SHORT MESSAGE SERVICE (SMS) COMMAND AND CONTROL (C2) AWARENESS IN ANDROID-BASED SMARTPHONES USING KERNEL-LEVEL AUDITING

THESIS

Robert J. Olipane, Captain, USAF

AFIT/GCO/ENG/12-21

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, the Department of Defense, or the United States Government.

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

AFIT/GCO/ENG/12-21

SHORT MESSAGE SERVICE (SMS) COMMAND AND CONTROL (C2) AWARENESS IN ANDROID-BASED SMARTPHONES USING KERNEL-LEVEL AUDITING

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

Robert J. Olipane, B.S.C.S.

Captain, USAF

June 2012

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

AFIT/GCO/ENG/12-21

SHORT MESSAGE SERVICE (SMS) COMMAND AND CONTROL (C2) AWARENESS IN ANDROID-BASED SMARTPHONES USING KERNEL-LEVEL AUDITING

Robert J. Olipane, B.S.C.S.
Captain, USAF

Approved:

// Signed //	22 May 2012
Robert F. Mills, PhD (Chairman)	Date
// Signed //	22 May 2012
Michael R. Grimaila, PhD, CISM, CISSP (Committee Member)	Date
// Signed //	22 May 2012
Barry E. Mullins, PhD (Committee Member)	Date

AFIT/GCO/ENG/12-21

ABSTRACT

This thesis addresses the emerging threat of botnets in the smartphone domain and focuses on the Android platform and botnets using short message service (SMS) as the command and control (C2) channel. With any botnet, C2 is the most important component contributing to its overall resilience, stealthiness, and effectiveness. This thesis develops a passive host-based approach for identifying covert SMS traffic and providing awareness to the user. Modifying the kernel and implementing this awareness mechanism is achieved by developing and inserting a loadable kernel module that logs all inbound SMS messages as they are sent from the baseband radio to the application processor. The design is successfully implemented on an HTC Nexus One Android smartphone and validated with tests using an Android SMS bot from the literature. The module successfully logs all messages including bot messages that are hidden from user applications. Suspicious messages are then identified by comparing the SMS application message list with the kernel log's list of events. This approach lays the groundwork for future host-based countermeasures for smartphone botnets and SMS-based botnets.

To my sons, because of you, I've already succeeded

ACKNOWLEDGMENTS

I would like to give my sincere thanks and appreciation to Dr. Robert Mills for enlightening me to the research process and providing me the freedom, empowerment, support and guidance to take this research effort to fruition.

To my friends and to my family, thank you all for your support and motivation helping me through this effort; time away from the research was just as important as the time spent doing the research in accomplishing the goal. Special thanks to Tim Wilson for always willing to be a sounding board and notable mentions for Matt Sievers, Ben Pacer, Loui Hashmi, and Eric Merrit; experiences are more memorable when shared with others. Thanks for sharing and letting me share along the journey.

Robert J. Olipane

TABLE OF CONTENTS

Page

ABSTRACT iv

DEDICATION v

ACKNOWLEDGMENTS vi

LIST OF FIGURES x

LIST OF TABLES xi

LIST OF ABBREVIATIONS xii

1 INTRODUCTION 1
1.1 Research Motivation 1
1.2 Problem Statement 1
1.3 Research Objectives 2
1.4 Approach 3
1.5 Assumptions and Limitations 3
1.6 Thesis Overview 5

2 LITERATURE REVIEW AND RELATED WORKS 6
2.1 Overview of the Botnet Phenomena 6
2.1.1 History and Terminology 6
2.1.2 Botnet Phases 7
2.1.2.1 Injection and Spreading 8
2.1.2.2 Command and Control Models 9
2.1.2.3 Malicious Applications 11
2.1.3 Botnet Defense Techniques 12
2.2 Smartphone Architecture and Short Message Service 12
2.2.1 Smartphone Architecture 13
2.2.2 Short Message Service (SMS) 14
2.2.2.1 Sending and Receiving SMS 14
2.2.2.2 SMS and the ATtention (AT) modem commands 15
2.3 Introduction to Google Android 17
2.3.1 Android Architecture 18
2.3.2 Android Security 18
2.3.3 Android Radio Interface Layer (RIL) 20
2.4 Smartphone Botnets, SMS Botnets, and Android Botnets 22

2.4.1 Smartphone Botnets . . . 23
2.4.2 Android Botnets in the Wild . . . 23
2.4.3 SMS Botnets . . . 24
2.5 Summary . . . 24

3 METHODOLOGY . . . 25
3.1 Introduction . . . 25
3.2 Transparent SMS Bot . . . 25
3.3 Design . . . 27
3.4 Implementation . . . 30
3.4.1 Target Device: HTC Nexus One . . . 31
3.4.2 Development Environment . . . 33
3.4.3 Deploying the Module . . . 35
3.4.3.1 Android Partition Layout . . . 35
3.4.3.2 Vendor-specific Security Mechanisms . . . 36
3.5 Test Environment and Experimental Design . . . 37
3.5.1 Experimental Design Overview . . . 38
3.5.2 Test 1: Functionality Test (Logging non-C2 Messages) . . . 39
3.5.3 Test 2: Functionality Test (Logging C2 messages) . . . 40
3.5.4 Test 3: Utility Test (Scenario 1 - Well timed C2) . . . 41
3.5.5 Test 4: Utility Test (Scenario 2 - Poorly timed C2) . . . 42
3.6 Summary . . . 42

4 RESULTS AND ANALYSIS . . . 44
4.1 Overview . . . 44
4.2 Test 1: Functionality Test (Logging non-C2 Messages) . . . 44
4.3 Test 2: Functionality Test (Logging C2 Messages) . . . 45
4.4 Test 3: Utility Test (Scenario 1 - Well timed C2) . . . 47
4.5 Test 4: Utility Test (Scenario 2 - Poorly Timed C2) . . . 48
4.6 Further Analysis . . . 49
4.6.1 Design Analysis . . . 49
4.6.2 Implementation Analysis . . . 50
4.6.3 Test Analysis . . . 50
4.7 Summary . . . 51

5 CONCLUSIONS AND RECOMMENDATIONS . . . 52
5.1 Overview . . . 52
5.2 Significance of Research . . . 52
5.3 Recommendations for Future Research . . . 53

APPENDIX A: LOADABLE KERNEL MODULE CODE . . . 56

APPENDIX B: DEVELOPMENT ENVIRONMENT . . . 59

APPENDIX C: TEST RESULTS DATA 66

BIBLIOGRAPHY 72

LIST OF FIGURES

FIGURE PAGE

2.1 Centralized C2 Botnet Structures [17] 10
2.2 Example of P2P C2 Botnet Structure [17] 11
2.3 Conceptual Design of Modern Smartphone Architecture adapted from [2, 22] . 13
2.4 SMS Delivery and Receipt at the Cellular Network Level based on [22, 34] . . 15
2.5 Sending and Receiving SMS at the Device Component Level adapted from [22, 34] 16
2.6 Android System Architecture [8] 17
2.7 Overview of Android's Telephony Stack and Radio Interface Layer [33] 21
3.1 Bot Location and Control Flow on Android's Telephony Stack [33, 34] 26
3.2 Security Module and Control Flow on Android's Telephony Stack 28
3.3 Security Module and Bot Locations with Control Flows on Android Telephony Stack 29
3.4 Instrumenting *ch_read* Using a Jprobe 31
3.5 HTC Nexus One with Technical Specifications 33
3.6 Anritsu MD8470 Signal Analyzer with Wireless Network Simulator Software . 38
3.7 Component Overview of Test Environment 38
4.1 Test 1: Terminal Display Showing Kernel Logging non-C2 Messages 45
4.2 Test 2: Terminal Display Showing Kernel Logging C2 Messages 46
4.3 Test 3: Side-by-Side Comparison of SMS App and Kernel Log 47
4.4 Test 4: Side-by-Side Comparison of SMS App and Kernel Log 48
B.1 Linux Kernel Configuration Menu 64

LIST OF TABLES

TABLE PAGE

2.1 Botnet Phases and Description 8

2.2 Sample SMS Commands (Extended AT Commands) 16

3.1 Virtual Machine Development System Specifications 34

3.2 Partition Layout for Most Android Smartphones 36

3.3 Summary of Experimental Tests 39

3.4 Test 1 - Logging non-C2 Messages 40

3.5 Test 2 - Logging C2 Messages 40

3.6 Test 3 - Well Timed C2 Sequence of Events 41

3.7 Test 4 - Poorly Timed C2 Sequence of Events 42

LIST OF ABBREVIATIONS

ABBREVIATION		PAGE
C2	Command and Control	2
SMS	Short Message Service	3
LKM	Loadable Kernel Module	3
GSM	Global System for Mobile Communications	4
SMD	Shared Memory Driver	4
OS	Operating System	5
IRC	Internet Relay Chat	6
P2P	Peer-to-Peer	8
DDoS	Distributed Denial of Service	11
IDS	Intrusion Detection System	12
PDA	Personal Digital Assistant	12
SOC	System on a Chip	13
OTA	Over-the-Air	14
WAP	Wireless Application Protocol	14
SMSC	SMS Center	14
AT	ATtention	15
API	Application Programming Interface	18
VM	Virtual Machine	18
ARM	Advanced RISC Machine	18
JNI	Java Native Interface	20
AOSP	Android Open Source Project	32

SHORT MESSAGE SERVICE (SMS) COMMAND AND CONTROL (C2) AWARENESS IN ANDROID-BASED SMARTPHONES USING KERNEL-LEVEL AUDITING

1 INTRODUCTION

1.1 Research Motivation

Over the years, smartphones have become more than just a staple in our everyday society; they are becoming the future of everyday computing. Smartphones have evolved from a mobile phone and personal digital assistant combined device, to a mobile computing environment hosting a local operating system, file system, full network connectivity and the ability to run third-party applications providing feature-rich capabilities. Now, at prices competitive with their dumb and feature-phone counterparts, they are no longer catered solely towards the business or information-technology professional, and have successfully crossed the threshold into the world of the average consumer.

As ubiquitous as smartphones have become and their evolution into mobile computing platforms, security of these devices becomes increasingly important. The ability to protect the sensitive data these systems contain and operate on is a risk all smartphone owners face. This research fits into the mobile security domain specifically focusing on the Android Operating System and the botnet phenomena.

1.2 Problem Statement

One of the most alluring aspects of a smartphone is its ability to run third-party applications which enhances the user's experience. Applications extend the basic functions of the device and provide capabilities including productivity for work and

finance to entertainment in streaming media, updating social networking sites or playing video games. Some applications have even been developed to manage home security systems, open garage doors, and remotely monitor and start personal vehicles.

The Android Market was developed as a central location for users to download applications directly to their phone. This market is open to any developer (possibly anonymous) who wants to submit an application for download with no filtering process. In general, Android application security relies heavily on the user to make decisions and accept security permissions upon installation with little or no granularity as to how these permissions will be used.

This scenario is prime for any cyber attacker because it provides a vector to distribute malicious code disguised as user-experience enhancing applications. With most average users viewing security as a hindrance on their experience and productivity as opposed to providing added benefit, attackers now have an ideal situation to target systems with access to sensitive information such as GPS location, financial information, e-mail traffic, text messaging traffic, and more. In 2011, an estimated 488 million smartphones shipped worldwide, surpassing client-PC figures, with projected increases in subsequent years [4]. With smartphones sales on the rise, an open environment for distributing maleware, and a device that is constantly connected, smartphone botnets represent a significant cyber threat. This threat of botnets spreading through the smartphone environment as they have in fixed networks and traditional desktop systems is inevitable. More research is required to study and develop measures for detection and mitigation of botnets in the mobile smartphone domain.

1.3 Research Objectives

The Command and Control (C2) component for any botnet in a fixed or mobile network is critical because it directly affects the stealthiness, robustness, and overall effectiveness of the botnet. A botmaster needs to devise crafty and stealthy ways to get

commands to the botnet. This translates to hiding inbound SMS messages from the user in a Short Message Service (SMS) botnet. This author is unaware of any detection methods specifically for smartphone botnets using SMS C2 at the time of this writing. This thesis lays the groundwork for identifying SMS as the C2 channel on smartphones. The research goals are the following:

- Design a security mechanism complementary to Android's existing security providing awareness for covert SMS messages and identifying potential C2 and bot presence on a system.
- Implement the design as a proof of concept on a retail Android smartphone.
- Validate the proof of concept using an Android SMS bot.

1.4 Approach

The goals of this research are realized by developing a loadable kernel module (LKM) that logs, at kernel level, all inbound SMS messages as they are received from the modem. Comparing the kernel log with the list of received messages in the SMS application facilitates detection of suspicious messages that may be indicative of a botnet. The module is implemented and validated using an SMS bot running on an HTC Nexus One Android smartphone. Details are described in Chapters 3 and 4.

1.5 Assumptions and Limitations

Several key assumptions and limitations were accepted in realizing the goals. The assumptions are:

- Assumption 1: A hybrid implementation of an SMS botnet can use SMS in concert with other channels for C2. This research assumes SMS is the only C2 channel for both inbound and outbound communication.

- Assumption 2: Cellular network operators use SMS for more than just providing their subscribers text messaging services. This research assumes all non-malicious SMS messages are meant for the user to see in an application and all hidden SMS messages are considered malicious or C2 related.
- Assumption 3: The focus is not on the actual injection and spreading phases of a botnet. This research assumes there are successful vectors for infecting a smartphone with a bot based on related works on Android security.

The limitations are:

- Limitation 1: The implementation is hardware specific and depends on the specific kernel-level driver a vendor chooses to communicate with the smartphone's modem. The current implementation is successful on an HTC Nexus One and will work on most HTC model phones and smartphones using a Global System for Mobile Communications (GSM) modem with the shared memory driver (SMD) as the kernel's interface to the baseband radio.
- Limitation 2: The process for identifying C2 is manual in nature and requires a comparison of the kernel logs to the list of received messages in the SMS user application.
- Limitation 3: The events are stored in the kernel log. The size of the kernel log's ring buffer is limited and limits the amount of total messages that can be logged. The current size of the buffer is 128KB.
- Limitation 4: Detecting C2 traffic on the system is an indicator of the presence of a bot on the system. The module only provides awareness and does not take any steps towards mitigation or eradication.

- Limitation 5: As a host-based system, detection is only for bot presence on the smartphone device and not detecting the network topology of the bot network itself.

1.6 Thesis Overview

The remainder of this document is structured as follows. Chapter 2 is a literature review summarizing the botnet phenomena on traditional computer systems including detection techniques, a high-level overview of general smartphone architecture, SMS, and Google's Android mobile Operating System (OS). The chapter concludes with a review of works related to SMS botnets on smartphones and Android-based smartphone security vulnerabilities. Chapter 3 presents the methodology for designing the mechanism to provide awareness for detecting SMS C2, implementing the module on an Android smartphone as a proof of concept, and testing the proof of concept with an SMS bot. Chapter 4 validates the proof of concept and demonstrates the utility of the security mechanism. Finally, Chapter 5 summarizes the research, discusses the impacts to the botnet and mobile smartphone communities, and suggests areas for extended and future research.

2 LITERATURE REVIEW AND RELATED WORKS

Developing a security module to identify covert SMS messages for the purpose of detecting potential C2 communication on Android-based smartphones requires an understanding of the different components involved in the system. This chapter provides a background and overview of those components. First, the botnet phenomena is discussed to understand how botnets function on fixed networks and desktop systems. Next, an explanation of general smartphones are discussed to understand their architecture and their process for sending and receiving SMS messages on cellular networks. Then, an overview of the Android operating system's architecture, security, and SMS messaging components is discussed. Finally, all the components are tied together and smartphone botnets and SMS smartphone botnets are discussed.

2.1 Overview of the Botnet Phenomena

Botnets have evolved into one of the largest sources of cyber crime perpetrators on the web at present over the last decade [17]. Several taxonomy works were developed identifying botnet characteristics including infection mechanisms, C2 structures, and detection and defensive techniques [7, 12, 17, 21, 36]. The following summarizes the botnet phenomena affecting traditional desktop computers and fixed network operators.

2.1.1 History and Terminology

Bots are used for both legitimate and illegitimate activities. A bot refers to code, a script, or set of scripts designed to perform predefined functions in an automated fashion. Before nefarious uses, bots were used legitimately by search engines to crawl the web, online game sites as artificial opponents, and for Internet Relay Chat (IRC) networks to automate the Administrator functions [26]. It was originally in the IRC networks where bots were modified and linked to other bots to create the first network of bots, or botnet.

These early botnets perform malicious activities including elevating IRC channel privileges, stealing personal information, and executing denial of service attacks on IRC servers. More detailed discussion on botnet history and evolution, including initial botnet family names, capabilities and frameworks, are discussed in [5, 21, 26].

A bot refers to malware on a computer waiting for instructions from an attacker for the purpose of this research. This malware is not an exploit to software or an OS; it is the payload allowing an attacker full control of the infected system. The distinguishing characteristic a bot has from other kinds of malware is the ability to communicate and receive commands and malicious capabilities from the attacker [21, 36]. A botnet refers to a collection or network of bots controlled by an attacker or group of attackers. The attacker or group of attackers controlling the botnet is known as the botmaster and the mechanisms allowing the botmaster to communicate with the botnet to perform the desired affects is known as C2. The way a botmaster chooses to implement C2 is the most important component. This decision directly affects the botnets stealthiness, ability to survive individual bots being removed from the network, and overall effectiveness. Related work detailing measuring performance metrics for robustness, efficiency, size, and utility are discussed in [6, 7].

2.1.2 Botnet Phases

The existence of a botnet can be described in phases. Initially, the botmaster must infect a system with the bot. The botmaster then continues to infect multiple systems in the injection and spreading phase using manual or self-propagation techniques to other systems on the network or across networks. The C2 communication phase between the bot and botmaster is initiated once a system is compromised. The malicious application phase can begin when the botmaster has amassed a large enough set of bots. Table 2.1 is a summary of the previously described botnet phases.

Table 2.1: Botnet Phases and Description

Phase		Description
Injection and Spreading		Distribution of malicious email OS and software vulnerabilities Instant Messaging Other botnets
Command & Control	Model & Topology	Centralized P2P / Distributed
	Application & Protocol	IRC Web: HTTP, HTTPS, DNS Email: SMTP Instant Messenger P2P
	Communication	Inbound / Bi-directional Push or Pull
Application		DDoS Attacks Spam & Advertising Hosting malicious resources Personal or Corporate espianage

2.1.2.1 Injection and Spreading. There are several methods for bots to inject and spread throughout a network. The most common methods include malicious emails, software vulnerabilities, instant messaging, peer-to-peer (P2P) file sharing networks, and using other botnets [17]:

- Distributing malicious emails: Coupled with social engineering (e.g., Phishing), a system is compromised when users execute a malicious binary in an attachment or clicking on an embedded link to the remote binary.
- Software vulnerabilities: New systems are commandeered by exploiting vulnerabilities in the operating system or application software. Automated tools to scan and exploit vulnerable systems are typically used.
- Instant messaging: Similar to distributing malicious emails, users receive and execute a malicious binary in the form of an attachment or embedded link from an individual on their buddy list.

- P2P file sharing network: The malicious binary is placed on a users system in a shared location and social engineering is used to get the user to execute the binary.

- Using other botnets: Either a botnet is used to propagate and amass new bots to create a separate botnet or a botnet is used to take over an existing botnet using the methods described above.

2.1.2.2 Command and Control Models. The C2 model is the most important component of the botnet. The C2 model determines how the botmaster maintains and employs the botnet. Communication can be either inbound or bi-directional from the bot's point of view. A botmaster can push commands and updates or the botnet can check-in and pull the updates and commands. Two important decisions a botmaster considers for resilience and effectiveness are channel protocol and C2 structure. Early botnet C2 favored the IRC protocol because of their origin in IRC networks. Currently most professional networks now block IRC traffic. Botnet communication evolved to use email (SMTP) and web (HTTP, HTTPS, DNS) protocols, in addition to IRC as a result of networks blocking IRC traffic and other detection and mitigation techniques.

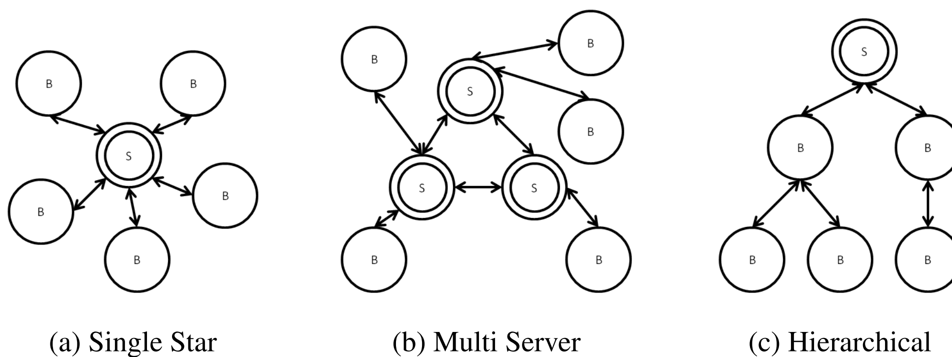
Centralized and Peer-to-Peer (P2P) are the two primary types of C2 structures. The centralized C2 model is characterized by the botmaster controlling a single centralized server hosting services such as IRC, HTTP, and SMTP. A new bot will communicate with the botmaster through this server once infected. Three topologies within this model are:

- Single Star: The simplest topology, a single central server is the C2 master.

- Multi-server Star: This incorporates multiple servers with bots distributed amongst them to increase redundancy and scalability. The servers coordinate and synchronize with each other and appear as a central location to the entire botnet.

- Hierarchical: Employs select bots to act as proxy servers to the actual C2 server to increase the resilience of the C2 server. This minimizes the number of bots knowing the actual location of the C2 server.

Figure 2.1 shows the different centralized C2 structures. The principle reason for implementing a centralized C2 model is its relative ease in design and management, its relatively quick response to both updates and executing commands, and stealthy network traffic which avoids detection. A major drawback with this model is the single point of failure. Once identified, the central server along with the entire botnet can be hijacked or terminated.



(a) Single Star (b) Multi Server (c) Hierarchical

Figure 2.1: Centralized C2 Botnet Structures [17]

In a distributed P2P model each bot acts as both client and server. The principle reason for implementing this model is the overall resilience of the botnet from individual bot detection. Identifying or removing a number of bots will not necessarily lead to the destruction of the botnet. However, the drawbacks for this model include the level of difficulty in design and management, the un-reliable response time for propagating commands and updates, and potential anomalous traffic due to inter-communication within the botnet [17]. Figure 2.2 illustrates a sample P2P C2 structure.

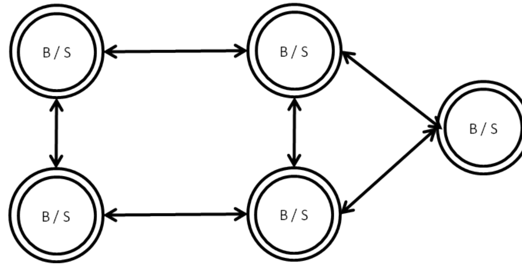


Figure 2.2: Example of P2P C2 Botnet Structure [17]

2.1.2.3 Malicious Applications. The lucrative aspect of botnets for cyber criminals is the ability to monetize their application. Malicious activities include but are not limited to:

- Distributed Denial of Service (DDoS) Attacks: A botmaster can effectively attack a targeted server and remain anonymous.
- Spam and Advertisements: A botmaster can use infected machines to send mass advertising emails either via proxy or relay. Valid email addresses from infected machines can also be used to defeat spam filters.
- Hosting Malicious Resources: A botmaster can select a specific bot or bots to store malicious binaries, fraudulent sites, or phishing sites.
- Personal and Corporate Espionage: A botmaster also has the data on an infected system or the data that passes through an infected system at their disposal. As a result, personal information such as passwords or credit card information can be accessed. Also at risk are business documentation, trade secrets, and other corporate sensitive information.

The process, roles, and description of the monetary flow resulting from a botmaster executing these activities are described in [20].

2.1.3 Botnet Defense Techniques

As with any malware, there are general anti-malware countermeasures falling into three categories: detection, prevention, and eradication [16]. Detection is the ability to recognize and/or locate malware on a system. At the network level, Intrusion Detection Systems (IDS), Honeypots, and active and passive monitoring are the primary countermeasures for detecting botnets; extensive research has been accomplished using Honeypots, Honeynets, and other network monitoring schemes for detecting botnets [13, 36, 37]. IDSs may use a signature-based technique, anomaly-based technique, or both. Detection requires monitors on the system that analyze the system internals as opposed to network traffic to detect botnet activity at the host level. Prevention is keeping malware from executing on or entering a system. A proactive approach, from the user perspective, includes gaining awareness of all types of malware including botnets, maintaining software and operating system updates, and investing system resources towards some form of malware detection software (e.g., anti-virus). Eradication is removing the detected malware and all its traces. Typically, the only way to achieve this for a botnet is to re-install the operating system.

2.2 Smartphone Architecture and Short Message Service

Smartphones are set apart from their dumb phone and feature phone counterparts by the presence of a full-fledged operating system and file system. A feature phone is set apart from a dumb phone by its ability to perform more than just mobile telephony like having personal digital assistant (PDA) features including calendar, contacts, email, or running simple applications. However, they lack a dedicated operating system or file system allowing more user enhancing applications. The presence of this additional component requires additional hardware and a different internal architecture as well.

2.2.1 Smartphone Architecture

Dumb and feature phones' processor is the baseband radio processor. The baseband radio processor runs a specialized real-time operating system to handle communication with cellular networks. This processor is also responsible for handling the additional feature sets including the PDA capabilities or running simple Java-based applications on feature phones. Smartphones consist of two processors: a dedicated application processor that hosts the phone's operating system and a baseband processor. The application processor presents in the form of a System on a Chip (SOC) with additional integrated functional components. These processors collectively form the design of a modern smartphone when combined with other peripherals such as a touch screen, keyboard, audio input/output, GPS, and a gyro. Figure 2.3 depicts a generalized system design showing the two processors and peripherals.

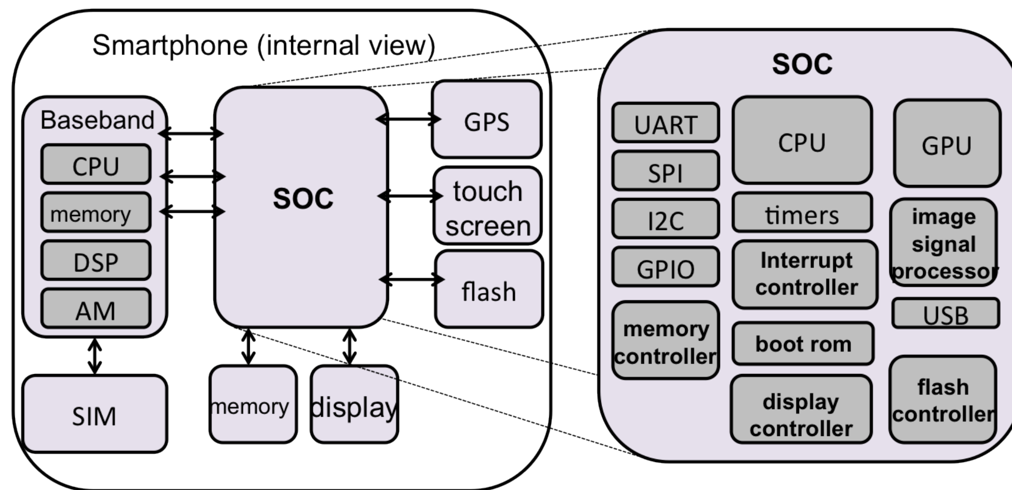


Figure 2.3: Conceptual Design of Modern Smartphone Architecture adapted from [2, 22]

This conceptual design of modern smartphones is generally how iPhone, Android, and Windows Mobile devices are implemented [2, 22, 35]. In addition to the operating system providing the user an interface for mobile computing and running feature rich

applications, it is also responsible for communicating with the baseband radio to maintain the telephony capabilities. This communication with the baseband processor is accomplished through serial, terminal, USB connections or shared memory between the two processors [35].

2.2.2 Short Message Service (SMS)

A feature available on nearly all mobile phones is SMS. Mobile network operators use this service for various reasons, but it is mainly known by subscribers as the service providing text messaging capabilities. SMS can also provide other functions such as a control channel for voice mail notifications, remote over-the-air (OTA) phone configurations, and a transport for the Wireless Application Protocol (WAP).

2.2.2.1 Sending and Receiving SMS. When sending or receiving an SMS message, the important components in the cellular network are the base transceiver station (base station) and the SMS Center (SMSC). SMS messages generated by a mobile phone are transmitted from the modem to the base station. The base station then forwards the message to an intermediary component, the SMSC, which provides the store and forward service and queues messages if the receiving phone is unavailable. The SMSC also provides an avenue to send SMS messages from other than mobile devices. This is how internet services send SMS messages to mobile devices. Messages are forwarded directly to the receiving base station which forwards the message to the receiving mobile device if the recipient is on the same network and handled by the same SMSC. Otherwise it will forward the message to the recipient's SMSC to handle. SMS text messaging is provided as a best effort service and delivery is not guaranteed. Figure 2.4 displays this simplified overview of sending and receiving SMS messages at the network level; additional detail about the cellular network components and SMS messaging can be found in [18].

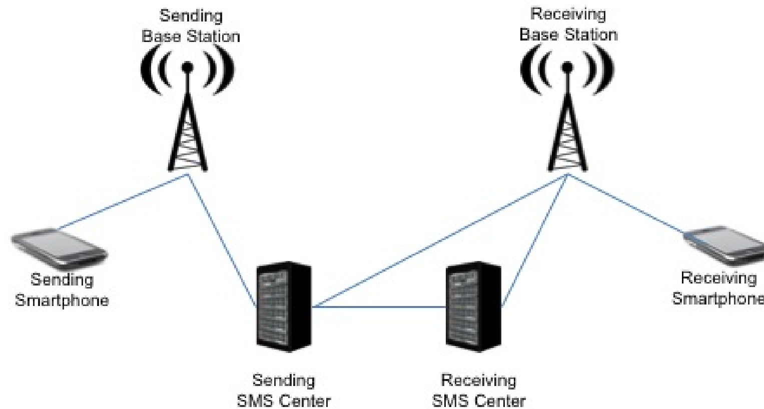


Figure 2.4: SMS Delivery and Receipt at the Cellular Network Level based on [22, 34]

The important components are the phone's telephony stack, device/modem driver, and the baseband radio at the device level. The telephony stack provides all the layers of abstraction and handles all communication between the application processor and the modem. It consists of all the components and interfaces for developers to create applications using the modem. Any use of the modem, data, voice or SMS, requires the use of special modem commands. These commands are communicated between the two processors through the device driver via the serial, terminal, USB or shared memory connections. Figure 2.5 shows the smartphone components involved in sending and receiving SMS messages starting at the top with the application, the radio application programming interfaces (APIs) and Radio Interface Layer, the modem driver, and modem.

2.2.2.2 SMS and the ATtention (AT) modem commands. All modem functions require the use of AT commands. AT is an abbreviation for ATtention and all commands begin with "AT". Basic AT commands include a set of modem configuration controls and simple functional commands like dial, answer call, and hang up. SMS AT commands are a subset of an extended AT command set that all begin with a "+" . Table 2.2 shows a sample list of basic SMS related AT commands for both sending and

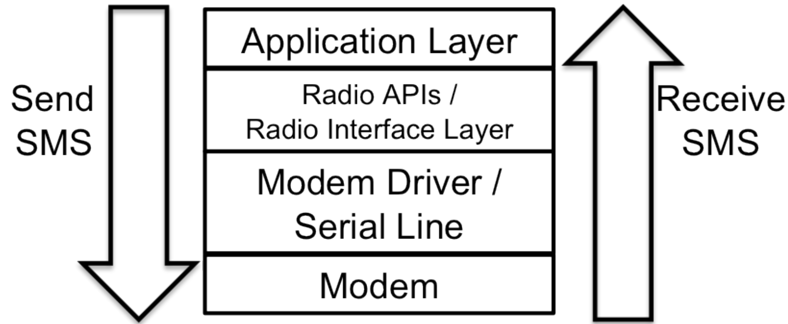


Figure 2.5: Sending and Receiving SMS at the Device Component Level adapted from [22, 34]

receiving messages on the phone. Not all SMS commands are implemented on all phones. Additional details about SMS and the SMS AT-command set can be are found in [1, 9].

Table 2.2: Sample SMS Commands (Extended AT Commands)

Sending (solicited) SMS Commands	**Receiving (unsolicited) SMS Commands**
AT+CMGS (send message)	AT+CMT (forward message to computer)
AT+CMGR (read message)	AT+CMB (forward broadcast message to computer)
AT+CMGL (list message)	AT+CDS (forward received status report to computer
AT+CMGC (send command)	

This section discussed the general smartphone architecture and how SMS is implemented both at a high and low level. It is easy to recognize at a high level the built-in persistence SMS provides which is a good characteristic for any botnet C2. Understanding the components and details of the process at a low level is import for developing ways to both hide and detect covert SMS messages. The next section provides an overview of the Android operating system architecture, its security, and the Android-specific components that allow the SMS capabilities.

2.3 Introduction to Google Android

The Android platform was designed from the ground up by a consortium of industry partners including mobile operators, handset manufacturers, semiconductor companies, software companies and commercialization companies looking to provide an open platform allowing developers to continue to improve the user's experience in a mobile environment with new and innovative capabilities [24]. The Android architecture has four layers with five components: Applications, Application Framework, Libraries, Android Runtime, and the Linux Kernel. Figure 2.6 shows a representation of the Android architecture.

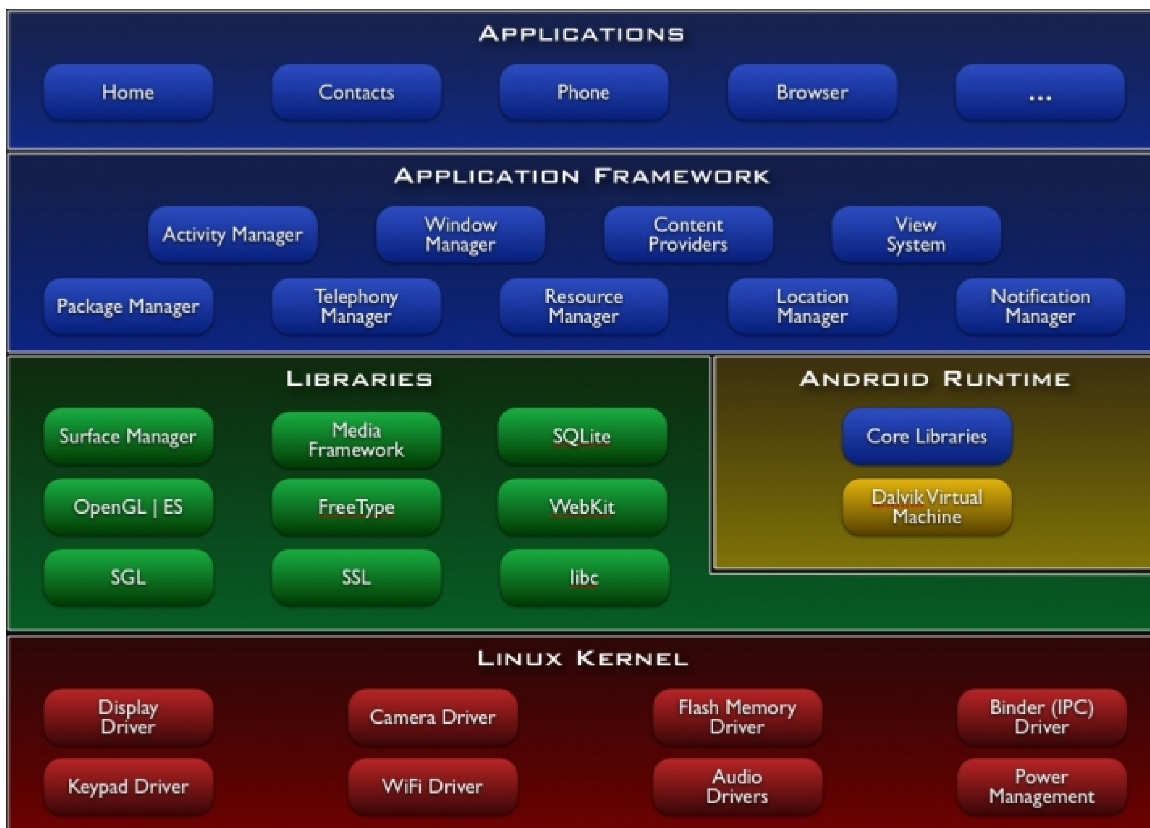


Figure 2.6: Android System Architecture [8]

2.3.1 Android Architecture

At the top of the stack is the Applications Layer. Applications are written in Java and provide users the functionality and capabilities expected in smartphones including phone, messaging, camera, productivity, web, entertainment and gaming. Below the Applications layer is the Application Framework layer. This layer is also written completely in Java and provides developers Google-proprietary extensions and services and access to the application programming interfaces (API) to the native libraries. The intent is to help developers and simplify the reuse of components from application to application. The next layer down includes the native libraries and the Android runtime environment. The native libraries are written in C/C++ and are used by various components of the system. The Android runtime component contains the core libraries and the Dalvik virtual machine (VM). The core libraries provide most of the functionality available in the core libraries of the Java programming language [8]. The Dalvik VM is Google's proprietary implementation of the Java VM, optimized for running multiple instances efficiently and with a minimal memory footprint; details on the major differences between the Java VM and Google's Davlik VM is discussed in [10]. An Android application runs as its own process with its own instance of the Dalvik VM. At the foundation of the stack lies a stripped down version of Linux kernel 2.6 which provides the core system services and abstraction layer between hardware and the rest of the stack.

2.3.2 Android Security

This subsection provides a summary of Android security related work [10, 11, 23, 29, 30]. The microprocessor of choice for the architecture of Android devices are based on is the Advanced RISC Machine (ARM) family of processors which provides variety of performance benefits suitable for mobile devices. The cons from a security perspective include having an executable heap and stack and no memory map

randomization. ARM architecture also provides security mechanisms such as TrustZone technology and no-execution page protection; however, Android currently does not use these features. In general, Android devices are susceptible to all current ARM vulnerabilities and exploits.

Application distribution is provided in a central location called the Android market. Developers publish their applications for users to download. Only a developer license and application signing is required to publish to the market. Anyone can sign up or become a developer for just twenty-five dollars and this can even be done anonymously. There are no Certificate Authorities; certificates are self-signed by developers. The combination of opportunity and anonymity provide an enticing environment for attackers to widely disseminate malicious code. Google does have the ability to push kill messages to devices to remotely remove applications from both phones and the Android market, but this is only after the malicious software has been identified and reported.

Android has a permissions-based security model with respect to application installation. The Application Framework layer handles initial installation permissions and enforcing system permissions during runtime. The application requests permissions upon installation to access data from other applications it needs to interact with (e.g., contacts, calendar, SMS, email, social networks logins, etc.) which the user must approve for a successful installation. The burden lies on the users to understand the impacts of sharing data between applications; however, users are not given the granularity of what data elements from which applications are being accessed. Approval is all or nothing. Most users choose to click through warnings just to install the application not knowing what they agreed to.

As discussed earlier, when an application is installed, it is given Unix-style user-ID/group-ID based on approved permissions implementing a separation of privileges; however, [29] describes a scenario for installing two seemingly unrelated applications by

the same developer using the shared user-ID permission to collaborate and leak information from the device.

When an application is running, it runs in its own process in its own instance of the Dalvik VM with the intent of maintaining all code execution within the VM. A work around is using the Java Native Interface (JNI) providing access to Android's native libraries allowing code execution outside of the VM and opening return-to-libc-type attacks [23, 27]. A "return-to-libc" attack is typically a variation to the buffer overflow attack. A buffer overflow is used to write executable code on the stack often generating a terminal shell. Executing code on the stack is prohibited in some operating systems. A way to circumvent this security mechanism is to use the buffer overflow to jump to the address of existing executable code memory. Typically, the "system()" function in libc, which is already loaded into memory, is used as an address to jump to and generate the shell.

2.3.3 Android Radio Interface Layer (RIL)

Android's telephony stack provides all the layers of abstraction to the radio hardware. At the top level, Android's application layer, the Phone and SMS applications provide users with the phone and messaging features. The telephony services and RIL reside in the application framework layer. Android's telephony services (android.telephony) provide developers with the APIs to develop their own phone and messaging applications or applications that utilize the radio hardware. The RIL resides between the hardware and telephony services and is the abstraction layer for the radio hardware and handles all communication with the modem. Figure 2.7 shows Android's telephony stack and the components of the RIL; additional details about Android's RIL can be found in [3, 33].

As discussed in Section 2.3, all modem communication is in the form of AT commands; Android's RIL supports two forms of communication:

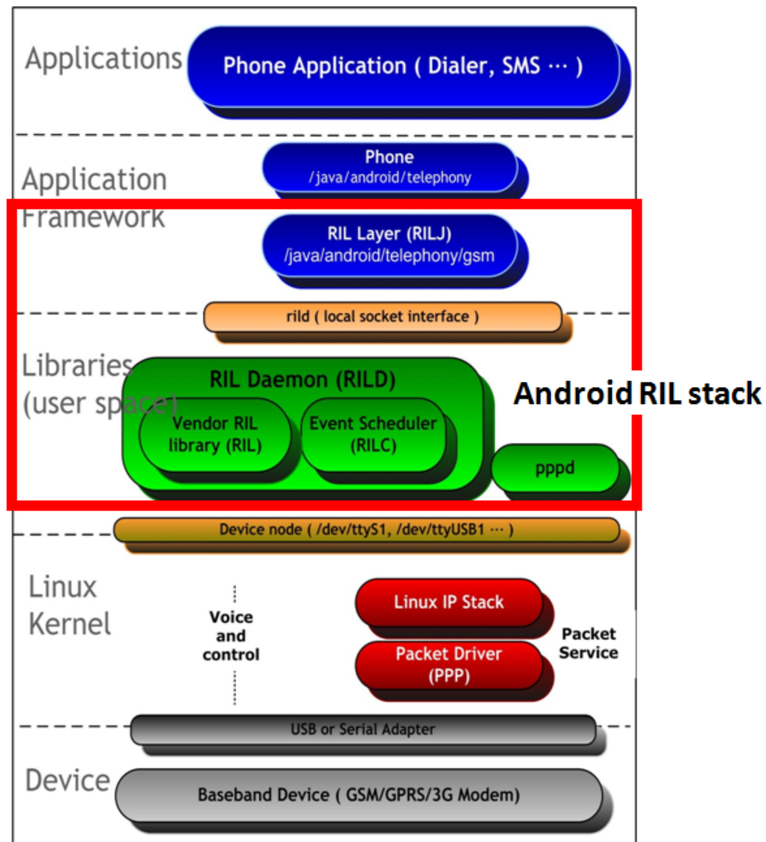


Figure 2.7: Overview of Android's Telephony Stack and Radio Interface Layer [33]

- Solicited Requests: Any commands originating from the user and originating from the RILJ. There are over 60 solicited AT commands implemented, examples include send SMS, dial phone number, hang up, etc.

- Unsolicited Requests: Any commands originating from the baseband radio. There are 11 unsolicited AT commands implemented, examples include receive SMS, hang up, inbound phone call, etc.

The RIL consists of four main components to handle all communication requests and flow to and from the radio hardware. The four components and their roles, identified by their system log tag names (RILJ, RILD, RILC, RIL), are:

- Vendor RIL (RIL): RIL acts as a driver for RILJ, it is a proprietary shared library specific to each modem, responsible for initiating and handling all communication with the hardware radio through the kernel.
- Event Scheduler (RILC): RILC serves as the bridge between RILJ and RIL. It handles all solicited requests from RILJ and dispatches them to RIL. It also handles both the unsolicited requests and the responses for solicited requests from the RIL and dispatches them to RILJ.
- RIL Daemon (RILD): The purpose of the RILD is to initiate the event scheduler and RIL. To initiate RIL, RILD locates the vendor library, maps the functions to the required RILJ functions via the RIL_Init function. Once both RILC and RIL are initiated, the process sleeps forever.
- Java RIL (RILJ): Part of telephony services and provides developers the telephony APIs. The purpose of RILJ is to dispatch solicited requests to RILC by parcel through a local socket interface and handle the responses and unsolicited requests dispatched to it from RILC.

This section provided an overview of Android's architecture discussing each component of the stack. A review of the existing security was also provided and vulnerabilities were identified through existing research. Finally, the Android-specific components implementing the telephone and SMS capabilities were discussed. The next section ties the previous three sections together and discusses botnets on smartphones, Android smartphone botnets, and SMS botnets.

2.4 Smartphone Botnets, SMS Botnets, and Android Botnets

The majority of botnet research is focused on desktop computer systems and fixed networks. Malware in a mobile environment is not a novel idea. The shift to use botnets

on smartphones and mobile networks is increasing in popularity and focus. Smartphone devices have networking capabilities via High-Speed Downlink Packet Access (HSDPA), Evolution-Data Optimized (EV-DO), Universal Mobile Telecommunication System (UMTS), Enhanced Data Rates for GSM Evolution (EDGE), General Packet Radio Service (GPRS), and wireless networks (WLAN), among other cellular networking technology and infrastructure. Coupled with the additional C2 channels including SMS and Bluetooth, persistent connectivity, and the increased use in various business and home communities, mobile botnets are becoming more lucrative than traditional botnets. This section reviews related works on botnets on smartphones, SMS-based botnets, and Android botnets.

2.4.1 Smartphone Botnets

Attack vectors introducing malware into a mobile device or smartphone via SMS, MMS, downloaded executable, and Bluetooth have been documented [31]. Android devices are just as susceptible to these attack vectors. In [23], the process is discussed for using the JNI and ARM exploits in an Android application to bootstrap a rootkit to provide a platform for building a botnet. Using Android's Marketplace the persistent connectivity through a combination of cellular radio, wireless LAN or Bluetooth, the botnet injection and spreading phase is accomplished easily.

2.4.2 Android Botnets in the Wild

Smartphone botnets are a reality. In the wild, the first botnet on mobile devices was detected in 2009 on Symbian OS based systems [15]. Also in 2009, several countries in Europe and Australia experienced a botnet on Apple's iOS iPhones, iKee.B; a breakdown analysis was performed detailing the installation, propagation, C2, and approximation of the original source [25]. On Android systems there have been two identified instances. The first botnet, Droiddream, was detected in March of 2011, and the security group

Lookout Mobile Security provided an extensive analysis of how the Trojan attempted to gain root access and communicated with its C2 server to download and update [28]. Variants, including a Droiddream lite, have been detected as well. In June of 2011, jSMSHider, was identified. This Trojan attacked custom ROMs by exploiting the vulnerability of custom signed ROMs from the Android Open Source Project [32].

2.4.3 SMS Botnets

Smartphones provide additional C2 channels less common to desktop systems. Related work has been completed developing proof of concept botnets successfully opening the possibility for botmasters to use Bluetooth and SMS as a means for C2. In [31], a simulation was used to demonstrate the botnet. A hybrid C2 structure was used identifying select nodes as command entry points into the botnet via SMS. In [34], the botnet was implemented on hardware and also describes a hybrid C2 structure utilizing select Android-based phones with GSM modems to assist in stealth and robustness.

2.5 Summary

This chapter presents overviews and background information related to botnets, smartphones, SMS and Android. Significant research has been dedicated towards understanding and defending against botnets on traditional computer systems but not towards mobile botnets. Mobile botnets are now becoming a trend in the malware world. This research leverages the proof of concept Android botnet discussed in [34] and describes the development of a proof of concept security module to provide user awareness for SMS-based C2. The next chapter describes in detail the design approach, implementation on a retail Android smartphone and validates the success against the proof of concept SMS bot. This research extends the works related to C2 detection countermeasures for botnets specifically in the smartphone domain.

3 METHODOLOGY

3.1 Introduction

The most important component for any botnet is C2 because it directly affects the botnet's stealthiness, resilience, and overall effectiveness. If the C2 mechanism is identified, removing a bot from the network, disrupting, dismantling, or hijacking the botnet becomes straightforward. The cat and mouse game in the botnet world is hiding and detecting the C2. Attackers continually look for creative ways of hiding C2 in traditional communication channels and are branching out into new channels. When sending a C2 message, two factors botmasters consider are: latency, how quickly the botnet responds to the message; and contact percentage, how many of the bots actually receive the message. A botmaster wants to ensure both a high contact percentage and a quick response when executing a DDos attack. However, contact percentage is weighted more heavily during C2 messages intended for updating the botnet's command set, or executing an attack like retrieving personal or sensitive information. SMS as a C2 channel is enticing to botmasters because of the importance of contact percentage and the built-in persistence SMS provides. This chapter discusses the design, implementation and validation of a security module for identifying covert SMS C2 messages on Android smartphones.

3.2 Transparent SMS Bot

It is important to understand how SMS C2 can be hidden on an Android smartphone. A proof of concept bot described in [34] achieves this by executing a man-in-the-middle attack between the application layer and the kernel/modem driver on the telephony stack. Creating this proxy below the application layer is an effective way to transparently intercept and filter inbound bot-related messages from the user.

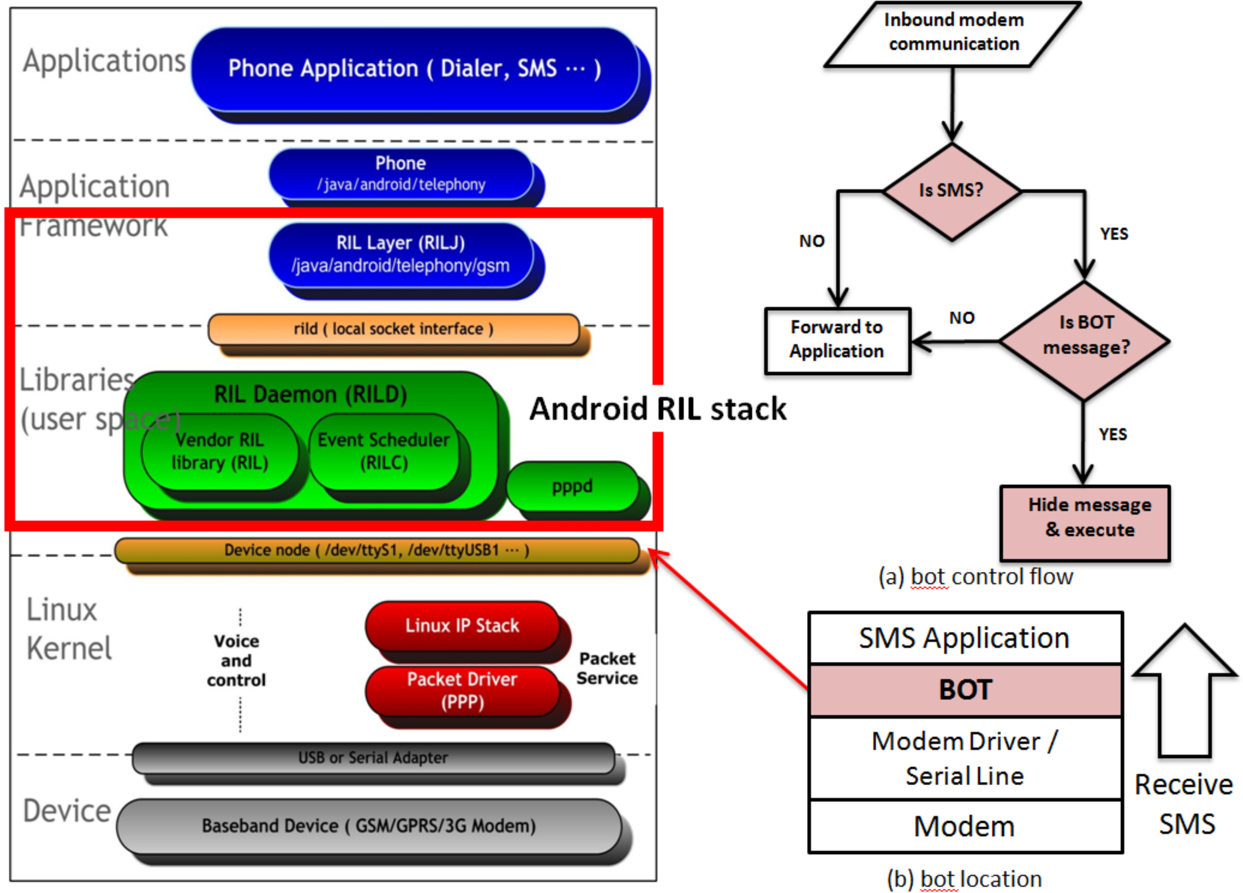


Figure 3.1: Bot Location and Control Flow on Android's Telephony Stack [33, 34]

The bot first determines whether or not information coming from the modem is SMS related. It passes the information immediately up the stack to the user application if the data is determined not to be SMS related. If so, it then determines if the message is bot-related, and if not, the message is immediately passed up the stack. Otherwise, the bot consumes the message hiding it from the user application and executes the command contained in the message. Figure 3.1 illustrates both the bot's control flow and location on Android's telephony stack.

3.3 Design

The proof of concept SMS bot effectively filters SMS messages by staying below the application layer. This research proposes to defeat this type of botnet by getting below the botnet and in-between the driver and physical modem, in kernel-space, to log inbound messages before they can be filtered by another process. Considering a defense-in-depth approach, the design for detecting SMS-specific C2 is host-based, reactive in nature and uses passive monitoring.

- Network vs. Host-based: Detection and mitigation for SMS and other botnets can be implemented at the network level. Network service providers would be required to upgrade infrastructure with security mechanisms to protect their subscribers. Possible solutions and financial justification to present providers are not in scope of this research. A host-based solution is the chosen approach.

- Proactive vs. Reactive: Preventing any malware from entering a system is an effective deterrent for cyber crime. Because of the Android security vulnerabilities discussed in Section 2.4, the approach is to focus on detection and not the injection or mitigation phases. Before a reactive eradication process can be executed, identification is required. The design is reactive in nature by assuming the system is already infected and only provides an avenue to identify potential C2 communication.

- Active vs. Passive Monitoring: With an SMS-based bot, the bot receives an SMS to process as either bot-related or not. The bot consumes bot-related messages and does nothing with other messages. Passive monitoring is effective if the messages are captured early in the process of entering the system and no later than right before the bot processes the message.

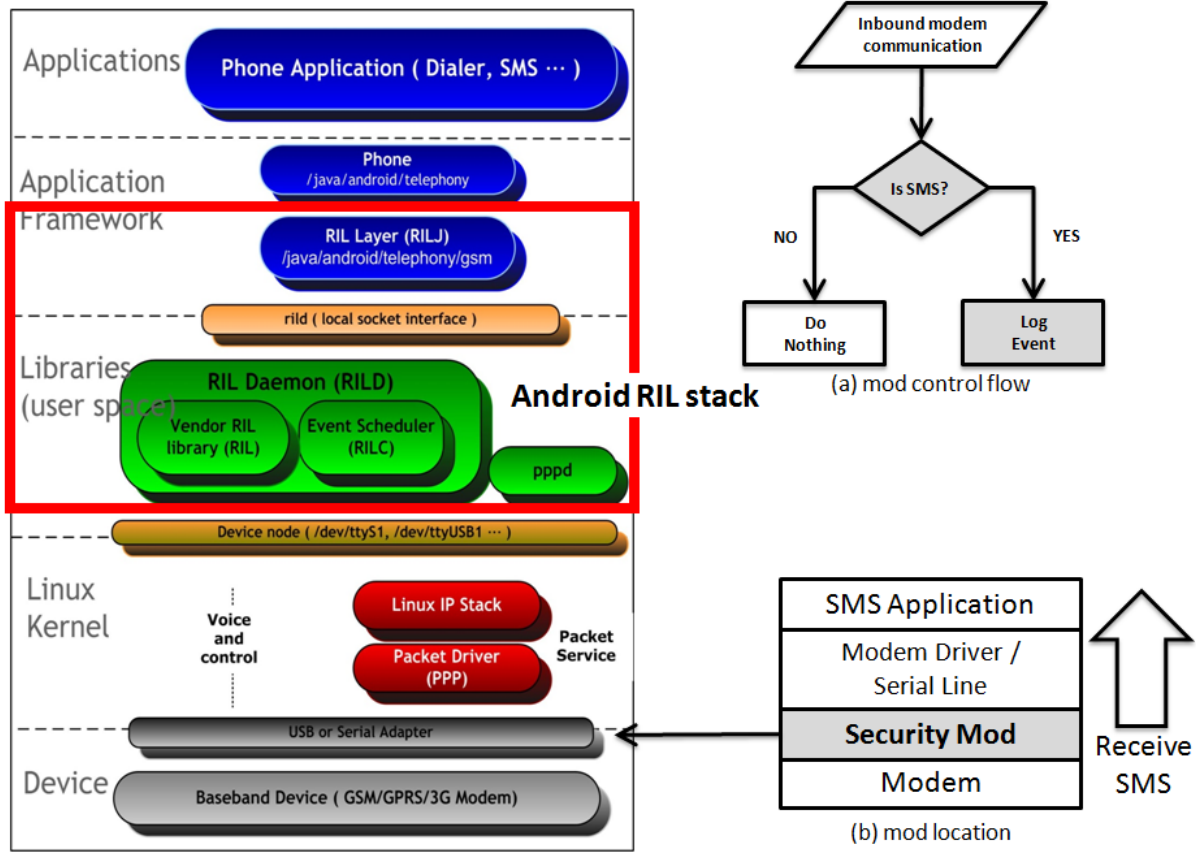


Figure 3.2: Security Module and Control Flow on Android's Telephony Stack

SMS messages enter the system through the modem on smartphones. Capturing messages as they enter through the modem is ideal; however, baseband radios are closed proprietary systems and not a feasible approach. The earliest messages can be intercepted is the point where the radio processor sends the data to the application processor. Figure 3.2 illustrates the security module in this location and on Android's telephony stack with the control flow.

It is assumed only SMS C2 or other malicious SMS messages are hidden from users. As a result, logging inbound messages at this location is effective in identifying whether or not a bot is hiding inbound SMS messages. The control flow of the security module is

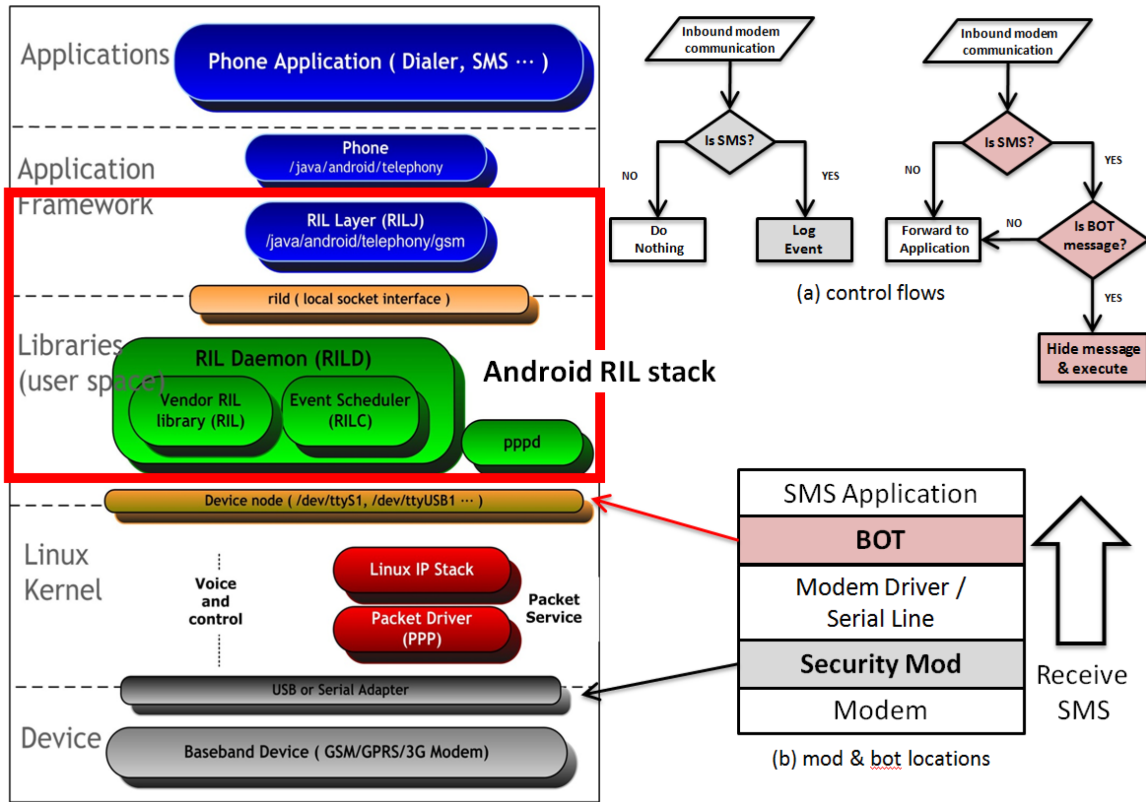


Figure 3.3: Security Module and Bot Locations with Control Flows on Android Telephony Stack

similar to the bot. The event is logged if inbound data is SMS related. Otherwise it does nothing. The difference between the two processes occurs after the data is determined to be an inbound SMS, no other processing is required. The inbound data is not modified or stored; only the event (i.e., "Message Received") and time stamp are required for the log. Finding discrepancies in the inbound message events log compared to the list of received messages in a user's application successfully identifies hidden C2 communication. Figure 3.3 illustrates both the security module and the SMS bot on Android's telephony stack and their control flows.

3.4 Implementation

This sections discusses the details for implementing the design on an HTC Nexus One, including the development environment, tools, and general process. Implementation is hardware specific. There are a few approaches to achieving the goal of capturing messages as they are received from the modem:

- New modem driver: This approach provides an avenue for a complete system and more control; however, the level of complexity involved in developing a new driver to handle modem communication is un-necessary to prove this design concept.
- Modify existing driver: Modifying the existing driver is less complex than developing a driver from scratch and could provide the same level of control and completeness; however, deployment to a production system is more involved and requires the kernel to be recompiled and flashed to the target system.
- Loadable kernel module: Loadable kernel modules (LKM) exist for the purpose of modifying or extending the running kernel without having to recompile and redeploy to a system. Modules can be loaded and removed as necessary to manage resources. This is the approach chosen for the proof of concept.

The LKM uses a kernel probe to instrument a routine that reads the buffer to which the modem writes data. A kernel probe is a built-in Linux kernel mechanism for seamlessly and dynamically debugging the system. Any routine in the kernel can be instrumented with kernel probes given the address of the instruction. There are three types of kernel probes: kprobe, jprobe and kretprobe. Jprobes and kretprobes are implementations of the general kprobe with breakpoints placed at a routine's entry or exit, respectively. Benefits of a jprobe include the ability to conveniently access the routine's arguments. A jprobe seamlessly executes a trampoline effect by setting a break point at the instruction address, calling a pre-handler function to make copies of the arguments on

the stack, executes the jprobe instructions, then returns to the original execution flow; the original routine's arguments remain unmodified. On the target system, the *ch_read()* routine in the SMD is instrumented. Figure 3.4 is a simplified illustration of instrumenting ch_read with a jprobe, for the source code see Appendix A. For more information on kernel probes, documentation can be found in the Linux source code [19].

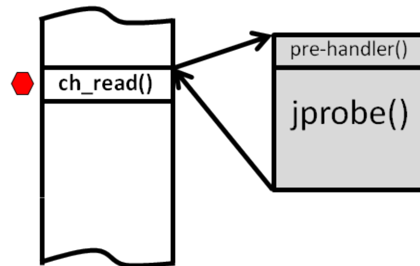


Figure 3.4: Instrumenting *ch_read* Using a Jprobe

3.4.1 Target Device: HTC Nexus One

Originally, it was believed that a module to perform SMS auditing could be developed to function on all Android phones. However, through the course of the research, it was learned that developing software to function at the kernel level is highly dependent upon the underlying hardware implementation. Thus, developing software for the kernel level turned out to be much more complicated than originally conceived, and is certainly much more difficult than developing conventional user applications.

Although the Android operating system and kernel are "open", vendors have full control over the development of their production devices. Vendors typically modify and compile the OS and kernel to accommodate their specific hardware needs and apply their own security mechanisms to prevent tampering. Modifications developed at the kernel level for one vendor's smartphone will generally need to be recompiled along with another

version of the kernel source to run directly on other hardware devices because of this tight coupling between the hardware platform and the kernel.

A specific smartphone needed to be selected to proceed with this research due to the strong dependency between the kernel and underlying hardware implementation. Factors to be considered included the following:

- Vendor and model: This is probably the most important factor in successfully developing software for Android devices when focusing on levels below the application layer. Developing beyond the application layer requires access to the OS and kernel source code. Also, knowledge of the vendor-specific security and processes for disabling them are required. Working below the application layer can permanently disable (e.g., "brick") a device. Having knowledge of the recovery system along with a compatible third-party backup and recovery system is beneficial.

- Android OS version: The Android Open Source Project (AOSP) is an initiative for developers who embrace the openness of the Android platform and want to use it to further the mobile device experience. It is also where developers go to access both the Android OS and kernel sources. The AOSP community and Google provide the most focus and support on the latest OS version and the main line of the kernel. Typically, a device is built for a specific OS version with a specific kernel. Various versions of source code are device specific, and not all versions work on all devices. Finally, building the source is both device and kernel specific, because the correct kernel version and proprietary binaries for a given device are required for a successful build on the target system.

- Linux kernel source: Although the Linux kernel is open source, vendors typically modify the source code to accommodate their hardware and do not always make

Feature	Value
Android OS version	2.3.6 (Gingerbread)
Baseband version	32.41.00.32H_5.08.00.04
Display	480 x 800 pixels (3.7 inches)
CPU	Qualcomm QSD8250 1GHz
Memory (internal)	512MB RAM / 512 MB ROM
Kernel version	2.6.35.7-ge0fb012

Figure 3.5: HTC Nexus One with Technical Specifications

their code available. At the very least, source code for the device's motherboard and chipset is required.

- SMS Bot Compatibility: Developing a kernel level module to work with the SMS bot requires a device that uses SMD as the communication interface with a GSM modem.

Based on these considerations, the HTC Nexus One, also known as the Android Development Phone 3 (ADP3) and shown in Figure 3.5 with its technical specifications, was chosen as a suitable and representative smartphone for this research.

3.4.2 Development Environment

The development system is a VM running 2 dual-core processors (4 processors total), 3GB of RAM, 65GB of hard drive space, and the 64-bit Ubuntu TLS (10.04) Linux distribution with latest updates (at the time of development). The VM specifications were chosen based on a combination of available resources from the host system and recommendations from the AOSP website for initializing a build environment for the

Android OS version 2.3.6 (Gingerbread, git branch tag *android-2.3.6_r1*). Table 3.1 is an overview of the development system technical specifications.

Table 3.1: Virtual Machine Development System Specifications

Feature	**Value**
Operating System	Ubuntu LTS (10.04)
CPUs	2 dual-core (4 total)
RAM	3GB
Hard Drive	65GB

Three utilities are required for developing and testing kernel modules in the Android environment are provided by downloading the Android source:

- Android Debugger Bridge (adb): adb is a command line utility that allows communication with an emulator or a connected Android device. adb is used for shell access to the device and transferring files and images to and from the VM for implementation purposes.
- fastboot: fastboot is a command line utility that allows for flashing images to partitions on Android devices from a system over USB.
- ARM compiler: The Android source comes with pre-built toolchains, libraries and compilers for developing on *x86* architecture and cross-compiling for the ARM architecture.

The final piece of the development environment is the Android kernel. Kernel source code for most Android phones is made available for download by the vendor. The source for the HTC Nexus One was unavailable (at the time of development and writing). The

MSM kernel provided by the AOSP and compatible with the HTC Nexus One's motherboard and Qualcomm chipset, is used for this development (specifically, git branch tag *msm-2.6.35*). For detailed steps in setting up the development environment including downloading and compiling the Android source, utilities, kernel, and LKM, see Appendix B.

3.4.3 Deploying the Module

Successfully inserting an LKM on a retail production system is more involved than merely building the module and using the *insmod* command. The first hurdle to overcome deals with the access to the kernel source. Production system kernels are configured to check the version of the kernel the module was built from. Modules will not load if the kernel versions do not match for security and stability reasons. The source code for the specific kernel running on the device is required.

Replacing the existing kernel in the development smartphone is required because a kernel compatible with the motherboard and chipset is used. An understanding of the Android system partitions and vendor-specific security mechanisms is required to accomplish this.

3.4.3.1 Android Partition Layout. The Android system consists of multiple partitions, shown in Table 3.2. The boot partition contains the kernel and is packed together with the RAMDISK, which is a set of files required to initialize the system including the init process. The fastboot utility allows you to "test" a kernel image to see if it will boot on a device without permanently ruining the device upon failure. Restarting the device replaces the old kernel. For a more permanent solution, the new kernel image needs to be packed together with the RAMDISK to form a new boot image and then the boot image needs to be flashed to the device.

Table 3.2: Partition Layout for Most Android Smartphones

dev:	**Name**	**Description**
mtd0:	“misc”	Stores misc system settings
mtd1:	“recovery”	Alternative boot partition
mtd2:	“boot”	Stores boot image (kernel & RAMDISK)
mtd3:	“system”	Operating system (minus kernel & RAMDISK)
mtd4:	“cache”	Stores frequently accessed app data and components
mtd5:	“userdata”	Partition containing user’s data

3.4.3.2 Vendor-specific Security Mechanisms. In addition to the application security Android implements, vendors add additional layers of security. Two widely known security mechanisms vendors employ are denying root-level access and locking the bootloader. Locking the bootloader prevents users from flashing new images or firmware to the device and generally prevents the use of the fastboot utility. To flash a new image and use the fastboot utility, the bootloader needs to be unlocked. On the HTC Nexus One, unlocking the bootloader is as simple as using adb, opening a shell, and executing a command. There are pros and cons to having a phone with and without root access. Having a device without root access helps maintain a certain security level and prevents novice users from inadvertently lowering system defenses like changing the system partion from read-only to read/write. From a vendor perspective, it provides a way to lock certain system features like tethering a network connection to other mobile devices for financial gains. For the tech savvy, obtaining root-level access to a phone opens a full set of customizations and unlocks the full potential of the mobile device. For installing an LKM, root access is required. This can be achieved by flashing new firmware once the bootloader is unlocked.

The following summarizes the general steps required to deploy an LKM on an Android device whose kernel source is unavailable:

1. Download and build an Android kernel suitable for the target system (see Appendix B for details on HTC devices).
2. If the bootloader is locked, unlock the boot loader.
3. Test the new kernel image with the fastboot utility.
4. Unpack the kernel from the existing boot image and re-pack it with the new kernel to create a new boot image.
5. Flash the new boot image to the target device using fastboot.
6. If root access is not available, obtain root access on the device.
7. Use adb to push and load the module to the phone.

3.5 Test Environment and Experimental Design

SMS is a best effort delivery service. A simulated network was chosen to avoid any potential message delivery variability and having to determine if the service provider is not delivering messages or if the module is affecting the system. Using a simulated network also prevents the need to purchase service and data contracts from a network provider.

The hardware used to simulate the cellular network base station and SMSC is an Anritsu MD8470A Signal Analyzer which includes Wireless Network Simulator (WNS) software to test voice, data, MMS and SMS. Both are shown in Figure 3.6. Only the SMS send features are used for the experiments. The signal analyzer simulates both the cellular base station and SMSC. Figure 3.7 illustrates a component diagram of the test environment including the HTC Nexus One, security module, and SMS bot.

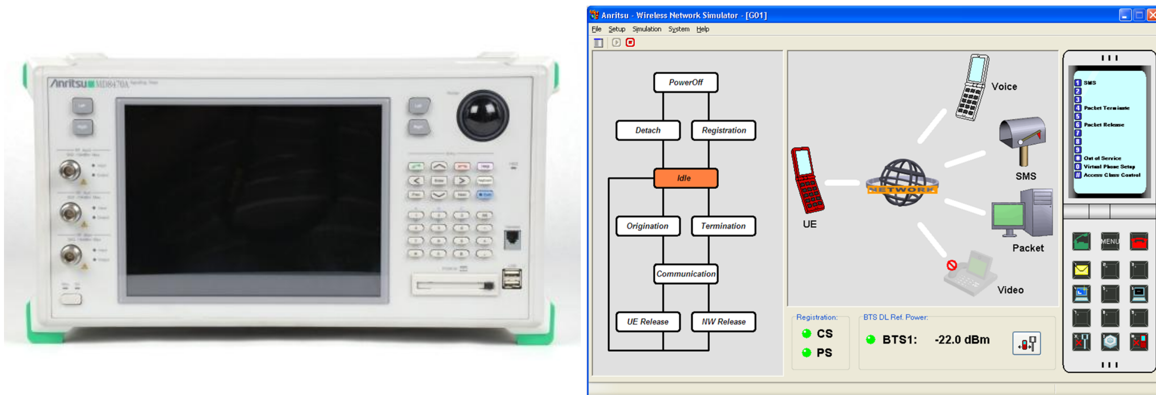


Figure 3.6: Anritsu MD8470 Signal Analyzer with Wireless Network Simulator Software

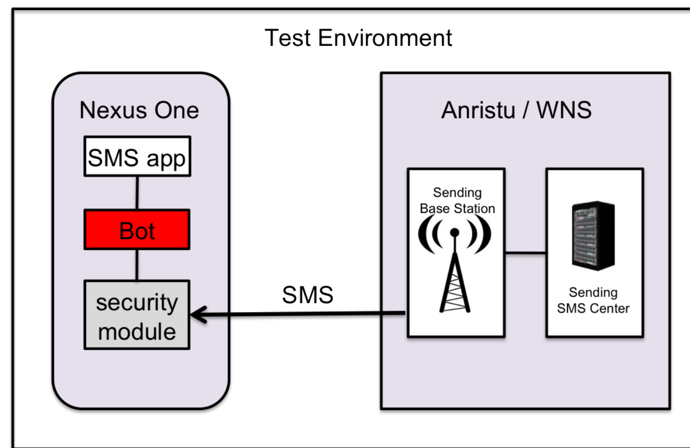


Figure 3.7: Component Overview of Test Environment

3.5.1 Experimental Design Overview

The tests are designed only to show functionality and utility of this proof of concept for detecting covert SMS messages. Because the target device is a production system used in a dedicated simulated cellular network, little variability is expected with the test results and tests are repeated only five times. The SMS bot has been presented at several security conferences and was shown to work and behave as expected. The bot also behaved as

Table 3.3: Summary of Experimental Tests

Test No.	Purpose	Description
1	Functionality	Test logging non-C2 messages
2	Functionality	Test logging C2 messages
3	Utility	Scenario 1: Well timed C2 message
4	Utility	Scenario 2: Poorly timed C2 message

expected when deployed to the target system prior to testing although not shown in this work. All tests include the SMS bot on the system in an active state. The bot is a safebot and does not execute any payload; it only provides the transparent SMS C2 capabilities. It hides bot-related messages based on a bot-key in the received message. The bot-key implemented is "BOT:". Any SMS message beginning with the bot-key will be hidden and all other messages are passed up the stack to the user application. The security module uses a ring buffer in the kernel to log the messages. All tests are performed with an empty kernel buffer and an empty message list in the SMS application, the stock SMS application is used in all testing. The actual implementation of the module sends the event string, "Message Received" with a time stamp to the kernel log. At the end of each test, the SMS application is compared to the list of logged messages in the kernel log to determine if the C2 messages can be identified. Table 3.3 provides a summary of the tests identifying their purpose and description.

3.5.2 Test 1: Functionality Test (Logging non-C2 Messages)

This tests the module's basic ability to log non-C2 messages. The average user sends and receives approximately 110 messages daily [14]. The workload for this test is 110 messages sent at 1 second intervals from the SMSC using the WNS software. The module

is only concerned with inbound messages. Receiving 110 messages at 1 second intervals represents an extreme user case and tests the module's ability to log messages not consumed by the bot. Table 3.4 is an overview of the test and measure of success.

Table 3.4: Test 1 - Logging non-C2 Messages

No. of Messages	**Message Type**
110	non-C2
Success: all non-C2 messages identified in kernel log Failure: all non-C2 messages not identified in kernel log	

3.5.3 Test 2: Functionality Test (Logging C2 messages)

This tests the module's basic ability to log C2 messages, messages hidden from the user application by the bot. The workload is the same as in Test 1 where 110 messages are sent in succession at 1 second intervals by the simulated SMSC. Although a poor representation of stealthy botnet C2 communication, it is a simple and relevant method in testing the security module's ability to log bot-related C2 messages. Table 3.5 is an overview of the test and measure of success.

Table 3.5: Test 2 - Logging C2 Messages

No. of Messages	**Message Type**
110	C2
Success: all C2 message identified in kernel log Failure: all C2 message not identified in kernel log	

3.5.4 Test 3: Utility Test (Scenario 1 - Well timed C2)

This tests a specific scenario for a well timed C2 message, or a message that may be difficult to identify in the kernel log using a manual comparison process. A C2 message is sent in this scenario during a time when the user is receiving plenty of messages from other users. Four different senders are utilized. Sender 1, identified by the phone number 12081208, will send only one message. Sender 2, identified by the phone number 26662666, will send two messages. After five minutes, sender 3, identified by the number 13371337, will send 4 messages at 45 second intervals. Sender 4, identified by phone number 40084008, will then send only one message. The last 5 messages will be sent within 5 minutes of each other and the C2 message will be sent within that time frame. Table 3.6 outlines the test sequence and indicates the measure for success.

Table 3.6: Test 3 - Well Timed C2 Sequence of Events

Sender	**No. of Messages**	**Time** ($t = minutes$)
12081208	1	t_0
26662666	2	$t_1 - t_2$
13371337	4	$t_7 - t_{10}$
bot	1	t_8
40084008	1	t_{11}
Success: C2 message identified in kernel log Failure: C2 message not identified in kernel log		

3.5.5 Test 4: Utility Test (Scenario 2 - Poorly timed C2)

This tests a specific scenario for a poorly timed C2 message or a message that is easily detected in the kernel log using a manual comparison process. Here, a C2 message is sent during a period of inactivity. Four different senders are utilized. Sender 1, identified by phone number 13371337, will send 3 messages at 55 second intervals. Sender 2, identified by the phone number 40084008, will send 1 message. After 10 minutes of inactivity, sender 3, identified by the number 26662666, will send 1 message. Sender 4, identified by phone number 12081208, will also send only one message. During the 10 minutes of activity, at minute 8, the bot will send a C2 message. Table 3.7 outlines the test sequence and indicates the measure for success.

Table 3.7: Test 4 - Poorly Timed C2 Sequence of Events

Sender	**No. of Messages**	**Time** (t = *minutes*)
13371337	3	$t_0 - t_2$
40084008	1	t_3
bot	1	t_8
26662666	1	t_{13}
12081208	3	$t_{14} - t_{16}$
Success: C2 message identified in kernel log Failure: C2 message not identified in kernel log		

3.6 Summary

This chapter discussed in detail the design and design considerations for a security module that detects hidden SMS C2 messages on Android-based smartphones. The design

is implemented on a retail HTC Nexus One smartphone. The steps for setting up the development environment and deploying the module to the phone were examined. Finally, the proof of concept is tested to show functionality and utility using an SMS bot in a test environment with a simulated base station and SMSC. The results and analysis are discussed in the following chapter.

4 RESULTS AND ANALYSIS

4.1 Overview

This chapter provides detailed results and analysis for the experiments outlined in Section 3.5 and also provides additional analysis for the overall development, implementation, and testing phases.

A terminal emulator application on the phone (downloaded from the Android Marketplace) is used to provide the screen shots in the figures. A terminal application is not standard on Android phones. The application is used to avoid the need to be connected to a host machine and use the adb utility to gain access to the kernel logs. The kernel logs are displayed to the device using the command, *dmesg | grep Message*, from the terminal. The results were piped through the grep utility and filtered for the word "Message" to filter any other kernel messages that may have been logged during a test run. The results of the *dmesg* command were also sent to a text file whose logs are shown in Appendix C. The focus of the tests and results are based on functionality and utility of the proof of concept. Performance was not a factor in the design or results of the tests.

4.2 Test 1: Functionality Test (Logging non-C2 Messages)

Purpose: The purpose of this test is to provide a base case and show the security module's ability to log all inbound messages. The success criterion is 100% accuracy for the module logging the inbound message events. A missed message results in failure.

Results & Analysis: Figure 4.1 shows the test results in the phone's terminal application. For this particular test, validating all 110 messages using the device's terminal emulator is tedious but achievable. The log data was also sent to a log file to document the results. The log file found in Appendix C clearly shows all 110 inbound message events logged. The stock SMS application was not under test; however, it did

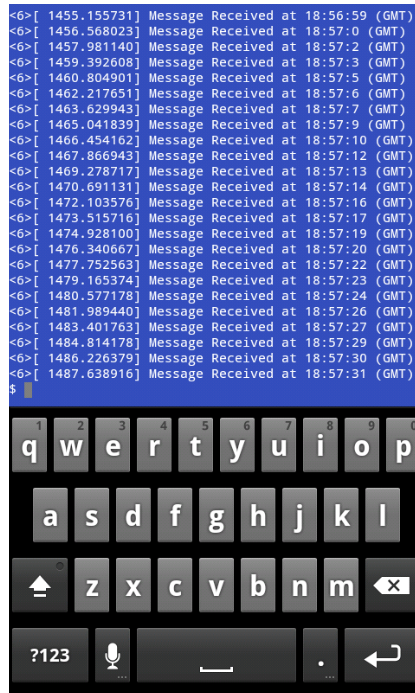


Figure 4.1: Test 1: Terminal Display Showing Kernel Logging non-C2 Messages

receive all 110 messages and shows the module does not disrupt normal operation of the SMS application. This successfully validates the module's ability to log received messages as events in the kernel log.

4.3 Test 2: Functionality Test (Logging C2 Messages)

Purpose: Similar to Test 1, the purpose of this test is to show the security module's ability to log bot-related inbound messages. The only difference from Test 1 is the type of messages being received; only C2 messages are sent in this test. The success criterion for this test is also 100% accuracy at logging C2 messages. A missed message in the log is considered a failure.

Results & Analysis: Figure 4.2 shows the test results in the phone's terminal application. Like Test 1, the results were tedious to validate in the terminal application

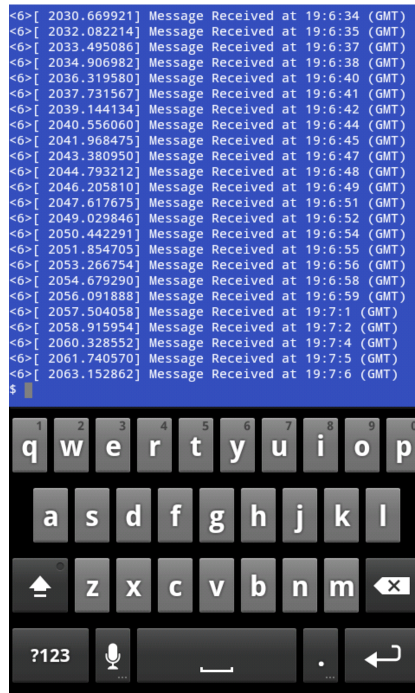


Figure 4.2: Test 2: Terminal Display Showing Kernel Logging C2 Messages

alone, but achievable. The log file in Appendix C clearly shows all 110 inbound message events logged. Zero messages were received by the SMS application successfully validating the bot working and the module's ability to log all inbound messages including C2 messages.

The bot was not under test, however, zero messages were received in the SMS application. This shows the bot working and successfully validated the module's ability to log all inbound messages including C2 messages.

Tests 1 and 2 were designed to strictly show the functionality of the module and validate the ability to independently log both regular messages and C2 messages. The following two tests mix in covert C2 messages with regular messages and shows the utility of being able to log both types.

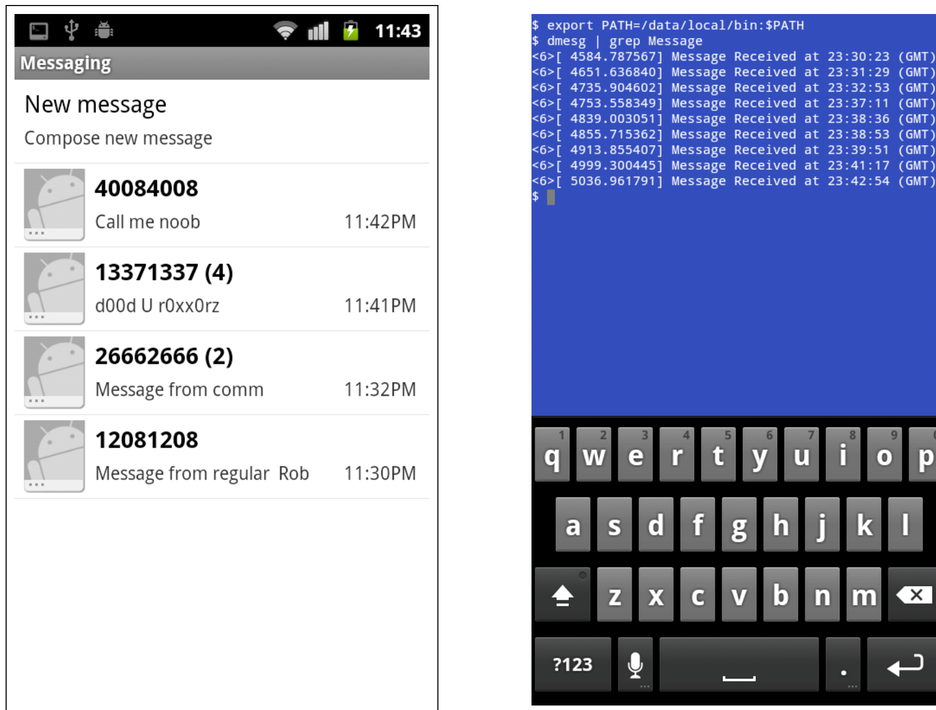


Figure 4.3: Test 3: Side-by-Side Comparison of SMS App and Kernel Log

4.4 Test 3: Utility Test (Scenario 1 - Well timed C2)

Purpose: The manual nature of the comparison process allows for user error in detecting a C2 message even if the message is logged as an event. The purpose of this test is to provide a scenario of a stealthy C2 message by sending the message during a period of activity. This test differs from the previous two tests by mixing C2 messages with regular SMS messages. The success criteria is still whether or not the message is logged; however, this scenario is designed to show the difficulty a user may experience in detecting a covert message if it is sent when the user is receiving a lot of messages at about the same time.

Results & Analysis: Figure 4.4 shows a side-by-side comparison of the test results. The side-by-side analysis shows the SMS application receiving 8 messages in a 12 minute

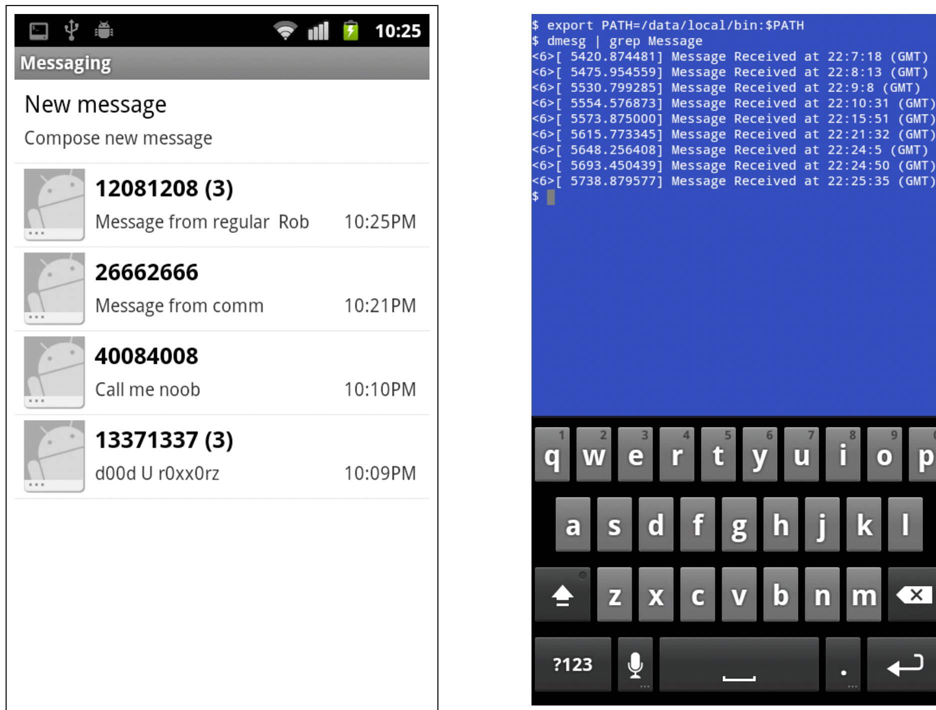


Figure 4.4: Test 4: Side-by-Side Comparison of SMS App and Kernel Log

time frame, from 11:30PM to 11:42PM. A quick look at the kernel log shows "Message Received" events between 11:30PM and 11:42PM; at first glance nothing seems anomalous. A more detailed inspection of the kernel log will show there are a total of 9 events where there are only 8 received messages in the SMS application. Upon further inspection, we can narrow down the suspicious message as one of the 5 messages sent between 11:37 - 11:41PM. Although this shows a scenario for a craftily sent C2 message, the module still logged the event and successfully showed C2 can be detected and distinguished from regular SMS traffic.

4.5 Test 4: Utility Test (Scenario 2 - Poorly Timed C2)

Purpose: The previous test highlights a situation where a logged C2 message can be stealthy given the manual comparison process. The purpose of this test is to provide a

scenario showing a more obvious C2 message by sending the message during a time of reduced SMS activity. As with the previous test, the success and failure is based on the message being logged and not the difficulty or ease of identifying the message in the kernel log.

Results & Analysis: Figure 4.4 shows the results and side-by-side comparison of the test. It is more apparent which of the kernel log events is the C2 message in this side-by-side analysis. The C2 message is not sent outside the time frame of received messages as in the last test where it would be clear to see the "hidden" message. The SMS application shows 3 messages received from user 13371337 with the last one received at 10:09PM. The kernel log shows 3 messages from 10:07 - 10:09PM. The next message is received at 10:10PM as shown in the kernel log. The next inbound message in the SMS application is received 11 minutes later at 10:21PM. Looking at the kernel log, we see the message received at 10:21PM; however, there is also a message received at 10:15PM. This message is not shown in the SMS application and is clearly a hidden message. This test shows a scenario where a hidden message is less difficult to identify in the kernel log and successfully shows the C2 message is still logged as a received message event.

4.6 Further Analysis

This section provides additional analysis on the overall research effort highlighting insights for each step of the process.

4.6.1 Design Analysis

This design for detecting covert SMS messages by capturing them as they are entering the operating system from the modem is successful and provides a starting point for future defensive techniques. The design is specific to the Android platform, but because of the common 2-processor architecture in most modern smartphones, this model can be extended to be applied across multiple platforms.

4.6.2 Implementation Analysis

Implementation from mobile OS to mobile OS will vary and it was realized that implementation within Android's mobile OS platform varies depending on the vendor and specific hardware implementations. The current Android implementation will work on most HTC devices and other Android smartphones using a GSM modem and SMD combination. If the source for a device's kernel is available, and the device uses the GSM/SMD combination, the source code provided in Appendix C can be built with the kernel and loaded onto a system. In general, taking the LKM approach, the process is to obtain the source of the appropriate kernel version, identify which kernel driver the vendor is using as an interface to the modem, identifying which routine is capturing the data before it leaves the kernel, then build and deploy the module. An understanding of the vendor-specific security mechanisms and processes for circumventing or disabling them as identified in Section 3.4 is also required to successfully deploy the module to the device.

4.6.3 Test Analysis

As expected, each test performed showed zero variability in the results. The complete kernel log for only 1 out of the 5 runs for each test is provided in Appendix C. In the two utility tests (Test 3 and 4), it is conceptually simple in such ideal conditions where both the kernel buffer along with the SMS message list is cleared, yet tedious by manual methods to identify C2 by comparing the SMS application's total inbound messages to the total count of inbound message events logged. In a realistic scenario, as time goes on it will be harder to identify C2 messages between the kernel log and list of messages in the application. This was taken into consideration when designing the tests and the way the results were analyzed. The goals of the designed tests were to show basic functionality and utility of the proof of concept and show the design can successfully be implemented on retail hardware. The design of the tests and their results successfully achieved both.

Based on the methodology used in Tests 3 and 4, an automated solution would provide added benefit over a manual comparison process. Implementing this automated process requires an application-level solution that can compare whether a kernel-logged message was received by the application layer. The current implementation is already vendor, hardware, and kernel specific and implementing a user-level application was deemed outside the scope of research. The goal of creating a security module and proving the utility of logging inbound SMS messages at the point where application processor receives it from the radio processor is successfully achieved. More research is required to extend this across all Android devices, at which point, an application layer automation process is warranted.

4.7 Summary

This chapter provides details for the results and analysis of the each of the tests performed and discussed in Section 3.5. It follows with additional analysis for the overall research effort specifically focusing on design, implementation, and test. The next chapter summarizes the entire research effort.

5 CONCLUSIONS AND RECOMMENDATIONS

5.1 Overview

This chapter concludes the research efforts discussed in this thesis. The following sections discuss the significance of the work accomplished, recommendations for future work, and a summary of the entire effort.

5.2 Significance of Research

Botnets are dangerous in the cyber world. They are known to be lucrative in the hands of cyber criminals because of their powerful distributed nature and stealthiness. The smartphone domain shows an increasing trend of computational power comparable to their desktop, laptop, and tablet counterparts. Increasing computational power combined with decreasing costs equates to consumer demand purchasing smartphones at an exponential rate. This translates into the portable and nearly as powerful smartphones replacing traditional laptops and desktops as the future of modern daily computing.

Cyber criminals will undoubtedly remain on the forefront of developing and implementing botnets on smartphones for profit and other personal gain. Cyber defenders must therefore understand the intricacies of botnets in the smartphone domain to better protect against them. This research provides details for implementing a security design concept on Android retail hardware, identifies the important components required for implementation, and realizes that, although Android systems are advertised as open devices, any development beyond the application layer is highly dependent on a vendor's hardware implementation and requires extensive knowledge outside of Android's architecture stack. This research also highlights that the point where the two processors in a smart phone communicate is vulnerable to attack. This research showed how the inter

processor communication could be monitored to detect covert SMS messages; attackers could use similar techniques for malicious intent.

This research extends the work of countermeasures for botnets in the smartphone and mobile computing domain by successfully designing a mechanism to identify covert SMS messages, and implementing on retail hardware.

5.3 Recommendations for Future Research

The design and proof of concept is a stepping stone for more work in the defense against smartphone botnets. Specific ways the proposed implementation can be improved upon are the following:

- **Direct User Notification**: As identified in Section 4.6.3, an automated process would provide added benefit. Several factors are required for this to be accomplished. First, all Android devices would have to be loaded with a modified version of the developed LKM or modified version of the LKM functionality would have to be incorporated into all Android kernels. If this were the case, implementation could be achieved the following way. Rather than logging messages in the kernel's log buffer, the module would send a signal from the kernel to a user-space process. An application would require a segment of C code to accept the signal from the kernel and notify the application. This can be achieved using the Android's Software Development Kit in conjunction with the JNI and Android's Native Development Kit. Then, when notified by the kernel, the application would check to see if a message was received by the SMS application. If a message was not received by the SMS application, a notification would immediately be sent to the user's screen identifying a covert SMS message and potential malicious activity.
- **Detect Hidden Sent Messages**: Modifying the current module to detect hidden outbound messages would help identify bot responses to the botmaster or help

identify specific botnet attacks such as sending premium rate SMS messages or SPAM advertising messages via SMS, which are ways botmasters monetize their smartphone botnet. This could be achieved by using the same jprobe technique as capturing inbound SMS messages. Finding the correct routine to instrument depending on the kernel driver being used is the limiting factor. This was not implemented in this research because the simulated cellular network in the test environment was not set up for user-end equipment to send SMS messages from one device to another or from the device to the simulated SMSC.

- **SMS Filtering**: Inspection of the SMS data can be performed in the current design, but was not implemented. Analyzing the messages in concert with signature-based techniques with known bot-keys can help determine whether or not there is malicious intent in the message. For example, using the SMS bot in this research, the known bot key is “BOT:”. Searching the first four characters of the message data and comparing it to “BOT:” would determine malicious intent. The challenge would be identifying and maintaining an accurate list of bot keys.

- **Whitelisting / Blacklisting**: providing options for a whitelist or blacklist based on process ID, phone number, or other identifying characteristic would begin crossing the threshold of detection and into the realm of mitigation. Like the filtering process, implementing a white or black list can be achieved at the same location and would require a location to store approve or disapproved list items. Storing the list in the kernel would be more secure, but less flexible as updates would require kernel modifications and possibly recompiling and re-deploying to the system. Storing the list in user space, while more flexible, allows more opportunities for tampering.

5.4 Summary

This research identifies a threat with botnets on smartphones and lays the groundwork for addressing the threat by setting and achieving the following goals.

- **Design a security mechanism, complementary to Android's existing security, providing awareness for covert SMS messages, and identifying potential C2 and bot presence on a system.** At the user level, SMS messages can be hidden from user applications. An approach is logging all inbound SMS traffic at the kernel level allowing the user to determine whether or not their SMS application is receiving all messages. A loadable kernel module accessing the data the modem is sending to the kernel is used to achieve this. The implementation does not interfere with any of the device's other functions including the built-in Android platform security.

- **Implement the design as a proof of concept on a retail Android smartphone to identify the steps required.** The loadable kernel module approach is both hardware and kernel specific. The target for development is for an HTC Nexus One smartphone and works with most HTC smartphones or phones using the Shared Memory Driver (SMD) to communicate with the modem.

- **Validate design and proof of concept with an Android SMS bot.** Using a manual comparison process, tests results validate the design and proof of concept on the HTC Nexus One. Comparing the kernel log events to the SMS application shows the design successfully logs C2 messages.

This research succeeded in realizing the goals and sets a foundation for detecting and mitigating SMS botnets on smartphones and extends the research for general security and countermeasures for botnets on mobile computing systems.

APPENDIX A: LOADABLE KERNEL MODULE CODE

```
1  /*
2   * by Robert Olipane, USAF
3   * Feb 2012
4   *
5   * Original code from /samples/kprobes/jprobe_example.c
6   * For more information on theory of operation of jprobes, see
7   * Documentation/kprobes.txt
8   *
9   * Build and insert the kernel module as done in the kprobe example.
10  */
11
12 #include <linux/kernel.h>
13 #include <linux/module.h>
14 #include <linux/kprobes.h>
15 #include <linux/kallsyms.h>
16 #include <linux/string.h>
17 #include <linux/time.h>
18
19 #include <mach/msm_smd.h>
20 #include <mach/msm_iomap.h>
21 #include <mach/system.h>
22
23 #include "smd_private.h"
24
25 /*
26  * Proxy routine having the same arguments as actual ch_read() routine
27  * in /arch/arm/mach-msm/smd.c
28  */
29 static int my_ch_read(struct smd_channel *ch, void *_data, int len)
30 {
31
32   void *ptr;
33   unsigned n;
34   unsigned char *data = _data;
35
36   unsigned long get_time;
37   int sec, hr, min, tmp1, tmp2;
38   struct timeval tv;
39
40   /* get current time for output */
41   do_gettimeofday(&tv);
```

```
42    get_time = tv.tv_sec;
43    sec = get_time % 60;
44    tmp1 = get_time / 60;
45    min = tmp1 % 60;
46    tmp2 = tmp1 / 60;
47    hr = tmp2 % 24;
48    /* end get current time, format hr:min:sec */
49
50    /* parse through data if there is any */
51    while(len > 0) {
52
53      /*** start of ch_read_buffer ***/
54      unsigned head = ch->recv->head;
55      unsigned tail = ch->recv->tail;
56      ptr = (void *) (ch->recv_data + tail);
57
58      if (tail <= head)
59        n = head - tail;
60      else
61        n = ch->fifo_size - tail;
62      /*** end of ch_read_buffer ***/
63
64      if (n == 0)
65        break;
66
67      if (n > len)
68        n = len;
69      if (_data)
70        memcpy(data, ptr, n);
71
72      /* Look for unsolicited AT result code */
73      if (strncmp("+CMT", data, 4) == 0)
74      {
75        printk(KERN_INFO "Message Received at %d:%d:%d (GMT)",hr,min,sec);
76
77        /*
78         * data = unsolicited result code and data
79         * PDU format (7-bit OCTETS)
80         * can see that data here...
81         */
82        //printk(KERN_INFO "SMS: \n %s \n", data);
83        break;
84      }
```

```
85
86       data += n;
87       len -= n;
88     }
89
90     // Always end with a call to jprobe_return().
91     jprobe_return();
92     return 0;
93   }
94
95   static struct jprobe my_jprobe = {
96     .entry = (kprobe_opcode_t *) my_ch_read
97   };
98
99   static int __init jprobe_init(void)
100  {
101    int ret;
102
103    my_jprobe.kp.addr =
104      (kprobe_opcode_t *) kallsyms_lookup_name("ch_read");
105
106    ret = register_jprobe(&my_jprobe);
107    if (ret < 0) {
108      printk(KERN_INFO "register_jprobe failed, returned %d\n", ret);
109      return -1;
110    }
111    printk(KERN_INFO "Planted jprobe at %p, handler addr %p\n",
112           my_jprobe.kp.addr, my_jprobe.entry);
113    return 0;
114  }
115
116  static void __exit jprobe_exit(void)
117  {
118    unregister_jprobe(&my_jprobe);
119    printk(KERN_INFO "jprobe at %p unregistered\n", my_jprobe.kp.addr);
120  }
121
122  module_init(jprobe_init)
123  module_exit(jprobe_exit)
124
125  MODULE_LICENSE("GPL");
126  MODULE_AUTHOR("afitstud");
```

APPENDIX B: DEVELOPMENT ENVIRONMENT

This Appendix is setup with the following sections and instructions:

B.1. Setting up the build environment
B.2. Downloading the Android source
B.3. Downloading and building the Android linux kernel
B.4. Building and inserting LKMs

The components of the development environment include:

- VM running the Ubuntu 10.04 LTS Linux distribution
- Andoid source code (repository branch tag *android-2.3.6_r1*)
- Android Debugger Bridge (adb), provided as part of the Android source
- fastboot, provided as part of the Android source
- Cross compiler toolchains, provided as part of Android source
- Android Linux Kernel source (repository branch tag *msm-2.6.35*)

B.1 Setting up the build environment

Step-by-step instructions for initializing the build environment for the Android source can be found at the AOSP website, http://source.android.com/. It is updated frequently and caterd to the latest verion of the Android OS and the master branch of development. The following are steps taken for the recommended Gingerbread branch for the HTC Nexus One.

1. Download and install Ubuntu 10.04 LTS, 64-bit version: a unix environment is required for development, Google tests and recommends the Ubuntu 10.04 LTS distribution of Linux, support for Mac is provided on the website. VMware Workstation 8.0 was used to create the VM and allows installation from an image file, the Ubuntu 10.04 LTS image was downloaded from the Ubuntu website (http://www.ubuntu.com/download) and the image was created. At the time of the development, the website recommended a minimum of 10GB of HD space with 30GB required for a full build and 50GB set as a cache for speeding up the build process for rebuilds. This development did not require a full build, only the prebuilt toolchains for cross compiling and the adb and fastboot utilities were required.

2. Update and install required packages: Ubuntu has a package update manager that will automatically check installed packages for updates, Google recommends all standard updates. In general, the following packages are required:

    - Python 2.5 - 2.7
    - GNU Make 3.81 - 3.82
    - Java Development Kit (JDK) 6 - for Gingerbread and higher

- Git 1.7 or newer

Additional packages include: git-core, gnupg, flex, bison, gperf, build-essential, zip, curl, zlib1g-dev, libc6-dev, lib32ncurses5-dev, ia32-libs, x11proto-core-dev, libx11-dev, lib32readline5-dev, lib32z-dev, libgl1-mesa-dev, g++-multilib, mingw32, tofrodos, python-markdown, libxml2-utils, xsltproc. The following commands will install the listed packages in Ubuntu from the command line:

Installing JDK

```
$ sudo add-apt-repository ”deb http://archive.canonical.com/ lucid partner”
$ sudo apt-get update
$ sudo apt-get install sun-java6-jdk
```

Installing other pacakges

```
$ sudo apt-get install git-core gnupg flex bison gperf build-essential zip curl zlib1g-dev
libc6-dev lib32ncurses5-dev ia32-libs x11proto-core-dev libx11-dev lib32readline5-dev
lib32z-dev libgl1-mesa-dev g++-multilib mingw32 tofrodos python-markdown libxml2-
utils xsltproc
```

3. Enable USB access: Linux systems need to be configured to access USB devices, the recommended approach is to create a file in /etc/udev/rules.d/ with root priviledges identifying who can access the devices. Vendor-specific codes are required for enabling access. Details of creating this file and list of vendor codes is found at, http://developer.android.com/guide/developing/device.html. Below is a sample *.rules* file (*51-android.rules*), the ”idProduct” attribute is not required for USB access (this attribute is available on the AOSP website for development certain target systems):

Sample .rules file

```
# adb protocol on passion (Nexus One)
SUBSYSTEM==”usb”, ATTR{idVendor}==”18d1”, ATTR{idProduct}==”4e12”, MODE=”0666”, OWNER=”<username>”
# fastboot protocol on passion (Nexus One)
SUBSYSTEM==”usb”, ATTR{idVendor}==”0bb4”, ATTR{idProduct}==”0fff”, MODE=”0600”, OWNER=”<username>”
```

4. At this point, the system is ready for downloading the android source which is required to successfully compile the kernel and develop lkms. The adb and fastboot utilities are in the android source and are used to shell access to devices and flashing images to system partitions, respectively.

B.2 Downloading the Android source

This section discusses the steps required for downloading the Android source. The general steps are downloading and installing the repo utility, create a working directory, check-out a copy of the source code, then synchronizing the files to be worked on locally. Building the source code is not necessary for the purpose of this research. The tools that required in the source are only the ARM compiler and toolchains, fastboot, and adb.

1. Downloading and installing repo: repo is a tool Google created to work with git specifically for Android projects. First, create a local bin directory and add it to the path environment variable, use curl to download repo to the local bin diretory, the repo binary attributes are changed to be executable by all. The list of commands are below.

Downloading and installing repo

```
$ mkdir ˜/bin
$ PATH=˜/bin:$PATH
$ curl https://dl-ssl.googl.com/dl/googlesource/git-repo/repo > ˜/bin/repo
$ chmod a+x ˜/bin/repo
```

2. Obtaining the source: Obtaining the source is just a matter of creating a working directory and checking out the appropriate branch. The AOSP website lists the branches and branch tags available for checkout. The branch checked out in this development was version 2.3.6._r1 (Gingerbread). The repo ”init” and ”sync” commands are used to perform these actions.

Obtaining the source

```
$ mkdir android-2.3.6_r1
$ cd android-2.3.6_r1
$ repo init -u https://android.googlesource.com/platform/manifest -b android-2.3.6_r1
$ repo sync
```

Note1: checking out a branch other than the ”master” branch requires the branch tag preceded by ”-b”, i.e. -b <branch_tag>. To check out the master branch, run the command without it.
Note2: synching the source will take some time depending on the connection, the master branch is approximately 6GB (at time of development).

3. ARM compiler, fastboot, and adb: building the source is not required to build and compile loadable kernel modules. The arm-eabi-gcc compiler is located in the pre-built folder of the root directory. The full path is:

   /prebuilt/linux-x86/toolchain/arm-eabi-4.4.3/bin/

   The two utilities, fastboot and adb, do need to be built. Use make to compile the source, the path to the binaries should be automatically added to the environment path, if not, add it to the path. The full path to fastboot and adb is:

   /out/host/linux-x86/bin/

Building fastboot and adb

$ make fastboot adb
 	
To check your path use
$ echo PATH
 	
To add the path use
$ export PATH=$PATH:<path-to-android-2.3.6_r1>/out/host/linux/bin

Note1: checking out a branch other than the ”master” branch requires the branch tag preceded by ”-b”, i.e. -b <branch_tag> Note2: synching the source will take some time depending on the connection, the master branch is approximately 6GB.

B.3 Downloading and building the linux kernel

Android's linux kernel is not a standard linux kernel, it has been modified for hardware in a mobile environment. Some differences include how it handles inter-process communication (binder process in Android), the use of a shared memory driver for applications, additional power management control, and a custom implementation of the standard C library (known as Bionic). The prerequisites for successfully building the kernel is having the prebuilt toolchains and the right kernel for the hardware. Many HTC products use Qualcomm processors which uses the Android msm kernel, this is the version that will be used in this section. The following are the steps to download, configure and build the kernel.

1. Downloading the kernel: The most difficult part of downloading the kernel is knowing which version to download and which branch. The Nexus One at the time of development was running version 2.6.35. Knowing this, all that is required is using git to clone the repository and check out the 2.6.35 branch.

Building fastboot and adb

```
$ git clone https://android.googlesource.com/kernel/msm.git
$ cd msm
$ git checkout -b android-msm-2.6.35 /origin/android-msm-2.6.35
```

2. Setting environment variables and kernel configuration: To compile successfully, enivronment variables need to be set including adding the path of the toolchains and setting the kernel configurations. The following commands set the environment variables and add the path to the toolchain if not already in the path.

Setting environment variables

```
$ export ARCH=arm
$ export SUBARCH=arm
$ export CROSS_COMPILE=arm-eabi-
$ export PATH=$PATH:<android-2.3.6_r1>/prebuilt/linux-x86/toolchain/arm-eabi-4.4.3/bin
```

There are several ways to configure the kernel options. Running the *make menuconfig* command will bring up a text-based menu for setting all the options in the kernel. For the Nexus One, there is a default kernel configuration named 'mahimahi_defconfig'. Running *make mahimahi_defconfig* will compile the kernel with this default configuration for the Nexus One hardware. Another option is to get the config file from the Nexus One and load it before compiling. For ease, the default mahimahi configuration is used. First the kernel will be built with default Nexus One configuration, then the kernel will be built with kernel probes enabled.

Building the kernel

```
$ make mahimahi_defconfig
$ make menuconfig
```

Note: Depending on the system building the kernel, *make menuconfig* may take a significant amount of time. Figure B.1 shows the text-based menu configuration from running *make menuconf*. Enabling kprobes is under the General setup area.

3. Build the kernel: At this point, the environment and configuration is set to build the Android kernel and begin lkm development using kprobes. From the command line and from the root directory of the kernel, run *make*.

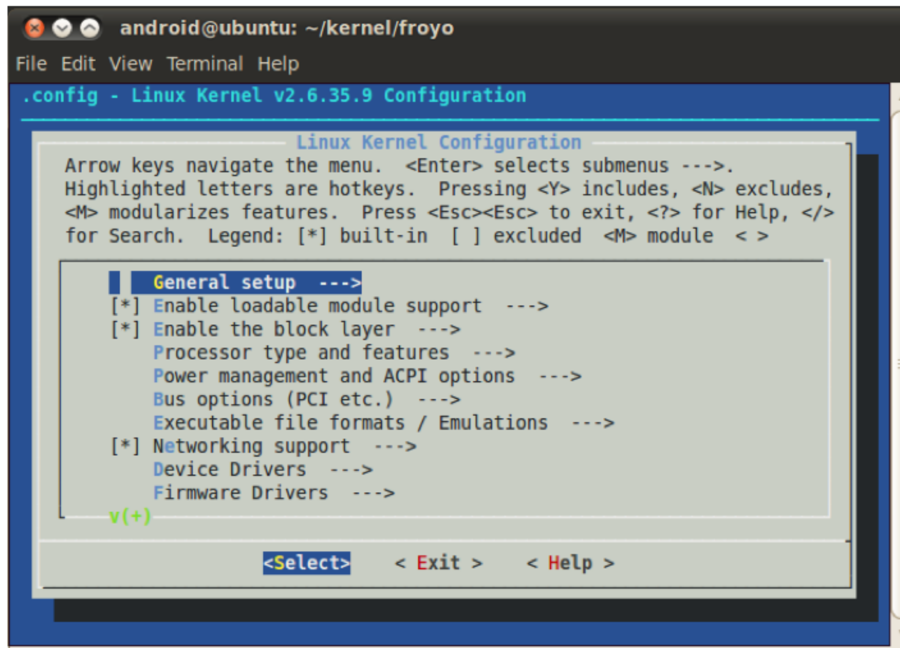


Figure B.1: Linux Kernel Configuration Menu

B.4 Building and inserting LKMs

The previous steps were required in creating an environment to build loadable kernel modules. Building the modules requires the same environment variables set as building the kernel and a make file. The lsmod, insmod, and rmmod commands will list the existing loaded kernel models on the system, load a kernel module to the system, and remove a kernel module from the system, respectively.

Sample LKM Makefile

```
# builds the kprobes example kernel modules
# then to use one (as root): insmod <module_name.ko >

# Update COMPILER path with path to your toolchain compiler location
# Update KERNEL_DIR path to your kernel's root directory location
obj-m := smdmod.o
COMPILER := <path-to-compiler-directory >
CROSS := $(COMPILER)
ARCH := arm
KERNEL_DIR := <path-to-kernel-directory >
PWD := $(shell pwd)

default:
$(MAKE) -C $(KERNEL_DIR) M=$(PWD) ARCH=$(ARCH) CROSS=$(COMPILER)
clean:
$(MAKE) -C $(KDIR) M=$(PWD) clean
```

Running make will create the kernel module in the form of a .ko file. The adb utility is used to push the module to the phone an insert it using the insmod command. If successfully loaded, the lsmod command will show the module as loaded.

APPENDIX C: TEST RESULTS DATA

C.1 Test 1 Log Results

```
1  <6>[ 1909.211914] Message Received at 19:4:33 (GMT)
2  <6>[ 1910.624267] Message Received at 19:4:34 (GMT)
3  <6>[ 1912.036712] Message Received at 19:4:35 (GMT)
4  <6>[ 1913.449188] Message Received at 19:4:37 (GMT)
5  <6>[ 1914.861206] Message Received at 19:4:38 (GMT)
6  <6>[ 1916.273498] Message Received at 19:4:40 (GMT)
7  <6>[ 1917.685729] Message Received at 19:4:41 (GMT)
8  <6>[ 1919.098052] Message Received at 19:4:42 (GMT)
9  <6>[ 1920.510650] Message Received at 19:4:44 (GMT)
10 <6>[ 1921.923034] Message Received at 19:4:45 (GMT)
11 <6>[ 1923.335418] Message Received at 19:4:47 (GMT)
12 <6>[ 1924.747253] Message Received at 19:4:48 (GMT)
13 <6>[ 1926.159851] Message Received at 19:4:49 (GMT)
14 <6>[ 1927.571868] Message Received at 19:4:51 (GMT)
15 <6>[ 1928.984191] Message Received at 19:4:52 (GMT)
16 <6>[ 1930.396606] Message Received at 19:4:54 (GMT)
17 <6>[ 1931.808959] Message Received at 19:4:55 (GMT)
18 <6>[ 1933.221069] Message Received at 19:4:57 (GMT)
19 <6>[ 1934.633392] Message Received at 19:4:58 (GMT)
20 <6>[ 1936.045928] Message Received at 19:4:59 (GMT)
21 <6>[ 1937.458282] Message Received at 19:5:1 (GMT)
22 <6>[ 1938.870880] Message Received at 19:5:2 (GMT)
23 <6>[ 1940.282745] Message Received at 19:5:4 (GMT)
24 <6>[ 1941.694915] Message Received at 19:5:5 (GMT)
25 <6>[ 1943.107238] Message Received at 19:5:6 (GMT)
26 <6>[ 1944.519592] Message Received at 19:5:8 (GMT)
27 <6>[ 1945.931976] Message Received at 19:5:9 (GMT)
28 <6>[ 1947.344238] Message Received at 19:5:11 (GMT)
29 <6>[ 1948.756469] Message Received at 19:5:12 (GMT)
30 <6>[ 1950.169036] Message Received at 19:5:13 (GMT)
31 <6>[ 1951.581237] Message Received at 19:5:15 (GMT)
32 <6>[ 1952.993347] Message Received at 19:5:16 (GMT)
33 <6>[ 1954.405761] Message Received at 19:5:18 (GMT)
34 <6>[ 1955.817993] Message Received at 19:5:19 (GMT)
35 <6>[ 1957.230407] Message Received at 19:5:21 (GMT)
36 <6>[ 1958.642547] Message Received at 19:5:22 (GMT)
37 <6>[ 1960.054840] Message Received at 19:5:23 (GMT)
38 <6>[ 1961.467468] Message Received at 19:5:25 (GMT)
39 <6>[ 1962.879638] Message Received at 19:5:26 (GMT)
```

```
40  <6>[ 1964.291839] Message Received at 19:5:28 (GMT)
41  <6>[ 1965.704071] Message Received at 19:5:29 (GMT)
42  <6>[ 1967.116394] Message Received at 19:5:30 (GMT)
43  <6>[ 1968.529022] Message Received at 19:5:32 (GMT)
44  <6>[ 1969.941314] Message Received at 19:5:33 (GMT)
45  <6>[ 1971.353240] Message Received at 19:5:35 (GMT)
46  <6>[ 1972.766113] Message Received at 19:5:36 (GMT)
47  <6>[ 1974.178161] Message Received at 19:5:37 (GMT)
48  <6>[ 1975.590209] Message Received at 19:5:39 (GMT)
49  <6>[ 1977.003082] Message Received at 19:5:40 (GMT)
50  <6>[ 1978.414825] Message Received at 19:5:42 (GMT)
51  <6>[ 1979.827087] Message Received at 19:5:43 (GMT)
52  <6>[ 1981.239898] Message Received at 19:5:45 (GMT)
53  <6>[ 1982.652191] Message Received at 19:5:46 (GMT)
54  <6>[ 1984.063995] Message Received at 19:5:47 (GMT)
55  <6>[ 1985.476348] Message Received at 19:5:49 (GMT)
56  <6>[ 1986.889251] Message Received at 19:5:50 (GMT)
57  <6>[ 1988.301330] Message Received at 19:5:52 (GMT)
58  <6>[ 1989.713623] Message Received at 19:5:53 (GMT)
59  <6>[ 1991.125518] Message Received at 19:5:54 (GMT)
60  <6>[ 1992.538208] Message Received at 19:5:56 (GMT)
61  <6>[ 1993.950469] Message Received at 19:5:57 (GMT)
62  <6>[ 1995.362487] Message Received at 19:5:59 (GMT)
63  <6>[ 1996.774749] Message Received at 19:6:0 (GMT)
64  <6>[ 1998.187133] Message Received at 19:6:1 (GMT)
65  <6>[ 1999.599334] Message Received at 19:6:3 (GMT)
66  <6>[ 2001.011993] Message Received at 19:6:4 (GMT)
67  <6>[ 2002.423919] Message Received at 19:6:6 (GMT)
68  <6>[ 2003.836334] Message Received at 19:6:7 (GMT)
69  <6>[ 2005.248535] Message Received at 19:6:9 (GMT)
70  <6>[ 2006.660797] Message Received at 19:6:10 (GMT)
71  <6>[ 2008.073120] Message Received at 19:6:11 (GMT)
72  <6>[ 2009.485931] Message Received at 19:6:13 (GMT)
73  <6>[ 2010.897705] Message Received at 19:6:14 (GMT)
74  <6>[ 2012.310180] Message Received at 19:6:16 (GMT)
75  <6>[ 2013.722320] Message Received at 19:6:17 (GMT)
76  <6>[ 2015.134704] Message Received at 19:6:18 (GMT)
77  <6>[ 2016.547698] Message Received at 19:6:20 (GMT)
78  <6>[ 2017.959228] Message Received at 19:6:21 (GMT)
79  <6>[ 2019.371734] Message Received at 19:6:23 (GMT)
80  <6>[ 2020.783935] Message Received at 19:6:24 (GMT)
81  <6>[ 2022.196533] Message Received at 19:6:25 (GMT)
82  <6>[ 2023.608520] Message Received at 19:6:27 (GMT)
```

```
83  <6>[ 2025.020965] Message Received at 19:6:28 (GMT)
84  <6>[ 2026.433044] Message Received at 19:6:30 (GMT)
85  <6>[ 2027.845489] Message Received at 19:6:31 (GMT)
86  <6>[ 2029.258148] Message Received at 19:6:32 (GMT)
87  <6>[ 2030.669921] Message Received at 19:6:34 (GMT)
88  <6>[ 2032.082214] Message Received at 19:6:35 (GMT)
89  <6>[ 2033.495086] Message Received at 19:6:37 (GMT)
90  <6>[ 2034.906982] Message Received at 19:6:38 (GMT)
91  <6>[ 2036.319580] Message Received at 19:6:40 (GMT)
92  <6>[ 2037.731567] Message Received at 19:6:41 (GMT)
93  <6>[ 2039.144134] Message Received at 19:6:42 (GMT)
94  <6>[ 2040.556060] Message Received at 19:6:44 (GMT)
95  <6>[ 2041.968475] Message Received at 19:6:45 (GMT)
96  <6>[ 2043.380950] Message Received at 19:6:47 (GMT)
97  <6>[ 2044.793212] Message Received at 19:6:48 (GMT)
98  <6>[ 2046.205810] Message Received at 19:6:49 (GMT)
99  <6>[ 2047.617675] Message Received at 19:6:51 (GMT)
100 <6>[ 2049.029846] Message Received at 19:6:52 (GMT)
101 <6>[ 2050.442291] Message Received at 19:6:54 (GMT)
102 <6>[ 2051.854705] Message Received at 19:6:55 (GMT)
103 <6>[ 2053.266754] Message Received at 19:6:56 (GMT)
104 <6>[ 2054.679290] Message Received at 19:6:58 (GMT)
105 <6>[ 2056.091888] Message Received at 19:6:59 (GMT)
106 <6>[ 2057.504058] Message Received at 19:7:1 (GMT)
107 <6>[ 2058.915954] Message Received at 19:7:2 (GMT)
108 <6>[ 2060.328552] Message Received at 19:7:4 (GMT)
109 <6>[ 2061.740570] Message Received at 19:7:5 (GMT)
110 <6>[ 2063.152862] Message Received at 19:7:6 (GMT)
```

C.2 Test 2 Log Results

```
1  <6>[  260.065032] Message Received at 21:5:22 (GMT)
2  <6>[  261.477386] Message Received at 21:5:23 (GMT)
3  <6>[  262.887359] Message Received at 21:5:25 (GMT)
4  <6>[  264.302001] Message Received at 21:5:26 (GMT)
5  <6>[  265.712890] Message Received at 21:5:28 (GMT)
6  <6>[  267.126617] Message Received at 21:5:29 (GMT)
7  <6>[  268.538909] Message Received at 21:5:30 (GMT)
8  <6>[  269.950592] Message Received at 21:5:32 (GMT)
9  <6>[  271.363555] Message Received at 21:5:33 (GMT)
10 <6>[  272.775848] Message Received at 21:5:35 (GMT)
11 <6>[  274.185913] Message Received at 21:5:36 (GMT)
12 <6>[  275.600524] Message Received at 21:5:38 (GMT)
13 <6>[  277.012786] Message Received at 21:5:39 (GMT)
```

```
14 <6>[   278.422943] Message Received at 21:5:40 (GMT)
15 <6>[   279.837371] Message Received at 21:5:42 (GMT)
16 <6>[   281.247344] Message Received at 21:5:43 (GMT)
17 <6>[   282.662078] Message Received at 21:5:45 (GMT)
18 <6>[   284.071960] Message Received at 21:5:46 (GMT)
19 <6>[   285.484252] Message Received at 21:5:47 (GMT)
20 <6>[   286.896728] Message Received at 21:5:49 (GMT)
21 <6>[   288.308868] Message Received at 21:5:50 (GMT)
22 <6>[   289.721191] Message Received at 21:5:52 (GMT)
23 <6>[   291.135833] Message Received at 21:5:53 (GMT)
24 <6>[   292.548126] Message Received at 21:5:54 (GMT)
25 <6>[   293.960449] Message Received at 21:5:56 (GMT)
26 <6>[   295.370452] Message Received at 21:5:57 (GMT)
27 <6>[   296.783569] Message Received at 21:5:59 (GMT)
28 <6>[   298.197387] Message Received at 21:6:0 (GMT)
29 <6>[   299.609710] Message Received at 21:6:2 (GMT)
30 <6>[   301.022003] Message Received at 21:6:3 (GMT)
31 <6>[   302.432006] Message Received at 21:6:4 (GMT)
32 <6>[   303.844299] Message Received at 21:6:6 (GMT)
33 <6>[   305.256591] Message Received at 21:6:7 (GMT)
34 <6>[   306.671234] Message Received at 21:6:9 (GMT)
35 <6>[   308.083526] Message Received at 21:6:10 (GMT)
36 <6>[   309.495880] Message Received at 21:6:11 (GMT)
37 <6>[   310.908142] Message Received at 21:6:13 (GMT)
38 <6>[   312.321960] Message Received at 21:6:14 (GMT)
39 <6>[   313.732757] Message Received at 21:6:16 (GMT)
40 <6>[   315.142730] Message Received at 21:6:17 (GMT)
41 <6>[   316.557373] Message Received at 21:6:18 (GMT)
42 <6>[   317.971130] Message Received at 21:6:20 (GMT)
43 <6>[   319.379669] Message Received at 21:6:21 (GMT)
44 <6>[   320.794281] Message Received at 21:6:23 (GMT)
45 <6>[   322.204254] Message Received at 21:6:24 (GMT)
46 <6>[   323.618865] Message Received at 21:6:26 (GMT)
47 <6>[   325.031188] Message Received at 21:6:27 (GMT)
48 <6>[   326.441192] Message Received at 21:6:28 (GMT)
49 <6>[   327.853515] Message Received at 21:6:30 (GMT)
50 <6>[   329.265930] Message Received at 21:6:31 (GMT)
51 <6>[   330.678100] Message Received at 21:6:33 (GMT)
52 <6>[   332.090423] Message Received at 21:6:34 (GMT)
53 <6>[   333.502899] Message Received at 21:6:35 (GMT)
54 <6>[   334.915039] Message Received at 21:6:37 (GMT)
55 <6>[   336.328247] Message Received at 21:6:38 (GMT)
56 <6>[   337.742309] Message Received at 21:6:40 (GMT)
```

```
57 <6>[  339.154266] Message Received at 21:6:41 (GMT)
58 <6>[  340.564453] Message Received at 21:6:42 (GMT)
59 <6>[  341.978851] Message Received at 21:6:44 (GMT)
60 <6>[  343.388854] Message Received at 21:6:45 (GMT)
61 <6>[  344.801269] Message Received at 21:6:47 (GMT)
62 <6>[  346.216247] Message Received at 21:6:48 (GMT)
63 <6>[  347.628112] Message Received at 21:6:49 (GMT)
64 <6>[  349.040405] Message Received at 21:6:51 (GMT)
65 <6>[  350.450469] Message Received at 21:6:52 (GMT)
66 <6>[  351.862701] Message Received at 21:6:54 (GMT)
67 <6>[  353.277313] Message Received at 21:6:55 (GMT)
68 <6>[  354.689666] Message Received at 21:6:57 (GMT)
69 <6>[  356.099639] Message Received at 21:6:58 (GMT)
70 <6>[  357.514617] Message Received at 21:6:59 (GMT)
71 <6>[  358.924316] Message Received at 21:7:1 (GMT)
72 <6>[  360.338867] Message Received at 21:7:2 (GMT)
73 <6>[  361.751190] Message Received at 21:7:4 (GMT)
74 <6>[  363.163513] Message Received at 21:7:5 (GMT)
75 <6>[  364.573547] Message Received at 21:7:6 (GMT)
76 <6>[  365.985778] Message Received at 21:7:8 (GMT)
77 <6>[  367.400421] Message Received at 21:7:9 (GMT)
78 <6>[  368.812713] Message Received at 21:7:11 (GMT)
79 <6>[  370.222717] Message Received at 21:7:12 (GMT)
80 <6>[  371.635711] Message Received at 21:7:13 (GMT)
81 <6>[  373.049652] Message Received at 21:7:15 (GMT)
82 <6>[  374.459625] Message Received at 21:7:16 (GMT)
83 <6>[  375.871887] Message Received at 21:7:18 (GMT)
84 <6>[  377.284210] Message Received at 21:7:19 (GMT)
85 <6>[  378.699249] Message Received at 21:7:21 (GMT)
86 <6>[  380.108856] Message Received at 21:7:22 (GMT)
87 <6>[  381.521148] Message Received at 21:7:23 (GMT)
88 <6>[  382.933502] Message Received at 21:7:25 (GMT)
89 <6>[  384.345764] Message Received at 21:7:26 (GMT)
90 <6>[  385.761932] Message Received at 21:7:28 (GMT)
91 <6>[  387.174072] Message Received at 21:7:29 (GMT)
92 <6>[  388.582702] Message Received at 21:7:30 (GMT)
93 <6>[  389.998809] Message Received at 21:7:32 (GMT)
94 <6>[  391.407531] Message Received at 21:7:33 (GMT)
95 <6>[  392.822784] Message Received at 21:7:35 (GMT)
96 <6>[  394.231933] Message Received at 21:7:36 (GMT)
97 <6>[  395.644226] Message Received at 21:7:37 (GMT)
98 <6>[  397.056518] Message Received at 21:7:39 (GMT)
99 <6>[  398.469207] Message Received at 21:7:40 (GMT)
```

```
100 <6>[  399.883483] Message Received at 21:7:42 (GMT)
101 <6>[  401.295776] Message Received at 21:7:43 (GMT)
102 <6>[  402.705871] Message Received at 21:7:45 (GMT)
103 <6>[  404.120422] Message Received at 21:7:46 (GMT)
104 <6>[  405.530487] Message Received at 21:7:47 (GMT)
105 <6>[  406.943572] Message Received at 21:7:49 (GMT)
106 <6>[  408.355590] Message Received at 21:7:50 (GMT)
107 <6>[  409.769622] Message Received at 21:7:52 (GMT)
108 <6>[  411.179626] Message Received at 21:7:53 (GMT)
109 <6>[  412.594268] Message Received at 21:7:54 (GMT)
110 <6>[  414.006561] Message Received at 21:7:56 (GMT)
```

C.3 Test 3 Log Results

```
1 <6>[ 4584.787567] Message Received at 23:30:23 (GMT)
2 <6>[ 4651.636840] Message Received at 23:31:29 (GMT)
3 <6>[ 4735.904602] Message Received at 23:32:53 (GMT)
4 <6>[ 4753.558349] Message Received at 23:37:11 (GMT)
5 <6>[ 4839.003051] Message Received at 23:38:36 (GMT)
6 <6>[ 4855.715362] Message Received at 23:38:53 (GMT)
7 <6>[ 4913.855407] Message Received at 23:39:51 (GMT)
8 <6>[ 4999.300445] Message Received at 23:41:17 (GMT)
9 <6>[ 5036.961791] Message Received at 23:42:54 (GMT)
```

C.4 Test 4 Log Results

```
1 <6>[ 5420.874481] Message Received at 22:7:18 (GMT)
2 <6>[ 5475.954559] Message Received at 22:8:13 (GMT)
3 <6>[ 5530.799285] Message Received at 22:9:8 (GMT)
4 <6>[ 5554.576873] Message Received at 22:10:31 (GMT)
5 <6>[ 5573.875000] Message Received at 22:15:51 (GMT)
6 <6>[ 5615.773345] Message Received at 22:21:32 (GMT)
7 <6>[ 5648.256408] Message Received at 22:24:5 (GMT)
8 <6>[ 5693.450439] Message Received at 22:24:50 (GMT)
9 <6>[ 5738.879577] Message Received at 22:25:35 (GMT)
```

BIBLIOGRAPHY

[1] 3GPP/ETSI. "Technical realization of the Short Message Service (SMS)", 2011. [ONLINE]. Available: http://www.3gpp.org/ftp/Specs/html-info/23040.htm.

[2] Andrus, J., C. Dall, A. V. Hof, O. Laadan, and J. Nieh. "Cells: a virtual mobile smartphone architecture". *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP), 2011*, 173–187. ACM, 2011.

[3] AOSP. "Radio Interface Layer", 2011. [ONLINE]. Available: http://www.kandroid.org/online-pdk/guide/telephony.html.

[4] Canalys. "Smartphones overtake client PCs in 2011", 2012. [ONLINE]. Available: http://www.canalys.com/newsroom/smart-phones-overtake-client-pcs-2011.

[5] Canavan, J. "The evolution of malicious IRC bots". *Virus Bulletin Conference*, 104–114. Citeseer, 2005.

[6] Dagon, D., G. Gu, C. P. Lee, and W. Lee. "A taxonomy of botnet structures". *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC), 2007*, 325–339. IEEE Computer Society, 2007.

[7] Dagon, D., G. Gu, C. Zou, J. Grizzard, S. Dwivedi, W. Lee, and R. Lipton. "A taxonomy of botnets". *Unpublished paper, c*, 2005.

[8] Developers, Android. "What is Android?", October 2011. [ONLINE]. Available: http://developer.android.com/guide/basics/what-is-android.html.

[9] developershome.com. "Introduction to SMS Messaging", 2011. [ONLINE]. Available: http://www.developershome.com/sms/smsIntro.asp.

[10] Enck, W., D. Octeau, P. McDaniel, and S. Chaudhuri. "A study of Android application security". *Proceedings of the 20th USENIX Security Symposium, 2011*. 2011.

[11] Enck, W., M. Ongtang, and P. McDaniel. "Understanding Android Security". *Security and Privacy, IEEE*, 7(1):50–57, 2009.

[12] Fabian, M. A. R. J. Z. and M. A. Terzis. "A Multifaceted Approach to Understanding the Botnet Phenomenon". *Proceedings of the 2006 ACM SIGCOMM Internet Measurement Conference (IMC), 2006*. 2006.

[13] Feily, M., A. Shahrestani, and S. Ramadass. "A Survey of Botnet and Botnet Detection". *Proceedings of the Third International Conference on Emerging Security Information, Systems and Technologies (SECURWARE), 2009*, 268–273. 2009.

[14] Gayomali, C. "Jaw-Dropper: 18 to 24 Year Olds Average 110 Text Messages per Day", 2011. [ONLINE]. Available: http://techland.time.com/2011/09/20/jaw-dropper-18-to-24-year-olds-average-110-text-messages-per-day/.

[15] Giles, J. "Virus May Signal First 'Zombie' Cellphone Network", 2009. [ONLINE]. Available: http://abcnews.go.com/Technology/AheadoftheCurve/story?id=8112308&page=1.

[16] Goertzel, K. M. and T. Winograd. *Tools Report on Anti-Malware*. Technical report, 2009.

[17] Hachem, N., Y. Ben Mustapha, G. G. Granadillo, and H. Debar. "Botnets: Lifecycle and Taxonomy". *Proceedings of the Conference on Network and Information Systems Security (SAR-SSI), 2011*, 1–8. 2011.

[18] Hartmann, J. Eberspacher; H. Vogel; C. Bettstetter; C. *GSM - Architecture, Protocols, and Services 3rd Ed.* John Wiley and Sons, 2009.

[19] Keniston, Panchamukhi P. Hiramatsu M., J. "Kernel Probes (Kprobes)", 2012. [ONLINE]. Available: http://www.kernel.org/doc/Documentation/kprobes.txt.

[20] Kok, J. and B. Kurz. "Analysis of the BotNet Ecosystem". *CTTE 2011*, 2011.

[21] Micro, T. "Taxonomy of botnet threats". *Trend Micro Enterprise Security Library*, 2006.

[22] Mulliner, C. and C. Miller. "Fuzzing the phone in your phone". *Black Hat (June 2009)*, 2009.

[23] Oberheide, J. "Android hax". *SummerCon, July*, 2010.

[24] Open Handset Alliance, (OHA). "Open Handset Alliance: Android", 2011. [ONLINE]. Available: http://www.openhandsetalliance.com/.

[25] Porras, P., H. Saïdi, and V. Yegneswaran. "An Analysis of the iKee. B iPhone Botnet". *Security and Privacy in Mobile Information and Communication Systems*, 141–152, 2010.

[26] Puri, R. "Bots and botnet: An overview". *SANS Institute 2003*, 2003.

[27] Schmidt, A. D., H. G. Schmidt, L. Batyuk, J. H. Clausen, S. A. Camtepe, S. Albayrak, and C. Yildizli. "Smartphone malware evolution revisited: Android next target?" *Proceedings of the 4th International Conference on Malicious and Unwanted Software (MALWARE), 2009*, 1–7. IEEE, 2009.

[28] Security., Lookout Mobile. *Lookout Mobile Security Technical Tear Down: Droiddream, Payload One and Payload Two*. Technical report, 2011.

[29] Shabtai, A., Y. Fledel, U. Kanonov, Y. Elovici, and S. Dolev. "Google Android: A state-of-the-art review of security mechanisms". *CoRR abs/0912.5101*, 2009.

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

[31] Singh, K., S. Sangal, N. Jain, P. Traynor, and W. Lee. "Evaluating bluetooth as a medium for botnet command and control". *Detection of Intrusions and Malware, and Vulnerability Assessment*, 61–80, 2010.

[32] Strazzere, T. "Security Alert: Malware Found Targeting Custom ROMs (jSMSHider)", 2011. [ONLINE]. Available: http://blog.mylookout.com/2011/06/security-alert-malware-found-targeting-custom-roms-jsmshider/.

[33] Wang, J. "Android Radio Interface Layer", 2011. [ONLINE]. Available: http://www.slideshare.net/ssusere3af56/android-radio-layer-interface.

[34] Weidman, G. "Transparent Botnet Command and Control for Smartphones over SMS Shmoocon 2011". 2011 2011.

[35] Welte, H. "Anatomy of contemporary GSM cellphone hardware". *Unpublished paper, c*, 2010.

[36] Zeidanloo, H. R., M. J. Z. Shooshtari, P. V. Amoli, M. Safari, and M. Zamani. "A taxonomy of Botnet detection techniques". *Proceedings of the 3rd IEEE International Conference on Computer Science and Information Technology (ICCSIT), 2010*, volume 2, 158–162. 2010.

[37] Zou, C. C. and R. Cunningham. "Honeypot-aware advanced botnet construction and maintenance". *Proceedings of the International Conference on Dependable Systems and Networks (DSN), 2006*, 199–208. IEEE, 2006.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704–0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202–4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**

1. REPORT DATE *(DD–MM–YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From — To)*
14–06–2012	Master's Thesis	Sep 2010 — 14 Jun 2012

4. TITLE AND SUBTITLE

Short Message Service (SMS) Command and Control (C2) Awareness in Android-based Smartphones Using Kernel-Level Auditing

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Olipane, Robert J., Capt, 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/GCO/ENG/12-21

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

Intentionally left blank

10. SPONSOR/MONITOR'S ACRONYM(S)

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

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A: APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

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

14. ABSTRACT

This thesis addresses the emerging threat of botnets in the smartphone domain and focuses on the Android platform and botnets using short message service (SMS) as the command and control (C2) channel. With any botnet, C2 is the most important component contributing to its overall resilience, stealthiness, and effectiveness. This thesis develops a passive host-based approach for identifying covert SMS traffic and providing awareness to the user. Modifying the kernel and implementing this awareness mechanism is achieved by developing and inserting a loadable kernel module that logs all inbound SMS messages as they are sent from the baseband radio to the application processor. The design is successfully implemented on an HTC Nexus One Android smartphone and validated with tests using an Android SMS bot from the literature. The module successfully logs all messages including bot messages that are hidden from user applications. Suspicious messages are then identified by comparing the SMS application message list with the kernel log's list of events. This approach lays the groundwork for future host-based countermeasures for smartphone botnets and SMS-based botnets.

15. SUBJECT TERMS

SMS, Android, Botnet, Smartphone

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Dr. Robert F. Mills (ENG)
U	U	U	UU	88	**19b. TELEPHONE NUMBER** *(include area code)* (937) 255-3636, x4527; robert.mills@afit.edu