Automated Identification of Installed Malicious Android Applications

By

Mark Guido, Justin Grover, Jared Ondricek, Dave Wilburn, Drew Hunt and Thanh Nguyen

From the proceedings of

The Digital Forensic Research Conference

DFRWS 2013 USA

Monterey, CA (Aug 4th - 7th)

Digital Investigation 10 (2013) S96–S104

Contents lists available at SciVerse ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin

Automated identification of installed malicious Android applications

Mark Guido*, Jared Ondricek, Justin Grover, David Wilburn, Thanh Nguyen, Andrew Hunt

The MITRE Corporation, 7515 Colshire Drive, Mclean, VA 22102, USA

Keywords:
Android
Mobile forensics
Periodic
Enterprise
Monitoring

A B S T R A C T

Increasingly, Android smartphones are becoming more pervasive within the government and industry, despite the limited ways to detect malicious applications installed to these phones' operating systems. Although enterprise security mechanisms are being developed for use on Android devices, these methods cannot detect previously unknown malicious applications. As more sensitive enterprise information becomes available and accessible on these smartphones, the risk of data loss inherently increases. A malicious application's actions could potentially leave sensitive data exposed with little recourse. Without an effective corporate monitoring solution in place for these mobile devices, organizations will continue to lack the ability to determine when a compromise has occurred. This paper presents research that applies traditional digital forensic techniques to remotely monitor and audit Android smartphones. The smartphone sends changed file system data to a remote server, allowing for expensive forensic processing and the offline application of traditional tools and techniques rarely applied to the mobile environment. The research aims at ascertaining new ways of identifying malicious Android applications and ultimately attempts to improve the state of enterprise smartphone monitoring. An on-phone client, server, database, and analysis framework was developed and tested using real mobile malware. The results are promising that the developed detection techniques identify changes to important system partitions; recognize file system changes, including file deletions; and find persistence and triggering mechanisms in newly installed applications. It is believed that these detection techniques should be performed by enterprises to identify malicious applications affecting their phone infrastructure.

1. Introduction

Android malware is increasingly becoming a problem for both enterprise and individual users alike. The number of malicious applications targeting the Android operating system is dramatically increasing. Kaspersky Labs reported that the total number of Android malware samples tripled compared to the previous quarters during the first six months of 2012.[1]

To combat Android malware, a current industry approach is to install a mobile virus scanner on a phone, much like an administrator would do on an enterprise laptop or desktop. These virus scanners typically run as applications inside the Dalvik virtual machine and compare newly installed applications against a known repository of malware signatures. This "blacklisting" technique has known weaknesses that can be exploited by malware distributors (Vidas et al, 2011b). A zero-day malicious application could surreptitiously escalate privileges by modifying critical system files and altering the phone's behavior, thus rendering virus-scanning engines blind to attacks.

Because smartphones are mobile and often operate outside the confines of the enterprise, network level

* Corresponding author.
E-mail address: mguido@mitre.org (M. Guido).

[1] Y. Namestnikov. (2012, Aug. 8). "IT Threat Evolution: Q2 2012," *SecureList*. Kaspersky Lab ZAO. Available: http://www.securelist.com/en/analysis/204792239/IT_Threat_Evolution_Q2_2012.

1742-2876/$ – see front matter
http://dx.doi.org/10.1016/j.diin.2013.06.011

monitoring approaches are typically not effective. Therefore, it is necessary to have robust methods of monitoring the devices themselves for potentially malicious activities. Monitoring the usage pattern and stored data of an enterprise smartphone, whether performed by security personnel or malicious actors, can reveal unique information about users and the organizations they work for.

1.1. Project background

This research applies traditional forensic techniques to monitor and audit Android smartphones. The initial focus was on the enterprise use case, described in Section 1.2, where phone users inadvertently or intentionally install malicious applications. The focus of the research was to ascertain new ways of identifying malicious Android applications when they are installed or as they deliver their malicious payloads. Android phones are an appropriate target for this research because they are increasingly being selected for enterprise deployments.

1.2. Enterprise use case

The use case selected for this effort was detecting the unintentional or deliberate installation of malicious applications on Android smartphones associated with an enterprise; these smartphones may have been distributed to the enterprise user population and may have access to resources on the enterprise network. Enterprise users are assumed to have no privileged access on these phones, and part of this research is to keep enterprise users from obtaining privileged access. The malicious applications may lower the security posture of both the smartphone and the enterprise network to which it connects.

2. Related work

While the research incorporates several commonly used forensic tools and techniques, the methodology of pairing these tools and techniques into one system that would operate on the enterprise is unique. The computer forensics field is trending primarily toward live forensic analysis, as seen in traditional tools and systems such as Encase Enterprise and AccessData Enterprise. This research also falls under this category and is believed to be the first system of its kind for smartphones. Mobile forensics, considered "live" because of its reliance on the target smartphone's running kernel, typically uses commercially available mobile phone kits that take a one-time logical or physical image of a target phone (Lessard and Kessler, 2010). If these kits do not support certain target phones, specialized, albeit inconsistent, methods are used. Vidas, Zhang, and Christin (Vidas et al, 2011a) discussed these inconsistencies and provided a generalized method for forensic acquisition on Android phones where investigators have no prior knowledge of their contents. The research team drew upon these findings during the system design process.

Utilizing a running kernel to perform live forensic acquisitions in the enterprise can be problematic. It is theoretically possible for a malicious user or a malicious application to modify the running kernel to hide certain actions and skew the system's measurements. Weinstein (Weinstein, 2012) proposed a method of trusting the kernel, where achieving a practical amount of trust in a system's measurements would be possible.

The methodology used to send small updates of data from the phone over-the-air (OTA) is considered fairly novel, primarily because the use case was unique (Section 1.2). The system described herein was implemented for the enterprise and does not take into account evidentiary issues that may restrict some of the techniques used during a forensic acquisition. Teleporter, a previous capability that documented a technique for obtaining hard-drive images over limited bandwidth connections, was identified (Watkins et al., 2009). The research team analyzed their methods and incorporated a similar hashing technique, but the overall system design differed significantly from Teleporter.

The market for mobile forensic tools and techniques has grown as more mobile devices with capabilities rivaling laptops and desktops have flooded the market. Mobile operating systems now support and use file systems that traditionally have been found in laptops and desktops. This fact is significant that it allows the system to incorporate offline traditional forensic techniques when analyzing smartphones. Fairbanks (Fairbanks, 2012) researched the fourth extended file system (ext4) for digital forensics and incorporated support of the ext4 file system into The Sleuth Kit (TSK), a popular open-source forensic tool set that is easily incorporated into analytical systems.[2] The research team implemented Fairbanks' work and used it to identify added, modified, and deleted files in the ext4 file system found on many recent Android phones.

Another technique incorporated into the system was using the forensic tool `fiwalk` to produce Digital Forensics Extensible Markup Language (DFXML) (Garfinkel, 2011). This use of DFXML allowed system processes to pass data into the detector architecture and analyze it in a standard way.

The Android Malware Genome Project (AMGP) was another related work effort that helped the research. In addition to being a source for the malware we used for experimentation, the classification work (Zhou and Jiang, 2012) on over 1200 malware samples influenced the system's development direction. These classifications inspired a detector that looked for malware that, when installed and executed, would establish persistence on a phone to survive a reboot. The research team theorized that malware could alternatively use a modified system.img, boot.img, or bootloader to install persistence mechanisms. One research effort that used a modified boot.img to exploit Universal Serial Bus (USB) connectivity between a USB client and a USB host (Wang and Stavrou, 2010) was identified and lent credence to the theory.

3. Periodic mobile forensics

The system performs periodic scans of an Android phone's block devices, identifying changes to specific bit

[2] B. Carrier *The Sleuth Kit* Available: http://www.sleuthkit.org/.

sequences of data. The offset locations of the changed bit sequences are stored in a local database on the phone. The system sends copies of the changed bit sequences to a remote enterprise database via Wi-Fi for storage, and references those bit sequences by collection time. Specialized tools allow authorized users to reconstruct bit for bit copies (images) of each block device at a given collection time (henceforth referred to as "snapshot"). It is conceivable that some bit sequences may change many times between snapshots – the system will only reconstruct images based on the snapshot times.

As a starting point in the enterprise context, each unique block device on the population of enterprise phones was forensically imaged prior to their use, establishing a baseline image. Any reconstructed snapshot images can then be compared against the baseline image to identify malicious artifacts. A framework was developed to run a set of forensic tools and techniques on the reconstructed snapshot images. Results are again stored in the server database and can be used to set up further forensic analysis. Any audit output generated from these automated processes can be sent to an organization's central auditing system such as Splunk or ArcSight.[3,4]

3.1. System components

The system is composed of a service named Tractor Beam, which runs on the smartphone; a central server available via a Wi-Fi network; a framework for executing a series of automated forensic processes called the Analysis Framework; and a remote enterprise database for storing the phone's changed bit sequences and its derived event data.

3.1.1. Tractor Beam

Tractor Beam is a custom daemon with a native component that runs on an Android phone. The daemon is started by init, which allows Tractor Beam to have the proper privileges to read the block devices. The daemon and native component are added to `/system/bin` and `init.rc` was modified to start Tractor Beam at startup. Tractor Beam identifies the offset of changed bit sequences (the bit sequence size is configurable) on each block device since its last runtime by comparing a SHA256 hash to a previous measurement stored in a local SQLite database unavailable to the unprivileged enterprise phone user.[5] If bit sequences at the specified offset have a different hash value compared to the last time it was scanned, the offset is marked as changed in the local database. When the smartphone is able to communicate via Wi-Fi with the listening service on the Central Server, Tractor Beam initiates secure, authenticated communications, reading and copying each changed bit sequence from the phone block devices over the network for storage and analysis. Tractor Beam's local storage requirements are small because no block device bit sequences are ever stored locally, only their SHA256 hashes. SHA256 hashing can be expensive to process for every changed bit sequence, but these operations can be executed in the background on the phone or during execution of a partial WakeLock.

3.1.2. Central server

The Central Server listens on a negotiable port via Hypertext Transport Protocol Secure (HTTPS). All phone Tractor Beam clients can perform server authentication by examining the server's certificate. The Central Server's service listens for incoming bit sequences and offset values, and stores them in the normalized database. This "sending" process will take longer to complete if there are more changed bit sequences on the phone since the last send. During laboratory testing OTA, an average of 200 megabytes (MB) worth of block device changes were read from the phone file systems and sent to the Central Server in roughly 2 min. The Central Server also reconstructs images of the target phone's block devices on a file system scratch space by using both the baseline image and the incoming bit sequences and offset locations stored in the database, as seen in Fig. 1.

3.1.3. Analysis framework

The Analysis Framework provides a dynamic capability to run a series of automated forensic processes, called detectors or loggers, on the reconstructed images. It also helps manage the file system scratch space by reusing images that may have been previously reconstructed.

Detectors and loggers are run against the reconstructed images in the order in which they are imported into the framework. Techniques that identify malicious activity are typically developed into detectors, while loggers simply record events that may or may not be considered malicious, such as modified, accessed, created, and entry modified (MACE) time changes. Event data may be further processed by a later detector, or it could be executed solely to provide data for future human analysis. The framework was developed with modularity in mind so that loggers and detectors can be added or dropped at any time, and they will be dynamically run within the framework. Some detectors and loggers relevant to this research are described below:

Detect system changes (*detect_systemdelta.py*): This detector identifies any changes to the phone's system.img. System.img changes may indicate new binaries or applications being installed on the phone. These changes could indicate that the malicious applications are establishing persistence.

Detect boot changes (*detect_bootdelta.py*): This detector identifies any changes to the phone's boot.img. It then performs further analysis to determine whether the change occurred in the phone's kernel or ramdisk, which contains the phone's root file system and core system files. Malicious applications can modify the kernel in a targeted attack to make the kernel behave differently as in (Wang and Stavrou, 2010). Also, core startup files like `init.rc` can be modified, and a malicious application could add executable files to the root file system that could survive a reboot.

[3] *Splunk*. Splunk, Inc. Available: http://www.splunk.com/.

[4] "HP ArcSight Security Intelligence," *HP Enterprise Security*. Hewlett–Packard Development Company, L.P. Available: http://www.hpenterprisesecurity.com/products/hp-arcsight-security-intelligence/.

[5] SQLite, SQLite Consortium. Available: http://www.sqlite.org.

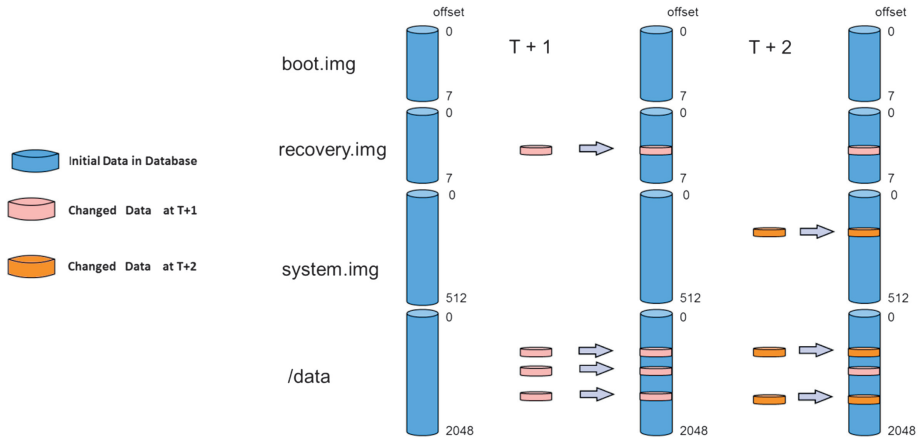

Fig. 1. Reconstruction of images based on changed data offsets.

Detect bootloader changes (*detect_bootloaderdelta.py*): This detector identifies any changes to the phone's bootloader. In the enterprise context, observed unplanned changes to the bootloader should rarely happen; therefore, this kind of activity is almost always considered malicious.

Detect recovery changes (*detect_recoverydelta.py*): This detector identifies changes to the phone's recovery.img, which holds the running kernel and root filesystem when the phone is booted into recovery mode. Changes to the recovery.img may not indicate truly malicious behavior—for example, they may just be due to the phone user installing a custom recovery.img to enable backup and restoring functionality—an enterprise should not allow such changes on its phones because this supplies the enterprise phone population or even malicious attackers with elevated privileges to areas of the phone storage that are typically protected while the phone is running in normal mode.

Log file MACE (*log_fileMACE.py*): This logger executes the forensic tool `fiwalk` to output DFXML, which is processed to identify any files that were added or changed on the file systems.[6] The resulting log output is time stamped based upon the MACE times and stored in the database.

Log FLS parse (*log_flsParse.py*): This logger executes the forensic tool `fls` from the Sleuth Kit and then processes the results to output any observed deleted files.[7] The resulting log output is time stamped based upon the MACE times and is stored in the database. If any seemingly legitimate application contains exploit files in its assets directory and subsequently writes them to the file system, executes them, and deletes them, this logger will log the existence of any remnants of the deleted exploit files on the file system.

Detect inspect APK (*detect_inspectapk.py*): This detector observes Android packages (APKs) that have been installed since the last snapshot by first running fiwalk to identify the APK file's installation directory and inode, then using icat to extract the APK file.[8] Once the APK file has been extracted, the resulting AndroidManifest.xml file is parsed and then inspected for the presence of a BOOT_COMPLETED event registration. If BOOT_COMPLETED is not found, the detector will simply log the installation of a new application. Zhou and Jiang (Zhou and Jiang, 2012) found that the BOOT_COMPLETED event registration was used in over 83% of the AMGP sample set, and the assumption was that this event registration would occur less frequently in legitimate applications and would be a good indicator for identifying malicious applications.

3.1.4. Enterprise database

The Central Server uses a relational database to store both the collected bytes from the phones and any detector output or logger events. The database is designed to eliminate the duplication of identical bit sequences of data. This drastically cuts down the amount of storage space needed for each phone in the enterprise, as similar model and vendor phones can reference one baseline image. The AUDIT table records the output from both the detectors and the loggers. It is expected that the detectors and loggers could process any data stored in the database or the reconstructed snapshot images for their forensic analysis.

4. Malware experimentation

The Tractor Beam architecture and each developed detector were tested using both real malware and legitimate applications. Section 4.1 describes the sources of the mobile malware. Section 4.2 describes three rounds of experimentation and includes a more in-depth analysis of several interesting samples.

[6] "Fiwalk," *Digital Evaluation and Exploitation*. Naval Postgraduate School. Available: https://domex.nps.edu/deep/Fiwalk.html.

[7] B. Carrier. "FLS," *The Sleuth Kit*. Available: http://www.sleuthkit.org/sleuthkit/man/fls.html.

[8] B. Carrier. "Icat," *The Sleuth Kit*. Available: http://www.sleuthkit.org/sleuthkit/man/icat.html.

4.1. Malware source information

Started by Yajin Zhou and Xuxian Jiang of the North Carolina State University's Department of Computer Science, the AMGP aims at characterizing collected Android malware.[9] AMGP made its dataset of 1261 malware samples available for this research. Samples from this dataset were used for the bulk of the malware experimentation.

A total of 31 Android malware samples were used during three rounds of experimentation. Malware from AMGP accounted for 29 of the samples; two were from other sources. These two samples were used during testing because they were known to work on the model and version of the phone used.

4.2. Tests performed

4.2.1. Round 1

The first round of testing used 20 malware samples, all chosen randomly from a down-selected malware sample set. This initial round was used to establish the correct functionality of Tractor Beam and its components and the ability of the custom-developed detectors to identify artifacts indicative of malicious activity.

A Google Nexus S phone running Android Gingerbread version 2.3.1 was used during the testing, primarily because this version and phone were deemed the most vulnerable modern Android version in use at test time. The testing procedure began by resetting the phone to a known-good state using the Team Win Recovery Project recovery image. Due to constraints in the testing environment and the concern of malware contamination of enterprise network resources, the Tractor Beam service and all communication to the Central Server was modified for the testing to use the Android Debug Bridge (ADB) over USB. The phone was placed in a Faraday bag (which further reduced wireless capabilities). All subsequent tests included the restoration of the phone to an initial known good state. The malware was also installed over ADB and launched via a combination of ADB commands and the use of Android's Activity Manager. Tractor Beam took a measurement after the malware's installation, execution, and phone reboot. The Analysis Framework then executed all seven plug-ins described in Section 3.1.3. Testers reviewed the results to determine if the loggers and detectors successfully identified whether the malware executed its full malicious payload or if the loggers and detectors could verify that the malware was unsuccessful in exploiting the phone. This became the success criteria for the test.

It was unknown as to the full malicious payload of the malware or if the malware required triggers before fully executing its malicious payload. The malware was provided with some amount of functionality, including root privileges and a reboot to generate a BOOT_COMPLETED event notification, because the research team speculated that these may be possible triggers for some of the malware. The research team was unable to supply all possible triggers as there was no internet connectivity available in the lab environment. Results of the Round 1 tests are presented in Table 1.

Table 1
Test results from round 1.

Sample	Logged		Detected[a]		Detection result
	Installed	Files dropped	BOOT_ COMPLETED	system.img change	
1	x	x	x		Success
2	x	x	x		Success
3	x	x	x		Success
4	x		x		Success
5	x	x			Fail
6	x	x	x		Success
7	x	x	x		Success
8	x	x	x		Success
9	x	x	x		Success
10	x		x		Success
11	x	x	x		Success
12	x	x	x		Success
13	x	x		x	Success
14	x	x	x		Success
15	x	x	x		Success
16	x		x		Success
17	x	x	x		Success
18	x	x	x		Success
19	x	x	x		Success
20	x	x	x		Success

[a] Other detectors were run in addition to the two listed in Table 1; however, they were not triggered by any of the malware samples in Round 1.

All malware was observed installing on the phone. Many of the malware samples were logged writing asset files that were suspected of containing exploits to the persistent phone storage. The majority of the malware tested in Round 1 registered for the BOOT_COMPLETED event notification in their Android Manifest files. One sample modified the system.img. No samples successfully made modifications to the boot.img, bootloader, or recovery.img.

Many malware test samples from Round 1 were analyzed more to determine Tractor Beam's successful detection. Two are described below. Descriptions of actions taken by the malware and how Tractor Beam measured them are included.

4.2.1.1. Analysis of sample 5. Sample 5's malware consisted of an application that contained a version of the DroidDream malware which attempted to use an exploit named `rageagainstthecage (RATC)`.[10] The detectors did not alert on any malicious activity during the test; however, upon further inspection, the research team found that the `RATC` executable was dropped into a directory named `files`, along with several other scripts that contained instructions to remount and alter several of the phone's read-only partitions. Since no changes to the `/system` partition were detected, it was concluded that the malware did not

[9] Y. Zhou and X. Jiang. *Android Malware Genome Project*. Available: http://www.malgenomeproject.org/.

[10] DroidDream *The Lookout Blog* Availability: https://blog.lookout.com/blog/2011/03/01/security-alert-malware-found-in-official-android-market-droiddream/.

fully execute. In trying to determine where the malware may have failed or where Tractor Beam may have unsuccessfully verified proper phone operation, the researchers manually executed the exploit on the phone to test whether the phone was vulnerable. `RATC` forked many processes to increase the RLIMIT_NPROC value, which is the maximum number of simultaneous connections, and then reset the ADB daemon (`adbd`) as it was designed to do, but `adbd` was unable to obtain setuid privileges and failed to return a root shell. During the execution of this malware, the ADB connection was also not reset, so it was concluded that the `RATC` exploit was never called. An interesting side effect from this test came during the manual execution of the `RATC` exploit. Due to the large number of processes spawned as a result of running `RATC`, other applications, including Tractor Beam, were unable to create new processes until the phone was rebooted and the RLIMIT_NPROC value was reset. This caused a temporary denial of service to Tractor Beam because it was using a forked process for some functionality. Version 2.3.1 of the Android operating system featured an `adbd` impervious to this vulnerability, but it did not fully make the entire system completely invulnerable. Future Tractor Beam functionality should be modified to be invulnerable to an exploit of this type by eliminating the need for spawning a new process.

4.2.1.2. Analysis of sample 13. Sample 13's malware consisted of the Gingerbreak 1.3 application, which contained a well-known Gingerbreak binary exploit that achieves privilege escalation on a phone by exploiting the `vold` service on vulnerable Gingerbread Android versions.[11] This sample was not categorized as part of the AMGP, as it was included in the testing set to demonstrate Tractor Beam's capability to successfully detect changes on the phone's `/system` partition, a portion of the phone that is normally "read only" to unprivileged users. The Gingerbreak binary exploit is a common exploit for Gingerbread versions of Android and was observed in many samples in the AMGP, so including a non-weaponized version of the exploit that researchers could better control was not considered adverse in the testing. The Gingerbreak 1.3 application is proven to drop new files onto the `/system` partition that are necessary to gain permanent administrative privileges. In some environments (e.g., on personally owned devices, experimental use cases, or other environments where legitimate customers wish to gain root access to their phones), the Gingerbreak 1.3 application is not considered malware, as it is merely a means of obtaining root permissions to perform legitimate (and legal) actions; however, in an enterprise setting, any process that provides escalated privileges to non-administrative users may violate policy and may be potentially damaging to an enterprise's overall security. Tractor Beam was able to observe the Gingerbreak 1.3 application installing and was able to log assets files written to persistent storage that were identified as the Gingerbreak exploit. Tractor Beam was also able to observe changes to the `/system` partition.

After the execution of the Gingerbreak binary exploit on the phone, a secondary APK named Superuser.apk was installed into the `/system/app` directory. Both of these files (eu.chainfire.gingerbreak-1.apk and Superuser.apk) were logged as application installations by the Tractor Beam detectors and loggers. The subsequent modification of the `/system` partition, which resulted from installing Superuser.apk, was also successfully detected by the system.

4.2.2. Round 2

The second round of testing consisted of 10 malware test samples and 90 assumed legitimate applications. Zhou and Jiang (Zhou and Jiang, 2012) concluded that 83 percent of the malware samples in the AMGP exhibited the BOOT_COMPLETED event registration in their AndroidManifest.xml files. The primary objective of this round was to determine if the BOOT_COMPLETED event registration would be considered a good feature to use in a future classifier; the team wanted to determine at what rate the BOOT_COMPLETED event registration is performed in legitimate applications. The malware, as in Round 1, was selected randomly from the team's down-selected pool of samples. The legitimate applications were chosen from Google Play's most popular Android applications, with some ranked applications removed from the testing because the applications were already resident on the phone. The number of people that ranked these applications most popular was considered a decent gauge to conclude that these applications do not contain malicious payloads of any kind.

The testing procedure used in Round 2 was nearly identical to the one used in Round 1 (Section 4.2.1), except that the Round 2 procedure was broken down into ten tests, with each test consisting of nine legitimate applications being installed and started in conjunction with one malware sample. This minor procedure change allowed the testing team to work more efficiently by resetting the phone less frequently and also mimicked the more likely real world behavior of encountering more phone activity between Tractor Beam collection runs.

The other detectors and loggers were also executed during this round, and each malware was observed installing and writing assets to persistent storage. From the output of the other detectors besides detect_inspectapk.py, it was determined that many of the malware samples did not execute their entire payload. The detect_inspectapk.py detector's effectiveness at identifying malware is described in the Tables 2 and 3.

The statistics above confirm the findings from Zhou and Jiang (Zhou and Jiang, 2012) regarding the prevalence of the BOOT_COMPLETED event registration in malware. In fact, in all of the testing, the prevalence of the

Table 2
Round 2 identification by the detect_inspectapk.py detector.

		Predicted class	
		Malware	Non-malware
Actual class	Malware	10	0
	Non-malware	29	61

[11] (2011, April 21). "Yummy Yummy, Gingerbreak!" *C-Skills*. Available: http://c-skills.blogspot.com/2011/04/yummy-yummy-gingerbreak.html.

Table 3
Round 2 detect_inspectapk.py detector rates.

Accuracy	71.0%
Error	29.0%
Precision	25.6%
Recall	100.0%

BOOT_COMPLETED event registration was observed in the chosen malware at a rate of 90%. While the 100% recall rate for this test shows that all the actual malware samples were correctly identified as malware, the 25.6% precision rate reveals that many benign applications were incorrectly marked as malicious. The conclusion from these statistics is that the detect_inspectapk.py detector is an ineffective means of classifying malware by itself, primarily because of the high false positive rate.

Round 2 testing revealed a minor flaw in the Tractor Beam detection mechanisms; three benign applications were installed in the `/mnt/secure/asec` directory on the Secure Digital (SD) card without Tractor Beam's detection. Android documentation notes that the BOOT_COMPLETED event registration is not broadcast to components on the SD card, so any applications installed on the SD card cannot effectively use this intent as a persistence mechanism.[12]

Round 2 testing also revealed that while 10 APKs were installed during each round, only one deleted APK could be recovered from the `/data/local/tmp` directory. This directory typically stores APKs during application installations from markets or during side loading, after which these APKs are deleted from the directory. The file system was allocating and deallocating these blocks so quickly that only the last APK installed and deleted in this persistent storage location could be recovered. It was determined that this persistent storage location should not be used as a definitive source for the detection of APK installations.

4.2.3. Round 3

For round 3, the source code for an assumed benign open source Android game APK was located and downloaded from the Internet. The research team modified the source code, adding functionality to demonstrate the exploits documented in (Zhou and Jiang, 2012). The Android game application (henceforth referred to as Android game malware), exploits USB connectivity between itself and a USB host (presumably a laptop or desktop). This malware was packaged to include a weaponized version of the Gingerbreak exploit that was executed from certain states in the game. Once the exploit was successful, the phone restarts using new malicious kernel functionality that modifies the behavior of the USB interface. This new kernel functionality is also the means by which the malware establishes persistence on the phone, as subsequent reboots would start up the malicious kernel. For this round, the Android game malware was installed on the phone, the game was played to achieve the game states to execute the exploit, and the phone was observed rebooting. After a full reboot, a Tractor Beam measurement was taken and analyzed.

4.2.3.1. Analysis of Android game malware. The lifecycle of the Android game malware is described below:

1. The APK installs to the `/mnt/secure/asec` directory on the phone's SD card. For the target phone, this was an internal SD card.

2. After the phone's screen is locked during gameplay, the Android game malware launches a background service.

3. The background service spawns a process that roots the phone using a weaponized version of the Gingerbreak application. This application exploits the `vold` service and then installs Superuser.apk to the `/system/app` directory using its new elevated privileges. Superuser.apk also writes the `su` binary to the `/system/bin/` directory.

4. Once Gingerbreak executes successfully, the background service calls an external script that uses the `dd` binary to write a new boot.img to the phone.

5. The background service then restarts the phone. The phone restarts using the malicious kernel contained in the newly installed boot.img.

Tractor Beam did not detect the installation of this malware since it was installed in the `/mnt/secure/asec` directory. No asset files were observed being written to the phone's persistent storage, and subsequent analysis indicated that all of the `asset` files were parsed directly into memory and executed. Modifications to both the boot.img and the system.img were observed by Tractor Beam detectors. The Superuser.apk file was observed being installed to the `/system/app` directory, which triggered another instance of the detect_inspectapk.py detector to be run on this Superuser.apk installation, which observed its writing the `su` binary to the `/system/bin/` directory.

5. Results

The test results showed promise in the research's overall approach to logging and detecting events that occurred on an Android smartphone. The mechanisms were able to effectively detect changes to protected areas of a phone and were able to reconstruct and analyze many of the added, deleted, and modified artifacts from the phone's file systems. With these results came several significant findings:

- The Tractor Beam detector that measured for the existence of the BOOT_COMPLETED event registration was a solid starting point for malware detection, but it was not effective by itself. All but three malware samples registered to receive the BOOT_COMPLETED event notification within their AndroidManifest.xml file, which was a significantly more common occurrence than in the tested benign apps.

[12] "Applications That Should NOT Install on External Storage," *Developers*. Google. Available: http://developer.android.com/guide/topics/data/install-location.html#ShouldNot.

- No malware was observed modifying the `/bootloader`, or recovery.img and only a single malware altered the target phone's boot.img. An exploit that does this would require the malware to be targeted towards a particular phone version and hardware specification.
- Tractor Beam lacked visibility into apps installed to a phone's SD card. The `/mnt/secure/asec` directory contained encrypted versions of the installed APKs. Tractor Beam processes only monitored the `/data/app` and `/system/app` directories for application installations, and therefore these encrypted application installations were missed. Further analysis showed that applications installed in `/mnt/secure/asec` would not receive the BOOT_COMPLETED event notification that was measured.
- Not all deleted applications were detected. The `/data/local/tmp` directory is used for temporary storage during market installation and side loading. During tests where more than one application was installed between Tractor Beam measurements, the team observed that only the last installed application was able to be extracted from the file system. This proved that this directory should not be relied upon as a definitive source to identify application installations.

6. Limitations and future work

6.1. Malware limitations

There are different types of available Android devices, and among them many different providers, manufacturers, and operating system (OS) versions. Applications and processes that function properly on one Android phone may not work on another. During the malware experimentation, most of the malware did not execute properly for several possible reasons:

1. The malware may only work on a limited number of environments (e.g., OS versions, phone models, manufacturers, etc.) or may need certain phone capabilities.
2. The malware may be dependent on certain phone settings having been set (e.g., wireless connectivity, Internet connectivity, language settings, etc.) or may be monitoring for certain trigger events to be received.
3. The malware may need to receive or access command and control communications to receive commands or 2nd stage payloads.
4. The malware was altered or damaged or may contain code that does not properly execute the malicious payload.

An example of these issues was observed in Round 1, Test 2. Analysis showed that the malware functions that could execute its malicious payload were never called during the overall code execution.

6.2. Tractor Beam limitations

Several factors can potentially limit the effectiveness of Tractor Beam's processes and detectors.

The BOOT_COMPLETED event registration is not effective alone. This observed persistence mechanism suffered from a false positive problem, where benign applications used this method to achieve their own persistence. It was concluded that this feature may be suitable for use in combination with other features inside of some classifier.

Future detectors that rely on finding declared intents inside an APK (other than BOOT_COMPLETED) may be challenged by intents that can be registered within an application's source code on-the-fly. The BOOT_COMPLETED intent was an exception to this rule, as it must be present in the AndroidManifest.xml file.

Applications installed to the SD card (specifically to `/mnt/secure/asec`) present visibility problems for Tractor Beam processes. APKs installed there are encrypted as `.asec` files, which are not able to be inspected by Tractor Beam's current set of detection methods. The unencrypted mount points for these applications do not exist when the file systems are reconstructed offline.

6.3. Future work

One important future enhancement is identifying new APKs installed to the SD card (specifically to `/mnt/secure/asec`). Development efforts may include capturing a phone's volatile memory to decrypt the `.asec` files later during image reconstruction, or exploring dynamically identifying unencrypted mount points when they exist and extracting their contents for later analysis.

Other application installation locations including `/vendor/app/data/app-private`, and `/system/framework`, need to be researched and potentially implemented to ensure that all application installations are logged.[13] Alternatively, a method of identifying applications installed anywhere on the persistent storage may be a more elegant solution.

Detectors that identify additional features to be used in conjunction with the BOOT_COMPLETED event registration feature will also be investigated.

7. Conclusion

As the Android platform continues to increase in popularity, so will the stakes and risks posed to enterprises by targeted malware. This research aims to offer new and alternative methods of detecting malicious Android applications as they are installed or as they deliver their malicious payloads. Tractor Beam and its components recognized file system changes to system partitions that shouldn't change often; found persistence mechanisms in the malware that would allow malware to restart after a reboot; and, found malicious files dropped to the file system in newly installed applications. Results thus far have been promising, limitations have been identified, and future work is planned to increase the capabilities and effectiveness of the current detection mechanisms.

[13] "Disable Export to APK from apps like APKInstaller," *Android Security Discussions*. Google. Available: https://groups.google.com/forum/?pli=1#!topic/android-security-discuss/UMcuWitSENo.

References

Fairbanks K. An analysis of EXT4 for digital forensics. Digital Investigation Aug, 2012;9(Suppl.):S118–30.
Garfinkel S. Digital forensics XML and the DFXML toolset. Arlington, VA: Naval Postgraduate School; Sept, 2011.
Lessard J, Kessler G. Android forensics: simplifying cell phone examinations. Small Scale Digital Device Forensics Journal 2010;4(1):1–12.
Vidas T, Zhang C, Christin N. Towards a general collection methodology for Android devices. Digital Investigation 2011a;8(Suppl.):S14–24.
Vidas T, Votipka D, Christin N. All your droid are belong to us: a survey of current Android attacks. In: Proceedings of the 5th USENIX conference on offensive technologies. USENIX Ass.; 2011b.
Wang Z, Stavrou A. Exploiting smart-phone USB connectivity for fun and profit. In: Annual computer security applications conference, Austin, TX Dec, 2010.
Watkins K, McWhorte M, Long J, Hill B. Teleporter: an analytically and forensically sound duplicate transfer system. Digital Investigation Sept, 2009;6(Suppl.):S43–7.
Weinstein D. A security hygienic smart charger for mobile devices. Mclean, VA: The MITRE Corp.; June 2012.
Zhou Y, Jiang X. Dissecting Android malware: characterization and evolution. In: IEEE Symposium on security and privacy, San Francisco, CA May 2012.

Further reading

Carrier B. File system forensic analysis. Boston, MA: Addison-Wesley; 2005.
Ferrill P, Welsh T, editors. Pro Android Python with SL4A. New York, NY: Apress; 2011.
Garfinkel S, Nelson A, Young J. A general strategy for differential forensic analysis. Digital Investigation Aug, 2012;9:S50–9.
Hoog A, Ward A, Scherer H, editors. Android forensics: investigation, analysis and mobile security for google Android. Waltham, MA: Syngress; 2011.
Ratabouil S, Hyames D, editors. Android NDK beginner's guide. Birmingham B3 2PB, UK: Packt Publishing Ltd; 2012.
Tan P, Steinbach M, Kumar V. "Classification: alternative techniques", in introduction to data mining. Boston, MA: Addison-Wesley; 2005. p. 207–326.