1-1-2014

A New Covert Channel Over Cellular Network Voice Channel

Bushra Sulaiman Aloraini

A New Covert Channel Over Cellular Network Voice Channel

by

Bushra Sulaiman Aloraini

Committee Members

Daryl Johnson

Bill Stackpole

Sumita Mishra

Thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science in Networking and System Administrations

Rochester Institute of Technology

B. Thomas Golisano College

of

Computing and Information Sciences

Department of Networking, Security, and Systems Administration

8/4/2014

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude and appreciation to all those who provided me the opportunity to complete this research. A special appreciation to my committee chair, Professor Daryl Johnson, for providing me with well-rounded skills appropriate for my long-term career goals and providing me with the foundation to become an instructor and an independent thinker. I would like to thank my committee members, Professor Bill Stackpole and Professor Sumita Mishra, for the invaluable knowledge, encouragement, and assistance that they provided at all levels of the research.

I would also like to thank my beloved parents, Sulaiman and Fatima, for all of the ultimate sacrifices that they have made on my behalf. Their support and love helped me to gain ambition and an ability to tackle challenges. I would also like to thank my sisters and brothers, as they were always supporting and incenting me to strive towards my goal. In particular, I want to acknowledge my husband and best friend, Waleed, without whose love, encouragement, and assistance, I would not have finished this research. In conclusion, I recognize that this research would not have been possible without the financial assistance of PNU, Princes Nora University.

A NEW COVERT CHANNEL OVER CELLULAR NETWORK VOICE CHANNEL

BUSHRA ALORAINI

ABSTRACT

Smartphone security has become increasingly more significant as smartphones become a more important part of many individuals' daily lives. Smartphones undergo all computer security issues; however, they also introduce a new set of security issues as various capabilities are added. Smartphone security researchers pay more attention to security issues inherited from the traditional computer security field than smartphone-related security issues. The primary network that smartphones are connected to is the cellular network, but little effort has been directed at investigating the potential security issues that could threaten this network and its end users.

A new possible threat that could occur in the cellular network is introduced in this paper. This research proves the ability to use the cellular network voice channel as a covert channel that can convey covert information as speech, thus breaking the network policies. The study involves designing and implementing multiple subsystems in order to prove the theory. First, a software audio modem that is able to convert digital data into audio waves

and inject the audio waves to the GSM voice channel was developed. Moreover, a user-mode rootkit was implemented in order to open the voice channels by stealthily answering the incoming voice call, thus breaking the security mechanisms of the smartphone.

Multiple scenarios also were tested in order to verify the effectiveness of the proposed covert channel. The first scenario is a covert communication between two parties that intends to hide their communications by using a network that is unknown to the adversary and not protected by network security guards. The two parties communicate through the cellular network voice channel to send and receive text messages. The second scenario is a side channel that is able to leak data such as SMS or the contact of a hacked smartphone through the cellular network voice channel. The third scenario is a botnet system that uses the voice channel as command and control channel (C2). This study identifies a new potential smartphone covert channel, so the outcome should be setting countermeasures against this kind of breach.

TABLE OF CONTENTS

NOMENCLATURE

ADB: Android debug bridge

ADC: Analog-to-digital converter

AP: Application processor

API: Application programming interfaces

ART: Android runtime

BP: Baseband processor

C2: Command and control

CDMA: Code division multiple access

DAC: Digital –to-analog

DEX: Dalvik executable

DTX: Discontinuous transmission

DVM: Dalvik virtual machine

FFT: Fast fourier transform

FSK: Frequency-shift keying

GSM: Global system for mobile communications, originally groupe spécial mobile

HAL: Audio hardware interface

ICC: Inter-component communication

JDK: Java development kit

JIT: Just in time

LTE: Long-term evolution

MAC: Mandatory access control

PCM: Pulse code modulation

RIL: Radio interface layer

RTOS: Real-time operating system

SDK: Software development kit

UMTS: Universal mobile telecommunications system

VAD: Voice activity detection

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

This chapter gives an introduction to the research undertaken in this thesis. It discusses the current problem and motivation for conducting this work, describes the aims and objectives of the research as well as the work assumptions and limitations, introduces an overview of the methodologies taken, and, finally, provides the structure of the rest of the paper.

1.1 Thesis Motivation

Smartphones have become increasingly ubiquitous and have replaced the role of personal computers because these phones introduce great convenience. In fact, smartphones have a unique nature as they save more personal data, have "always-on" network connectivity, and have "location-aware" capability, as well as contain more

personal data and private information than PCs. As a result, the security of these devices becomes increasingly important.

Smartphones usually are linked to many communication networks, such as Bluetooth, near field communication (NFC), and mainly Wi-Fi and cellular networks. Computer security researchers have shed some light on the PC security issues that have crept into the smartphone world, which usually are Wi-Fi related; however, little effort has been directed at studying the potential security breaches of the cellular network. Therefore, this research discusses how the cellular voice channel can form a covert channel for legitimate message exchange, information leakage, and malicious code distribution.

1.2 Problem Statement

A study shows that worldwide smartphone numbers exceed the number of PDAs and PCs combined [1], which makes their threats and risks more serious compared with other devices. Indeed, smartphone security is a thorny problem, as there are multiple parties that bear part of the security responsibility, such as smartphone hardware manufacturers, cellular service providers, smartphone software security companies, smartphone operating system security mechanisms, poor implementation of cellular communication standards, and potentially the users themselves!

When GSM, CDMA, and LTE standards were introduced, attacks against cell phones were not much of a concern; these standards became a dangerous gateway for

security threats. Also, the fact that there are no security countermeasures, such as firewalls and IDS to guard the cellular voice traffic in the core of the cellular network, makes these channels a good choice to launch such an attack.

Yet smartphone software security companies could not provide an effective solution to protect smartphone security due to the complexity of their communications and the lack of resources when compared to PCs, such as battery life, processor, and RAM capabilities. Therefore, solutions inspired by PCs do not fit smartphone security needs well, especially when addressing the cellular network.

On the other hand, smartphone hardware manufacturers keep their software implementation as closed secrets; however, they do not use proper security measures. As a result, multiple security issues have occurred that could exploit these secrets. All the issues described above contribute to the growth of smartphone security breaches.

Additionally, because cellular voice channel is not designed to convey data but rather voice, smartphone operating systems do not employ security mechanisms in order to prevent a potential threat that may come from the cellular voice channel. This kind of threat has never been attempted previously. What makes the situation even worse is the lack of security research that addresses the potential cellular network security breaches.

This research introduces one of the potential security breaches that is using cellular voice channel to carry out a covert communication in order to leak information or spread malware to smartphones. The potential threat of using cellular voice channel as a covert

channel is a general threat for all types of smartphone operating systems; however, an Android operating system was chosen to investigate the issue because of its openness.

1.3 Thesis Objectives

In the smartphone security field, examining the cellular voice channel as a covert channel has not been studied yet. Therefore, this channel was examined because of its ability to be a potential medium of information leakage and malware distribution through carrying modulated “speech-like” data covertly.

The research has four main objectives as follows:

1. Investigating the ability of the cellular voice channel to compose a smartphone covert channel.
2. Implementing an audio modem to convert digital data to analog waves to be carried on the cellular voice channel.
3. Building a smartphone rootkit that is able to stealthily open the channel and exchange data.
4. Examining Android’s operating system security mechanisms against this kind of threat.

1.4 Research Methodology

Many subsystems were developed during this research to prove the ability to use cellular voice channel as a covert channel able to carry modulated data in order to leak data or distribute malware. First, a modem that was able to convert digital data to a "speech-like" wave to be carried through the cellular voice channel was implemented. In addition, a user-mode rootkit was developed to answer an incoming voice call covertly from a specific phone number and to leak information such as SMS or to receive commands simulating a botnet system. Furthermore, Android security mechanisms were bypassed in order to develop this kind of system.

The systems were implemented using Android studio development environment and validated using Galaxy Nexus and Galaxy S 3 Android smartphones. In addition, the systems are able to run on a rooted official stock Android ROM and a custom Android ROM. The T-Mobile GSM circuit switched core also was chosen to validate this kind of covert channel. Further details about system development are mentioned in Chapters 3 and 4.

1.5 Assumptions and Limitations

Several assumptions and limitations are accepted in order to achieve the goals.

1.5.1 The assumptions

Assumption 1: The developed system could be implemented on any cellular communication system such as GSM, CDMA, and LTE. However, this research was validated in only GSM voice channel.

Assumption 2: The audio modem could be implemented to simulate a natural sound such as a bird tweeting, a piano playing, and a cricket chirping to increase the secrecy of the covert channel. However, the audio modem was implemented using the traditional modulation technique named Frequency-shift keying (FSK) modem.

1.5.2 The limitations

Limitation 1: The developed system in this research was only validated with GSM core network. Other cellular standards have not been tested.

Limitation 2: Because the cellular network voice channel is designed to carry frequencies that fit into bandwidths between 300Hz and 3400Hz, and the smartphone audio hardware is designed to play back and capture frequencies that are audible to the average human, the developed audio modem communications could be heard. However, this modem could be made to simulate nature voices such as birds or crickets that will increase the covertness but reduce the channel capacity.

Limitation 3: The current system implementation is hardware specific and depends on certain smartphone devices. The current implementation is successful on a Galaxy Nexus and Galaxy S 3, and it works on most Galaxy S models.

Limitation 4: The system only provides proof of concept and does not take any steps towards mitigation or suppression.

1.6 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 provides background information that discusses cellular network infrastructure and traffic flow, cellular voice channel characteristics, and smartphone architecture. Chapter 2 also explores Android OS in terms of the Android architecture, Android security, Android Radio Interface Layer (RIL), and the Android media system. An overview of covert channel and the related security issues such as rootkit, botnet, and data exfiltration are additionally highlighted. Also provided is a literature review of works related to smartphone malware, covert channel phenomenon, and carrying data over the cellular voice channel.

Chapter 3 describes the methodology that was used to conduct the research work. The chapter also provides the system design and implementation to prove the concept of using the cellular voice channel as a covert channel. The chapter then provides the tested scenarios and the development environment. Then Chapter 4 analyzes the system and verifies the proof of concept. Finally, Chapter 5 provides the research work summary and recommends areas for further research.

CHAPTER II

BACKGROUND INFORMATION AND LITERATURE REVIEW

2.1 Introduction

Investigating the use of the cellular network voice channel as a covert channel on Android smartphones requires an understanding of the manifold aspects of the subject. In this chapter, an overview of these aspects is highlighted. First, an overview of cellular network infrastructure and its traffic and the cellular voice channel overview are introduced. Then, the smartphone architecture is provided in order to understand how smartphones function. Android OS is discussed in terms of Android architecture, Android security, the Android RIL, and the Android audio system. Covert channel and the rootkit also are discussed. Finally, a literature review of related work is provided.

2.2 Background Information

2.2.1 Cellular Network Infrastructure Overview

A cellular network is a radio network distributed over land through base stations. It usually consists of five components: mobile station (MS), base station (BS), base station controller (BSC), mobile telephone switching office (MTSO), and public switched telephone network (PSTN). The MSs represent the mobile phones that communicate to the BS directly in the same cell. The MS has a subscriber identity module (SIM) that provides the MS with a unique identity. The base stations connect the mobile switching center (MTSO) and mobile phones together. The BSC handles the handoff operation between the BSs as a subscriber moves among the cells. In addition, a cluster of BSCs are connected to a MTSO, which is the control center for the cellular system. See Figure 2.1.

The MTSO—also known as mobile switching center (MSC)—provides overall control of the cellular system and acts as a central office exchange to interconnect calls and connect into the PSTN. MTSO allows subscribers to place calls, and it controls them as required. Moreover, it holds databases that store the last known locations of the cellphones. It also contains authentication center and billing information facilities. PSTN is the land-based section of the network that aggregates all the circuit-switched telephone networks in the entire world to provide infrastructure and services for public telecommunication [2].

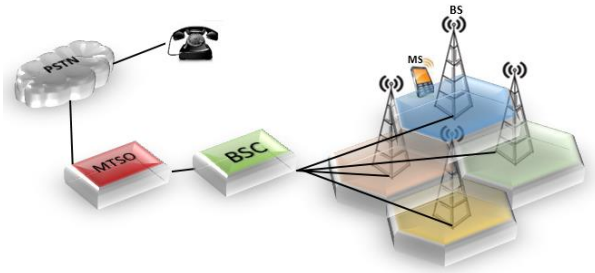


Figure 2. 1: Cellular Network Infrastructure

A main function of the MTSO is to constantly monitor the quality of the communications signal. In addition, MTSO usually employs Lawful Interception Systems (LIS) on all voice and data traffic (such as PSTN, GSM, GPRS, 3G, UMTS, LTE, CDMA, and VoIP). Regarding data traffic, this kind of system is usually utilized for infrastructure protection and cybersecurity defense purposes. In the case of voice traffic, it has been used in real time interception and listening to conversations as needed, as well as for data retention purposes [3]. However, the primary aim of voice traffic data retention is for governmental security purposes, such as tracking or identifying the locations of individuals. Since it is unusual to have audio waves that carry malicious contents, there is no feature of these systems that analyzes the content of the audio waves to detect the malicious acts that could threaten the security of the core network or the end users.

2.2.2 Cellular Voice Channel Overview

Understanding how cellular voice channels function is essential to design an audio modem capable of generating audio waves that can travel among these channels. In digital cellular systems like GSM and CDMA, when someone makes a phone call, and once the

voice has passed to the microphone, it would pass the analog-to-digital converter (ADC) that converts the analog wave into digital data using Pulse Code Modulation (PCM) to be understood by the cell phone. The PCM method is utilized to represent sampled analog signals in digital form by recording a binary representation of the magnitude of the audio sample, after which the sample is encoded as an integer. The data stream then is processed and transmitted through the cellular core network in digital form.

Audio data stream travels among the cellular channels and is compressed to allow greater channel capacity. The compression rate relies on the type of voice coded. The cellular channels are band-limited channels, so audio signals should have frequencies within the telephone voice band, which is between 300 and 3400 Hz. When the data stream is received on the other side, it is reset to the original source signal. Then the digital –to-analog (DAC) module converts the bit stream to audio wave again.

In order to reduce the bandwidth of the voice call and save the power of the cellphone, Discontinuous Transmission (DTX) is used to allow the transmitter of the cellphone to be turned off when the user is not talking. To do so and to detect silence, DTX uses Voice Activity Detection (VAD), which is a unit that determines whether the speech frame includes speech or a speech pause to reduce the transmission to only speech and to reject noise and silence. Once the frame has been labeled as non-speech, it is dismissed instead of being transmitted [2] [4].

Currently, voice calls are carried out by using the 2G technology that includes GSM and CDMA. The 2G is a digital cellular that has a circuit-switched telephone networks core. However, in the future voice calls will be carried out using packet-switched core

network. In both networks, cellular voice channels are not meant to carry any other kind of data but audio waves, and these audio waves usually are not utilized to carry a malicious content.

2.2.3 Smartphone Architecture

Cellphones are classified into two kinds: feature phones that provide basic functionalities, like voice calls and sending and receiving SMS, and smartphones that provide a variety of features in addition to the main cellular phone functionalities, such as installing applications, browsing the Web, and sending and receiving email. Feature phones usually have a single processor that has one operating system that handles the baseband software stack, the user interface, and the applications.

On the other hand, smartphones include at least two processors, the baseband processor (BP) and the application processor (AP). BP handles radio access to the cellular network and provides communication protocols such as GSM, GPRS, and UMTS. In addition, it has a Real-Time Operating System (RTOS) such as Nucleus and ThreadX. AP is responsible for the user interface and applications. Some smartphones employ an shared-memory architecture between the baseband processor and the application processor. However, the modern design has better separation, so the BP and the AP have separate memories and communicate through a dedicated channel [5][6]. See Figure 2.2, adapted from [5].

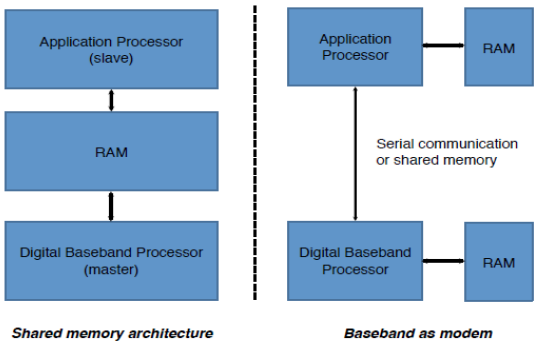
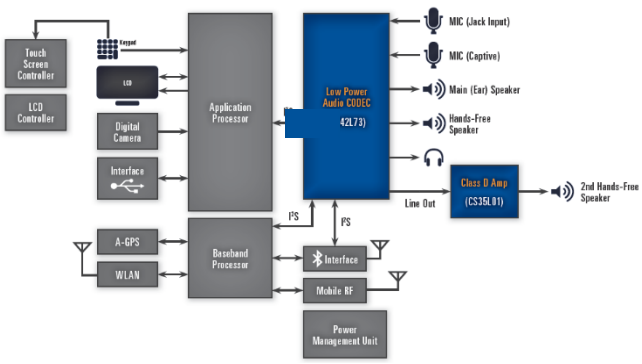


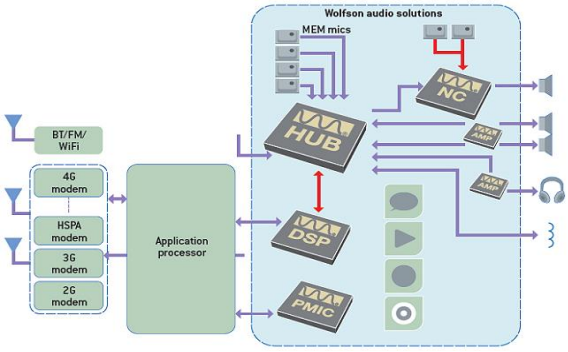
Figure 2.2: Smartphone Design Architectures

In smartphones, the cellular voice call routing and control is achieved typically by the baseband processor only, while the application processor handles all the multimedia functions. However, as users spend more time using mobile phones for more media-rich applications, most smartphone hardware designers wanted to provide higher-quality audio and video performance and longer battery life [7]. See Figure 2.3, adapted from [7] [8]. Therefore, they introduced a separate dedicated processor for audio/video decoding to meet these increasing needs.

Currently in most smartphones, as has been discovered in this research, the audio routing functionalities, including voice call functionality, have been moved to be controlled by the application processor. This feature contributes to introducing a new security issue as the audio path to the cellular voice channel could be reached from the application processor.



a) Cirrus Logic audio architecture



b) Wolfson audio architecture

.

Figure 2.3: Smartphone Audio Subsystem Architecture

2.2.4 Android OS Overview

Android is an open source operating system developed by the Open Handset Alliance based on the Linux kernel. Android operating system consists of five software components within four main layers: Linux kernel, libraries and Android runtime, application framework, and applications. See Figure 2.4, adapted from [9]. The first layer is the Linux kernel layer that provides main system functionalities such as memory management and process management. In addition, the kernel takes care of networking and many device drivers as it communicates with hardware and baseband processor.

The second layer includes two components: a set of libraries and Android Runtime. The libraries are written in C/C++ and are responsible for providing multiple system services, such as storing and sharing data of applications, such as playing and recording

audio and video, and Internet security. The second component in the second layer, Android Runtime, has a set of core Java libraries that help Android application developers use standard Java programming language to write Android applications.

Android Runtime also has a main component called Dalvik Virtual Machine (DVM). DVM is an enhanced version of Java Virtual Machine, which is designed and developed for Android in order to utilize Linux core features, such as multi-threading and memory management. DVM allows every Android application to run in its own process under a unique UNIX UID. DVM is responsible for executing binaries of all applications located in the application layer.

The third layer is the application framework that delivers many services by offering Application Programming Interfaces (API) to third party application developers. API allows the developers to access the different hardware, sensors, and other components of the devices. The fourth layer is the applications layer, which is written fully in Java and represents the user application that could be installed, such as games, and social networking. Applications are written in Java and compiled to the Dalvik Executable (DEX) byte-code format. Every application executes within its own instance of DVM interpreter.

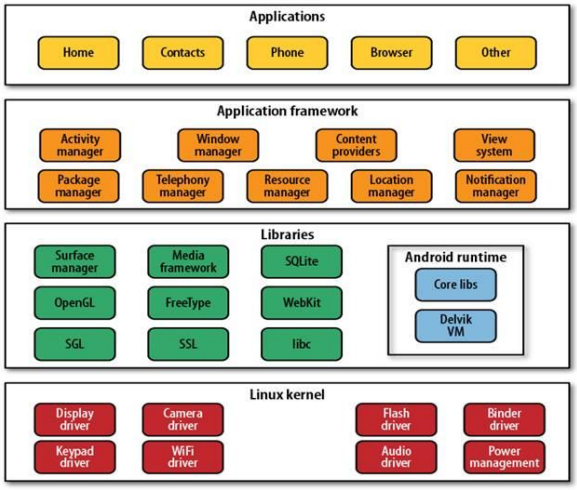


Figure 2.4: Android Architecture Layer

Android applications typically have four types of components: Activities, Services, Broadcast Receivers, and Content Providers. Table I shows the summary of the Android applications components. Android applications communicate together using inter-component communication (ICC) by sending an Intent. Intents are special messages that instruct applications to perform a specific task. In addition, applications could communicate using channels controlled by the Linux kernel, such as UNIX domain and Internet sockets or files.

TABLE I: ANDROID APPLICATION COMPONENT

Android application component	***Description***
Activity	represents the application's user interface
Service	works as background processes
Broadcast Receiver	responds to broadcast messages that come from other applications or from the system
Content Provider	is a database container

2.1.4.1 Android Security

Android framework involves multiple security mechanisms, such as application sandboxing, application signing, and Android permission framework. The application sandboxing mechanism is employed to isolate an application from system resources, actual hardware, and other application resources. Sandboxing is performed by assigning a unique user ID to each application to ensure that it has its own sandboxed execution environment. Application signing is a key security mechanism in Android. All Android applications should be signed independently with a self-created key using public key infrastructure. Moreover, all applications that are signed with the same key could share the same UID and play in the same sandbox.

The Android permission framework is utilized to control access to sensitive system resources by applying Mandatory Access Control (MAC) on Inter Component Communication (ICC). Therefore, each application can define a list of permissions that are needed for access to components to specify its protection domain and permission approved during the installation process. In addition, each component could be assigned an access permission label to specify an access policy that protects the application resources.

2.2.4.2 Android Telephony Framework

The Android telephony stack is responsible for communication between the baseband modem and the application layer. The Android telephony stack consists of four main layers: applications, framework services, radio interface layer, and baseband modem.

The application layer involves all the smartphone telephony applications, such as Dialer, SMS, and Antenna signal indicator. The phone application in the application layer communicates directly with the internal API in the telephony framework to place and tear down phone calls. See Figure 2.5, adapted from [10]:

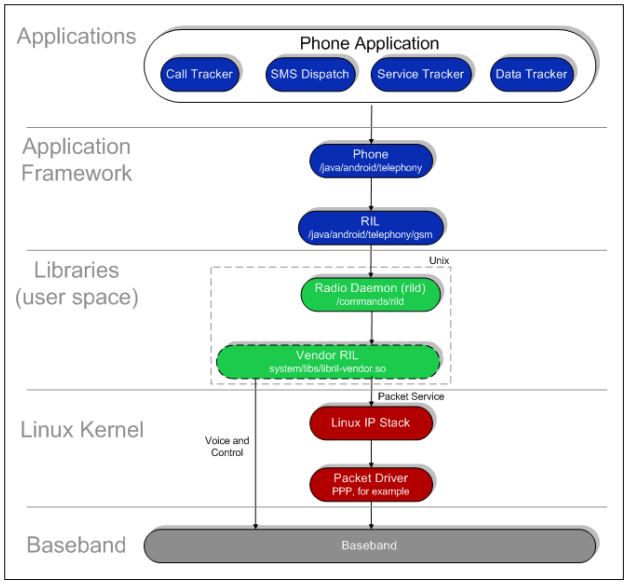


Figure 2.5: Android Telephony Architecture

Android telephony framework services is a layer that provides APIs for the phone application to make calls, and send SMS and MMS. However, APIs cannot be entered from any other applications that are not part of the Android system. Only the phone default application can utilize this internal API. The telephony requests are passed to the baseband modem. The modem replies to the application through the telephony framework.

Communication between the telephony framework and modem are handled by a RIL. The RIL communicates with the cellular modem by utilizing a single serial line. The

RIL has two main components: a RIL Daemon and a Vendor RIL. The RIL Daemon connects the telephony framework to the Vendor RIL and initiates the modem and reads the system property to find the proper library for the Vendor RIL.

The Vendor RIL is the baseband modem driver. Mobile vendors are various, and, therefore, each vendor has a different implementation of the vendor RIL. The baseband modem is a cellular telephony processor that comes in different models, and there are no standards for this kind of processor. Usually, the implementation of the baseband modem software is kept secret.

2.2.4.3 Android Media Framework

Since it is important to understand how a smartphone cellular voice call takes place, it is essential to realize how Android handles audio stream, which forms an important part in carrying out a cellular voice call. Android Application Framework takes care of Android's multimedia system. It uses the Android.media APIs to call media native code libraries in order to contact the audio hardware. See Figure 2.6, adapted from [11]. The Android audio libraries include two native layers that deal with audio software: Audio Flinger and Audio Hardware Interface (HAL).

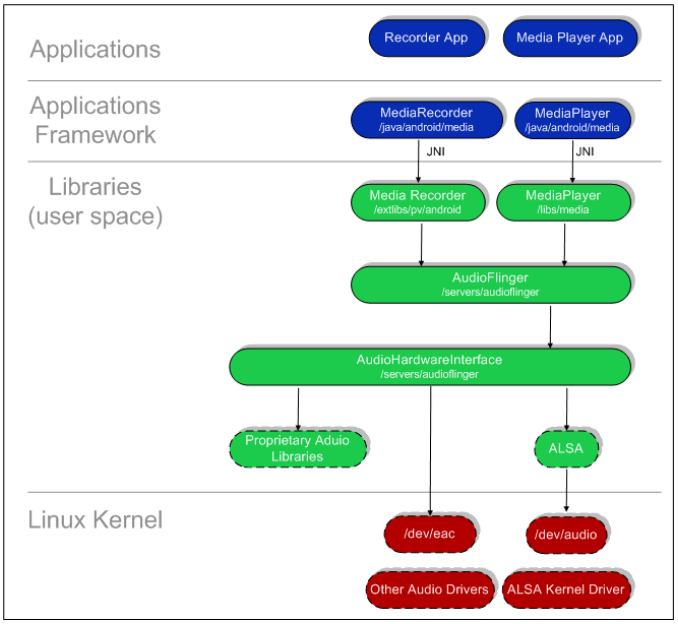


Figure 2.6: Android Audio System

Audio Flinger communicates with the HAL layer, which represents the hardware abstraction layer that hides audio drivers from the Android platform. Audio Flinger is the audio server that provides some required audio functions, such as audio stream routing and mixing, since the audio stream can be input or output to/from multiple microphones, speakers, and applications. Audio Flinger performs audio routing by setting a routing mode, such as MODE_IN_CALL, MODE_IN_COMMUNICATION, or MODE_NORMAL, which is then passed into audio.h interface in Audio HAL which, in turns, determines the routing path.

The audio HAL is vendor-specific and comprises three interfaces: audio.h, audio_policy.h, and audio_effect.h. Audio.h includes the main functions of an audio device. Audio_policy.h contains the audio policy manager that represents audio control

policies. Audio_effect.h encompasses audio effects functions like echo or noise suppression. The kernel driver contains the audio driver that communicates with the audio hardware and the HAL.

When some applications use the Audio Flinger to redirect the audio stream as STREAM_VOICE_CALL, some vendor-specific audio HAL implementations redirect the stream that comes from/to an application from/to the actual stream voice call by default in some smartphones. In the other smartphones, redirecting voice call stream from/to an application from/to the actual cellular voice stream is possible also, because the audio routing is currently accomplished using the application processor, not the baseband processor as mentioned earlier. This is achieved by making some modifications to audio HAL and some application framework components, such as Audio Service and Audio System, in smartphones that cannot redirect the required audio path by default.

2.2.4.4 Android Phone Calls

In Android, phone calls are achieved using both the telephony and media frameworks. When the user places a call, the phone application in the application layer sends a request to the Vendor RIL for a phone call with the phone number. The Vendor RIL sends the request to the baseband processor. The baseband sends a request to the base station. When a connection is established, the RIL updates the phone application interface with the ongoing call, and the phone's media system stream switches the phone audio mode to MODE_IN_CALL.

2.2.5 Covert Channel Overview

The covert channel concept was first presented by Lampson in 1973 as a communication channel that was neither designed nor intended to carry information at all [12]. A covert channel utilizes mechanisms that are not intended for communication purposes, thereby violating the network's security policy [13]. In fact, the use of a covert channel means that there is acollusion between the sender and receiver to pass information. A side channel is a type of covert channel, but only the receiver desires a successful communication and the information is leaked unintentionally, which could be conducted through network attacks in this channel.

Three key conditions were introduced that assist in the emergence of a covert channel: 1) the presence of a global shared resource between the sender and receiver; 2) the ability to alter the shared resource; and 3) a way to accomplish synchronization between the sender and receiver [14]. The cellular voice channel has all three conditions, making it an ideal channel for implementing a covert channel.

Usually the term "covert channel" refers to the hiding of information inside a carrier, which in this case represents the cellular voice channel. The information conveyed across this channel is referred to as covert information, which represents the audio signals that are generated by the audio modem.

There are several security research recommendations that count covert channels as a source of threats, which requires identifying the channels, assessing their capacity, and,

depending on that, closing or minimizing the information leak. Therefore, studying cellular voice channels as a potential source of such an attack was conducted in this research.

2.2.5.1 Covert Channel Taxonomy

Covert channels are usually classified based on the following criteria [15] [16]:

- **Storage and timing channels**

Usually covert channels are categorized as storage or timing channels. Storage channels involve a storage location written to by the sender and read by the receiver. Timing channels involve signaling information through a network by manipulating time values of resources so that the receiver can notice it and interpret the information.

- **Passive and active channels**

Passive channels use an existing connection that is already established to transfer covert information. It does not initiate any new network connections, so it is usually not detectable, but it has limited throughput. On the other hand, active channels initiate a new connection to transfer covert information. It can provide significantly higher throughput as compared to passive channels, but they are more prone to detection.

- **Noisy and noise-free channel**

A noisy channel can intentionally or unintentionally corrupt the data signal with errors and so affect the transmitted covert data rate. However, a noise-free channel has no errors that transmit covert data as is.

2.2.5.2 Covert Channel Evaluation Criteria

The effectiveness of a covert channel can be evaluated by the following criteria:

- Bandwidth
- Robustness
- Covertness

Channel bandwidth refers to the channel maximum error-free transmission rate. Bandwidth is usually expressed in bits per second. Robustness determines the ease of limiting the channel capacity by adding noise or the ease of removing the covert channel. Covertness determines to what extent the covert channel can be detected; in other words, it means the secrecy or stealth of the channel. Covertness of a channel is usually determined by comparing the traffic characteristics with a covert channel and real legitimate traffic. However, all of these factors cannot be combined together in one covert channel, as they are conflicted. For example, sending less data increases secrecy; however, it reduces capacity [17].

2.2.5.3 Covert Channel Applications

Several groups have an interest in keeping their communication stealthy. Using the traditional security techniques such as encryption and steganography do not help against the detection of the communication that is taking place. Therefore, a covert channel, a new trend in using a communication channel that is not anticipated by adversaries, is used. A

covert channel has many uses, such as providing a transfer channel for viruses, worms, or other types of malware. Another use of a covert or a side channel is data exfiltration, which is becoming a serious security issue.

In addition, a covert channel could be utilized to stealthily distribute commands for the purpose of launching botnets to avoid detection in traditional C2 channels. A botnet is a group of compromised machines (zombies) infected by a malicious bot to allow a bot-master to remotely control the devices. Botnets could pose a serious threat, such as leaking information, as well as disseminating malicious code, conducting distributed denial of service attacks, and performing identity theft. Moreover, usually covert channels are used in combination with a rootkit [17].

2.2.6 Rootkit overview

The “rootkit” term originally meant a software component developed by attackers to hide the presence of malware on a compromised operating system. The term was first introduced as a set of modified tools of system binaries used to gain and keep administrative privilege access on UNIX systems [18]. Rootkit also could be utilized to hide processes, alter programs, hide communication channels, etc., while remaining undetected. In addition, rootkit could be used with botnets to allow a hacker to gain access and get personal information without being noticed.

There are several rootkit categorizations depending on whether it survives system reboot or whether it runs in user mode or kernel mode. User mode rootkits are running in

the application space and intercepting API calls, so when an application makes a system call, the rootkit can change the values or the flow of those system calls. However, kernel mode rootkits usually change some kernel data structures after accessing kernel memory. A memory-based rootkit cannot activate each time the system starts, while the persistent rootkits can.

2.3 Literature Review

Compared to the PC security research field, the concern for smartphone security breaches is new. It began with the first mobile malware, named *Cabir*, which appeared in 2004. Since then, many smartphone malware have appeared, although with simple functionalities, until the emergence of the first mobile botnet SymbOS.Yxes [19] in 2009, which targets Symbian platform. In the same year, Ikee. B [20] appeared, and it targeted jail-broken iPhones that could leak sensitive information and distribute malicious code through a Bluetooth channel.

PlaceRaider [21] is a proof-of-concept visual smartphone surveillance that has been proposed. It allows attackers to remotely explore the victim's virtual environment through using the smartphone's sensors, such as camera and microphone, and sending the data through the Internet. In March 2011, malware named *DroidDream* [22], which aims to steal sensitive information and download malicious codes, had been detected embedded within at least 50 applications in the Android market.

As smartphones are mostly connected to the cellular network more than Wi-Fi, some security breaches that are initiated from the cellular network have been launched. For example, a mobile botnet named ZITMO targeting Symbian, BlackBerry, and Android platforms has been released into the wild. It aims to leak personal information and forward it through SMS messages to a predefined bot-master number. In addition, it monitors SMS communication and leaks mTAN passwords that are sent by banks to validate mobile transactions [23] [24].

However, all of these security breaches are conducted through an overt channel. An overt channel transfers information through a legitimate channel used as a medium for data transmission. On the other hand, a covert channel uses a medium that is not normally used as a data container to transfer information [14]. The covert channel has appeared in smartphone areas; for instance, Soundcomber is a proof-of-concept malware [25] that uses the smartphone microphone sensor and phone call through covert channels in between applications to get a small amount of sensitive information such as credit card numbers. Then the private data is transferred through an overt or covert channel using applications collusion attacks. Moreover, the paper [26] discussed several possible covert channels on Android OS that use different system states, such as volume or vibration settings, and file locks as data storage.

The current research field of implementing covert channels focuses on exploiting the weaknesses in common Internet protocols to embed a covert channel, such as TCP/IP [27], HTTP [28], VoIP [29], and SSH [30]. In the cellular network field, it has been proven that the ability to embed high-capacity covert channels in SMS, by composing the SMS in

Protocol Description Unit (PDU) mode, could be used as a data exfiltration channel [31]. In [32] the authors introduced stenographic algorithms to hide data in the context of MMS to be used in on-time password and key communication.

An application that modulates data to be "speech-like" is already in existence, such as a point-of-sale (POS) system. This application uses the cellular voice channel to transmit an audio-modulated data between the transaction center and POS terminal. Several previous efforts on transmission data over a GSM voice channel [33-36] have been already presented. In addition, advanced studies have introduced transferring encrypted data using GSM voice channel [37-39]. In [40], the authors proposed hiding secret data on compressed speech bit streams. However, all of these efforts in sending data over the GSM network voice channel were achieved by computer and GSM modem only, not with smartphones.

Although several security solutions have been developed in order to prohibit unauthorized data leakage, such as TaintDroid [41] and XManDroid [42] in Android platform, the proposed solutions are not applicable to cellular network communication. TaintDroid was developed to track the flow of sensitive information through third-party applications. XmanDroid monitors multiple channels in between applications and stops communication that may lead to a privilege escalation attack.

CHAPTER III

METHODOLOGY

3.1 Introduction

To prove the concept of using the cellular network voice channel as a covert channel, two subsystems were developed during this research: an audio modem that converts data from digital to analog to carry covert information and an Android user-mode rootkit able to open the channel covertly for data exchange. In addition, three scenarios were implemented to show the ability of the combined subsystems to form covert channels, side channels, and botnets. Therefore, in the following section the audio modem design and implementation are illustrated, the constraints and the challenges in designing such a modem are introduced, user-mode rootkit design and implementation are described, and multiple scenarios utilizing these subsystems together are explained.

3.2 Architecture and Implementation

3.2.1 Audio Modem Design

The first developed subsystem was a modem able to carry data through the cellular voice channel during a normal voice call. The modem consisted of an encoder and decoder pair that was designed to work entirely in software. The encoder converted raw data into "speech-like" waveforms to be injected into the microphone of the downlink voice stream. The audio signals then traveled among the cellular voice channel as speech in digital form, reached the second cellphone to be played over an uplink voice stream's speaker, and then demodulated and converted back into the original data. The designed modem utilized Morse code combined with Frequency Shift-Keying modulation. Figure 3.1 illustrates the audio modem design.

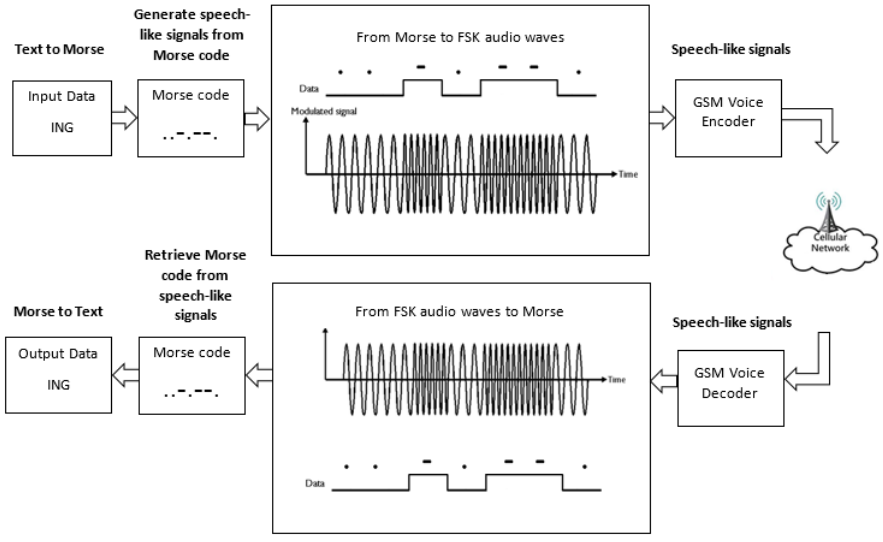


Figure 3.1 Audio Modem Design

3.2.1.1 The Encoder

The encoder encodes text information into Morse code as a series of dots and dashes. Then it generates pure sine waves of tow frequencies using FSK modulation: one frequency represents a dot and a different frequency represents a dash using the following equation for each sample in the buffer:

$$f(x) = Amplitude * sin(2\pi f) * (sample / sampleRat) \quad (3.1)$$

Where:

- Amplitude: the amplitude of the wave; it is a constant value for all waves.
- f: the frequency of the wave; this is a variable and depends on if the current code is a dot or a dash.
- Sample: the current sample in the buffer.
- SampleRate: the modem sampling rate, which is 8000 KHz.
- Sample / sampleRate: represents the current time.

The audio modem works as following:

1. Maps each character to its corresponding Morse code.
2. Plays a hail tone to indicate to the beginning of the audio data.
3. For every code:
    - Plays a tone with high frequency if the code is a dash.
    - Plays a tone with low frequency if the code is a dot.
4. Plays a silence to separate each character.
5. Plays a long silence to separate each word.

6. Forms each code to be a “speech-like” waveform of 20 ms frame length to fit into GSM encoder.

3.2.1.2. The Decoder

The decoder performs the reverse functionality of the encoder. The decoder performs spectral analysis to analyze each received sample and estimates its frequency to retrieve the original data. Fast Fourier Transform (FFT) algorithm was used to analyze the spectrum and detect the frequency pitch. Therefore, when the decoder receives an audio wave sample, the following steps are accomplished:

1. Low-pass filter is performed to the audio signal to get rid of noise and enhance the reliability of the pitch detection.
2. Hann window function is applied to the input data to increase the detection quality by looking into the important details of the signal rather than analyzing the entire signal and to avoid the signal leakage in both sides of the wave.
3. FFT is performed on the input data, which is an algorithm that quickly computes the frequency of a given signal. The output of the FFT is an array of N complex numbers that has both real and imaginary parts.
4. FFT bins are normalized to bring the peak amplitude to a normal level, hence making the entire received audio signal at the same volume so as to not affect the quality of frequency detection.
5. Magnitude values of each FFT bin are calculated using the following formula:

$$magnitude[i] = sqrt(real[i] * real[i] + imag[i] * imag[i]) \quad (3.2)$$

6. The bin of the maximum magnitude value is obtained in the FFT data.
7. Frequency of the index of the maximum magnitude value to detect the signal frequency is calculated by this equation:

$$freq = i * SampleRate / N \quad (3.3)$$

Where:

- i = index of magnitude
- SampleRate = the modem sampling rate
- N = size of FFT data

8. Finally, the received frequency is compared to the dot-and-dash frequency values with a minor tolerance.

3.2.2 Modem Design Constraints and Challenges

To develop a modem that copes with the cellular network channel characteristics, some constraints on the audio signal were taken into consideration.

1. The generated audio frequencies should fit into bandwidth between 300Hz and 3400Hz, which represents the voice channel bandwidth of the cellular network. Therefore, generating an ultrasonic signal that can't be heard by the human ear is not suitable for this channel.

2. In addition, the generated audio signals should pass through the voice channel without being omitted by Voice Activity Detector (VAD), which is primarily used to stop frame transmission at silence.
3. Cellular networks employ speech compression codecs in order to save the bandwidth and enhance the quality of sound. As a result these speech compression techniques could significantly distort audio signals that are not "speech-like," and so the generated signal should have the speech characteristics.

Therefore, the audio modem was designed to modulate the audio signal to possess speech characteristics to reduce distortions that could be produced by VAD and speech compression, as well as to fit into the cellular voice bandwidth to avoid distortions produced by filtering. In addition to the constraints mentioned above, there are a number of challenges in implementing such a modem. In fact, designing an audio modem for the cellular voice channel communication is difficult, because the cellular voice channel and the cellphone devices audio hardware have multiple limitations and challenges.

One of the challenges of the cellular voice channel is that the voice call between two cellphones is achieved in real time; hence, there is no retransmission in case of a missing voice frame. Sometimes a voice frame could not be received by the other cellphone because of the noisy air interface, or the voice frame could be dropped intentionally by the core network when a frame-stealing scenario happens. Frame stealing happens during a call in some cases, such as when a user receives an incoming waiting call request and the network drops the current voice frame in the conversation to send a notification to the user. On the other hand, smartphone capabilities, such as the processor and battery life, as well

as the quality of audio hardware, vary among cellphones. For example, a cellphone speaker that sounds clear and loud could be low or unclear in another, since there are no standardizations in cellphone speakers and microphone specifications.

3.2.3 Audio Modem Implementation

The audio modem was implemented as an Android application that could be installed in the application layer. The modem works as follows: When is there an ongoing cellular voice call, the receiver side should press the receive button to obtain the required data. At the sender side, a text message is sent to AudioUtils class that, in turn, calls Morse class in order to encode the text data into Morse code. The EncodeMorse method in Morse class will take the input data and perform the conversation that generates Morse code then composes an Integer array that contains the frequencies of dots and dashes. This array then is sent to the encoder to be sampled as sine waves. See Figure 3.2 below:

```
for (int j = 0; j < input.size(); j++) {
    // Generate a sine waves
    for (int i = 0; i < kSamplesPerDuration; ++i) {
        InputBuffer[i + j * kSamplesPerDuration] = Amplitude * Math.sin(TwoPi * i / (kSamplingFrequency / input.get(j)));
    }
```

Figure 3.2: Generating Sine Waves by the Encoder

The encoder then creates a byte array to be sent to PlayThread class that contains an Audio Track to play the audio tones and inject them into the cellular voice stream. See

Figure 3.3 below:

```
AudioTrack atrack = new AudioTrack(AudioManager.STREAM_VOICE_CALL,
                            (int) kSamplingFrequency,
                            AudioFormat.CHANNEL_OUT_MONO,
                            AudioFormat.ENCODING_PCM_16BIT,
                            buffer.length,
                            AudioTrack.MODE_STREAM);
```

Figure 3.3: Injecting Audio into the Voice Call Stream

At the other end, MicrophoneListener constantly records the incoming audio waves and sends the buffer to StreamDecoder. See Figure 3.4 below:

```
AudioRecord arec = new  AudioRecord(MediaRecorder.AudioSource.VOICE_CALL,
        (int) Constants.kSamplingFrequency,
        AudioFormat.CHANNEL_IN_MONO,
        AudioFormat.ENCODING_PCM_16BIT,
        buffSize);
```

Figure 3.4: Recording Voice Stream Call Buffer

StreamDecoder monitors the recorded buffer continuously to look for the hail key, which refers to the beginning of the real incoming audio data. The hail key has a different frequency from the dots and dashes and is generated once at the encoder to indicate the beginning of audio data to be recorded. If the hail key was not found, the detection mode would run to check the incoming data over again. If the key was found, the StreamDecoder would send the recorded buffer to the Decoder to decode the audio waves. The decoder

hires the math package that has an FFT algorithm and other signal processing functions to process the incoming signals and detect the frequencies. See Figure 3.5 below:

```
 //low pass filter
int nPoints = signal.length * 2;
double [] data_lowpassed = Tools.lowpass(signal, nPoints);

 //apply window
double data_window [] = DFT.window(data_lowpassed, DFT.HANN);
double [] data = FFT.magnitudeSpectrum(data_window);
N= data.length;

//normalize data
double [] data_FFt = normalize(data);
int freq = 0 ;

int max = getMaxPeek(data_FFt);
```

Figure 3.5: Decoding Audio Signals Using FFT Algorithm

Once the audio signal frequencies have been determined, the decoder creates an Integer array of the detected frequencies that will be sent to the Morse decoder to retrieve the original data. Figure 3.6 shows the data flow of the implemented modem.

The frequencies that were chosen with the modem are 600Hz and 1000Hz to satisfy the demands of the cellular network voice channel bandwidth (300Hz and 3400Hz). In addition, the sampling rate was selected at 8000 kHz to suit the telephony system sampling rate that carries speech signals. Each 20 ms a voice frame will be injected into the GSM encoder to avoid the distortion that could be issued from the VAD.

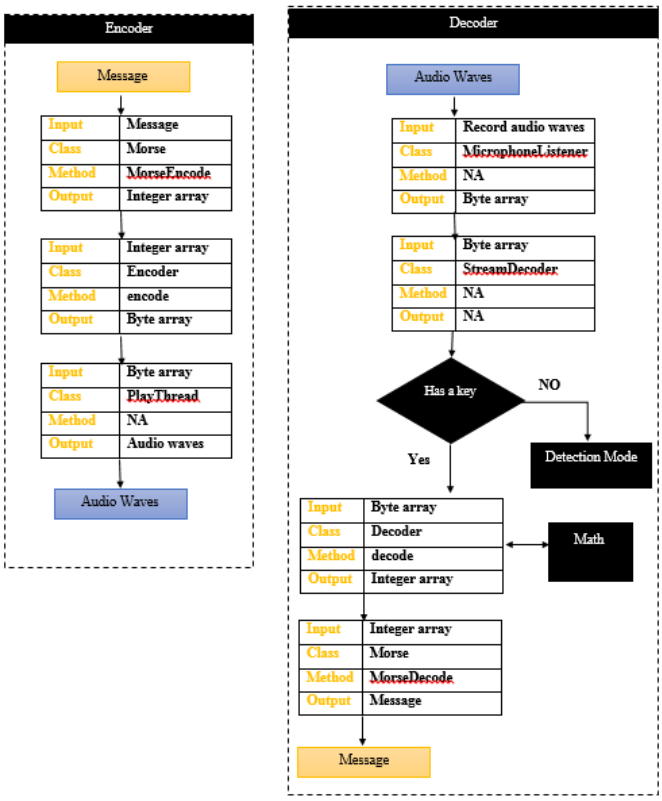


Figure 3.6: Audio Modem Decoder and Encoder Data Flow

3.2.4 Voice Call Rootkit Design

Besides an audio modem that converts digital data to audio waves and sends the data during a normal call, another system was needed in order to compromise smartphone operating systems and open the cellular voice channels by stealthily answering voice calls and performing data exchange. This kind of system was accomplished by utilizing a rootkit that can hide its presence in the operating system to quietly allow continuous privileged access.

3.2.4.1. Bypassing Android Security Mechanisms

Answering a voice call in Android is achieved by a Phone App that resides in the application layer of Android. A Phone App runs inside **com.android.phone** process that comprises multiple telephony components such as SIP, SMS, Simphonebook, and Phone App. All these components utilize the telephony framework APIs to communicate to the baseband modem through the RIL socket. Hence, the Phone App communicates to the baseband modem by calling the framework APIs to initiate and receive voice calls.

To help rootkit catch the incoming calls before being delivered to Phone App, it was necessary to find a way to make the rootkit able to reach the RIL socket in order to listen to the baseband modem messages. RIL socket is a Linux socket that is owned by com.android.phone process only, and it cannot be entered from outside the phone process. Thus, building rootkit inside the com.android.phone process to communicate to the socket was a good choice to secretly communicate to the baseband modem and intercept its messages.

Designing a rootkit that works inside the phone process requires overcoming the Android application sandboxing mechanism. This is achieved by assigning a unique User ID for each application so that the other applications cannot access its data and resources unless they have shared a User ID and process, as well as their certificate match. The rootkit takes advantages of android:sharedUserId and android:process attributes to play inside the com.android.phone process sandbox and use its resources to be able to access the RIL socket.

The android:sharedUserId attribute allows an application to share the same User ID with another one; hence it can reach the others' application data and resources. The android:process sets the process where all the application components should run. Setting both attributes, sharedUserId and process, allows both applications to run in the same process and access each other's resources. Setting sharedUserId and process attributes on the manifest allows rootkit to run similar to the com.android.phone process; however, it also requires that either the rootkit certificate match the Phone App certificate or the signature verification process in Android bypassed during the installation time.

Phone App is signed by the system platform certificate and runs in the shared system sandbox as a system user, which has more privileges to access the system resources. Signing rootkit with the platform signature was possible with some custom ROMs such as Cyanogenmod, because they are using the default certificate that is known as it is released publicly.

Another method also was used in this research—bypassing the signature verification method in order to infect the rooted Android official ROMs. That was achieved by modifying Services.jar. Services.jar has multiple java classes, and one of them is PackageManagerService class that is responsible for matching the signature of the application during installation time. The method in PackageManagerService is called CompareSignatures, which is a Boolean method that returns true if the signatures match and false if not. Therefore, the method was modified to always return true, which means the signature is always matched. See Appendix A for more detail.

After overcoming security obstacles, rootkit was built successfully inside the phone process and needs to go further to intercept the telephony framework APIs calls and take control of answering the incoming voice calls. To find a tactic to intercept the telephony APIs calls and control the call flow, it was important to understand the intricacies of telephony applications to realize their weaknesses and vulnerabilities. However, as there is no documentation about the telephony applications, analyzing the Phone App and internal telephony framework source code was essential. Therefore, in the following section a discussion about Android application architecture and the Phone App incoming call data flow is provided in order to discover any vulnerability that helps in achieving the goal.

3.2.4.2. Android Application Architecture

In this section, an overview of how the Android application generally works is highlighted. When an Android application is not running, and one of its components just starts working, the Android system initiates a new Linux process with a single main thread of execution for the application. Then, when another component of the same application also starts, it will run in the same process and the same main thread.

The main thread is essential because all the application's components will run in it by default. The main thread is responsible for the user interface interactions and events handling and handles system calls to each application component. Android thread can associate with a looper, which runs a message loop for a thread to process different

messages. A looper contains a MessageQueue that contains a list of the messages. The interaction with the message queue is performed by using a Handler class.

The handler is associated to the looper and its associated thread. Handler is responsible for handling messages within the MessageQueue. When the message is delivered to the handler, a new message arrives in the MessageQueue. When a thread creates a new handler, it bounds the handler to it and to its message queue. Then the handler works to send messages to the message queue and handles messages as they arrive from the message queue. See Figure 3.7 below:

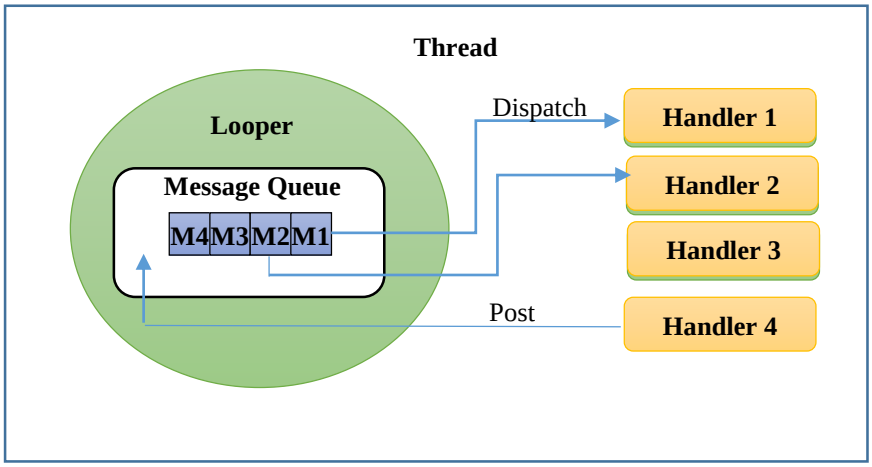


Figure 3.7: Thread's Looper and Handler

By exploring the source code of the telephony application and framework, it was discovered that all the important components inside the com.android.phone process are running in the same main thread and hence are connected to PhoneFactory's looper and its message queue, which is located in the internal telephony framework.

Therefore, the rootkit was designed to run inside the system sandbox and in the phone process as a background service. Once rootkit service has been created inside the phone process, it runs in the phone process main thread, thus hooking itself to the main thread looper and its message queue. The rootkit creates a handler to process the looper messages that belong to PhoneFactory so that it can receive/send messages from/to the PhoneFactory looper.

3.2.4.3. Incoming Call Data Flow

When an incoming call request arrives to the baseband modem, it is delivered to the Vendor RIL and, in its turn, passes it to the RIL daemon that eventually delivers the request to the telephony framework layer. The request message is passed among the telephony framework classes in this order: RIL, Basecommand, GSMCallTracker, GSMPhone, BasePhone, and then CallManager, which, in its turn, communicates to the Phone App in the application layer of Android to show an incoming call. See Figure 3.8.

Each class in the telephony framework can obtain notifications about the arriving messages by using the RegistrantList object, which is simply an object that holds java ArrayList of Registrants objects. Therefore, each class registers itself with the pre-class list in order to obtain the notifications. Rootkit takes advantage of this design and registers itself with the PhoneBase; hence it gets the incoming notifications when they are delivered to the CallManager and acts before CallManager handler passes the incoming notification to the Phone App. Once rootkit has received the incoming notification about a new

incoming call, it obtains the incoming caller ID and compares it with a predefined number. If the number does not match, it leaves the notification to arrive at its final destination, which is the Phone App that shows an incoming call. However, if the caller ID matches the predefined number, it answers the call covertly.

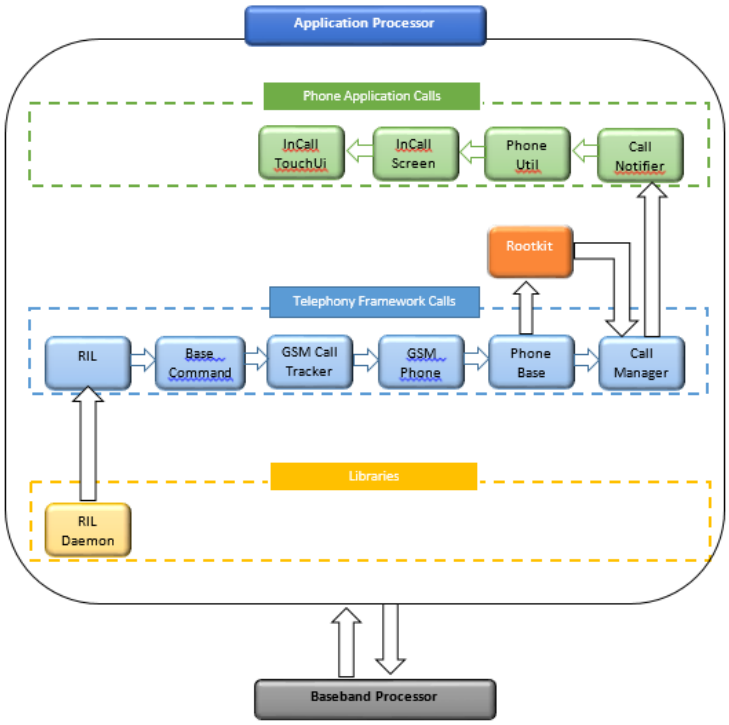


Figure 3.8: An Incoming Voice Call Request Flow in Android

Android thread can only perform one task at a time, if a long running task has been executed in the handler, the message queue cannot handle and process any other messages. In addition, it was discovered during this research if the message queue has been starved by sending two messages at the front of the message queue, it hinders the looper and block the thread from delivering the new messages as. Therefore, when rootkit want to answer

the call covertly, it hinders the looper from delivering the other messages until finishing the call. Thus the looper in the phone's main thread will not be able to deliver the other messages to the phone application.

Rootkit acts as a proxy between the application layer and the telephony framework to transparently intercept and filter incoming voice calls from the baseband modem. Whenever an incoming voice call request comes from the baseband modem to the Phone App, rootkit intercepts that incoming call request and decides whether to answer the call covertly in the case of calls from a specified number or pass the call request to the phone app to show the incoming call. See Figure 3.9 below:

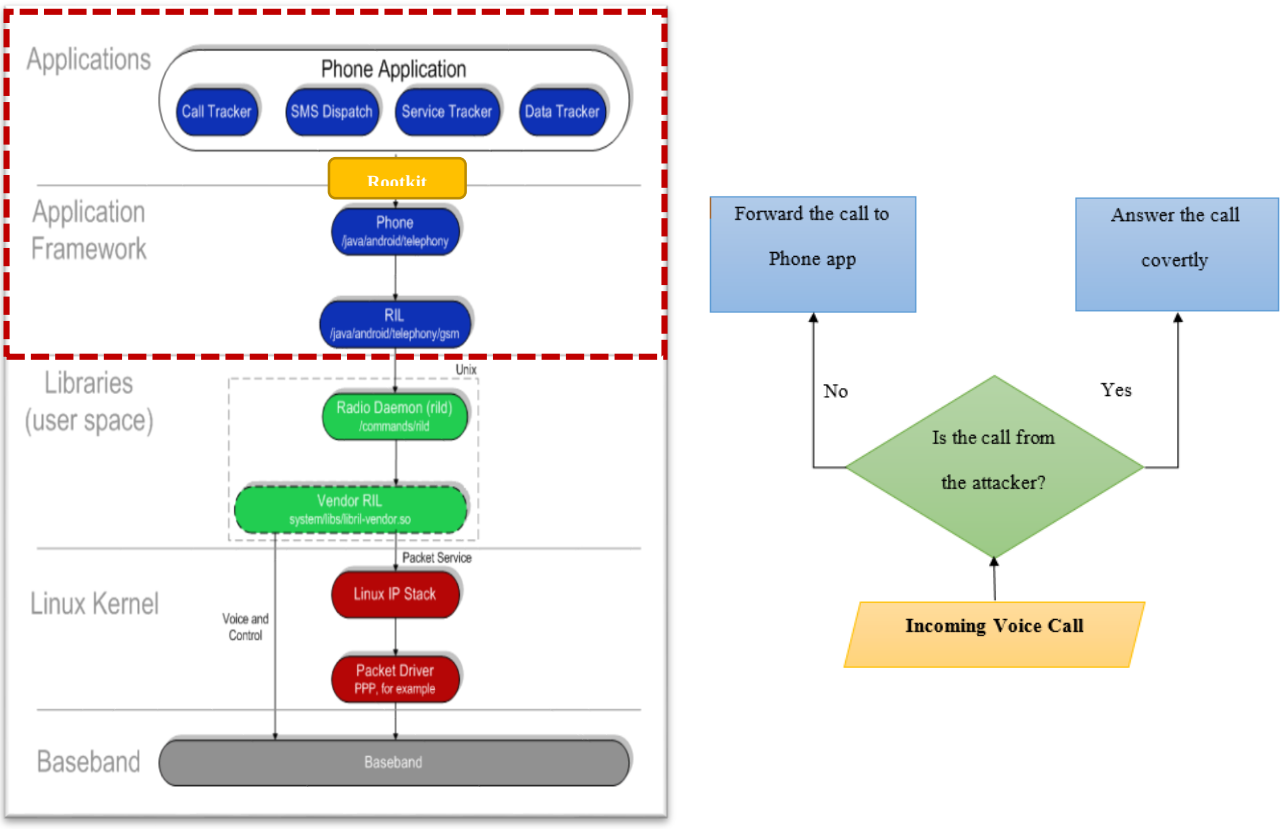


Figure 3.9: Rootkit Location and Control Flow on Android's Telephony

3.2.5. The Rootkit Implementation

Rootkit was implemented as an Android service that should be installed in the system partition of the application layer because it needs a privileged access. Rootkit works as follows: rootkit service starts with the system boot and works as a persistent service, so when it is killed by the system due to a shortage of system resources, it is restarted again. When the service runs, it works inside the phone process and the phone process main thread. The service creates a handler and hooks itself and the handler to the main looper of the main thread in phone process. Once the handler has been created, it becomes able to obtain and send messages from/to the phone main thread message queue.

When the handler starts, it registers itself with the PhoneBase's Registrant List to be able to listen to messages that come from Phone Base object and to receive notifications before CallManager. When the handler receives a new ringing connection message from the Phone Base, it gets the caller ID and compares it with a predefined number. If the caller ID does not match, it leaves the message to arrive at its destination. If it matches, it answers the call before the Phone App has been notified and hinders the main looper by starving its message queue. The handler sends two messages in front of the message queue, hence hindering it.

In fact, while implementing rootkit, it was essential to use the Android telephony internal classes and hidden components. However, as stated in Android documentation, these internal and hidden API telephony classes could not be reached by other applications other than the phone process. This rule was simply achieved by excluding the jar files of

the internal and hidden APIs from the developer environment, such as Android Studio and Eclipse environments, and no extra steps were taken in Android to prevent using these APIs at the run time. So the solution was simply to include these jar files in the developing environments to be able to use them. See Appendix A for more details.

3.3 Experimental Design

Using a real cellular infrastructure is more reliable than a simulated network; therefore, a real cellular network infrastructure and real smartphone were used during the experience to validate the theory. The T-Mobile GSM network was chosen to test the ability to employ the cellular voice channel as a covert channel that transfers covert information. The developed systems in this research could be run in all Android phones. However, in some Android devices there is a need to modify some Android system audio files in order to control the voice call stream.

The smartphones used in this experiment are Samsung Galaxy S3 and Galaxy Nexus as shown in Table II and Table III with their technical specifications. These two phones were chosen because of their ability to control the voice call stream by default; thus no extra work was needed to modify some system audio files to control the stream. The voice call was only used for the experiments. Both smartphones run Android Jelly Bean 4.3.3. Galaxy Nexus runs Cyanogenmod custom ROMs, and Samsung Galaxy S3 runs a rooted official ROM.

TABLE II: GALAXY NEXUS WITH TECHNICAL SPECIFICATIONS

Feature	Description
Device name	Galaxy Nexus
Android OS version	Android 4.3
System chip	Texas Instruments OMAP 4460
System memory	1 GB
CPU	1.2 GHz dual-core ARM Cortex-A9
GSM	850, 900, 1800, 1900 MHz
Additional microphone/s	Yes

TABLE III: SAMSUNG GALAXY SIII WITH TECHNICAL SPECIFICATIONS

Feature	Description
Device name	Samsung Galaxy SIII 19300
Android OS version	Android 4.3
System chip	Samsung Exynos 4
System memory	1 GB
CPU	1.4 GHz quad-core ARM Cortex-A9
GSM	850, 900, 1800, 1900 MHz
Additional microphone/s	Yes

3.3.2 Experimental Design Overview

When a channel is used to transfer information against the system design policy, it is considered a covert channel. In order for a covert channel to be utilized, an individual or malware must exist to convey the data. Therefore, different scenarios are discussed in order to show different statuses that could occur with this kind of security threat. These tests were executed to prove functionality and effectiveness of this proof of concept for using the cellular network as a covert channel.

3.3.2.1 First Scenario

The first scenario was implemented to investigate the ability to utilize the cellular network as a bidirectional covert channel. An Android application was developed to allow two individuals to communicate by manipulating a shared resource, which is the cellular voice channel, in an unexpected way, thus violating information flow policies. In this scenario, two individuals exchange text messages using a smartphone application and the cellular voice channel as a shared resource. The messages are modulated as speech-like and then travel through the cellular network core. The other smartphone application should receive the speech-like waves and demodulate them to get the original data. In this scenario, only the software audio modem was used and combined with a user interface to help the two individuals to send and receive the messages. The application resides in the application layer as a regular application that works as a software audio modem. Thus, any kind of

information could be leaked intentionally from the sender to the receiver. Figure 3.10 shows the covert channel experiment:

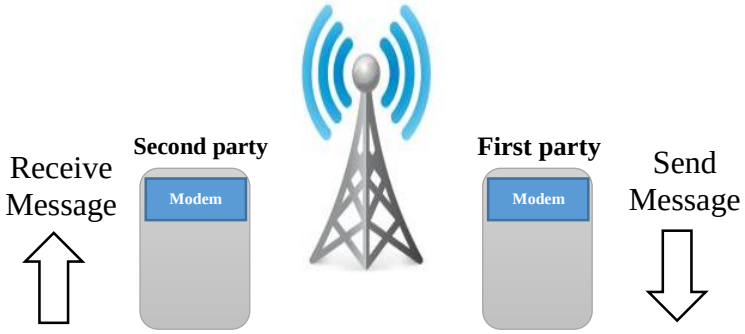


Figure 3.10: Covert Channel Experiment

3.3.2.2 Second Scenario

The second scenario examines the ability to use the proposed covert channel to leak information unintentionally from one side, while the other side desires a successful communication. This kind of communication is known as side channel. In this scenario rootkit works in the hacked smartphone and monitors and filters the incoming calls. Once rootkit has received an incoming voice call from a predefined number, it opens the voice channel covertly before it shows up in the smartphone's Phone App to allow data exchange.

In this scenario, when the channel is opened, the last SMS in the hacked smartphone will be modulated and sent over the covert channel to be acquired by the other side. The other side uses the software audio modem application to modulate the audio waves and see the last SMS. In addition, theoretically any kind of data could be leaked such as a picture

or video; however, this implementation focuses on text data and was used only to validate the proposed covert channel. Figure 3.11 shows the side channel experiment:

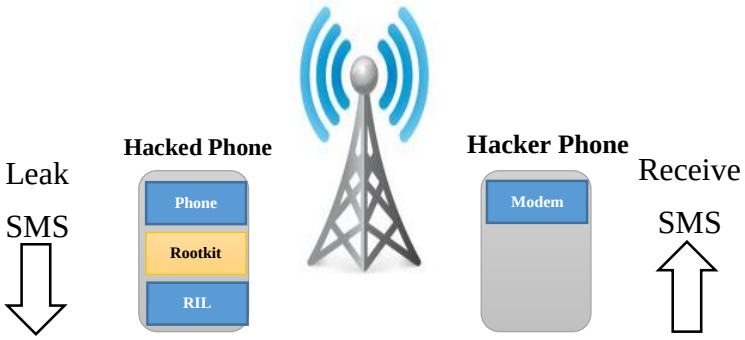


Figure 3.11: Side Channel Experiment

3.3.2.3 Third Scenario

The third scenario was implemented to test the proposed covert channel to be involved in botnets as C2 channel. With any botnet, C2 channel is the most significant part that contributes into its covertness and effectiveness. The designed system is similar to the second scenario, as it combines both the audio modem and rootkit. However, when the rootkit opens the incoming voice channel based on a decision made regarding the caller ID number, the rootkit listens to any command that is sent by the other side and executes it directly instead of leaking information from the hacked device. See Figure 3.12. In this case, rootkit will act as a botnet that listens to commands and executes them as required. Table IV includes some of the offered commands in this scenario:

TABLE IV: SOME COMMANDS WITH THE IMPLEMENTED BOT

Command	Description
Reboot	Reboot the system.
Clrlog	Clear call log.
Blueto	Switch Bluetooth on.

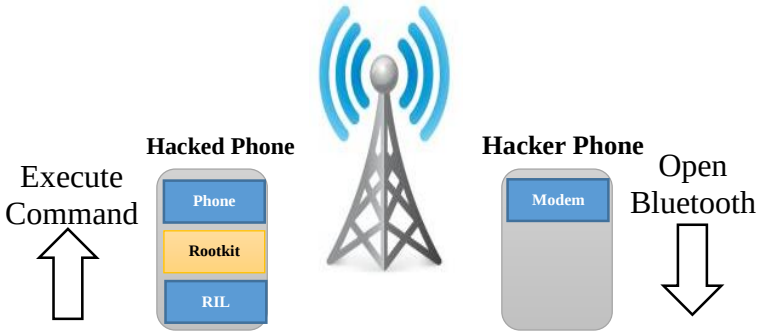


Figure 3.12: Botnet Experiment

3.3.2.4 Rootkit Situations

Rootkit makes some decisions based on the current satiation of the phone call state as follows:

- When the hacker calls during a hung up call between the victim and another party: rootkit will automatically reject the hacker's call without being noticed by the victim.

- When the hacker calls during as idle phone state, rootkit will answer the call covertly and allow data exchange.
- When the victim tries to initiate an outgoing call during a hung up call, or when the victim receives an incoming call waiting request during a hung up call by the hacker, the Phone App will not respond to the request and will be idle, because its main loop has been strived by a denial of service attack that was accomplished by rootkit.

3.4 Development Environment

The development system was 64-bit Windows 7 with 8 GB of RAM, Intel core i7, 450 GB of hard drive space. Android Studio Environment was utilized to develop the systems. The telephony internal classes and hidden components were added to the Android Studio Environment to enable them in the development environment. In addition, multiple tools were required for implementing and testing the systems in the Android environment as follows:

- Android Debugger Bridge (adb), which is a command line interface to establish communication between the developing computer devices and smartphone devices that are connected by USB cable. This tool is utilized to access a shell terminal in Android devices for transferring an application.

- Smali/baksmali is a tool that is used as an assembler/disassembler for the dex format used by DVM. This tool was used to compile and decompile the service.jar in order to modify it and bypass the signature verification.
- Dex2jar is a tool to read the Dalvik Executable (.dex/.odex) format.
- 7-zip is a tool is used for manipulating archives.
- OpenSSL is a tool used in the research to create and manage the private keys and public keys.

For further details describing how to set up the development environment, see Appendix A.

3.5 Deployment

Installing the developed software audio modem system was accomplished simply by installing it in the application partition of Android. However, the rootkit that involves the audio modem needed to be installed in the system partition, because it needs root privileges in order to run. The following instructions highlight the important steps that are followed to deploy the rootkit to the Android smartphone:

1. Launch a terminal.
2. Start the connected device shell terminal.
3. Obtain root access.
4. Mount Android system.

5. Install rootkit in system app folder, and install the audio modem in data app folder.
6. Unmount Android system.

Appendix A includes more details about the system deployment.

3.6 Summary

This chapter described in detail the architecture and design considerations for developing an audio modem and rootkit to proof the capability of the cellular voice channel to convey covert information. The systems were implemented and validated using T-Mobile GSM cellular network. Samsung Galaxy S3 and Galaxy Nexus smartphone devices were chosen for the research. The development environment and deploying method also were explained. In addition, different scenarios that were tested to validate the proof of concept were illustrated. The test results and system analysis are introduced in the next chapter.

CHAPTER IV

RESULTS AND DISCUSSION

4.1 Introduction

This section introduces a discussion about the system design and implementation analysis and the different tested scenario results that were mentioned in chapter 3. In addition, it discusses the overall analysis of the covert channel criteria. Screenshots of the smartphone screen during the experiments are provided. The tests and results discussion focuses on the capability to achieve a bidirectional communication over the cellular voice channel and from/to smartphone application layer to prove the theory. However, the performance is not covered in this discussion.

4.2 Modem Design and Implementation Analysis

The audio modem design aims to test the capability to generate an audio wave that could be modulated/demodulated successfully between the smartphones through the cellular voice channel. Another purpose is to verify the ability to develop a software audio modem that could be installed in the smartphone application layer and access the voice call stream to inject audio data.

The modem was tested successfully in Samsung Galaxy S 3 I9300, Galaxy Nexus, and Nexus S GSM versions. In fact, any smartphone that has the ability to record the voice call is prone to the botnet scenario, as it can receive any audio wave as a command and execute it. Samsung Galaxy S 3 was one of these devices. Understanding the vendor-specific audio subsystem mechanisms is important in order to successfully deploy the modem to the smartphone.

The modem implementation works in most Android smartphones, but the ability to access the voice call stream varies among the Android smartphones. Some smartphones can reach the voice stream from Audio Manager directly; however, reaching the voice stream of the cellular call in the other smartphones could be accomplished by modifying some Android system files. The current modem implementation with the capability to reach the voice call stream by default works on most Samsung Galaxy S series and Nexus smartphones.

The modem modulation technique that was used in this research is not the perfect one that could be implemented. It was used only to prove the concept as it has some

drawbacks that affect conveying data. Hence this design could be optimized in order to achieve better performance. The modem works only in Android platform; however, it can be also implemented in other platforms. Overall, the implemented modem successfully proved the ability to employ the cellular voice channel to act as a covert channel, and this can be achieved using smartphones.

4.3 Rootkit Design and Implementation Analysis

User-mode rootkit design has two primary purposes—building a rootkit that is able to communicate to the baseband modem and run covertly in any rooted Android OS, and filtering the incoming call to answer the voice calls, if needed, before they show up on the Android screen. In addition, it was important to verify the ability to implement a portable rootkit that can work easily without modifying underlying system files or applications. In fact, the portability factor is significant to test the ability to deploy the rootkit in any smartphone device, which indicates the ease of its distribution among smartphones.

Rootkit has not been tested in the new Android version Kit Kat 4.4. Although Kit Kat added a new feature to the DVM, which is called Android Runtime (ART), this operating system runs DVM as default and ART is still in its testing phase. The difference between DVM and ART is that DVM uses the just in time (JIT) method to compile the Android app, so each time the app runs, DVM compiles part of it to machine code. ART, on the other hand, compiles the whole app Dalvik bytecode during the installation into

a system-dependent binary, and it does that one time. However, Kit Kat OS also could be prone to infection by this rootkit.

Rootkit implementation successfully verified what it was meant to do. Rootkit was able to be portable and work silently. It runs in only Android platform, because the study focused on Android platform since its sources are accessible. Rootkit works in all Android-rooted stock ROMs, as well as most custom ROMs that have Jelly Bean 4.3.3 version or below. Rootkit was tested successfully in Samsung Galaxy S 3 I9300, Galaxy Nexus and Nexus S, Samsung Galaxy S 4, and Samsung Galaxy Duos y GSM versions, and it is believed to work in most GSM and CDMA Android smartphones.

4.4 Covert Channel Analyses

4.4.1 Covertness

The channel is difficult to detect because it is not known and predicted by an adversary. Another factor that contributes to the covertness is that these channels and the core network do not hire any network security guard, such as Firewalls and IDS to analyze the network traffic, which could carry a malicious content since the voice channel has never been used to exchange data between two cellphones.

4.4.2 Capacity

The channel capacity is related to the covertness. The channel is more covert when less data is transferred at a time; however, in this channel the voice call throughput was entirely occupied in order to convey covert information. This kind of channel is considered an active channel that establishes a new connection to exchange covert information. Active channels usually offer significantly higher throughput as compared to passive channels but that could be from ease of detection. In addition, a cellular voice channel fits into the noisy covert channel classification, as it could intentionally or unintentionally drop the data frame and so increase the transmitted covert data error rate.

The used GSM speech codecs affect the bit rate of the data. In addition, the speech codec can be switched during calls, because cellular network providers also can control and increase the number of active calls within one base station by switching cellphones to a low bitrate speech codec. That will impact the data transfer rate, as it could vary among the speech codecs and also in one speech codec at the time when the base station experiences an overload.

4.4.3 Robustness

The channel robustness determines the ease of limiting the channel capacity by adding noise or the ease of removing the covert channel. Therefore, the channel robustness is high, because it is not reliable to remove or add noise to this kind of channel, since it will affect the legitimate data transferring quality.

4.5 Scenario Analyses

The audio modem experiment that was performed in different scenarios showed some variability in results. In tested scenarios, when ideal conditions occurred where the surrounding environment is quiet, the smartphone hardware is loud and clear, the air interface is not noisy, and the call is carried out over one speech compression technique. The results were perfect and accurate, as the second party got the exact sent message.

In a realistic scenario, these conditions are not always guaranteed, so any conditions can easily either hinder the message from being transformed or omit some frames in the sent message. However, the audio modem design can be enhanced and optimized to overcome these constrains. The objective of the audio modem implementation is only to verify basic functionality and show possible scenarios that can be implemented successfully on real smartphones.

The goals of rootkit implementation were to prove the ability to build a tool that answers the voice call secretly without the knowledge of the smartphone's owner or the Phone App and to show the ability to leak information or receive commands through the voice channel. The test results show that rootkit works as it was designed. However, the only problem is when the rootkit answers the call from the attacker; the phone process message queue will be blocked, and any new messages cannot be delivered such as a new incoming call.

4.5.1 First Scenario: Covert Channel

Test objective

This test aims to show the capability to hire the cellular voice medium for exchanging data covertly between two cooperating parties and to examine the reachability of smartphone voice call stream within the application layer of Android.

Test Results

Figure 4.1 displays a screenshot of the smartphones to show test results. During an active voice call between two individuals, when the receive button in the audio modem application was clicked on one smartphone, it received the audio signals that had been sent by the other party over the cellular network. The second smartphone received the audio signals and demodulated them successfully, and it was able to show the original data that has been sent from the other smartphone. This successfully validates the modem's ability to utilize the cellular voice channel as a carrier of the generated audio signals by the smartphone. However, the receiver sometimes could not get the exact original data for several reasons, such as noisy environment, the frame stealing scenario, smartphone audio hardware quality, and the modem implementation.

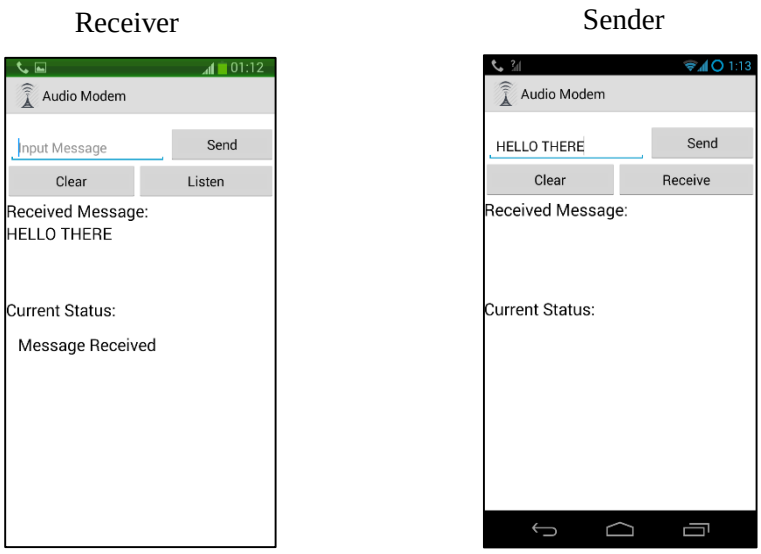


Figure 4.1: Covert Channel Test Screenshots

4.5.2 Second Scenario: Side Channel

Test Objective

This test examines the ability of the rootkit to open the voice channel secretly on the hacked smartphone and leak data from a hacked smartphone through the cellular voice channel located in the hacker smartphone.

Test Results

Figure 4.2 displays a screenshot of the smartphones to show test results. When the attacker made a call to the victim, rootkit recognized the attacker caller ID and based on that fact, it answered the call without showing up on the victim's screen. The victim had no idea about the ongoing voice call in his smartphone. Rootkit leaked the last received SMS in the victim's device by using the software audio modem. The attacker obtained the SMS by using the developed audio modem. However, the victim might hear audio waves

played in his/her phone, and this issue can be overcome by using modulation techniques that simulate a natural sound like a bird or cricket, as these sounds could be played as a notification in some applications.

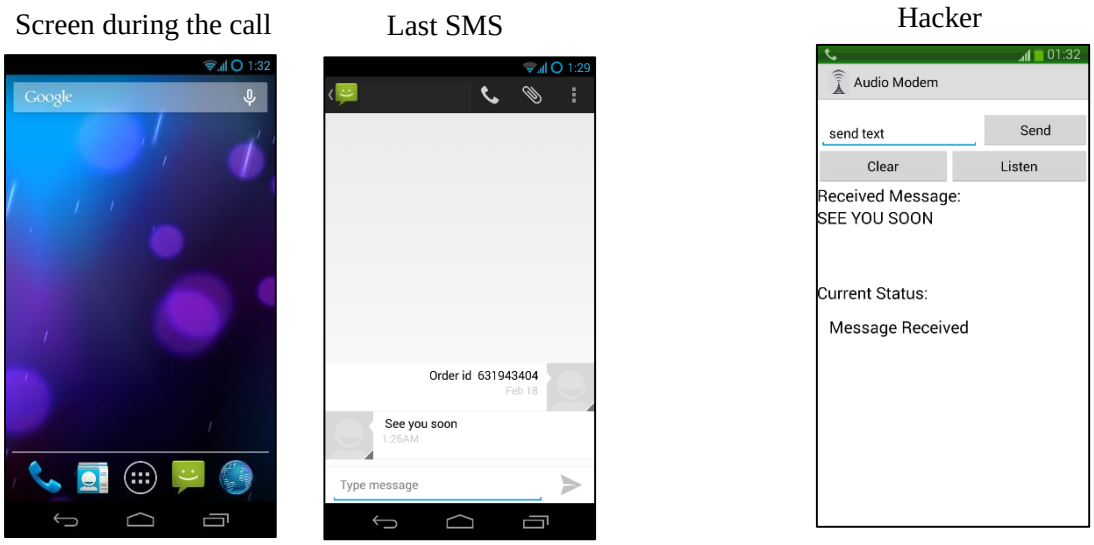


Figure 4.2: Side Channel Test Screenshots

4.5.3 Third Scenario: Botnet

Test objective

This experiment tests the capability to use rootkit to open the voice channel secretly when the attacker calls. Then the hacked smartphone receives commands through the cellular voice channel from the attacker's smartphone to execute them.

Test Results

Figure 4.3 includes screenshots of the smartphones to show test results. When the attacker made a call to the victim, rootkit recognized the attacker caller ID and based on that answered the call without showing up on the victim's screen. Rootkit then waited to receive a command, and once it was obtained, it was executed. The attacker sent the command using the developed audio modem.

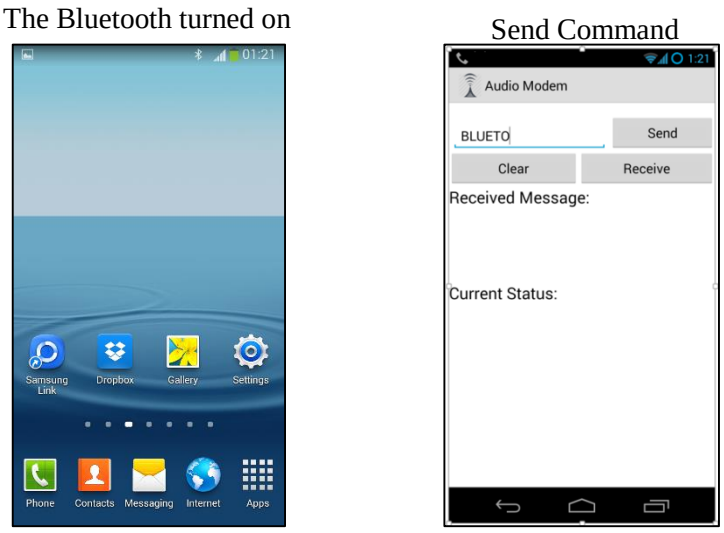


Figure 4.3: Botnet Test Screenshots

4.6 Summary

The developed systems and scenario results and analyses were provided in detail in this chapter. The audio modem and rootkit design and implementation analysis were discussed. In addition, the tested scenarios' objectives and test results were explained. The next chapter provides a conclusion of the research and recommendations for future work.

CHAPTER V

CONCLUSION AND FUTURE WORK

5.1 Introduction

This chapter concludes the efforts undertaken in this thesis. It discusses the significance of the achieved research, provides recommendations for future research, and summarizes the entire effort.

5.2 Significance of Research

As smartphones are trending to increase their computational capabilities, employees and individuals increasingly rely on smartphones to perform their tasks, and as

a result smartphone security becomes more significant than ever before. One of the most serious threats to information security, whether within organization or individual, is covert channels, because they could be employed to leak sensitive information, divert the ordinary use of a system, or coordinate attacks on a system. Therefore, detection of the covert channel is considered an essential task.

This research takes a step in this direction by detecting a potential covert channel which could affect smartphone security. It provides a proof of concept of the ability to use the cellular voice channel as a covert channel to leak information or distribute malware. It introduces details of designing and implementing the system and the challenges and constrains that have been faced to accomplish the system. It has been realized during this research that as smartphone hardware and software designs have changed recently, it allowed and contributed to the issue discussed in this research, as the new design allows access to the voice stream path from the application processor.

This research also proves that communication between the application processor and the baseband processors is vulnerable to attack in Android OS. In addition, it discusses some of the Android security mechanisms that were easily bypassed to accomplish the mission. The paper illustrates some discovered flaws in Android application architecture that allow a break in significant and critical Android operations.

5.3 Future Recommendations

When the covert channel had been discovered, implementing and testing was important to verify its effectiveness. Multiple scenarios were provided to give insights into how to eliminate or hinder its functionality. The research shows that this kind of covert channel can be practically applied and used. Bandwidth of this channel was sufficient for bidirectional communication. The covertness and the reliability were also good.

After identifying the covert channel, usually some countermeasures should be performed that might include eliminating the channel, limiting the bandwidth of the channel, auditing the channel, and documenting the channel. Obviously, this kind of channel could not be eliminated and its capacity could not be reduced, because these choices would affect significantly on the voice call availability and quality. However, auditing and documenting should be involved at setting the countermeasures in future work.

APPENDIX A: DEVELOPMENT ENVIRONMENT

This Appendix illustrates the development environment and includes the following sections:

A.1 Downloading the Android Environment

Step-by-step instructions for downloading the Android Studio Environment can be found at [43]. The following are steps taken from the website to set up the environment in Windows OS.

1. Download and install JDK 6 or greater.
2. Select **Start menu > Computer > System Properties > Advanced System Properties**. Then open **Advanced tab > Environment Variables** and add a new system.
3. Set an environment variable JAVA_HOME to indicate the JDK folder.
4. Download the **Android Studio** package.
5. Install Android Studio and the SDK tools.

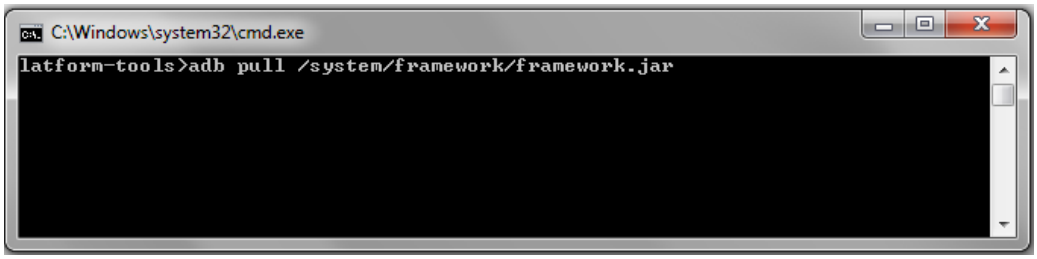
6. Launch the downloaded EXE file, android-studio-bundle-<version>.exe.
7. Follow the setup wizard to install Android Studio.

A.2. Adding the internal telephony APIs and hidden components into Android Environment

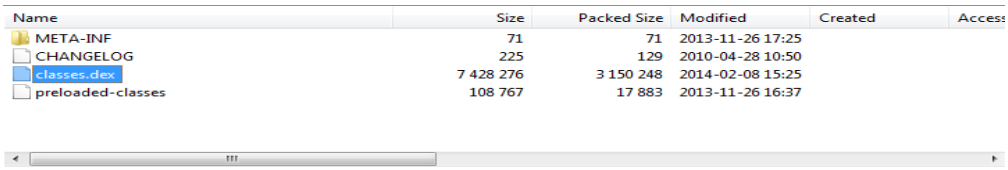
During the development of an Android project using Android Studio Environment, the environment uses Android classes that reside in a jar file called android.jar. The android.jar could be found in Android SDK platform directory SDK /platforms/platform-X/android.jar, where X is the API level number. The Android internal classes and hidden components have been eliminated from the android.jar file. When launching the developed application on a real smartphone, the smartphone loads framework.jar from the smartphone /system/framework/ framework.jar that has the same classes in android.jar and all the internal API classes and hidden API functions.

This section has instructions to add internal and hidden APIs to Android Studio Environment that are helpful during developing time [44]:

1. Select Android platform X as a target. In this research, API Level 18 platform has been utilized.
2. Plug the device or the emulator to pull the jar files: framework.jar, framework2.jar, and telephony-common.jar from /system/framework/ folder



3. Open the files framework.jar, framework2.jar, and telephony-common.jar using 7-ZIP File Manager to copy classes.dex file.



4. Using dex2jar tool, convert classed.dex files to classes.jar.

5. This produces classes-dex2jar.jar file, Rename theses files to framework-classes.jar, framework2-classes.jar, and telephony-classes.jar.
6. Now open the SDK location that is combined with Android Studio Environment.
7. Go to SDK/platform/ and copy the target platform folder android-X to android-X-modified. In this research the generated folder was android-18-modified.
8. Open android.jar in android-X-modified.
9. Open framework-classes.jar, framework2-classes.jar, and telephony-classes.jar.

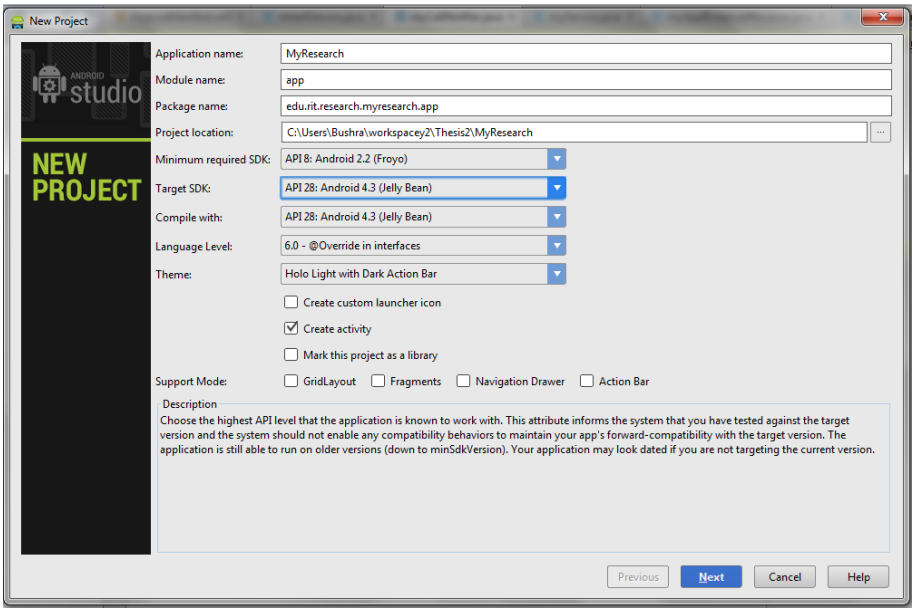
10. Copy the contents of framework-classes.jar, framework2-classes.jar, and telephony-classes.jar to android.jar, replacing all existing files.
11. Open build.porp file in android-18-modified folder, then change the number of the API level to distinguish between original and modified in the following line: ro.build.version.sdk = 18 to ro.build.version.sdk = 28.

```
# begin build properties
# autogenerated by buildinfo.sh
ro.build.id=JB_MR2
ro.build.display.id=sdk-eng 4.3 JB_MR2 819563 test-keys
ro.build.version.incremental=819563
ro.build.version.sdk=28
ro.build.version.codename=REL
ro.build.version.release=4.3
ro.build.date=Tue Sep 10 18:43:31 UTC 2013
```

12. Open source.properties file in android-18-modified folder, and then change the line: AndroidVersion.ApiLevel= 18 to AndroidVersion.ApiLevel = 28.

```
Pkg.Desc=Android SDK Platform 4.3
Pkg.UserSrc=false
Platform.Version=4.3
Platform.CodeName=Jelly Bean
Pkg.Revision=2
AndroidVersion.ApiLevel=28
Layoutlib.Api=10
Layoutlib.Revision=1
Platform.MinToolsRev=21
```

13. Now open Android Studio Environment.
14. Create new project and choose the modified SDK.

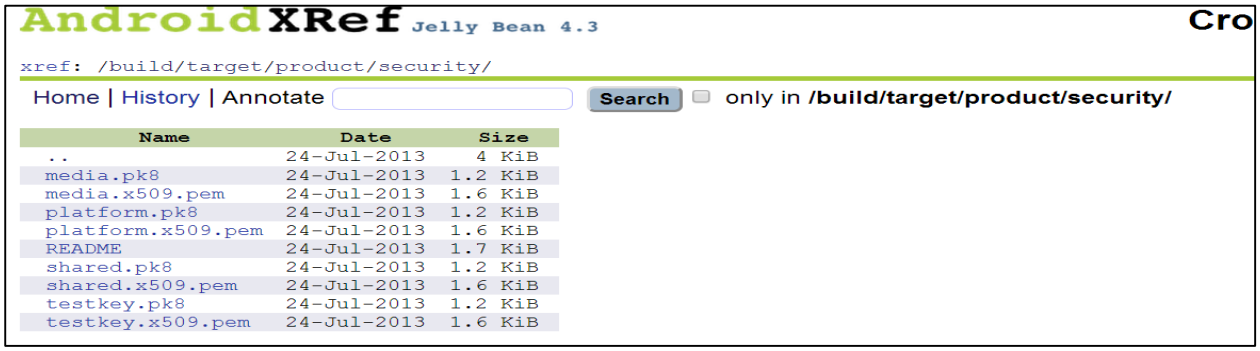


A.3. Obtaining and issuing the certificate

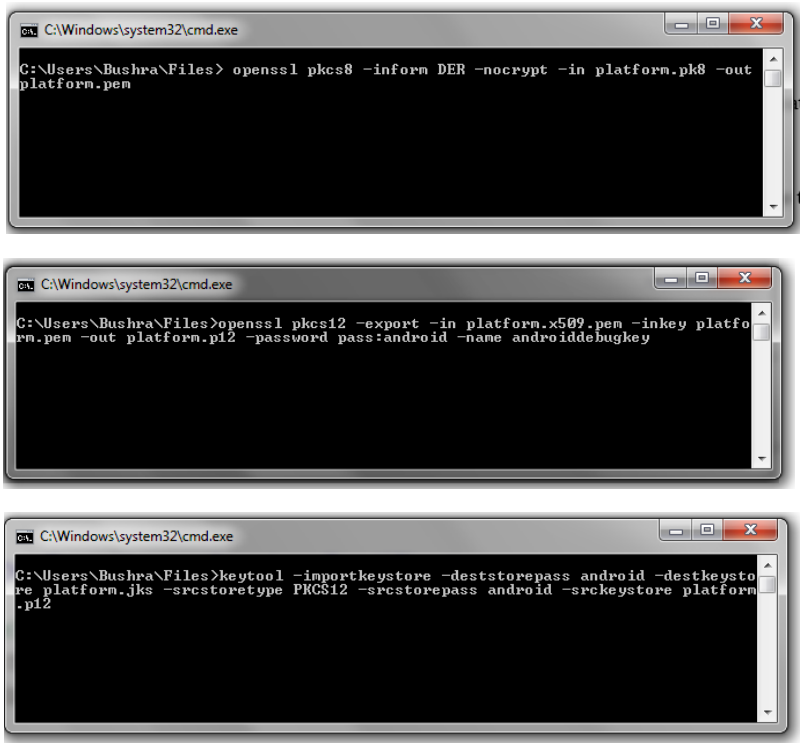
To generate the certificate used with some custom ROMs like CyanogenMod, follow the instructions below:

- Obtain the platform key and certificate, platform.pk8, and platform.x509.pem from AndroidXref in the following URL:

  http://androidxref.com/4.3_r2.1/xref/build/target/product/security/



- After that, obtain Openssl and SignApk.jar, which is a tool that creates the private key from the certificate and public keys.
- Copy the files SignApk, platform.pk8, and platform.x509.pem to Openssl directory.
- Then create the keystore, open cmd.exe on Windows within Openssl directory and issue the following commands:

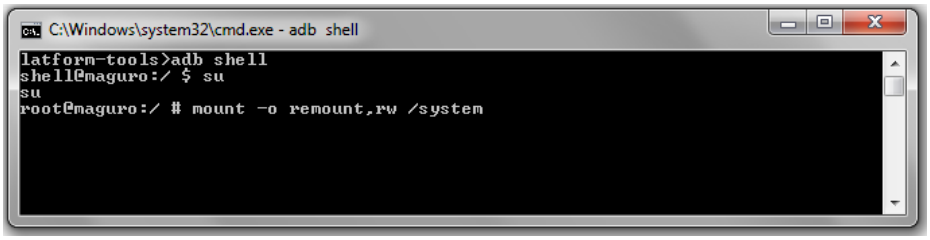


- Save the generated file platform.jks to use with Android Studio to generate the certificate.
- After compiling the Android project with Android Studio, sign the project with the generated file above.

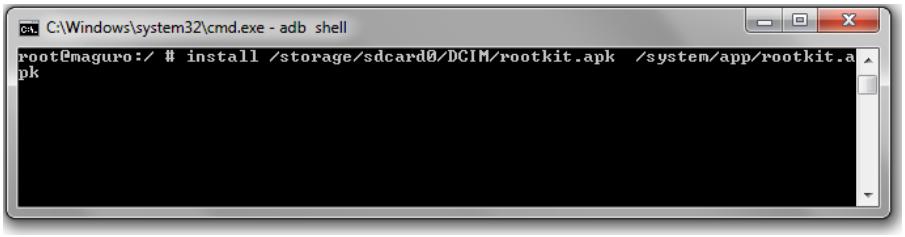
- And open Android Studio -> Build -> Generate Signed apk->enter the password->chose the platform.jks file.

Now the signed Android app can be deployed as follows:

- Plug in the smartphone.
- Copy the signed apk to the smartphone device storage.
- Open the adb path.
- Remount the system partition read-write.



- Install the application to the system apps folder.



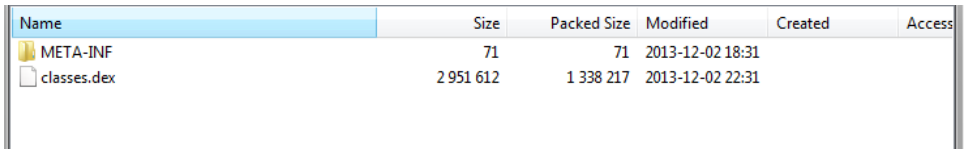
- Remount the system partition read-only.

A.4. Modifying the Android Services.jar

To modify Android Services.jar, the next instructions were followed [45]:

- Pull Services.jar file from the Android smartphones.

- Open the .jar file with 7zip, extract the classes.dex, and copy it into a new folder.



- Copy baksmali.jar and smali.jar to the created folder also.
- Open the cmd terminal and change directory to the created folder.
- Issue the following command to decompile the classes.dex file:

- This command creates a new folder called classout where all of your smali files will be found.
- Go into the classout folder and look for the following file: \com\android\server\pm\PackageManagerService.smali and replace the entire following method:

```
pareSignatures([Landroid/content/pm/Signature;[Landroid/content/pm/Signature;)I
    .locals 7

    if-nez p0, :cond_8

    if-nez p1, :cond_6

    const/4 v6, 0x1

    :goto_5
    return v6

    :cond_6
    const/4 v6, -0x1

    goto :goto_5

    :cond_8
    if-nez p1, :cond_c

    const/4 v6, -0x2

    goto :goto_5

    :cond c
    new-instance v3, Ljava/util/HashSet;

    invoke-direct {v3}, Ljava/util/HashSet;-><init>()V

    move-object v0, p0

    array-length v2, v0

    const/4 v1, 0x0

    :goto_14
    if-ge v1, v2, :cond 1e

    aget-object v5, v0, v1

    invoke-virtual {v3, v5}, Ljava/util/HashSet;->add(Ljava/lang/Object;)Z

    add-int/lit8 v1, v1, 0x1

    goto :goto 14

    :cond_1e
    new-instance v4, Ljava/util/HashSet;

    invoke-direct {v4}, Ljava/util/HashSet;-><init>()V

    move-object v0, p1

    array-length v2, v0

    const/4 v1, 0x0

    :goto 26
    if-ge v1, v2, :cond_30
```

```
    aget-object v5, v0, v1

    invoke-virtual {v4, v5}, Ljava/util/HashSet;->add(Ljava/lang/Object;)Z

    add-int/lit8 v1, v1, 0x1

    goto :goto_26

    :cond_30
    invoke-virtual {v3, v4}, Ljava/util/HashSet;->equals(Ljava/lang/Object;)Z

    move-result v6

    if-eqz v6, :cond_38

    const/4 v6, 0x0

    goto :goto_5

    :cond_38
    const/4 v6, -0x3

    goto :goto_5
.end method
```

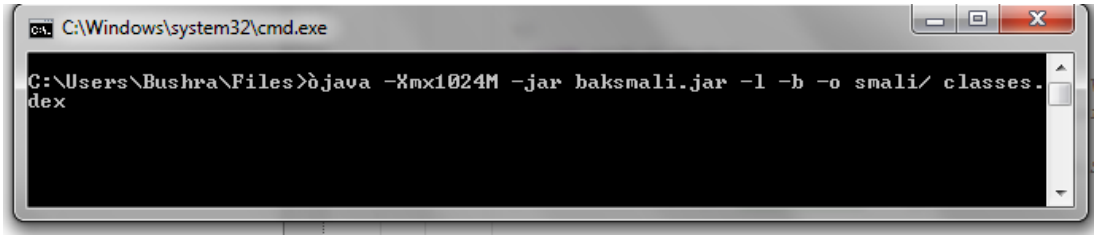
- Change and replace the entire .method above to this code:

```
.method                                                                    static
compareSignatures([Landroid/content/pm/Signature;[Landroid/content/pm/Signature;)I
    .locals 7

    const/4 v0, 0x0

    return v0
.end method
```

- Compile the classout folder into a new classes.dex by issuing the following command:



REFERENCES

[1] "Industry first: Smartphones pass PCs in sales - Google 24/7 -Fortune Tech." [Online]. Available: http://tech.fortune.cnn.com/2011/02/07/idc-smartphone-shipment-numbers-passed-pc-in-q4-2010/.

[2] J. Eberspächer, C. Bettstetter, and H.-J. Vögel, *GSM - Architecture, Protocols and Services*. Hoboken, NJ, USA: Wiley, 2009.

[3] "J-STD-025 Rev. A- Lawfully Authorized Electronic Surveillance." Telecommunications Industry Association 1997-2000, 2000.

[4] P. Traynor, P. McDaniel, and T. L. Porta, *Security for Telecommunications Networks*. Springer, 2008.

[5] H. Welte, "Anatomy of contemporary GSM cellphone hardware." 2010.

[6] R.-P. Weinmann, "Baseband Attacks: Remote Exploitation of Memory Corruptions in Cellular Protocol Stacks," in *Proceedings of the 6th USENIX Conference on Offensive Technologies*, Berkeley, CA, USA, 2012, pp. 2–2.

[7] R. Kratsas, "Unleashing the Audio Potential of Smartphones." Cirrus Logic, 2010.

[8] T. Fryer, "Better sound to single chip," Jun. 2013.

[9] "Android Architecture – The Key Concepts of Android OS," 2012. [Online]. Available: http://www.android-app-market.com/android-architecture.html.

[10] "Radio Layer Interface." [Online]. Available: http://www.kandroid.org/online-pdk/guide/telephony.html.

[11] "Android Platform Development Kit-Audio Subsystem." [Online]. Available: http://www.netmite.com/android/mydroid/development/pdk/docs/audio_sub_system.html .

[12] B. W. Lampson, "A Note on the Confinement Problem," *Commun. ACM*, vol. 16, no. 10, pp. 613–615, Oct. 1973.

[13] "National Institute of Standards and Technology-Trusted Computer System Evaluation Criteria." Aug. 1983.

[14] R. A. Kemmerer, "A Practical Approach to Identifying Storage and Timing Channels: Twenty Years Later," in *Proceedings of the 18th Annual Computer Security Applications Conference*, Washington, DC, USA, 2002, p. 109.

[15] N. Salwan, S. Singh, S. Arora, and A. Singh, "An Insight to Covert Channels," *arXiv:1306.2252 [cs]*, Jun. 2013.

[16] "Chapter 2 Covert Channels," *Docstoc.com*. [Online]. Available: http://www.docstoc.com/docs/120433303/Chapter-2-Covert-Channels.

[17] A. Giani, V. H. Berk, and G. V. Cybenko, “Data exfiltration and covert channels,” 2006, vol. 6201, pp. 620103–620103–11.

[18] S. Ring and E. Cole, “Taking a lesson from stealthy rootkits,” *IEEE Security Privacy*, vol. 2, no. 4, pp. 38–45, Jul. 2004.

[19] A. Apvrille, “Symbian worm Yxes: Towards mobile botnets?” *J Comput Virol*, vol. 8, no. 4, pp. 117–131, Nov. 2012.

[20] P. Porras, H. Saïdi, and V. Yegneswaran, “An Analysis of the iKee.B iPhone Botnet,” in *Security and Privacy in Mobile Information and Communication Systems*, A. U. Schmidt, G. Russello, A. Lioy, N. R. Prasad, and S. Lian, Eds. Springer Berlin Heidelberg, 2010, pp. 141–152.

[21] R. Templeman, Z. Rahman, D. Crandall, and A. Kapadia, “PlaceRaider: Virtual Theft in Physical Spaces with Smartphones,” *arXiv:1209.5982 [cs]*, Sep. 2012.

[22] “More droiddream details emerge: It was building a mobile botnet.” [Online]. Available: http://readwrite.com/2011/03/06/droiddream_malware_was_going_to_install_more_apps_on_your_phone#awesm=~oz6kzUDBY6bNiK.

[23] “BBOS_ZITMO.B,” *Trend Micro Threat Encyclopedia*. 2011.

[24] Y. Wang, K. Streff, and S. Raman, “Smartphone Security Challenges,” *Computer*, vol. 45, no. 12, pp. 52–58, Dec. 2012.

[25] K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang, “Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones,” in *the 18th Annual Network and Distributed System Security Symposium (NDSS).* 2011.

[26] M. Hansen, R. Hill, and S. Wimberly, “Detecting covert communication on Android,” in *2012 IEEE 37th Conference on Local Computer Networks (LCN)*, 2012, pp. 300–303.

[27] S. J. Murdoch and S. Lewis, “Embedding Covert Channels into TCP/IP,” in *Information Hiding*, M. Barni, J. Herrera-Joancomartí, S. Katzenbeisser, and F. Pérez-González, Eds. Springer Berlin Heidelberg, 2005, pp. 247–261.

[28] M. Bauer, “New Covert Channels in HTTP: Adding Unwitting Web Browsers to Anonymity Sets,” in *In Proceedings of the Workshop on Privacy in the Electronic Society (WPES 2003)*, 2003, pp. 72–78.

[29] T. Takahashi and W. Lee, “An Assessment of VoIP Covert Channel Threats,” in *Third International Conference on Security and Privacy in Communications Networks and the Workshops, 2007. SecureComm 2007*, 2007, pp. 371–380.

[30] N. B. Lucena, J. Pease, P. Yadollahpour, and S. J. Chapin, “Syntax and Semantics-Preserving Application-Layer Protocol Steganography,” in *Information Hiding*, J. Fridrich, Ed. Springer Berlin Heidelberg, 2005, pp. 164–179.

[31] M. Z. Rafique, M. K. Khan, K. Alghatbar, and M. Farooq, “Embedding High Capacity Covert Channels in Short Message Service (SMS),” in *Secure and Trust*

Computing, Data Management and Applications, J. J. Park, J. Lopez, S.-S. Yeo, T. Shon, and D. Taniar, Eds. Springer Berlin Heidelberg, 2011, pp. 1–10.

[32] K. Papapanagiotou, E. Kellinis, G. F. Marias, and P. Georgiadis, "Alternatives for Multimedia Messaging System Steganography," in *Computational Intelligence and Security*, Y. Hao, J. Liu, Y.-P. Wang, Y. Cheung, H. Yin, L. Jiao, J. Ma, and Y.-C. Jiao, Eds. Springer Berlin Heidelberg, 2005, pp. 589–596.

[33] C. K. LaDue, V. V. Sapozhnykov, and K. S. Fienberg, "A Data Modem for GSM Voice Channel," *IEEE Transactions on Vehicular Technology*, vol. 57, no. 4, pp. 2205–2218, Jul. 2008.

[34] M. Rashidi, A. Sayadiyan, and P. Mowlaee, "A Harmonic Approach to Data Transmission over GSM Voice Channel," in *3rd International Conference on Information and Communication Technologies: From Theory to Applications, 2008. ICTTA 2008*, 2008, pp. 1–4.

[35] M. Rashidi and A. Sayadiyan, "A New Approach for Digital Data Transmission over GSM Voice Channel," in *Proceedings of the 2Nd WSEAS International Conference on Circuits, Systems, Signal and Telecommunications*, Stevens Point, Wisc., USA, 2008, pp. 193–196.

[36] A. Dhananjay, A. Sharma, M. Paik, J. Chen, T. K. Kuppusamy, J. Li, and L. Subramanian, "Hermes: Data Transmission over Unknown Voice Channels," in *Proceedings of the Sixteenth Annual International Conference on Mobile Computing and Networking*, New York, NY, USA, 2010, pp. 113–124.

[37] Y. Yang, S. Feng, W. Ye, and X. Ji, “A Transmission Scheme for Encrypted Speech over GSM Network,” in *International Symposium on Computer Science and Computational Technology, 2008. ISCSCT ’08*, 2008, vol. 2, pp. 805–808.

[38] L. Chen and Q. Guo, “An OFDM-based Secure Data Communicating Scheme in GSM Voice Channel,” in *2011 International Conference on Electronics, Communications and Control (ICECC)*, 2011, pp. 723–726.

[39] N. Katugampala, “Secure voice over GSM and other low bit rate systems,” 2003, vol. 2003, pp. 3–3.

[40] A. Shahbazi, E. Soltanmohammadi, A. H. Rezaei, A. Sayadiyan, and S. Mosayyebpour, “Content Dependent Data Hiding on GSM Full Rate Encoded Speech,” in *International Conference on Signal Acquisition and Processing, 2010. ICSAP ’10*, 2010, pp. 68–72.

[41] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, “TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones,” in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, Berkeley, CA, USA, 2010, pp. 1–6.

[42] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A. Sadeghi, "Xmandroid: A new android evolution to mitigate privilege escalation attacks." *Technische Universität Darmstadt, Technical Report,* 2011. *TR-2011-04*

[43] “Android Developer,” *Android Developer*. [Online]. Available: http://developer.android.com/sdk/installing/studio.html.

[44] “Using internal (com.android.internal) and hidden (@hide) APIs.” [Online]. Available: https://devmaze.wordpress.com/2011/01/18/using-com-android-internal-part-1-introduction/.

[45] “To disable signature checks.” [Online]. Available: http://forum.xda-developers.com/showthread.php?t=1698352.

[46] “Digital-voices.” [Online]. Available: https://github.com/diva/digital-voices/.