.

2.2 Dynamic Analysis

Dynamic analysis studies the run-time behavior of programs to classify them as malicious. Min et al. [19] propose a run time-based, behavioral analysis method for detecting malicious applications. They employ a customized emulator by applying API hook technology to the loadable kernel module. Thus, when an app is running through a customized emulator, all sensitive information is logged and sent to the analyzer via Logcat. Their log parser categorizes behavior into

the following categories: 1) application use of intent permission, 2) third part advertisement, 3) leakage of private data, and 4) sending SMS signatures. Based on the amount of information leakage, an application is classified as malicious or benign.

Burguera, et. al., [25] present a behavior-based malware detection system that uses similar dynamic analysis techniques. Their Crowdroid implements a monitor that invokes system calls in order to create a frequency table of system calls on the client side. They employ a K-means algorithm to detect malicious behavior on the server side.

Mahmood et al. [22] presented a scalable dynamic analysis framework for evaluating Android application by utilizing the cloud. The platform utilizes robotium test automation in order to perform fuzz and dynamic analysis on android applications. Specifically, the paper describes a program analysis technique capable of fuzzing an Android application using a large set of test cases. The downside to this project is that black-box testing by robotium requires that applications are signed; thus, resigned applications that are automatically generated may show a decrease in functionality or failure.

Reina et al. [20] presents CopperDroid a dynamic analysis tool that characterizes low-level OS-Specific and high-level Android specific behaviors. CopperDroid utilizes QEMU [29] to automatically perform black-box dynamic analysis on Android applications. The VM-based centric system utilizes dynamic system call analysis in order to determine Android behaviors. CopperDroid also has the ability to determine whether a malware was initiated using Java, JNI, or native code. Results showed that from 1,600 samples of malware, CopperDroid was able to differentiate different behaviors.

In [23], Yan and Yin present DroidScope, an analysis tool that utilizes a virtual-based system in order to detect malware. Specifically, DroidScope reconstructs both the kernel and system level semantics in order to facilitate malware analysis. Furthermore, DroidScope utilizes three tiers of APIs to emulate an Android device. These three tiers included: the hardware, OS, and Dalvik Virtal Machine. Results indicates that the tool was affective in assessing malware samples with low overhead.

2.3 Risk Analysis

Grace, et. al., [21] put forth a system called RiskRanker which is designed to analyze apps for dangerous behaviors. Specifically, RiskRanker provides a proactive scheme in order to detect zero-day malware without relying on signature based algorithms for detection. RiskRanker provides scalability by automating behavior detection of applications. The aim of the research was to reduce the search space needed to detect malware. Thus, the system was designed for scalability, efficiency, and accuracy. The system uses a first-order and second-order analysis to assess potentially dangerous behavior in applications. First-order analysis is mainly designed to quickly evaluate untrusted apps for high and medium risk behaviors. High-risk behaviors exploit the kernel for vulnerabilities. This includes any apps that try to perform privilege escalation. Medium level exploits are classified as application that attempts to charge user money surreptitiously, upload sensitive data, or send SMS messages. Second-order analysis consisted of searching for encrypted native code in application and unsafe bytecode loading. Results for first-order analysis shows that out of 9877 applications that contained native code, 24 was found to have embedded rootkits ranging from 6 different malware family. Medium level risk results indicate that 2374 applications exhibit SMS sending behaviors in the background. Second-order results indicate that 315 samples were found to have encrypted native code execution causing malware installation. Furthermore, 184 unsafe dalvik code loading applications were found. In total, out of 118,318 applications from the Android markets analyzed, 718 malware samples from 29 different families were found.

Crussell, et. al. [14] proposed AnDarwin, a scalable framework that analyzes Android applications for plagiarism. Specifically, semantic information was used in order to detect repackaged applications. Results indicate that out of 265,359 applications, AnDarwin was able to identify 36,106 repackaged or rebranded applications. Furthermore, 88 new variants of malware were found.

While there has been substantial work conducted examining the detection of malicious code from static and dynamic perspectives, there is minimal substantive research that specifically investigates the effectiveness of these solutions in a re-packaged zero-day app that already contains excessive permissions.

3. Methodology

To support the hypothesis proposed in the introduction, the overall research was separated into two high-level stages. The first stage utilizes an initial iteration of a design science methodology as defined by Peffers, et. al, [28] to develop and implement the malicious code into an application. Any over-privileged application could have been chosen for this experiment. As a matter of convenience, excessive permission practice, large user base, and application

size, Snapchat was chosen as the target application. The second stage of this research utilizes the modifications in the first part to conduct a controlled experiment as defined by Sadish, et. al. [30]. The high-level problem statement examines the effectiveness of commercial and open source tools at detecting malicious code that has been injected and repackaged in a legitimate application.

3.1 Malicious Code

The first step in the *design and development* stage investigated what could be achieved with existing Snapchat permissions. Analysis of the AndroidManifest.xml file reveals that Snapchat requests 19 permissions from the user. The next step ran the Quick Android Review Kit [9], a static analyzer tool designed to look for security flaws and vulnerabilities. The application found components that were not protected by a permission. Specifically, the following components in 'com.snapchat.android.' were identified as not being protected:

- AppInstallBroadcastReceiver;
- notification.GcmMessageReceiver;
- LandingPageActivity;
- deeplink.DeepLinkActivity

The malicious application we developed as a result of the analysis is called AndroidTracker. It consists of implanting a hidden service in the form of a botnet that utilizes identified Snapchat permissions in order to steal sensitive data. Furthermore, the application utilizes a standard API to obtain sensitive data without requiring root access. Remote commands are communicated over WiFi, 3G, and/or 4G. It should be noted that for the purposes of this experiment, the malicious service was not obfuscated or packed when injected.

Command and control is created based on Google Cloud Messaging (GCM) by broadcasting messages to the hidden service. One of the main reasons for this choice is due to the amount of services and applications that take advantage of GCM in order to communicate notification and updates for products. Snapchat is a prime example of this. GCM service is embedded into the application easily by leveraging Snapchat's built-in GCM libraries. The GCM enables the device to utilize common APIs and libraries that normal applications would use in order to act as a benign service. Additionally, GCM does not quickly drain the battery of the device as normal sockets do.

The android devices selected for this experiment are detailed in Table 1-Smartphone Attributes. The devices were initialized in an attempt to remove previous data interaction on the device. The initialization process was repeated prior to the execution of each experiment.

The following steps were conducted to initialize the Android devices: First the device is turned off so that it can be booted into recovery mode. Second, the volume-down button is held down while holding the power button in order to boot into advance startup. Third, the volume-down button is used to scroll down to recovery mode. The power button is used to select this option. Fourth, the power button is held down while pressing the volume up button once. The power button is then released. Fifth, the wipe data/factory reset option is selected similar to step three. The final step is to reboot the system.

Table 1. Smart Mobile Device Attributes

Trait	Asus Nexus 4 Phone	Asus Memo Pad 7	Asus Nexus 5 Tablet	LG Leon H345
O.S.	Android KitKat 4.4	Android Kitkat 4.4.2	Android Lollipop 5.1	Android Lollipop 5.0
Internal Memory	16 GB	16 GB	16 GB	8GB
Memory Card	Yes	Yes	Yes	No

The device must then be configured to allow USB debugging in order to use ADB to extract and install applications. USB debugging is enabled by going to the 'Developer Options' in the system menu of the settings page.

3.2 Reverse Engineering & Injection Process

The AndroidTracker malware is coded as a service intent. Our service intent has the ability to listen to commands from an outside service in order to execute commands. The commands include: 1) taking a picture, 2) sending live GPS feed, 3) displaying messages on the phone, and 4) sending notifications. Figure 1 – Repackaging malware illustrates the attack vector we implemented to modify the application.

Figure 1. Repackaging malware

The injection process follows a structural procedure. The specific steps implemented to inject the code are as follows:

1. Initially, the latest Snapchat apk is pulled from the phone using adb. The following command identifies the exact location of the snapchat apk: "adb shell pm list packages -f |grep snapchat"
2. Once the location of the apk was identified, the apk was pulled using the following adb command: "adb pull /data/app/com.snapchat.android-1/base.apk"
3. Using apktool [7], we decompiled the current version of Snapchat into smali byte code using the following command: "apktool d Snapchat.apk"
4. In addition, the apktool was used to decompile our malicious service to obtain the malicious payload. The main purpose of this step is to obtain the actual payload in smali format. Furthermore, the main activity onCreate function can be utilized and copied into Snapchat's function to startup the activity. This step helps the reverse engineering process since the smali bytecode does not have to be completely coded by hand.
5. The manifest file, extracted from the disassembly of Snapchat with the apktool, was utilized in order to identify the startup activity payload injection site: com.snapchat.android.LandingPageActivity. Notice that this was one of the components that was listed as vulnerable based on the static analysis tool. The payload utilizes the main startup event from the startup activity.
6. Next, the service intent and broadcast receivers that the hidden service used was injected into the AndroidManifest. The Google Cloud Messaging system's receivers and services were utilized using the original Snapchat's permission.
7. The smali code obtained from AndroidTracker was copied and extracted into the Snapchat smali folder.
8. The main service activity from AndroidTracker's onCreate function was copied and pasted into the main activity of Snapchat's onCreate function. The function was altered to utilize Snapchat's class to correctly initialize the service without the application crashing. The placement is visible in Figure 3-Injection Code to Start AndroidTracker.
9. Snapchat is recompiled using the following command: "apktool b Snapchat"
10. Java's keytool was used to generate a signature with the following command: "keytool -genkey -v -keystore apk-key.keystore -alias apk-key -keyalg RSA -keysize 2048 -validity 10000"
11. Java's jarsigner was used to sign the generated apk from step nine using the following command: "jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1 -keystore apk-key.keystore 'Snapchat.apk' apk-key"
12. Finally, the apk was deployed onto the Android device using the following command: "adb install 'Snapchat.apk' -r"

```
new-instance v0, Landroid/content/Intent;

const-class v1, Lexample/com/trackerservice/MyService;

invoke-direct {v0, p0, v1}, Landroid/content/Intent;->
<init>{Landroid/content/Context;Ljava/lang/Class;)

.local v0, "intent":Landroid/content/Intent;

Invoke-virtual {p0, v0},
Lcom/example/com/trackerservice/TrackerService;->
startService(Landroid/content/Intent;)Landroid/content/ComponentName
;
```

Figure 3. Injection Code to Start AndroidTracker

3.3 Controlled Experiments

With the growth of malware, an influx of anti-virus solution can now be downloaded from the Android Play Store. For this test, we chose a sample of the top rated anti-virus apps by user from the store. Furthermore, we evaluated popular anti-virus solutions that currently have over one million downloads. Based on these two criteria, we chose a list of 12 anti-virus apps for our experiment. Furthermore, case one studies anti-virus engines that are publicly available online. Anti-virus engines are classified as any scalable online detection engine utilizing multiple suites of security tools.

Research tools were chosen with a web-based interface from a list of scanners acquired from research projects or papers. For online tools, we chose those that had at least one of these criteria: 1) static analysis, 2) dynamic analysis, or 3) combination of static and dynamic analysis. Tools that are limited in size are listed as well. For instance, Andrubis only allows for scanning files under 8 MB. Based on the tools selected, two experiments were conducted.

3.3.1 Experiment One. The first controlled experiment utilizes a structured methodology to test effectiveness of locally installed anti-virus and online antivirus scanning engines. The following provides overview for analysis of commercial anti-virus that have been locally installed. First, the Android device is initialized to the default states based on the steps in the initialization process presented previously.

Second, the commercial anti-virus is installed from the Play Store. Third, the settings are configured to real time monitoring where possible in the commercial anti-virus software. Fourth, the repackaged snapchat is installed with adb using the following command: “adb install ‘Snapchat.apk’ -r”. The fifth step initiates a full system scan. The sixth step initializes Snapchat to ensure functionality of AndroidTracker. The final step records the results and any warnings.

The online anti-virus engine suite follows a similar methodology as the commercial anti-virus applications. However, the online anti-virus engine requires the user to upload the apk onto the server first. The process then follows step six.

3.2.2 Experiment Two. The second controlled experiment runs the repackaged application through online scanner tools that are either static analyzer, dynamic analyzer, or a combination of both. Risk ranking scores are recorded based on the tool’s scoring system. Thus, the risk ranking system can be either numeric or descriptive. Furthermore, file size limits are recorded to assess the scope of the scanner. Since the modified version of Snapchat is 29 MB, if the online engine cannot support the file size, then the tool is deemed unsuccessful. Evidence of residual data is marked successful if any sign of the hidden service name is picked up from the results by manual analysis. Repackage detection assesses whether or not the tool was successfully able to identify a modified version of Snapchat. Finally, the tool was deemed successful if it was able to classify the malware.

4. Results and analysis

Injection of the repackaged app was successful in all four different Android devices that were tested. Each device was tested for functionality of Snapchat and AndroidTracker to ensure successful code injection. Snapchat was able to operate as normal on all four devices. When the user started the modified application, AndroidTracker was initialized in the background as well. The background service communicated with the central http command server using Google Cloud Messaging in order to register devices. In addition, the background service was successful in storing the device data and information in a remote SQL database. Querying of the commands utilized GCM in order to send messages to the device. Figure 4 displays the result of querying our Nexus 4 device for a live picture feed and GPS data. The live picture feed accurately displayed the front camera taking a screenshot of the desktop screen. The GPS data was accurate in displaying the tablet current location as well.

Furthermore, the hidden service was able to listen to query commands including: message notification, message toast, and turn off phone. This was conducted to verify that the botnet was successfully built and deployed using a common API to tests the effectiveness of anti-viruses at differentiating between benign and malicious API calls.

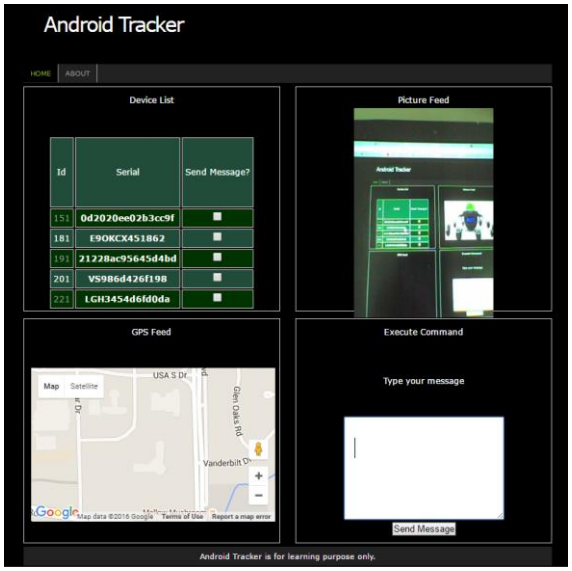

Figure 4. AndroidTracker http command server

The installation and successful communication with the HTTP remote server demonstrates that even though common API calls and libraries were used, a botnet was able to be created. Thus, signature based algorithms have trouble determining if a function’s signature is malicious or a false positive.

4.1 Experiment one

Results from experiment one indicate that current anti-virus applications are not detecting the newly repackaged malware. However, CM Security did give a warning indicating that the application may be leaking data. The data leakage warning was not surprising as the Quick Android Review Tool pointed out potential data leakage from components during the development process. Our results show that all 12 popular anti-viruses did not properly detect our sample. Table 1 - Anti-Virus Detection provides a summary of the test results.

Similar results were displayed from the online apk anti-virus scanners. No scanners were able to detect the presence of malware in the modified Snapchat. It

should be noted that each of the locally installed and on-line anti-virus suites utilize a signature based algorithm to check for malware. It is known that signature based algorithms are prone to not being able to affectively distinguish code variants [17]; the results indicate that they also have problems detecting new vulnerabilities in repackaged applications.

Table 1: Anti-Virus Detection

AV App	Malware?	Warnings
AVG Anti-Virus	N	N
CM Security	N	Y
Dr. Web Light	N	N
Avast	N	N
Norton Mobile	N	N
Kaspersky	N	N
Lookout	N	N
Malwarebyte	N	N
ESET Mobile Security	N	N
Avira Anti-Virus	N	N
NQ Security Anti-Virus	N	N
McAfee Mobile Security	N	N

While there is no holistic or heuritstic checks for malware, it is interesting that no warnings were displayed for any online anti-virus engine. Table 2 summarizes the results for online engines.

Table 2. Online Anti-Virus Engines Detection

	File Size Limit	Successful Detection of Malware?
SandDroid	50 MB	X
Androtoital	128 MB	X
Virus Total	Unlimited	X
OPSWAT	Unlimited	X

4.2 Experiment two

The results from the second controlled experiment were more promising in terms of overall detection. The online tools that use anti-virus engines did not classify our sample as malware. However, tools that use either static analysis, dynamic analysis, or a combination of both did give high risk scores or alerts that the application may be malicious, although no engine declared our sample to be malware. The tools did provide risk scores which appear to be based primarily on use of sensitive API calls and permissions. Scores range from numerical values to descriptive ratings. Furthermore, one of the main constraints with some of the static, dynamic, and combinational analysis tools was the file intake size. Since the repackaged Snapchat was 29MB, some of the tools were not able to handle a file that large.

The second controlled experiment was subdivided into two separate experiments. The first looked at Static analysis and the second looked at tools that utilize a combination of static and dynamic analysis. Dynamic analysis by itself was not included due to limited information given and file size constraint. Some dynamic results showed recording of the applications simply just opening and closing without any data generated while others simply displayed IP address and Geolocations. Thus, work towards a better scalable, automated dynamic engine for malware analysis is needed. Android Static analysis provided the most useful information compared to dynamic analysis. One of the problems with dynamic analysis is being unable to trigger the correct event for data analysis. Since AndroidTracker required Google Play Store to be installed, dynamic analysis utilizing sandbox environments were not able to activate the malicious conditions.

4.2.1 Static Analysis. The fully functional static analysis tools included AVC UnDroid, MobiSec Eacus, and Visual Threat. AVC UnDroid had a file size limit of 7 MB; thus, it was insufficient for dedicating malware. Static analysis results displayed the following types of features such as permission listing, website access strings, geolocation of networks that the application is talking too, sensitive API calls display, component analysis, and certification analysis MobiSec showed unique features since it was able to give a risk ranking score in combination with detecting that the apk was repackaged. Visual Threat gave a risk score of 55 out of 100 due to the permission usage and type of services associated with the application. None of the three static analysis engines were able to detect malware in the repackaged Snapchat. Table 3 summarizes the engines that we used and the results. Overall, static analysis did provide useful information for reverse engineering the malware.

Table 3: Static Analysis Engines Summary

	AVC UnDroid	MobiSec Eacus	Visual Threat
File Size Limit	7 MB	Not Specified	Not Specified
Risk Ranking Score/Scale	X	Low	55/100
Evidence of Residual Data?	X	X	✓
Repackage Detection?	X	✓	X
Successful Detection of Malware?	X	X	X

4.2.2 Combinational analysis. Like static and dynamic analysis, combinational analysis faced issues with file size limit as well. Andrubis and APK Analyzer were not tested due to their 8 MB and 20 MB limit respectively. SandDroid was able to display a risk ranking score of 100. SandDroid defines 1) connecting to the internet, 2) executing shell code, 3) having unused permission, 4) getting GPS info, 5) opening the camera, 6) recording audio, 7) getting unique device id and IMEI, and 8) executing internal requests as all risky behaviors [5]. The normal application that we piggybacked off of uses all of these permissions and behaviors non-maliciously and is flagged as risky: thus, SandDroid also considered our repackaged version risky. The dynamic analysis portion simply displays a gif of the application starting up and closing. No further details were provided for dynamic analysis.

NVISO ApkScan file size limited was unlisted, but it was able to successfully scan the repackaged Snapchat. Results show that the application only shows a medium level of risk since it marked the application as suspicious out of the three options: no malicious behavior, suspicious behavior, and confirmed malicious. Dynamic analysis was unsuccessful at providing any useful information.

Combinational analysis provided as much information as static analysis. The dynamic analysis did not provide useful information since no malicious behaviors were triggered. In addition, none of the engines were able to detect malware in the repackaged Snapchat. Table 4 provides a summary of the results.

Table 4. Combination of Static and Dynamic Analysis Engines Summary

	SandDroid	Andrubis	NVISO ApkScan	APK Analyzer
File Size Limit	50 MB	8 MB	Not Specified	20 MB
Risk Ranking Score/Scale	100/100	X	Suspicious	X
Evidence of Residual Data?	✓	X	✓	X
Repackage Detection?	X	X	X	X
Successful Detection of Malware?	X	X	X	X

4.3 Residual data analysis

The results for case three indicates that static analyzers can trace and detect the actual service registration and sensitive API calls. Figure 5 shows an example of a static analysis tool indicating sensitive API calls made by our malware sample with the threat level associated with the call. Furthermore, the static analysis tool was able to determine services started by the application, including our hidden service. SandDroid, specifically, was able to link permission requests with API calls in order to produce threat level evaluations. Furthermore, SandDroid was able to display the receivers, services and remote server address from AndroidTracker. NVISO ApkScan was able to provide the same information as SandDroid, but it was not able to provide the API call mapping that SandDroid provided. Visual threat was only able to provide residual data for services and remote server address. NVISO ApkScan's main advantages lies in its ability to provide adb logcat dump from the dynamic analysis run. However, no residual data was found in the logcat file. The remaining static and dynamic analysis tools were not able to provide any residual data in their results.

Dynamic analysis did not produce actionable results from any of the tools we tested. More than likely, this was because conditions were not triggered at runtime that executed maliciously inserted code. Most sophisticated malware will trigger only when correct timing conditions are met in order to avoid detection.

While residual data are useful, they are very time consuming to manually find. The experiments rely on the fact that the services and receivers' names were known prior to injection. Thus, it was easy to manually analyze the results to find the residual data.

- **API: Landroid/hardware/Camera;->open**
- Description: Open camera
- Caller Code: Lexample/com/trackerservice/CameraManager;->takePicture()V
- Threat Level:
- Path Index: 20

- **API: Landroid/hardware/Camera;->takePicture**
- Description: Takes picture
- Caller Code: Lexample/com/trackerservice/CameraManager;->takePicture()V
- Threat Level:
- Path Index: 120

- **API: Landroid/location/LocationManager;->getLastKnownLocation**
- Description: Gets geographic location
- Caller Code: Lcom/flurry/sdk/dz;->b(Ljava/lang/String;)Landroid/location/Location;
- Threat Level:
- Path Index: 18

Figure 5. Static Analysis Trace of Repackaged Malware

However, more sophisticated malware can blend its name with the application into which it is injecting. Additionally, obfuscation and packing of the Android application by using the Dex class loader leads to perplexing strings and function names. Thus, residual data becomes increasingly hard to find. Table 5 provides a summary of the residual data that was obtained in the experiment.

Table 5: Residual Data Obtained

	Activities Leaked	Receivers Leaked	Services Leaked	Remote Server Address Leaked
SandDroid	✓	✓	✓	✓
Andrubis	X	X	X	X
NVISO ApkScan	✓	✓	✓	✓
APK Analyzer	X	X	X	X
Tracedroid	X	X	X	X
AVC UnDroid	X	X	X	X
MobiSec Eacus	X	X	X	X
CopperDroid	X	X	X	X
Visual Threat	X	X	✓	✓

5. Conclusions and future work

The popularity of mobile devices and the Android operating system coupled with the continued integration of these devices into all aspects of society is generating an environment that is attractive to attackers. For the purposes of this research, the Snapchat application was chosen for modification due to excessive use of permissions and services along with its popularity.

A repackaged malicious Snapchat was produced in order to stimulate sophisticated zero-day malware. Using a controlled experiment, we assessed the effectiveness of current commercial anti-virus products for Android against a repackaged application that uses piggybacked API permissions. In addition, static analysis, dynamic analysis, and combination of static and dynamic analysis tool analysis were assessed in their detection ability on zero-day repackaged malware.

Detecting new repackaged zero-day malware remains a hard problem in today's environment. Security, to a large extent, relies on the detection abilities provided by markets to monitor published apps. The initial investigation indicates that commercial anti-virus software obtained from Google Play Store may not provide additional protection for such new threats as our contrived malware example. In addition, online detection suites were not able to detect the malware in the repackaged Snapchat.

Static analysis, dynamic analysis, and combination of static and dynamic analysis engines were able to provide better results and warnings for end-users. SandDroid, NVISO ApkScan, and Visual Threat were able to provide residual data that indicated that part or all of our hidden service was initialized. Furthermore, MobiSec Eacus was able to indicate that the apk was repackaged. However, many of the static and dynamic analysis engines were unable to provide any result due to its file size limit. Thus, larger, popular applications are not able to be processed through their engines for results.

The initial research results indicate that the static, dynamic, or combination analysis solutions used in this experiment do not detect malware in repackaged applications. They also indicate that analysis tools used in this experiment provided minimal to no indication of malicious code activity. These results support the hypothesis that commercial and open source tools will not detect a repackaged application with excessive permissions that contain malicious code. They also indicate that there are opportunities for developing improvements in commercial anti-virus, static, and dynamic analysis tools in order to provide better support for sophisticated zero-day malware.

While the scope of this research focuses on the identification of a reverse engineering methodology that utilized dynamic and static analysis tools in conjunction with zero day malware. Hence, future research will focus on expanding the implementation of the methodology to include injection non-zero day malware into popular applications. This research will also investigate the performance of market antivirus and analysis solutions to determine practical malware recognition. In addition, future research will investigate effective obfuscation techniques along with how these techniques perform when analyzed with real-world detection tools. This effort will also examine the development of effective algorithms that detect obfuscated malware. A related stream of research will investigate malware that utilizes the cloud as a propagation mechanism along with effective and efficient mitigation strategies.

6. References

[1] Mobile Security: Why App Stores Don't Keep Users Safe, http://www.darkreading.com/vulnerabilities---threats/mobile-security-why-app-stores-dont-keep-users-safe/a/d-id/1324829, accessed 05/23/, 2016.

[2] Sneaky Android Malware Makes Its Way on the Google Play Store, Again, http://news.softpedia.com/news/sneaky-malware-makes-its-way-on-the-google-play-store-again-503691.shtml, accessed 05/23/, 2016.

[3] Phishing Apps Posing as Popular Payment Services Infiltrate Google Play, http://www.pcworld.com/article/3063474/security/phishing-apps-posing-as-popular-payment-services-infiltrate-google-play.html, accessed 05/23, 2016.

[4] Press Release: Gartner Says Worldwide Smartphone Sales Grew 9.7 Percent in Fourth Quarter of 2015, http://www.gartner.com/newsroom/, accessed 05/19, 2016.

[5] Sanddroid - an Automatic Android Application Analysis System, http://sanddroid.xjtu.edu.cn

[6] Global Market Share Held by the Leading Smartphone Operating Systems in Sales to End Users from 1st Quarter 2009 to 1st Quarter 2016, http://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/, accessed 05/23, 2016.

[7] Apktool, http://ibotpeaches.github.io/Apktool/

[8] Keep the Iot in Check with Penetration Testing, http://www.iotevolutionworld.com/iot/articles/421263-keep-iot-check-with-penetration-testing.htm, accessed 05/19, 2016.

[9] Quick Android Review Kit (Qark), https://github.com/linkedin/qark

[10] Berman, K., W. B. Glisson, and L. M. Glisson, "Investigating the Impact of Global Positioning System (Gps) Evidence in Court Cases", Hawaii International Conference on System Sciences (HICSS-48), 2015

[11] Press Release: Gartner Says 6.4 Billion Connected "Things" Will Be in Use in 2016, up 30 Percent from 2015, http://www.gartner.com/newsroom/, accessed 05/19, 2016.

[12] Feldman, S., D. Stadther, and B. Wang, "Manilyzer: Automated Android Malware Detection through Manifest Analysis", IEEE, 2014, pp. 767-772.

[13] Sanz, B., I. Santos, X. Ugarte-Pedrero, C. Laorden, J. Nieves, and P. G. Bringas, "Anomaly Detection Using String Analysis for Android Malware Detection", Springer, 2014, pp. 469-478.

[14] Crussell, J., C. Gibler, and H. Chen, "Andarwin: Scalable Detection of Semantically Similar Android Applications": Computer Security–Esorics 2013, Springer, 2013, pp. 182-199.

[15] Press Release: Gartner Says Mobile App Stores Will See Annual Downloads Reach 102 Billion in 2013, http://www.gartner.com/newsroom/, accessed 05/19, 2016.

[16] Glisson, W. B., and T. Storer, "Investigating Information Security Risks of Mobile Device Use within Organizations ", Americas Conference on Information Systems (AMCIS), 2013

[17] Kamarudin, I. E., S. a. M. Sharif, and T. Herawan, "On Analysis and Effectiveness of Signature Based in Detecting Metamorphic Virus", International Journal of Security and Its Applications, 7(4), 2013, pp. 375-384.

[18] Mcmillan, J., W. B. Glisson, and M. Bromby, "Investigating the Increase in Mobile Phone Evidence in Criminal Activities", Hawaii International Conference on System Sciences (HICSS-46), 2013

[19] Min, L. X., and Q. H. Cao, "Runtime-Based Behavior Dynamic Analysis System for Android Malware Detection", Trans Tech Publ, 2013, pp. 2220-2225.

[20] Reina, A., A. Fattori, and L. Cavallaro, "A System Call-Centric Analysis and Stimulation Technique to Automatically Reconstruct Android Malware Behaviors", EuroSec, April, 2013,

[21] Grace, M., Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "Riskranker: Scalable and Accurate Zero-Day Android Malware Detection", ACM, 2012, pp. 281-294.

[22] Mahmood, R., N. Esfahani, T. Kacem, N. Mirzaei, S. Malek, and A. Stavrou, "A Whitebox Approach for Automated Security Testing of Android Applications on the Cloud", IEEE, 2012, pp. 22-28.

[23] Yan, L. K., and H. Yin, "Droidscope: Seamlessly Reconstructing the Os and Dalvik Semantic Views for Dynamic Android Malware Analysis", 2012, pp. 569-584.

[24] Zhou, Y., and X. Jiang, "Dissecting Android Malware: Characterization and Evolution", IEEE, 2012, pp. 95-109.

[25] Burguera, I., U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: Behavior-Based Malware Detection System for Android", ACM, 2011, pp. 15-26.

[26] Felt, A. P., E. Chin, S. Hanna, D. Song, and D. Wagner, "Android Permissions Demystified", ACM, 2011, pp. 627-638.

[27] Glisson, W. B., T. Storer, G. Mayall, I. Moug, and G. Grispos, "Electronic Retention: What Does Your Mobile Phone Reveal About You?", International Journal of Information Security, 10(6), 2011, pp. 337-349.

[28] Peffers, K., T. Tuunanen, M. Rothenberger, and S. Chatterjee, "A Design Science Research Methodology for Information Systems Research", J. Manage. Inf. Syst., 24(3), 2007, pp. 45-77.

[29] Bellard, F., "Qemu, a Fast and Portable Dynamic Translator", 2005, pp. 41-46.

[30] Shadish, W. R., T. D. Cook, and D. T. Campbell, Experimental and Quasi-Experimental Designs for Generalized Causal Inference, Wadsworth Publishing, 2001.