Fall 2015

# A Completely Covert Audio Channel in Android

Sukanya Thakur
*San Jose State University*

# A Completely Covert Audio Channel in Android

A Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Sukanya Thakur

December 2015

The Designated Project Committee Approves the Project Titled

A Completely Covert Audio Channel in Android

by

Sukanya Thakur

APPROVED FOR THE DEPARTMENTS OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

December 2015

|  |  |
|---|---|
| Dr. Mark Stamp | Department of Computer Science |
| Dr. Robert Chun | Department of Computer Science |
| Nikki Benecke Brandt | Monsanto Company |

## ABSTRACT

**A Completely Covert Audio Channel in Android**

**by Sukanya Thakur**

Exfilteration of private data is a potential security threat against mobile devices. Previous research concerning such threats has generally focused on techniques that are only valid over short distances (NFC, Bluetooth, electromagnetic emanations, and so on). In this research, we develop and analyze an exfilteration attack that has no distance limitation. Specifically, we take advantage of vulnerabilities in Android that enable us to covertly record and exfilterate a voice call. This paper presents a successful implementation of our attack, which records a call (both uplink and downlink voice streams), and inaudibly transmits the recorded voice over a subsequent inaudible call, without any visual or audio indication given to the victim. We provide a detailed analysis of our attack, and we suggest possible counter measures to thwart similar attacks.

## ACKNOWLEDGMENTS

Prof. Mark Stamp, my project guide, has supported me and kept me motivated during multiple hiccups and uncertainties I had been through in this project. He provided me with much needed guidance and information, while showing confidence in me especially at times when, in fact, I myself was uncertain on the outcome of my research and experimentation. I truly appreciate his patience and his encouragements all through this difficult path. His you-can-do-it encouragement and much needed timely tips have helped me develop independent thinking and climb the hill with confidence.

Prof. Robert Chun has inculcated in me the urge to develop conceptual clarity through his term paper assignments in his classes that I took during my graduate years in San Jose State. His presentation of key ideas in the most simple possible ways has always helped me see the points through the jargon loaded confusing cluster of literature on a subject. I am thankful to him for accepting to become my committee member.

Nikki Benecke Brandt's work on NFC fuzzing in Android has been the source of inspiration to me in deciding to go for this Android based project. She has always made herself available whenever I needed her inputs and help, even when it was a stretch for her during her busy schedule. I am gratified to have her as my committee member.

Finally I like to thank Apala, my daughter, for remaining awake with me in the nights, always ready to serve me coffee or tea, and for assisting me by making endless adversary calls during the experimentation.

## TABLE OF CONTENTS

CHAPTER

1 **Introduction** . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1

2 **Background** . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

2.1 Telephony Stack . . . . . . . . . . . . . . . . . . . . . . . . 5

2.2 Audio Stack . . . . . . . . . . . . . . . . . . . . . . . . . . 7

2.3 Audio Recording . . . . . . . . . . . . . . . . . . . . . . . . 9

2.4 Audio Replay . . . . . . . . . . . . . . . . . . . . . . . . . 10

2.5 Prior Work on Covert Channel using Audio Media . . . . . . . . . 11

2.6 Infection Mechanism . . . . . . . . . . . . . . . . . . . . . . 15

3 **Implementation** . . . . . . . . . . . . . . . . . . . . . . . . . 17

3.1 The Development Environment . . . . . . . . . . . . . . . . . . 18

3.2 Implanting a Call Monitor . . . . . . . . . . . . . . . . . . . 19

3.3 Implementing Soundles Replay of Audio Data . . . . . . . . . . . 22

3.4 Recording a voice call . . . . . . . . . . . . . . . . . . . . . 24

3.5 Installation . . . . . . . . . . . . . . . . . . . . . . . . . . 25

4 **Results** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

4.1 Experiment 1: Recording a Call conversation . . . . . . . . . . . 27

4.2 Experiment 2: Replaying covertly the recorded call to an incoming number . . . . . . . . . . . . . . . . . . . . . . . . . . . 28

4.3 Experiment 3: Replaying a large music file to a pre-determined incoming number . . . . . . . . . . . . . . . . . . . . . . . . 29

4.4 Experiment 4: Impact on normal phone behavior . . . . . . . . . 29

4.5 Discussion on the results . . . . . . . . . . . . . . . . . . . . . . . 30

**5 Countermeasures** . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31

**6 Conclusion and future work** . . . . . . . . . . . . . . . . . . . . . . 33

**APPENDIX**

**A Eclipse Setup** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36

A.1 Configuring SDK Path . . . . . . . . . . . . . . . . . . . . . . . . 36

A.2 Enabling Build of System App . . . . . . . . . . . . . . . . . . . . 36

A.3 ADT Hack: Allowing permission to use hidden API in Eclipse . . 37

**B Building and Integrating Android Components** . . . . . . . . . . 39

B.1 Build a shared library or jar file . . . . . . . . . . . . . . . . . . . 39

B.2 Manifest file to run the application inside Phone application . . . 40

B.3 How to sign using signApk . . . . . . . . . . . . . . . . . . . . . . 40

## LIST OF TABLES

1 Android Telephony and Audio stacks . . . . . . . . . . . . . . . . . . 5

2 Development Tools & Devices . . . . . . . . . . . . . . . . . . . . . . 19

3 Modules and their install destinations . . . . . . . . . . . . . . . . . 26

## LIST OF FIGURES

| | | |
|---|---|---|
| 1 | Main Modules of Telephony Stack | 6 |
| 2 | Main Modules of Audio Stack | 7 |
| 3 | AudioTrack implementation at the lower layer | 11 |
| 4 | Placing Call Monitor inside the telephony stack | 20 |
| 5 | Get the reference to the Phone using Reflection | 21 |
| 6 | Unregister the CallManager from Phone | 21 |
| 7 | Register the call monitor handler with the Phone | 21 |
| 8 | Modified AudioTrack uses direct call to tinyalsa | 23 |
| 9 | Writing the data directly into the pcm stream | 23 |
| 10 | Adding voice call device to the audio policy file | 24 |
| 11 | Adding voice call device profile to the audio policy file | 25 |
| 12 | Adding voice call channels as acceptable parameter value to AudioRecord API | 25 |
| 13 | Adding voice call streams to flinger | 26 |
| 14 | Calling Audio Record for recording for recording voice call | 26 |
| 15 | Experimental setup for showing covert call recording and replay | 28 |
| A.16 | Setting SDK path in Eclipse | 36 |
| A.17 | Enable System App build in Eclipse | 37 |
| B.18 | Entries in the in Manifest xml file to share the Phone process | 40 |
| B.19 | Signing the application | 41 |

# CHAPTER 1

# Introduction

Exfiltration of private data is a threat on mobile devices. Published attacks to exfiltrate data from mobile devices use various communication channels such as NFC, Bluetooth, Wi-Fi and FM radio receivers found in those devices [4, 8]. Availability of these media on mobile devices also provide opportunities for using them as receivers for data exfilteration from other non-mobile devices such as a computer using electromagnetic emanations [11]. While these media are commonly found in mobile devices, relying on such media results in attacks for which distance from the target is a limiting factor. This limitation, however, is absent when the cellular network itself is used for exfiltration. The cellular network can be vulnerable to attacks due to the fact that in most mobile phones the cellular interface (also known as baseband) is controlled by the main processor of the mobile device, a feature that leaves cellular media vulnerable to attack [10, 14]. Some published work explores the possibility of using a text channel provided by the cellular network as a medium for exfiltration [5]. While the texting over a cellular network does provide a viable option for removing any distance limitation, it suffers from the limitation that a data plan must be purchased for the mobile device, and text usage appears in the phone bill, which likely makes such an attack transient, at best.

In this project we have demonstrated that the audio media during a voice call can become a medium for exfiltration. There are several potential advantages in using audio during a voice call for exfiltration. As discussed above, voice does not have any distance limitation, and such usage is less likely to be observed on a bill. The main problem in using the audio media is that the voice calls are visually indicated and the audio transaction is audible.

We demonstrate that a voice call can be made completely covert—soundless and with no visual cue to the phone user. We demonstrate our implementation of covertly recording a call, and soundless replay of audio data to an incoming call and automatically connecting the exfiltration call without any visual indication. The implementation also demonstrates that incoming calls can be independently selected for either covert call recording or covert replay of audio data that includes recorded calls and other audio files, such as music. Soundlessness and the absence of visual cues together render a completely covert channel for exfiltration of a voice call.

In this paper we demonstrate covert exfiltration of two types of audio data—a recorded call conversation and a music file. There is published work [1, 8, 5] showing an ongoing trend of modulating data and video for exfiltration. Based on this trend, we believe that our work can easily be extended to exfiltrate data and video.

Our implementation of covert audio media during a voice call makes use of known vulnerabilities in Android, such as device rooting, easily exportable hidden APIs, installation of modules as shared libraries, and allowing an application to run inside the process of another application. Our purpose is to demonstrate the exploitation of these vulnerabilities so as to encourage the development of countermeasures to mitigate these potential security threats.

This paper is organized as follows. Chapter 2 provides the essential background information needed to understand our implementation. In Chapter 2, we give an overview of related work published in this field, and we discuss how our work contrasts with or builds upon the key ideas presented in previous work. Chapter 3 discusses our implementation in detail. Chapter 4 describes experiments that we have conducted to verify the claimed covertness of the implementation. Countermeasures to thwart the risk of attacks shown in our work are discussed in Chapter 5. In Chapter 6 we summarize our work and discuss a set of enhancements that could expose a much greater level of threat using the covertness of audio media during a voice call.

# CHAPTER 2

## Background

This chapter discusses necessary background of Android in order to understand the implementation approach and the outcome of this project. We will also discuss here briefly the prior work done in this area and how our work contrasts with or builds on the key ideas presented in those works. Android has a reference architecture simplified and illustrated by Table 1. The reference architecture is a layered architecture—the lower layers providing services to the upper layers. Android applications use the Java Application Programming Interfaces (API) exported from the Java/Framework layer. Framework layer API realize their functionality by using the services provided by the Hardware Abstraction Layer (HAL) components implemented in C/C++. Java API use Java Native Interface (JNI) to transact with the HAL API. Android Open Source Project (AOSP) provides the source code and build tools for Framework/Java API, JNI and HAL API for any published version of Android [2]. The HAL API interact with the linux kernel through software drivers provided mostly by device vendors. The lowest layer consists of various hardware such as audio devices, Baseband, and GPS. Baseband device interfaces with the Cellular network and naturally is one of the main hardware components in the mobile devices. Android functional modules such as Telephony and Audio modules conform to the Android reference architecture. Refer to the published Android Reference [13] for a detailed understanding of the Telephony stack. Refer to Android Audio Implementation site [3] for a detailed understanding of the Audio stack. Section 2.1 and Section 2.2 present the essential elements of the Telephony and the Audio stacks respectively. The material presented in these sections provide the necessary background for our implementation presented in Chapter 3.

Table 1: Android Telephony and Audio stacks

| **Android Ref Stack** | **Telephony** | **Audio** |
|---|---|---|
| Apps | Phone | Media player |
| Java API/Framework | PhoneGlobal, CallManager, Call Notifier, GSMPhone, RIL, ITelephony | AudioRecorder, AudioTrack |
| JNI | | libmedia_runtime |
| HAL | RIL daemon(rild) | libmedia, tinyalsa, AudioFlinger, AudioMixer |
| Linux Kernel | vendor RIL | ALSA |
| Hardware | Baseband Processor | Mic, Speaker, Bluetooth, earphone |

## 2.1 Telephony Stack

The Telephony stack implements the control path of a voice call whereas, the voice path is implemented in the Audio stack. The purpose of the control path is to process the signals from the underlying baseband device as well as from the phone user and manage the life cycle of a voice call. The main components of our interest in this stack are the Radio Interface Layer(RIL), BasePhone, CallManager, and CallNotifier. Each of these components are class objects. RIL interfaces with the RIL daemon in the HAL via a linux socket for receiving or sending call signals to the underlying software device driver (aka vendor RIL) that controls the baseband device. Drake et al. explains RIL and its related components details [6]. An internal PhoneApp application configures the Telephony stack by interconnecting the components to each other so that they form a layered hierarchy. The upper layer component registers with the lower layer to get notifications for certain events. The lower layer notifies its registered upper layer components when the event occurs. RIL, BasePhone, CallManager and CallNotifier form a stack sequence, RIL being at the lowest. BasePhone holds the reference to the underlying implementation of a phone object such as GSMPhone or CDMAPhone. BasePhone in turn is referenced by the ProxyPhone object. BasePhone registers itself with RIL object to receive and send the call control signals. CallManager registers with the BasePhone for almost all telephony events and gets notified by the BasePhone. The CallNotifier implements the call indication User Interface (UI) for incoming calls. CallNotifier shows the call indications when CallManager notifies it on an incoming call. These components form a path sequence for upward flow of a call signal. For example, an incoming call signal in the Java layer can travel through a path—`RIL ->BasePhone ->CallManager ->CallNotifier`. Figure 1 illustrates this arrangement and shows the register-notify hierarchy of the components discussed above.

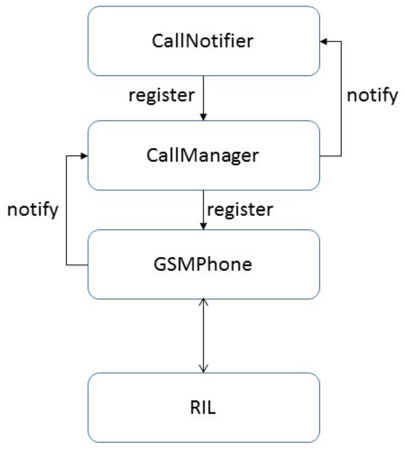


Figure 1: Main Modules of Telephony Stack

## 2.2 Audio Stack

The Audio stack manages the audio hardware devices such as speaker, earpiece, voice call device, bluetooth, headset, earphone and so on. The stack provides device control such as discovery of a device, enabling a device, and muting a device. In addition to controlling and managing the individual audio devices, one of the main functionality of the Audio stack is to manage and control the audio paths. Figure 2 shows the main components of Audio stack during a voice call. An audio path from any of the physical audio devices is called an audio channel. The digital audio data flowing through a channel is referred to as an audio stream. The voice channel and the voice stream refer to the audio channel and the audio stream during a voice call.

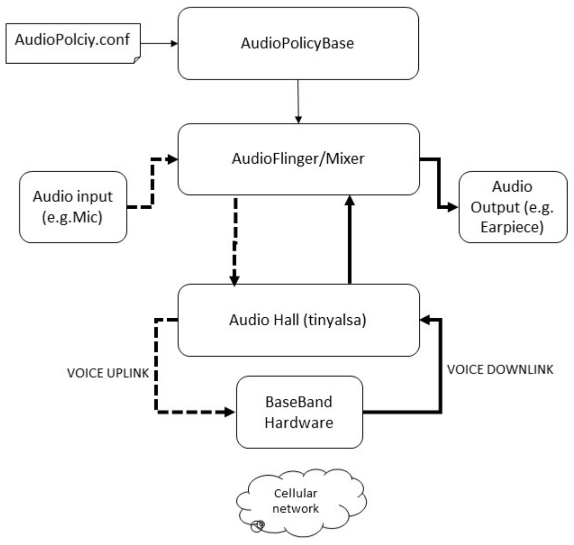


Figure 2: Main Modules of Audio Stack

In this discussion we are interested in the management of the voice streams and the voice channels during a voice call. A voice stream during a call is coded in Pulse Code Modulation (PCM) using 16 bits for each sample. The Voice Call device refers to a virtual device that provides and handles the voice channels and the voice streams during a voice call all the way to the Baseband via the HAL interface. The Audio stack uses tinyalsa as the HAL layer to control, read and write voice data during a call. The tinyalsa HAL layer provides API to read and write PCM data streams. These PCM streams are routed and mixed by AudioFlinger and AudioMixer to other devices. A voice path during a voice call consists of two channels - uplink and downlink. The uplink voice channel takes the voice stream from input devices such as microphone or bluetooth to the baseband. The downlink channel carries the voice streams from the baseband to the output devices such as speaker, earpiece, earphone or bluetooth.

One of the main functionality of the Audio stack, as mentioned above, is to provide pathways to audio streams by switching them among multiple channels. In addition to switch the streams among multiple channels, the stack provides also a mixer for re-sampling and mixing the audio streams to improve the audio quality and to feed the streams appropriately to channels. For example, both uplink and downlink voice streams are mixed and fed to the speaker. The switching and mixing of audio streams are controlled by a policy manager. The policy manager runs as a system service and is implemented by AudioPolicyManagerBase.cpp module in the HAL. The switch and the mixer are implemented by AudioFlinger and AudioMixer modules respectively in the HAL. The policy manager defines and creates the pathways. The policy manager uses an external configuration file, "audio_policy.conf", located in the /etc/ directory. The policy configuration file is a plain text file. It defines the input and output audio devices and their profiles. A profile define the device characteristics used by the mixer and flinger. For example, the profile for a voice call device defines number of bits per sample, sampling rate, number of channels to be mixed and so on. AudioPolicyManagerBase reads the configuration information at the boot time and creates internal data structure for the AudioFlinger. If the external audio policy configuration file is not present, the policy manager creates a default configuration for each device. The audio policy manager creates a data structure that represents the configuration information and used by the flinger and mixer modules. The flinger switches the streams among the defined pathways based on the configuration information. Prior to switching the streams to a channel, the flinger sends the stream to the mixer with appropriate mixing parameters. The output of the mixer is fed to the destination channel.

The Audio stack publishes AudioRecord and AudioTrack API to applications for reading or writing the audio streams to user files. AudioRecord outputs a specified stream for writing to an external file. AudioTrack takes a user stream and feeds it to an audio output device via the flinger. AudioRecord and AudioTrack are discussed in Section 2.3 and Section 2.4 respectively.

## 2.3 Audio Recording

AudioRecord is used for audio recording. The API returns audio stream from a user requested audio input device. The API takes the audio input device as the audio source parameter. AudioRecord.java implements the user interface of the API and AudioRecord.cpp implements the main functionality. In the user interface layer, the user provided parameters are validated and a call to JNI is made. The JNI part creates the AudioRecord.cpp object, and invokes the set method of this object. The set method in turn makes a call to the flinger to open an output channel that can be used to provide an outlet for the stream requested by the user. The flinger reads the device information and its profile from the configuration data created by the policy manager to validate the user request. At this point the user request for the stream output can be declined depending on the policy defined by the configuration information. Once the request is validated and accepted, the AudioRecord object sets up buffers that are shared between the user thread and the flinger. So the flinger will output the requested stream in the shared buffer. The user is expected to consume the stream data from the buffer to prevent any overrun. AudioRecord provides a read method to extract stream data from the buffer. After every read, the buffer is released.

Android official site mentions the voice call device as one of the possible input devices that can be accepted by the API. However, starting Android 4.0, the voice call device implies only the uplink voice channel from the microphone. The downlink voice channel is simply ignored by the API. This prevents recording of a two-way call conversation.

## 2.4 Audio Replay

AudioTrack is used by applications for writing user audio stream into an audio output device such as speaker and bluetooth. AudioTrack.java implements the user interface and AudioTrack.cpp implements the main functionality. A JNI interface provides the transition and data transfer between the two modules. Figure 3 shows how this API interfaces with the HAL layer. The user interface transfers control to JNI after a preliminary validation of user arguments. The JNI creates the AudioTrack.cpp object and calls its set method. The set method in turn makes a call to the flinger for opening an input channel that can provide the user stream a path to the requested device for output. The flinger validates the user request using the configuration information created by the policy manager. An input channel is opened or declined depending on the configuration information. The implementation does not support a user stream to feed a voice call directly as there is no standalone voice call device configuration available for direct output. In addition to this, the flinger and mixer are configured in such a way that the user audio stream is never routed to a voice call. The configuration allows the downstream voice channel to be connected to only audible devices such as speaker, headphone, earpiece or bluetooth. The upstream voice channel is fed from only microphone or bluetooth. Any alteration to this scheme would require a major code change in flinger. The above implementation of AudioTrack API makes it considerably difficult to make a voice call inaudible.

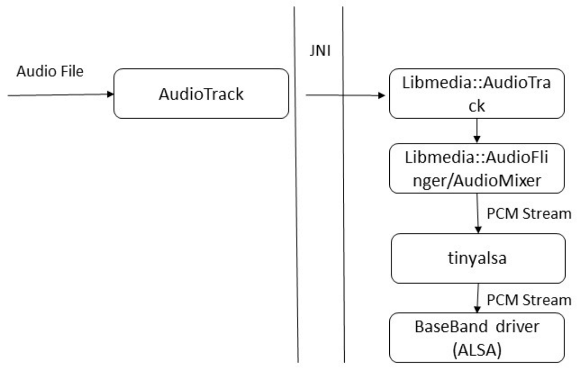


Figure 3: AudioTrack implementation at the lower layer

## 2.5 Prior Work on Covert Channel using Audio Media

Multiple works have been published demonstrating proof of concepts on implementing covert audio channels in Android for data exfilteration. These works fall under two main categories—a) Modulating the data and transmitting it in the inaudible frequency range through an audio media; and b) Modulating the data and transmitting it in the audible range using voice path. While transmitting data in inaudible range requires the adversary mobile device to be located in close proximity to the compromised device, data transmitted in the audible range is essentially not covert.

The work of Do et al. [5] presents use of SMS and audio media for covertly exfiltrating data from a mobile device. Do et al. argue that SMS and audio media are commonly available media on mobile devices and exfiltration over these media is difficult to be controlled. The authors propose the SMS medium for long range, large data exfiltration; and the audio medium for short range, small data exfiltration. Base64 is used to convert data to text prior to sending it via SMS. The entire base64 encoded data is split into a series of SMS messages prior to transmission. As messages via SMS are not guaranteed to be delivered in sequence, the receiver uses message index inserted by the sender to reassemble the messages in order. The messages are deleted immediately after sending in order to avoid detection at the victim phone. Scheme proposed by Do et al. for using audio medium for exfiltration relies on the fact that speakers and microphones on mobile devices can generate and received audio frequencies outside the human audible range. The work shows inaudible broadcast of user key-press sequences from the victim phone speaker and subsequent reception of the inaudible broadcast by the adversary microphone. The key press on the victim phone is tapped and the sequence of 1 and 0 of the key code is converted to inaudible frequency range to be broadcast from the speaker. 20 kHz, 22 kHz and 21 kHz are used respectively to represent 0, 1 and space between any two data bits. The sequence 101 is sent without any spacer frequency to indicate completion of the binary bit pattern for a key-press. Both the schemes presented by Do et al. suffer from serious limitations inherent in the chosen media of exfiltration. Use of SMS can face limited availability of media and can run into risk of detection. Availability of SMS depends on the service plan of the mobile phone, and can be detected by the usage record in the phone bill. The user can be visibly warned and/or further transmission can be blocked when the size of exfiltration exceeds the permissible data capacity of the phone by the service plan. These incidences can expose the attack. The audio scheme proposed by the authors suffers from distance limitation and data transmission errors due to noise present in the medium. The authors found that the reception accuracy was 100% only up to 1.7 m and the reception dropped entirely by 3.7 m. No data has been presented by the authors to clearly show the impact of noise on transmission errors, however they observed that in the cafe environment, reception dropped entirely at 2.9 m.

Work by Guri et al. [8] presents covert data exfitration from a computer with the help of a mobile phone using audio media. The data (text or binary) from the computer is first modulated to audio tones using either Audio Frequency Shift Keying (A-FSK) or Dual Tone Multiple-Frequency (DTMF). The audio signal is then frequency modulated using a carrier frequency in the FM radio range. The carrier frequency in FM radio range is generated by the video display unit by constructing an image of alternating sequence of black and white pixels. Once emanated from the video display unit, the signal is like a FM radio broadcast that is received by FM Radio in mobile phone. This FM audio is recorded by the mobile phone by modifying the MediaRecorder class, to extract and demodulate the data. In order to avoid potential detection by reception of changing tones in the FM radio in the mobile phone, the FM reception band is changed. Guri et al. work by using audio media for covert exfiltration suffers from distance limitation. Their results show that data could be transmitted with 97.73% accuracy up to a maximum distance of only 7 m.

Audio/FM modulated data transmission is slow, the results showing approximately 11 hours to transfer a raw data of size 0.5MB.

The work of Aloraini [1] attempts to use voice call as a covert media for transmitting audio modulated text data. In this work, a call monitor is inserted within the telephony stack to watch for an incoming call from a predetermined number. Upon receiving a call from that number, the call is connected without any visual indication. The text data is converted to audio using FSK modulation prior to transmission. The transmission of data however, remains audible. Moreover, the phone user cannot make any call during the exfiltration. The work justifies the covertness by arguing that the modulated signal sound is not human intelligible. The audibility and the user call blocking scheme however, makes the attack detectable, and the overall scheme of exfiltration fails to remain covert. In addition, the work is not suitable for large data exfiltration as the phone will remain blocked during the entire duration of exfiltration. The scheme is also not suitable for exfiltration of voice data, for example, a recorded call or music as it will become intelligibly audible.

Our work demonstrates a completely covert voice path that is inaudible with no visual indication. We have also provided a way to record any call conversation by breaking the limitation of AudioRecord to record only uplink voice. We also handle the call states within the call monitor to avoid blocking of user initiated calls during the exfiltration. In this project we have implemented and shown only audio data exfiltration. This is only a superficial limitation and can be removed by deploying the published work to convert non-audio data to audio data.

## 2.6 Infection Mechanism

With all its attempts to providing security to application and data, Android remains vulnerable because of its architectural limitations and practices. As Drake and Elenkov show in their respective [6, 7] A complete treatise on Android vulnerabilities are available in the separately published works of Drake et al. and Elenkov [6, 7]. We highlight some of those here as examples. Android uses shared libraries to implement API at the HAL layer and uses dex jar files for implementing API at the application layer. By knowing the Android version of a phone (which can be easily automated by yet another malicious software implanted in the device), it is easy to build a shared library or the dex jar files for that version in the AOSP and transport it to the device once the device is rooted. Rooting a device is routinely done by the adversaries. Another vulnerability of Android lies in the fact that an application can potentially run in the process of another application and can enjoy the privileges of the host application. The only real prevention provided by Android here is that the incumbent application should be signed with the same certificate as the host application. There are multiple ways of overcoming this restriction including use of publicly available signing tools. Another easily vulnerable security mechanism used by Android is to hide an API by using @hide pragma of Java and creating a rule in the ADT plugin used by SDK. This security arrangement is easily breakable with tools like d2j_dex2jar that can easily extract the Java jar files from dex code and include those in the SDK. A simple hack in ADT can easily be done, as shown in Appendix A to work around the ADT rule. After including the extracted jar files from dex jar files and removing the ADT rule, all the hidden API can be accessed using Java reflection. Java reflection allows access to even private fields and methods. Finally, it remains a fact that all security mechanisms implemented by Android ends at the

Java layer. Access to the HAL layer when a device is rooted, bypasses all the Android specific security layers. Once the device is rooted, the HAL layer opens up a wide attack surface. Modifying the HAL layer open source or using the needed API from this layer is easy. Our work demonstrates the exploitation of all the above Android vulnerabilities.

# CHAPTER 3

# Implementation

This chapter describes the implementation of our project. There are essentially three parts to our implementation. Section 3.2 discusses the first part of implementation for covertly watching for incoming phone numbers. This part of implementation is also responsible for triggering the covert recording of a call, and automatically connecting to an incoming call for covert audio exfiltration.

The second part implements a covert recording of an incoming call from a predetermined number. As discussed in the Section 2.3, Android does not allow recording of a two-way call conversation during a voice call. Only the uplink voice stream can be recorded. Section 3.4 discusses our implementation to remove this restriction.

The final part of our implementation deals with inaudible exfiltration or replay of an audio stream directly into a voice call. As discussed in Section 2.4, this feature is simply not available in Android. A major code change is required to overcome this limitation. Section 3.3 discusses our much simplified implementation for injecting any audio data covertly into a voice call.

Before delving into details, we would like to mention that learning Android programming environment and methodology is an essential skill to be acquired for the implementation. The Android programming concepts and examples illustrated in [12] have been extremely useful during our implementation.

## 3.1 The Development Environment

All Android applications and libraries are compiled and built on a host system using the cross compilation and build tootkits for the target mobile device. Hence a development environment is to be set up for cross-compiling and building the application and the libraries. Our host system consists of Ubuntu 14.04LTS running on Windows 7 based VMPlayer. We used Eclipse Luna with Android Development Tool (ADT) plugin. Eclipse provides an Integrated Development Environment (IDE) for developing Android applications. Both the Android SDK and the Android Open Source Project (AOSP) source tree are installed on the host. SDK and ADT plugin together provide the API, the toolkit and the build rules for Android application development. AOSP comes with the required toolkit for building the shared libraries and dex modules such as framework.jar. Refer to Android sites for downloading and configuring AOSP, ADT and SDK [2, 9]. In addition to the standard tools mentioned above, other tools are also needed to support our special needs of rooting the device, signing the application and extracting dex files. Table 2 lists the tools and devices used for the implementation.

Rooting of the target phone device is critical for our implementation. We could use the rooting process fully explained in [4]. We chose to build and download the Android on the target device in the debug mode, giving us automatic root access [6]. In order to use certain hidden API, the SDK needed to be extended with the class files of certain core classes in framework and telephony modules. This was achieved by extracting the required jar files from the dex modules in the target device using the d2j_dex2jar tool. Eclipse needed to be configured for setting up SDK path, building system application and enabling access to hidden API. Android ADT plugin in Eclipse needs some hacking for permitting use of hidden API. Appendix A gives the steps

Table 2: Development Tools & Devices

| Tools & Devices | Development Usage |
|---|---|
| Ubuntu 14.04LTS | Operating system on the host VM |
| Eclipse Luna | IDE for building application |
| ADT | Android plugin for Eclipse |
| SDK | Development kit |
| AOSP 4.3 | Building shared library |
| WugFresh Nexus Root Toolkit v1.8.2 | Rooting the target device |
| d2j_dex2jar 2.0 | Extract the java jar file from dex files |
| signApk | Signing the application |
| Android Phone firmware Maguro 4.3 JWR66Y | Experimental target phone device |

for setting up Eclipse including the hack for working around the ADT plugin rules. Appendix B gives the steps to carry out for building libraries in the AOSP, creating the required entries in the application Manifest file and signing the application.

## 3.2 Implanting a Call Monitor

The call monitor application needs to share the process of PhoneApp so that it can register for receiving call notifications. The call monitor is implemented as an Android Service application and is inserted inside the PhoneApp process representing a Man-in-Middle attack. The approach is similar to the work of Aloraini [1]. We, however, handle the call states in the call monitor as opposed to simply blocking the handlers. This makes the call tapping fully covert by avoiding any potential notice taken by the phone user.

In order to share the PhoneApp process, the application has to have the same user id and certificate as the PhoneApp. As the PhoneApp is installed as a system application, our application is installed under /system/app. Appendix A shows the entries in the Manifest.xml file of the application and describes the steps to sign the application.

In order to insert our call monitor between the BasePhone and the CallManager, the references to the BasePhone, the CallManager and their corresponding call handlers are obtained using Java reflection mechanism. CallManager has a single handler that is used to handle all the call related notifications. The call monitor application registers itself with the BasePhone and de-registers the CallManager handler from the BasePhone. The application then registers the CallManager handler to itself. This process inserts the application between the BasePhone and the CallManager making it receive the call notifications prior to the CallManager.Figure 4 illustrates how the call monitor is positioned within the Telephony stack.

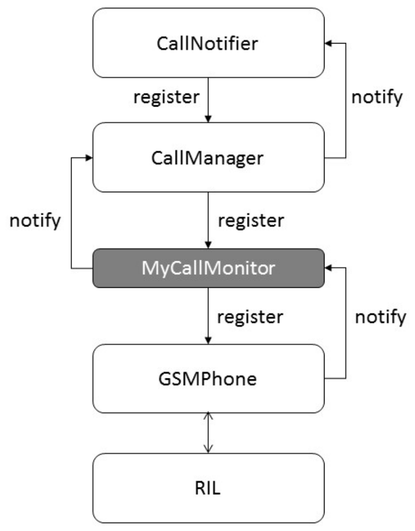


Figure 4: Placing Call Monitor inside the telephony stack

Figure 5, Figure 6 and Figure 7 shows essential code snippets for placing the call

```
Phone phone = null;
try {
  phone = (Phone) m.invoke(c);
  Log.i("Sukanya", "Phone␣Found");
} catch(IllegalArgumentException e) {
  //TODO Auto-generated catch block
  e.printStackTrace();
} catch(IllegalAccessException e) {
  //TODO Auto-generated catch block
  e.printStackTrace();
} catch(InvocationTargetException e) {
  //TODO Auto-generated catch block
  e.printStackTrace();
}
Phone basePhone = phone.getForegroundCall().getPhone();
```

Figure 5: Get the reference to the Phone using Reflection

monitor between the BasePhone and the CallManager.

```
try {
  myHandler = (Handler)mCMHandler.get(mCM);
  Log.i("Sukanya", "CallManager␣Handler␣found");
} catch(IllegalArgumentException e) {
    e.printStackTrace();
} catch(IllegalAccessException e) {
    e.printStackTrace();
}
basePhone.unregisterForIncomingRing(myHandler);
this.registerForIncomingRing(myHandler, EVENT_INCOMING_RING, null);
```

Figure 6: Unregister the CallManager from Phone

```
basePhone.registerForNewRingingConnection(mHandler,
   EVENT_NEW_RINGING_CONNECTION, null);
basePhone.registerForIncomingRing(mHandler, EVENT_INCOMING_RING, null);
```

Figure 7: Register the call monitor handler with the Phone

The call monitor is responsible for monitoring incoming calls and suppressing any visual indication when an audio replay is to be done. In our implementation we hard coded a phone number for secretly recording the call, and another number for replaying the recorded call. When the call monitor detects the incoming call from the recording number, it starts the recording by using the modified AudioRecord API. The incoming call notifications are also passed to the CallManager so that the call appears to be a normal call. Current implementation has a hard coded file name to store the recording. When the incoming call from the replay number is detected, the call monitor automatically connects the call and suppresses the incoming call notification from sending it to the CallManager. Hence the CallNotifier is never invoked to give any visual indication. The call monitor starts the replay by calling the MyAudioTrack. The MyAudioTrack API uses tinyalsa API for sending the audio stream directly into the voice call. Implementation of MyAudioTrack is explained in Section 3.3.

## 3.3 Implementing Soundles Replay of Audio Data

The flinger replays all audio files by feeding the stream into output device such as Speaker. A voice call streams are fed to the standard input and output devices. The flinger is coded in such a way that it would require considerable re-design and coding to disassociate the connections of a voice call streams from the channels of the audio devices. However our research has identified a shortcut path to directly feed the audio file stream into a voice call uplink stream. The mechanism uses the fact that with root permission, it is possible to read/write the PCM streams by making direct calls to tinyalsa API. This mechanism bypasses the flinger making the replay completely soundless. We have implemented a user level API, MyAudioTrack that interfaces with tinyalsa using the standard JNI and C++ object. This approach enables access to tinyalsa from user application level. Figure 8 illustrates the mechanism explained above.

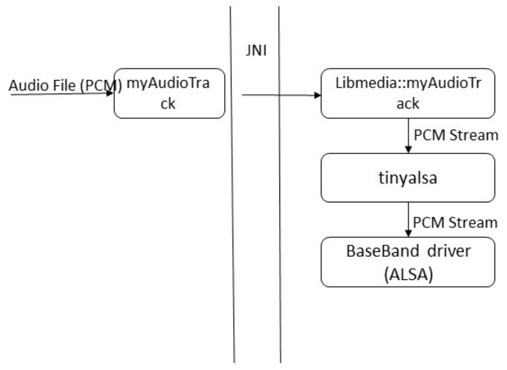


Figure 8: Modified AudioTrack uses direct call to tinyalsa

The user interface of MyAudioTrack is integrated with the framework.jar. The JNI for this API is created and integrated in libandroid_runtime.so and MyAudioTrack.cpp is created and integrated in the libmedia.so. MyAudioTrack provides a play method to play the audio to the voice call. The play method implemented in the C++ end of MyAudioTrack uses the tinyalsa APIs to write to baseband device directly.

Figure 9 gives a pseudo code used to insert the PCM data of a file directly into the baseband device.

```
pcmdev = open_pcm(card, device)
file = open_file(filename)
do:
  nbytes = fread(file, buffer, BUFSIZE)
  pcm_write(pcmdev, buffer, nbytes)
while nbytes > 0
```

Figure 9: Writing the data directly into the pcm stream

### 3.4 Recording a voice call

The existing audio policy implemented by Android restricts the voice recording stream only to be fed from the microphone input stream. This implies that only uplink voice during a call can be recorded as it is sourced from the microphone. Recording of both the uplink and the downlink streams of a voice call is prevented by excluding the voice call device from the list of input devices in the audio configuration file; and by dropping the channel entries for a voice call from the list of channels in sInChannelsNameToEnumTable array.

First we need to make the necessary entries in the "audio_policy.conf" file. To enable the recording of a voice call, the voice call device symbol, `AUDIO_DEVICE_IN_VOICE_CALL`, and its profile data were added to the policy configuration file. The required voice channels were entered in the sInChannelsNameToEnumTable array in the AudioPolicyManagerBase.cpp file. In addition, the hidden audioParamCheck method of AudioRecord class was made aware of the uplink and downlink voice channels. Figure 10 shows the entry for the voice call device. Figure 11 shows profile entry. Both entries are added to the audio policy configuration file.

```
global_configuration {
  attached_output_devices AUDIO_DEVICE_OUT_EARPIECE |
   AUDIO_DEVICE_OUT_SPEAKER
  default_output_device AUDIO_DEVICE_OUT_SPEAKER
  attached_input_devices AUDIO_DEVICE_IN_BUILTIN_MIC |
   AUDIO_DEVICE_IN_BACK_MIC | AUDIO_DEVICE_IN_VOICE_CALL
}
```

Figure 10: Adding voice call device to the audio policy file

Next, we need to make changes in the AudioRecord and AudioPolicyManagerBase.cpp files. Figure 12 gives the code snippet for adding the voice call channels to

```
inputs {
  primary {
    sampling_rates 8000|11025|16000|22050|24000|32000|44100|48000
    channel_masks AUDIO_CHANNEL_IN_MONO |AUDIO_CHANNEL_IN_STEREO |
AUDIO_CHANNEL_IN_FRONT_BACK | AUDIO_CHANNEL_IN_VOICE_CALL
    formats AUDIO_FORMAT_PCM_16_BIT devices AUDIO_DEVICE_IN_BUILTIN_MIC
    |AUDIO_DEVICE_IN_BLUETOOTH_SCO_HEADSET |
AUDIO_DEVICE_IN_WIRED_HEADSET | AUDIO_DEVICE_IN_BACK_MIC |
AUDIO_DEVICE_IN_VOICE_CALL
  }
}
```

Figure 11: Adding voice call device profile to the audio policy file

the audioParamCheck method of the user interface part of AudioRecord API. This modification is needed to make AudioRecord recognize these channels. Figure 13 shows insertion of the voice call channels in the sInChannelsNameToEnumTable list. This addition is needed for the flinger to recognize these channels. Finally Figure 14 illustrates how the AudioRecord API can be called for recording a voice call.

```
case (AudioFormat.CHANNEL_IN_VOICE_UPLINK | AudioFormat.
   CHANNEL_IN_VOICE_DNLINK):
  mChannelCount = 2;
  mChannels = channelConfig;
  break;
case AudioFormat.CHANNEL_IN_VOICE_DNLINK:
  mChannelCount = 1;
  mChannels = channelConfig;
  break;
case AudioFormat.CHANNEL_IN_VOICE_UPLINK:
  mChannelCount = 1;
  mChannels = channelConfig;
  break;
```

Figure 12: Adding voice call channels as acceptable parameter value to AudioRecord API

## 3.5 Installation

After making the code changes as described in the previous sections, the modules are compiled, built and downloaded to the device. Refer to Appendix B for these

```
#define AUDIO_CHANNEL_IN_VOICE_CALL
  (AudioSystem::CHANNEL_IN_VOICE_UPLINK)
const struct StringToEnum sInChannelsNameToEnumTable[] = {
  STRING_TO_ENUM(AUDIO_CHANNEL_IN_MONO),
  STRING_TO_ENUM(AUDIO_CHANNEL_IN_STEREO),
  STRING_TO_ENUM(AUDIO_CHANNEL_IN_FRONT_BACK),
  STRING_TO_ENUM(AUDIO_CHANNEL_IN_VOICE_CALL),
};
```

Figure 13: Adding voice call streams to flinger

```
record = new AudioRecord(MediaRecorder.AudioSource.VOICE_CALL,
  SAMPLE_RATE,
   AudioFormat.CHANNEL_IN_VOICE_UPLINK |
   AudioFormat.CHANNEL_IN_VOICE_DNLINK,
   // AudioFormat.CHANNEL_IN_STEREO,
   AudioFormat.ENCODING_PCM_16BIT, bufSize);
```

Figure 14: Calling Audio Record for recording for recording voice call

steps. Table 3 shows the modules that are built and their destination locations in the target phone device.

Table 3: Modules and their install destinations

| **New or Modified item** | **Destination** |
|---|---|
| MyAudioTrack | framework/framework.jar |
| MyAudioTrack | lib/libmedia.so |
| android_media_MyAudioTrack.cpp | lib/libandroid_runtime.so |
| AudioPolicyManagerBase.cpp | lib/hw/audio_policy.default.so |
| final.apk | /system/app |

# CHAPTER 4
# Results

We conducted a series of experiments and results to verify the audio and visual covertness of the audio data exfiltration. The experiment to covertly recording a two-way call coversation during a voice call is described in Section 4.1. Section 4.2 describes the exfiltration of a pre-recorded call, and Section 4.3 describes the exfiltration of a music file.

A series of experiments need to be conducted to show the behavior of victim phone in presence of the attack. For example the impact on battery life and the timing delay experienced during a normal call are to be observed and recorded here. Section 4.4 is currently a place holder for these experiments.

## 4.1 Experiment 1: Recording a Call conversation

This experiment verifies the recording of both uplink and downlink voice streams. Figure 15(a) illustrates this experiment.

When a call comes from a pre-determined number, the recording starts and stored under a file in the root directory. The file is then pulled to the host and replayed using 'aplay' available on Ubuntu 14.04LTS using the following command line:

```
aplay -t raw -c 2 -f S16_LE -r 44100 record.pcm
```

We found that the replay is clear for both streams. However the volume of downlink voice is much lower compared to the uplink voice.

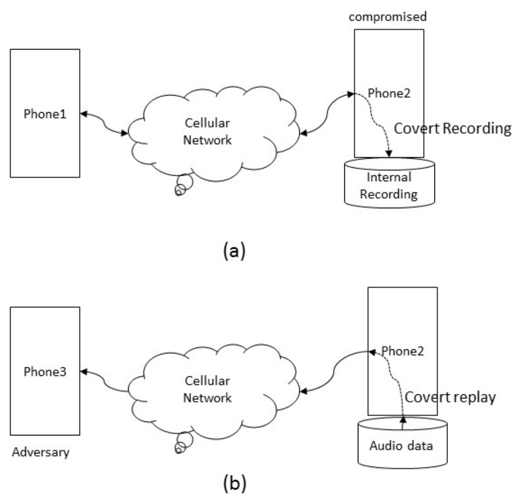


Figure 15: Experimental setup for showing covert call recording and replay

## 4.2 Experiment 2: Replaying covertly the recorded call to an incoming number

The second part of the experiment illustrated by Figure 15(b) was done to verify covert replay of the recorded audio file in the experiment in 4.1. The expectation was that when a call comes from a predetermined number, the call would be connected without showing any visual indication and the audio file will be replayed to the other end of the call without being heard at the local device. We verified this expected outcome.

The audibility of uplink voice was clear, but the downlink voice was a little difficult to hear. This can be due to the fact that we used only default static mixer when replaying the audio file to the voice call device.

## 4.3 Experiment 3: Replaying a large music file to a pre-determined incoming number

This experiment is similar to the experiment in 4.2, but this time a large WAV file was used for replay. The file played for around 4 minutes with clear audibility to the other end.

## 4.4 Experiment 4: Impact on normal phone behavior

We experimented the behavior of the phone for the following use cases to verify the covertness of the replay:

1. An incoming call to the victim phone during an exfiltration: The exfiltration call is disconnected, the incoming call is also disconnected.
2. An exfiltration call during an established incoming call at the victim phone: The exfiltration call is not allowed, and the foreground established call continues.
3. An exfiltration call during an established outgoing call from the victim phone: The exfiltration call is not allowed, and the foreground established call continues.
4. An attempt to make an outgoing call from victim phone during an exfiltration: The exfiltration is disconnected and the dialer pops up on the victim phone screen allowing the user to make the outgoing call.
5. The phone goes to sleeping mode during exfiltration: The exfiltration continues.
6. An exfiltration call when the phone is in the sleeping mode: The phone remains in the sleeping mode, and the exfiltration is allowed to proceed.

7. An exfiltration call during a non-phone applications such as web browsing and camera: The exfiltration proceeds without any call indication, the application also continues.

8. Opening of a non-phone application during an ongoing exfiltration: The application starts and the exfiltration continues.

## 4.5 Discussion on the results

The results for Section 4.1 is explained by the settings of sound volumes to match the different input and output audio devices. the volume of downlink voice is much lower compared to the volume of uplink voice. This results from the fact that the downlink voice volume is kept much lower to match the safety level of the earpiece. The uplink voice comes from the microphone and the volume is set to match the microphone which is higher than the volume set for the earpiece. Results of Section 4.2 shows that when the recorded call is replayed at the adversary end, the downlink voice in the recorded call is barely audible. This results from two factors. First, the replay is given to the earpiece of the adversary phone so the voice volume for both uplink and downlink channels in the recorded call is attenuated making the downlink voice volume significantly lower than the uplink voice volume. Another factor affecting the poor audibility is the fact that we have bypassed the Android optimized mixer at the victim end, and used a mixer that is not fully optimized.

Section 4.4 shows that we have managed the states of exfiltrating call to keep it covert. Most of the usual phone behavior have been tested to verify the covertness of the exfiltration. As there are numerous possible interactions, more testings need to be done for the verification of covertness in presence of complex interactions.

# CHAPTER 5

## Countermeasures

The main purpose of this project is to expose a few potential attacks on mobile phones that exploit the existing vulnerabilities of Android. Our potential attacks take advantage of the fact that shared libraries and dex jar files can be built and installed on any rooted mobile device once the device Android version is known. Unlike infecting the device by making changes in SMALI code of a dex file that can be easily thwarted by obfuscating the code, we argue that our approach poses significant challenges to countermeasures. Any such countermeasure has to be evaluated carefully as it may impact Android policy of open source and may not be justifiable from business point of view. In this chapter, we try to evaluate possible countermeasures to mitigate the risk of the potential attacks.

We have presented an attack that covertly records a call conversation otherwise not permitted by Android. The key vector to this attack is the plaintext "audio_policy.conf" file. We are easily able to include the voice call input device and its profile in this file. A possible mitigation will be to encrypt this file. It will then also be required that the decryption code is kept separate from the open source code. Hence a vendor supplied proprietary code must be used for decryption. However Android audio policy manager implements a default configuration built into the code when the external file is not available on the device. Hence the adversary will easily be tempted to remove the encrypted external file and then tamper the default configuration that is available in the open source tree. Removal of default code altogether is not effective as it can be easily reinstated.

Mitigation of the covert replay attack is also difficult and must be investigated for viability. The key attack vector here is the open source of tinyalsa that is used for baseband device access. Use of a proprietary API for baseband access can be a possible countermeasure. However, this decision falls in the space of Android open source policy and the countermeasure can pose certain business impact. Another approach may be to deploy in the target device a tamper detection mechanism for the shared libraries. Use of cryptographic hash seems to be a viable option. The cryptographic hash has to be kept in the pre-boot modules (such as initrd) to avoid tampering of those hashes themselves. Implementation of such tamper detection mechanism, however, should not come in the way of genuine experimentation or install of upgrades on the device. Coming out with an effective mitigation approach to protect a tampered shared library in Android calls for a research on its own.

# CHAPTER 6

## Conclusion and future work

Our project presents a proof of concept implementation for using a voice call as a potential covert channel for data exfiltration. In order to achieve this, a call monitor is inserted in the telephony stack for monitoring the incoming calls. When an incoming call from a predetermined number is detected by the call monitor, it triggers either the call recording or replay of an audio data depending on the incoming number. The call recording is kept covert and the user does not get any clue of the ongoing recording. The call monitor automatically connects the call for replay or exfiltration of audio data. On the victim phone the replay is inaudible and there is no visual indication of the call.

Our implementation breaks multiple security boundaries created by Android. The recording of two-way call conversation during a voice call overcomes the recording restriction of downlink voice stream posed by the Android audio policy manager. For replaying an audio stream inaudibly over the voice call bypasses the restricted audio path that prevents the flinger from sending the user audio stream directly to the baseband device.

In our approach we have exploited Android dependency on shared libraries. Shared libraries can be built easily in AOSP for any published Android version, and can easily be inserted into a rooted mobile device. We have argued that our approach presents difficult technical and business challenges to mitigate the proposed attacks. Tampering detection in the shared library by using cryptographic hashes is not an easily viable option as it will come in the way of doing genuine experimentation and upgrade on the device. Disassociating the vulnerable modules from AOSP and making those modules as vendor proprietary may not be easily achievable due to business reasons. Coming out with an effective mitigation of the potential attacks thus seeks an independent research on its own.

Our current implementation is limited to only audio data exfiltration. The published research [1, 8, 5] claim that a data can be audio modulated. Based on those works, our work can be enhanced for application to a generic data exfiltration. The data exfiltration built over voice call will break the distance limitations faced by the published research.

Our implementation has paved the path for direct usage of tinyalsa for covert replay of uplink voice streams over the voice call. We recommend to enhance this mechanism to covertly capture the downlink incoming voice. It can be done by using pcm_write call of tinyalsa. However, it requires additional research to keep the downlink recording of voice call inaudible. Ability to covertly capture the downlink voice stream will open up possibility of sending voice commands from a remote phone. The captured voice commands then can be analysed to decipher the sent command and action can be taken. This enhancement will offer a two-way covert links over voice calls. A botnet like framework can evolve that can use covert links over voice calls for sending and receiving voice and data commands. Attacks can be designed over such framework that can spread faster and pervasively as the voice call has not distance limitation and is available on all mobile devices without much censorship.

We conclude that the attack shown by our work has a far reaching influence on design of future attacks that may have potentially much larger and disastrous impact on mobile network. We seek out an immediate effective countermeasure to mitigate the risk of such attacks.

# LIST OF REFERENCES

[1] B. S. Aloraini, A New Covert Channel Over Cellular Network Voice Channel (2014), Thesis. Rochester Institute of Technology.

[2] Android Source Code, Android Source `http://source.android.com/source/index.html`

[3] Audio Implementation, Android Source `http://source.android.com/devices/audio/implement.html`

[4] N. B. Brandt, M. Stamp, Automating NFC Message Sending for Good and Evil, *Journal of Computer Virology and Hacking Techniques*, 10(4):273–297, 2014

[5] Q. Do, B. Martini et. al., Exfiltrating data from Android devices, *Computers & Security* 48:74–91, 2015 `http://www.elsevier.com/locate/cose`

[6] J. J. Drake, P. O. Fora et. al., *Android Hacker's Handbook*, Wiley, 2014

[7] N. Elenkov, *Android Security Internals An In-Depth Guide to Android's Security Architecture*, No Starch Press, 2015

[8] M. Guri, G. Kedma et. al., AirHopper:Bridging the Air-Gap between Isolated Networks and Mobile Phones using Radio Frequencies, *The 9th IEEE International Conference on Malicious and Unwanted Software*, 58–67, 2014

[9] Installing the Android SDK, Android Developer `http://developer.android.com/sdk/installing/index.html`

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

[11] M. G. Kuhn, R. J. Anderson, Soft Tempest:Hidden data transmission using electromagnetic emanations, *Information hiding*, 124–142, 1998

[12] L. T. PawPrints, *Beginning Android Development: Create Your Own Android Apps Today*, CreateSpace Independent Publishing Platform, 2014

[13] Telephony Manager, Android Developer `http://developer.android.com/reference/android/telephony/TelephonyManager.html`

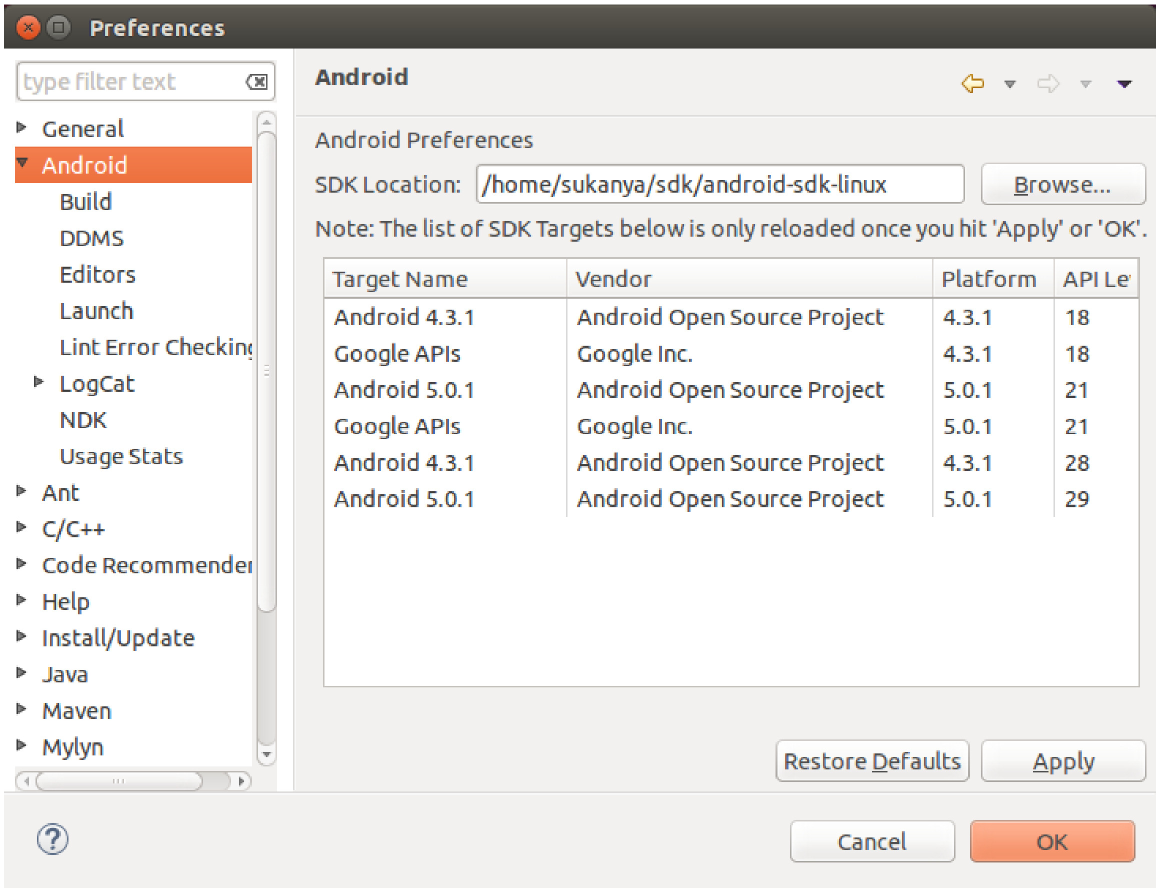
[14] R.P. Weinmann, Baseband Attacks: Remote Exploitation of Memory Corruptions in Cellular Protocol Stacks, *Proceedings of the 6th USENIX Conference on Offensive Technologies*, Berkeley, California, USA, 2012, p. 2 `https://www.usenix.org/conference/woot12/workshop-program/presentation/Weinmann`

# APPENDIX A

## Eclipse Setup

### A.1 Configuring SDK Path

To set SDK path, launch Eclipse and choose the following: Window -> Preferences -> Android -> SDK Location



Android Preferences

SDK Location: /home/sukanya/sdk/android-sdk-linux

Note: The list of SDK Targets below is only reloaded once you hit 'Apply' or 'OK'.

| Target Name | Vendor | Platform | API Le |
|---|---|---|---|
| Android 4.3.1 | Android Open Source Project | 4.3.1 | 18 |
| Google APIs | Google Inc. | 4.3.1 | 18 |
| Android 5.0.1 | Android Open Source Project | 5.0.1 | 21 |
| Google APIs | Google Inc. | 5.0.1 | 21 |
| Android 4.3.1 | Android Open Source Project | 4.3.1 | 28 |
| Android 5.0.1 | Android Open Source Project | 5.0.1 | 29 |

Figure A.16: Setting SDK path in Eclipse

### A.2 Enabling Build of System App

The Android plugin (ADT) in Eclipse enables Protected Permission in the Eclipse Lint setting. This option has to be disabled by performing the following sequence:

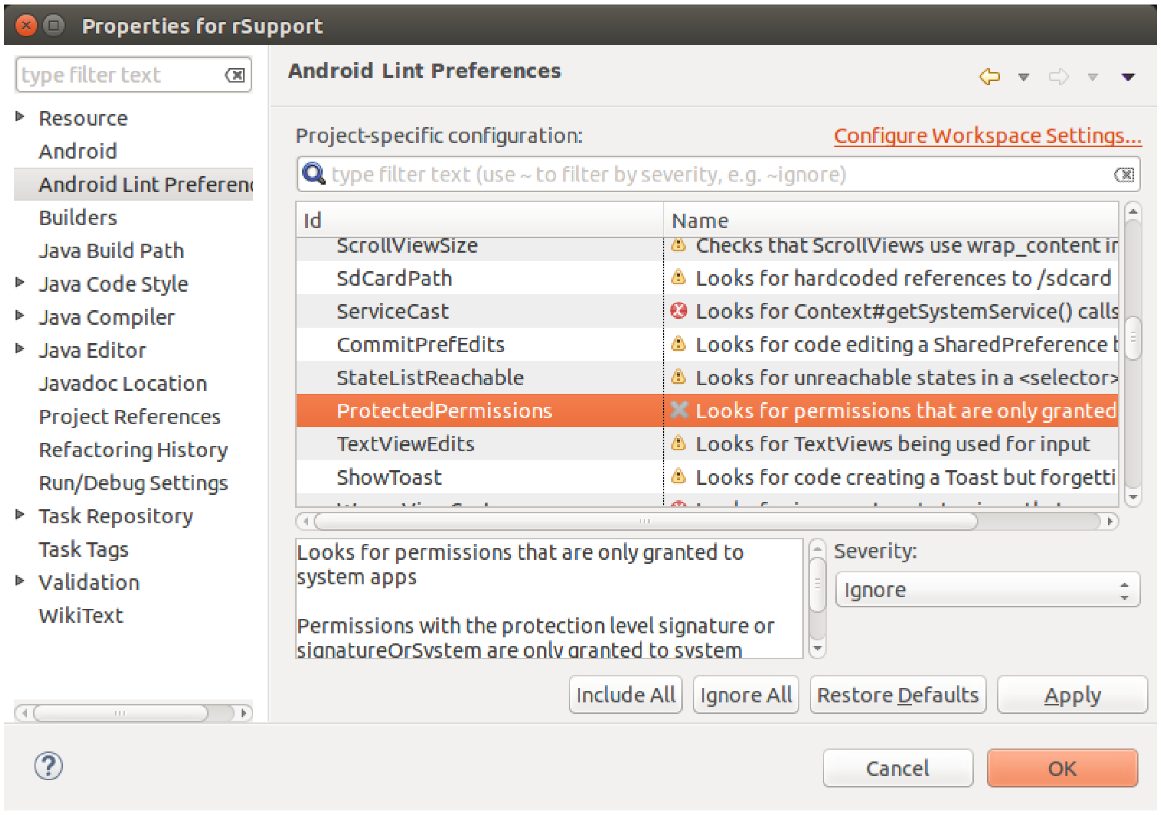
1. Select the project and right click on the project

2. Choose Build Path and then choose configure build path

3. A window opens as Figure A.17

4. Select Android Lint Preferences on the left menu

5. Scroll down to Protected Permissions on the right menu, select and disable this option



Properties for rSupport

Android Lint Preferences

Project-specific configuration: Configure Workspace Settings...

| Id | Name |
|---|---|
| ScrollViewSize | Checks that ScrollViews use wrap_content ir |
| SdCardPath | Looks for hardcoded references to /sdcard |
| ServiceCast | Looks for Context#getSystemService() calls |
| CommitPrefEdits | Looks for code editing a SharedPreference t |
| StateListReachable | Looks for unreachable states in a <selector> |
| ProtectedPermissions | Looks for permissions that are only granted |
| TextViewEdits | Looks for TextViews being used for input |
| ShowToast | Looks for code creating a Toast but forgetti |

Looks for permissions that are only granted to system apps

Permissions with the protection level signature or signatureOrSystem are only granted to system

Severity: Ignore

Figure A.17: Enable System App build in Eclipse

## A.3 ADT Hack: Allowing permission to use hidden API in Eclipse

1. Search for the ADT jar in the eclipse folder. Most likely this will be in .eclipse/org.eclipse.platform_*_linux_gtk_x86_64/plugins folder The ADT

jar file will be com.android.ide.eclipse.adt_*.jar

2. Save the original jar file and then extract this in some temporary folder

3. Locate AndroidClasspathContainerInitializer file under com/android/ide/eclipse/adt/internal/project folder

4. Edit and replace /com/android/internal/** to something like /com/android/internax/** that does not change the length of this string.

5. Create a jar file with the same name and copy it to the original ADT jar

# APPENDIX B

# Building and Integrating Android Components

## B.1 Build a shared library or jar file

From the root directory of AOSP:

1. . build/envsetup.sh
2. lunch

   Pick up the required config from the above, hit Enter
3. make <modulename>

   For example make "libmedia" will create libmedia.so

   or

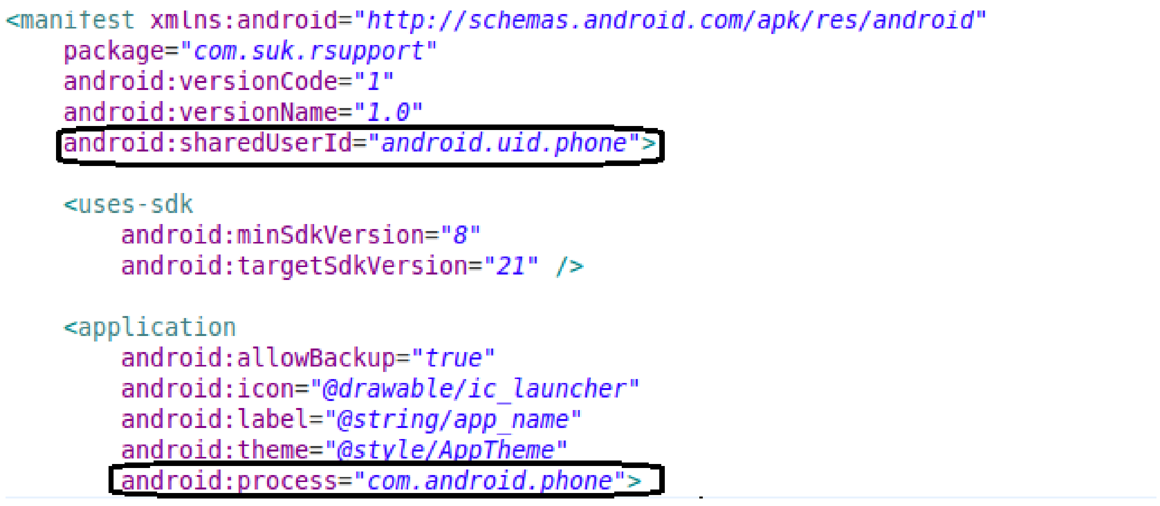
   make "framework" will create framework.jar

Next, use adb to push the newly built files to their respective directory in the target phone device. The following sequence shows how a libmedia.so file is pushed.

1. adb root
2. adb remount
3. adb push out/target/<pathname>/libmedia.so /system/lib/
4. adb reboot

## B.2 Manifest file to run the application inside Phone application

Figure B.18 illustrates the required entries in the Manifest.xml file for running the application inside the internal Phone process.



```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.suk.rsupport"
    android:versionCode="1"
    android:versionName="1.0"
    android:sharedUserId="android.uid.phone">

    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="21" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme"
        android:process="com.android.phone">
```

Figure B.18: Entries in the in Manifest xml file to share the Phone process

## B.3 How to sign using signApk

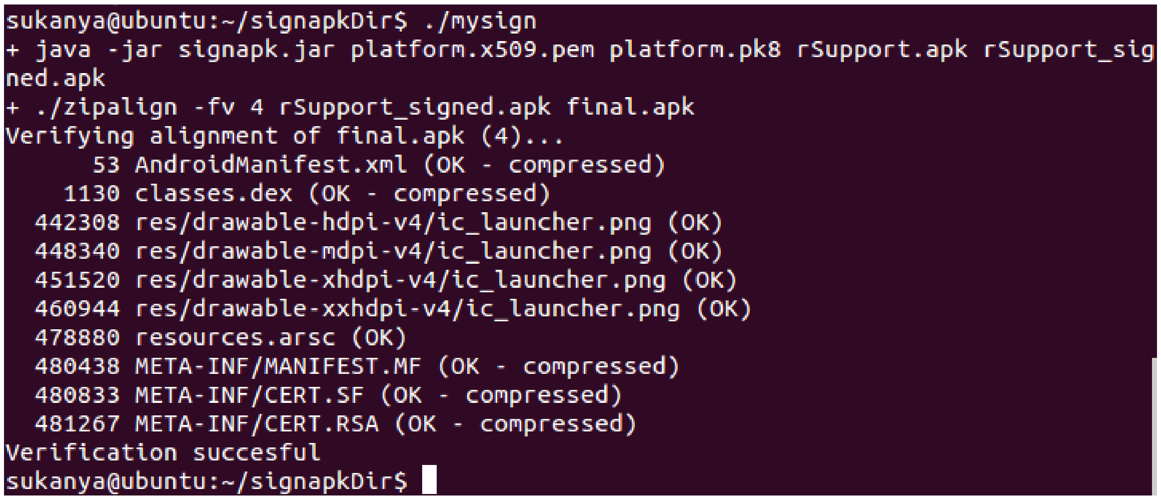
The application has to be exported from Eclipse using Unsigned Export. Prior to executing the following steps, install signapk and zipalign.

1. right-click on project -> Android tools -> export Unsigned Application Package
2. choose a directory to store the unsigned apk
3. Use the following in sequence by suitably replacing the user supplied files shown within <angular parenthesis>.

   ```
   java -jar signapk.jar platform.x509.pem platform.pk8
   <rSupport.apk> <rSupport_signed.apl>
   ```

   ```
   zialign -fv 4 <rSupport_signed.apk> <final.apk>
   ```

The Figure B.19 shows the screen capture of output.



```
sukanya@ubuntu:~/signapkDir$ ./mysign
+ java -jar signapk.jar platform.x509.pem platform.pk8 rSupport.apk rSupport_sig
ned.apk
+ ./zipalign -fv 4 rSupport_signed.apk final.apk
Verifying alignment of final.apk (4)...
      53 AndroidManifest.xml (OK - compressed)
    1130 classes.dex (OK - compressed)
  442308 res/drawable-hdpi-v4/ic_launcher.png (OK)
  448340 res/drawable-mdpi-v4/ic_launcher.png (OK)
  451520 res/drawable-xhdpi-v4/ic_launcher.png (OK)
  460944 res/drawable-xxhdpi-v4/ic_launcher.png (OK)
  478880 resources.arsc (OK)
  480438 META-INF/MANIFEST.MF (OK - compressed)
  480833 META-INF/CERT.SF (OK - compressed)
  481267 META-INF/CERT.RSA (OK - compressed)
Verification succesful
sukanya@ubuntu:~/signapkDir$
```

Figure B.19: Signing the application