A Survey of Security Vulnerabilities in Android Automotive Apps

Abdul Moiz, Manar H. Alalfi
a1moiz,manar.alalfi@ryerson.ca
Department of Computer Science, Ryerson University
Toronto, ON, Canada

ACM Reference Format:
Abdul Moiz, Manar H. Alalfi. 2022. A Survey of Security Vulnerabilities in Android Automotive Apps. In *The 3rd International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS'22), May 16, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3524489.3527300

1 ABSTRACT

In this study, we conduct an experiment to reproduce some of the existing vulnerabilities of Android platform in Android Auto and Android Automotive platform. Some vulnerabilities specific to automotive were also examined and verified in Android Automotive platform. In total, we examined 14 vulnerabilities out of which 11 were reproducible in Android Auto as these apps are basically Android mobile apps and run directly in user's smartphone device. Whereas for Android Automotive 9 vulnerabilities were reproducible, remaining 5 were not reproducible either due to permission restrictions or due to API deprecation in Android 9.0 (Pie). We also categorize these vulnerabilities as per their type and provided their severity levels. For some of the vulnerabilities we provided the compliant solution which can be useful to mitigate some vulnerabilities while others, like accessing precise location information and deriving speed from it, varies from app to app usage of that information.

2 INTRODUCTION

Android Operating System (OS) is designed for mobile devices but with time adopted for tablets, wearable, Internet of Things (IoT) devices, Android TV, and now it has been extended to support automotive infotainment systems. These infotainment systems are tightly coupled with automotive hardware and can listen to vehicle properties like speed, gear, driving state and more. They will also allow users to install third party apps of their choice. Therefore, it's important that these apps are properly analyzed for security vulnerabilities.

Several researchers shows the severity of vulnerabilities in automotive software. In 2011, Stephen et al. [5] demonstrated an example of remote code execution through CD player, bluetooth and telematic units. In 2015, Valasek and Miller [21] showed how they were able to remotely take over of 2014 Jeep Cherokee by exploiting their infotainment system. In 2016, Mazloom et al. [15] demonstrated that controlling vehicle CAN bus from In-Vehicles Infotainment (IVI) and 3rd party app is possible. Remote hijacking, Stealing sensitive information, injecting messages to CAN bus remotely are some possible scenarios but in this study we are focusing on existing Android vulnerabilities applicability to Android automotive OS.

Android automotive app development is similar to Android mobile app development, therefore, in this paper we study the most common mobile vulnerabilities provided by Open Web Application Security Project (OWASP) top 10 mobile vulnerabilities (Year2016) [17]. Majority of the vulnerabilities exist due to improper platform usage, stealing of sensitive information occurs due to insecure data storage and insecure communication in REST Application Programming Interface (API)s. Apps are hacked as authentication form contains default username and password obtained through reverse engineering of the Android Package Kit (APK) file. All these vulnerabilities are due to insecure coding practises and we will be looking into some of the existing vulnerabilities of Android that still applies to Android automotive apps. This paper provides the following contributions:

- We examine the reproducibility of 14 Android mobile vulnerabilities on Android Auto/Automotive platforms.
- We categorize these vulnerabilities as per their type and provided their severity levels
- We provide the compliant solution which can be useful to mitigate some of those vulnerabilities.

3 BACKGROUND

Android [1] is Google's mobile operating system based on Linux Kernel and other open source software. It's designed specifically for smartphone and tablet devices which is now powering million of devices around the globe. Due to its popularity and wide adoption it has been enhanced and extended to work with wearables, Android TV, IoT devices, vehicle's infotainment systems by plugging in Android smartphone devices (Android Auto), and finally it's been extended to work directly with vehicle's infotainment system (Android Automotive OS).

3.1 Android Automotive

Android Automotive OS is an extension of Android operating system to work directly with IVI systems. It links the vehicle hardware with Android software with the implementation of Hardware Abstraction Layer (HAL) by Original Equipment Manufacturer (OEM) manufacturer. It will allow users to use pre-installed first and second party Android apps along with third party Android apps. Since, it's an extension of Android OS, Android Automotive source code lives in same repository as Android, it offers same features for scalability, robustness, security models, developer tools, same infrastructure as

EnCyCriS'22 , May 16, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9290-7/22/05…$15.00
https://doi.org/10.1145/3524489.3527300

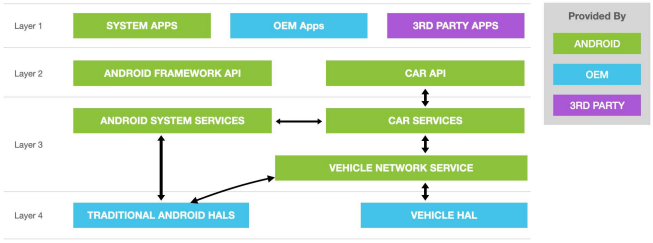

Figure 1: Android Automotive Architecture [3]

of Android along with additional APIs to link, support and control vehicle's features.

When compared to Android Auto, both allows users to install and use third party apps in their vehicle's infotainment system. However the way they allow it and their connection to vehicle's dashboard is different. Android Auto requires user to plug in their smart phone device to vehicle's dashboard either via USB cable or via wireless connection which then mirrors the mobile screen to vehicle's dashboard. It will allow users to use apps which are designed for Android Auto. In addtion, Android Automotive is an operating system. It runs natively in IVI systems which means car manufacturer are required to bundle this with their vehicle. Since, it's open source, it will allow them to alter Android Automotive OS as per their brand design and allow them to customized whole automotive experience. It will be capable to run apps designed for both Android Auto as well as Android Automotive. Android Automotive OS is announced by Google on March 2017, Since then car manufacturers; Polestar (Volvo's electric performance car), Renault–Nissan–Mitsubishi Alliance, General Motors, and Groupe PSA have announced to use Android Automotive to power their vehicle's IVI systems. Therefore, it's important to analyze these apps for possible vulnerabilities that may affect user's safety, security and privacy. This also justifies our focus to investigate Android Automotive apps.

3.2 Android Automotive Architecture

Android Automotive architecture can be broken into four layers. As seen in figure 1, First layer consist of system apps, OEM apps and third party apps. Second layer includes the Android framework API and Car API which interact with the Android Car and Vehicle network services, the third layer. Fourth layer is HAL which allows car manufacturer to implement these HAL to connect Android Automotive OS with vehicle and at the same time provide consistent interface to Android Automotive framework and app developers regardless of physical implementations. Android Automotive system and OEM apps will have more access to Car API and services as compared to third party apps and their connection to internet will allow users to access these apps and features remotely. They will be able to control Heating, Ventilation and Air Conditioning (HVAC), manage audio, remote engine start, lock/unlock doors and other sensors information. OEM apps will be able to communicate to vehicle HAL where as third party apps have limited capabilities.

When we tested Android Automotive Apps via the emulator, we have verified that third party apps can only listen to some of the information but can not access services to control vehicle features. Third party apps will be able to access limited vehicle information such as user's location, contacts, vehicle information, driving state which makes security analysis of these apps an important task as third party apps can observe these properties and can send this information to malicious servers.

3.3 Accessing Android Automotive Software Development Kit (SDK)

As of now, Android Automotive SDK is not publicly available to third party developers and current Android 9.0 (Pie) and 10.0 (Q) SDK allows for media or messaging apps development only. To access Car APIs, developers need to custom build their own Android Open Source Project (AOSP) by cloning AOSP repository and making sure it contains Car APIs. After the build process, developers can include compiled jar file into their Android project and explore Car APIs. The APIs that works with third party apps are mostly for observing vehicle information such as driving state, vehicle make, model and year. While using APIs that can controls vehicle features results in app termination which indicates that third party apps will not be able to control vehicle features.

3.4 Android App Structure

Android apps (Mobile, Auto and Automotive) unlike any typical desktop or program launchers which contains a single entry point have a complex structure. Each app is composed of Activities, Services, Content Providers and Broadcast Receivers. Each of these component serves a different entry point in an Android app. It's Android OS responsibility to decide and integrate the app into device's overall user experience. From security aspect, having multiple entry point in an app also increases the attack surfaces.

3.4.1 ***Permission Model.*** In Android each app need to declare permissions in order to access certain features of the platform. For example: accessing internet requires an app to declare Internet permission in it's manifest file. These permissions [2] can be of four different types *normal*, *dangerous*, *signature* and *signature_Or_System*. *normal* permissions can be granted by the system and are most commonly used. *dangerous* permissions require user's explicit approval before they can be accessed by an app. *signature* permissions will only be granted access when both the apps, one which declares the permission and other one which is requesting for permission are signed with same certificates. *signature_Or_System* permissions will be granted access if they satisfy signature condition or if an app requesting permission came pre-installed on the device.

3.4.2 ***Broadcast Receivers.*** The broadcast receivers in Android serves as a publish-subscribe design pattern. Apps can send or receive messages to or from Android system and other apps. Apps can register these broadcast receivers to receive specific broadcasts and it's system responsibility to route these messages to the subscribed receivers. Since, broadcast receiver's communication does not trigger any user interface changes they are perfect mechanism to leak sensitive information [4].

3.4.3 ***Inter Component Communication.*** In Android (Mobile, Auto and Automotive), each app exists in it's own sandbox environment which can not be accessed by other apps this ensures app security with respect to any modification done outside of it's sandbox. However, to communicate with app's components, within the same or between different app, programmers rely on Inter-Component Communication (ICC) communication. ICC provides a communication mechanism that can send or broadcast a message

Table 1: Vehicle specific vulnerabilities

Papers	Vulnerability Name	Category	Severity	Auto	Automotive
Eriksson et al.[9]	Sound Blast	Disturbance	Medium	TRUE	FALSE
	Fork Bomb	DoS	Medium	FALSE	FALSE
	Intent Storm	DoS	Medium	TRUE	TRUE
	Permissionless Speed	Privacy	Low	N/A	TRUE
New	Deriving Speed from Location Information	Privacy	Low	TRUE	TRUE

component called *intent* which can be received by activities, services or broadcast receivers. In case of a broadcast, *intent* is sent to all broadcast receivers in all the app's components installed on the device. Intent message must contain an action and data. Action defines a unique string which defines event that is happening and data can be any thing that other component requires to work on. Intent communication in an app requires great security consideration as it can lead to possible threats, discusses in our previous paper [16].

3.5 In-Vehicle Communication

Modern car's are equipped with Electronic Control Units (ECU) which control one or multiple subsystems in a vehicle such as braking, engine, and steering. These subsystems interconnects by means of different busses. These busses and protocols vary based on the manufacturer and different vehicle models of the same brand. Some example of these busses are: Controller Area Network (CAN) bus, designed to allow different micro controllers to communicate with each other's application without a host computer; Local Interconnect Network (LIN) bus, a serial network protocol used for communication between vehicle components; Media Oriented Systems Transport (MOST) bus, a serial communication system for transmitting audio, video and control data via fibre optic cables; BroadR-Reach technology, provides standards for ethernet connectivity within Automotive environment. These technologies provide no security or authentication therefore it's integration with infotainment systems requires proper security analysis.

4 VULNERABILITIES

This section focuses on vulnerabilities that are common in Android Mobile, Auto and Automotive platforms and some of which are specific to Android Automotive platform. Tables 1 list vulnerabilities that are automotive specifics and Table 2 list vulnerabilities that are common to all Android platforms including Mobile, Auto and Automotive.

Most of the vulnerabilities were discussed by Benjamin Eriksson et al. [9] and Amit et al. [14] in their papers while we added two extra vulnerabilities, namely: deriving speed from location information and precise location information. We also verified these vulnerabilities existence in Android Automotive platform, we created example projects and we carried out the test in both Android Auto and Android Automotive emulators which were part of Android SDK 9.0. In those tables, (true, false and not-applicable (N/A)) in auto and automotive columns defines which vulnerabilities were reproducible on those platforms.

4.1 Sound Blast

Sound Blast is a vulnerability through which a third party app can control audio volume in a vehicle. A specially designed use case

Table 2: Android vulnerabilities

Paper	Vulnerability Name	Category	Severity	Auto	Automotive
Eriksson et al. [9]	Permissionless Exfiltration	Privacy	Low	TRUE	TRUE
New	Precise Location information	Privacy	Low	TRUE	TRUE
Amit et al. [14]	Insecure File Storage	Data Exfiltration	Low	FALSE	FALSE
	Adding Custom Permission at Runtime		Medium	TRUE	FALSE
	Cross Site Scripting Vulnerabilities	XSSV	Low	TRUE	TRUE
	GPS Location Detection from WebView	Privacy	Low	TRUE	FALSE
	WebView URL Location Monitoring	Privacy	Low	TRUE	TRUE
	JavaScript Interfacing Vulnerabilities	Privacy, XSSV	Medium	TRUE	TRUE
Eriksson et al. [9]	Information Leakage Vulnerability	Privacy	Low	TRUE	TRUE

where sudden change of volume at vehicle high speed can be disastrous for the driver, passengers and vehicles around it. In Android apps, *AudioManager* class is responsible for adjusting audio volume programmatically. It does not require any special permission from the user to control audio. However, there is a fixed volume policy that can be configured by Android device manufacturers which prevents this API to adjust volume at run time. This policy is required in Android Automotive to prevent sound blast vulnerability. As seen in the Table 1, this vulnerability was reproducible on Android Auto but not on Android Automotive as their emulators are configured with fixed volume policy, but this will vary from different manufacturers and their adoption of this policy.

Listing 1: Sound blast vulnerability example code

```
1  private void setMaxVolume() {
2    // Initializing Audio Manager Instance
3    AudioManager am = (AudioManager) getSystemService(Context.AUDIO_SERVICE);
4    // Getting device max stream volume
5    int maxVolume = am.getStreamMaxVolume(AudioManager.STREAM_MUSIC);
6    // Setting device stream volume to max
7    am.setStreamVolume(AudioManager.STREAM_MUSIC, maxVolume, 0);}
8
9  private void setMinVolume() {
10   // Initializing Audio Manager Instance
11   AudioManager am = (AudioManager) getSystemService(Context.AUDIO_SERVICE);
12   // Getting device min stream volume
13   int minVolume = am.getStreamMinVolume(AudioManager.STREAM_MUSIC);
14   // Setting device stream volume to min
15   am.setStreamVolume(AudioManager.STREAM_MUSIC, minVolume, 0);}
```

Listing 1 shows the example usage of this API, *AudioManager* initialization is at line 3 & 11, setting volume to max is at line 5, 7 or min at line 13, 15. This a non compliant solution as *setStreamVolume* can be used to alter volume of the device instantly, which can be used to distract the driver if used in a malicious way.

Listing 2, shows a compliant example. We are using locks to prevent changing of volume quickly (line 6, 20), with each lock their is a delay of 1 second (line 18-20). This will ensure that volume change in a step by step manner and driver distraction can be avoided. Also, *adjustStreamVolume* methods with parameter *AudioManager.ADJUST_RAISE* or *AudioManager.ADJUST_LOWER* adjust volume in a step by step manner.

Listing 2: Sound blast vulnerability solution

```
1  private ReentrantLock lock = new ReentrantLock();
2  private Handler handler = new Handler();
3  private void adjustVolume() {
4    try {
5      // Applying lock so it can only be used if unlocked.
6      lock.lock();
7      // Initializing AudioManager instance.
8      AudioManager am = (AudioManager) getSystemService(Context.AUDIO_SERVICE);
9      // Increasing stream volume one step at a time.
10     am.adjustStreamVolume(AudioManager.STREAM_MUSIC, AudioManager.ADJUST_RAISE
           , 0);
```

```
    // ----- OR -----
    // Decreasing steam volume one step at a time.
    am.adjustStreamVolume(AudioManager.STREAM_MUSIC, AudioManager.ADJUST_LOWER
          , 0);}
    finally {
    // Using a handler to unlock the resource in one second.
    handler.postDelayed(new Runnable() {
      @Override
      public void run() {
        // unlocking the resource.
        lock.unlock();}
    }, 1000); // Time in milliseconds}}
```

4.2 Fork Bomb

This vulnerability can cause the Infotainment Head Unit (IHU) to occupy all the system resources, causing it to freeze or force a reboot. To perform this, root permissions were required on the device then any application can perform Remote Code Execution (RCE) on Android shell. This case was not reproducible from the Android Automotive emulator as they can not be rooted but it might be possible to root Android Automotive emulator if it was built manually from AOSP.

Listing 3: Fork bomb vulnerability

```
public class ShellExecutor {
  public static String execute(String command) {
    // String builder to capture shell output
    StringBuilder output = new StringBuilder();
    // Shell Process
    Process p;
    try {
      // Runtime class that executes shell commands and returns the shell
          process.
      p = Runtime.getRuntime().exec(command);
      p.waitFor();
      // Creating stream readers to read shell output
      InputStreamReader inputStreamReader = new InputStreamReader(p.
          getInputStream());
      BufferedReader reader = new BufferedReader(inputStreamReader);
      // Reading shell output line by line and appending it to string builder.
      String line = "";
      while ((line = reader.readLine()) != null) {
        output.append(line).append("\n");}
    } catch (Exception e) {
      e.printStackTrace();}
    // returning the captured output
    return output.toString();}}
// Executing shell command using class above.
ShellExecutor.execute("ls -a");
```

Listing 3, shows an example code of how to execute shell commands with Android apps. *Runtime* class (line 9) allows an Android app to interface with the environment in which it's running.

4.3 Intent Storm

Intents in Android can be used to launch activities, however continuously launching intents from a thread within a loop can cause Android Automotive User Interface (UI) unresponsive. A specially designed scenario in a malicious app can cause the automotive infotainment system unresponsive which can be distracted for the driver. These cases were tested on Android Automotive emulators where UI became unresponsive and took a good amount of time to respond or to kill the malicious application.

Listing 4, shows a non complaint code example of intent storm vulnerability. *MainActivity* is continuously spawned within an infinite loop causing UI to become unresponsive as it's occupying system resources. Whereas, listing 5, shows a complaint code example.

Listing 4: Intent storm vulnerability

```
private void launchActivity() {
  // Infinite loop to launch Activity continuously.
  // Rendering app as unresponsive
  while (true) {
    Thread thread = new Thread(new Runnable() {
      @Override
      public void run() {
        Intent intent = new Intent(MainActivity.this,
                    MainActivity.class);
        MainActivity.this.startActivity(intent); }));
    thread.start();}}
```

Listing 5: Intent storm compliant example

```
private void launchActivity() {
  Intent intent = new Intent(MainActivity.this, MainActivity.class);
  MainActivity.this.startActivity(intent);
}
```

4.4 Permissionless Speed

In Android Automotive, Android apps require explicit permission to access vehicle speed which users need to approve it. However, It's also possible to derive vehicle speed if Engine RPM, Current Gear and knowledge about wheel size are known. To access engine RPM information *CAR_POWERTRAIN* permission is required and to access Current Gear information *CAR_INFO* permission is required and both these permissions are implicit. Declaring them in the *AndroidManifest.xml* file will give the app permission to observe these properties.

Listing 6, shows an example of how to access engine RPM and current gear information. Car instance is initialized in *initCar* method (line 13) with car service lifecycle listener which notifies the application that it's safe to use car instance. Within the listener there is a method call to *watchPropertySensor* (line 17, 18) which initializes *CarPropertyManager* (line 20) service, this class provides the interface between car services and the application. Next, It prepares an array of properties that are relevant for deriving speed (line 21-26) and registering an event listeners (line 29-43) for these services. Any change to these services will trigger this event listener and from this way a malicious application can listen to current gear and engine RPM and with knowledge of wheel size, it can derive the vehicle speed.

Listing 6: Permissionless speed vulnerability

```
// AndroidManifest.xml
<uses-permission android:name="android.car.permission.CAR_INFO" />
<uses-permission android:name="android.car.permission.CAR_POWERTRAIN" />
// JAVA
public class CarSpeedActivity extends AppCompatActivity {
  private Car car;
  ...
  private void initCar() {
    PackageManager pm = getPackageManager();
    if (!pm.hasSystemFeature(PackageManager.FEATURE_AUTOMOTIVE)) { return; }
    if (car != null) { return; }
    // Car instance initialization
    car = Car.createCar(this, null, 100L, new
          Car.CarServiceLifecycleListener() {
      @Override
      public void onLifecycleChanged(Car car, boolean b) {
        if (car.isConnected() && !car.isConnecting()) {
          watchPropertySensor(car);}}});}
  private void watchPropertySensor(Car car) {
    // Initializing CarPropertyManager
    CarPropertyManager carPropertyManager = (CarPropertyManager) car.
          getCarManager(Car.PROPERTY_SERVICE);
    int[] properties = new int[] {
      VehiclePropertyIds.GEAR_SELECTION,
      VehiclePropertyIds.CURRENT_GEAR,
      VehiclePropertyIds.PERF_VEHICLE_SPEED,
      VehiclePropertyIds.ENGINE_RPM,
      VehiclePropertyIds.WHEEL_TICK };
    for (int property : properties) {
      // Registering callback listener for all relevant properties.
      carPropertyManager.registerCallback(new CarPropertyManager.
            CarPropertyEventCallback() {
        @Override
        public void onChangeEvent(CarPropertyValue carPropertyValue) {
          // Accessing property name
          String propertyName = VehiclePropertyIds.toString(carPropertyValue.
                getPropertyId());
          // Accessing property Value
          String propertyValue = carPropertyValue.getValue().toString()
          // Logging information
          Log.d("CarSpeedActivity", String.format("%s: %s", propertyName,
                propertyValue));
          // Broadcasting information
          Intent intent = Intent("com.example.app.broadcast");
          intent.putExtra("propertyName", propertyName);
          intent.putExtra("propertyValue", propertyValue);
          sendBroadcast(intent)
        } ...});}}}
```

4.5 Deriving Speed from Location Information

Android Automotive has special permission in order to access car speed information. However, speed can be derived from location information using distance formula. To access location information an app requires location permission from the user. Majority of apps rely on the user's location to operate. Such apps in automotive environment can easily inspect user speed, acceleration habits, and hard braking. Many of the car insurance apps use this technique to reward users with discounts based on their driving habits. As we experimented in our example code, this vulnerability is reproducible in both Android Auto and Android Automotive emulators.

Listing 7, shows an example of how to inspect location information and convert that into vehicle speed. Permission array at line 36 contains permissions that this app need from user to access location information. Method *onCreate* at line 42 is initializing location manager and requesting user for permission. *requestPermission* method (line 58) is either starting location listening process or prompting user for permission. Once user grant the permission *startLocationListening* process get called which registers *locationListener* and starts the listening process. *locationListener* (line 8-33) receives new location every 500 milliseconds or every 1 meter distance changed. In this listener, it takes previous and current location converts them in to distance (line 17) using method *distance* (not shown) and converts that into speed (line 25). This way any malicious app can derive speed just using location information.

Listing 7: Deriving speed from location vulnerability

```
1  public class DeriveSpeedActivity extends AppCompatActivity {
2     // Properties
3     private LocationManager locationManager;
4     private String provider = null;
5     private Location prevLocation = null;
6     private Date time = null;
7     // Location Listener --> This will listen all the location changes
8     private final LocationListener locationListener = new LocationListener() {
9        @Override
10       public void onLocationChanged(Location location) {
11          if (prevLocation == null && location != null) {
12             // Setting previous location if this is initializing for the
                  first time.
13          prevLocation = location;
14          time = new Date();
15       } else if (prevLocation != null && time != null && location != null)
                {
16          // Calculating Speed from location and time
17          Double dist = distance(
18                prevLocation.getLatitude(),
19                prevLocation.getLongitude(),
20                location.getLatitude(),
21                location.getLongitude());
22          // Converting distance to speed based on time spent covering that
                distance.
23          Double speed = (dist/(new Date().getTime()
24             /1000 - time.getTime()/1000)) * (1/3.6);
25          // Converting speed to string and setting it on the view's label
26          String speedStr = String.format("Car Speed: %f kph", speed);
27          Log.d("DeriveSpeedActivity", speedStr);
28
29          // Updating previous location with current location and time.
30          prevLocation = location;
31          time = new Date();}}
32          ... };
33    // Permissions required to access location
34    private final ArrayList<String> permissions = new ArrayList<String>() {
35     android.Manifest.permission.ACCESS_FINE_LOCATION,
36     android.Manifest.permission.ACCESS_COARSE_LOCATION,
37     android.Manifest.permission.ACCESS_BACKGROUND_LOCATION};
38    @Override
39    protected void onCreate(Bundle savedInstanceState) {
40       ...
41       // Initializing location manager and provider
42       locationManager = (LocationManager)
43             getSystemService(Context.LOCATION_SERVICE);
44       provider = locationManager.getBestProvider(new Criteria(), false);
45       // Requesting for permission from user
46       requestPermission(); }
47    private void requestPermission() {
48       // If permissions are already granted, start listening
49       if (checkSelfPermission(android.Manifest.permission.
               ACCESS_FINE_LOCATION) == PackageManager.PERMISSION_GRANTED) {
50          startLocationListening();
51       } else {
52          // Show permission prompt to user.
53          requestPermissions((String[])permissions.toArray(), 1); }}
54    private void startLocationListening() {
55       locationManager.requestLocationUpdates(provider, 500, 1.0f,
               locationListener);}
56    private void stopLocationListening() {
57       locationManager.removeUpdates(locationListener);}
```

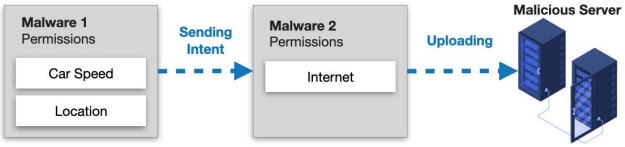

Figure 2: Permissionless exfiltration scenario

4.6 Permissionless Exfiltration

As the name implies, permissionless exfiltration vulnerability requires the use of two or more malicious apps to work together in order to leak sensitive information to malicious servers. As it can be seen in figure 2, A malicious app_1 with access to car speed and location information is passing that information to another malicious app_2 which has internet access, is uploading that information to a malicious server. Listing 8, shows a code example of how a malicious app can silently receive broadcast in background and can upload it to their server. In listing 6, method *watchPropertySensor* is broadcasting vehicle services property name and value (line 39-42) which can be received by the app in listing 8 as both their action and filter matches. In the listing 6 and 8 both the apps have different sets of permissions which makes it hard for the user to suspect these apps as malicious. This example was reproducible both in Android Auto and Android Automotive.

Listing 8: Permissionless exfiltration vulnerability

```
1  // AndroidManifest.xml
2  <uses-permission android:name="android.permission.INTERNET" />
3  // JAVA
4  class MainActivity extends AppCompatActivity {
5    // Initializing BroadcastReceiver instance
6    private final BroadcastReceiver receiver = new MyBroadcastReceiver();
7    @Override
8    public void onCreate(@Nullable Bundle savedInstanceState, @Nullable
           PersistableBundle persistentState) {
9      ...
10     // Configuring intent filter that will receive broadcast matching this
           filter.
11     IntentFilter filter = new IntentFilter("com.example.app.broadcast");
12     // Registering receiver so this activity can listens to broadcasts.
13     registerReceiver(receiver, filter);
14   }
15   ...}
16 class MyBroadcastReceiver extends BroadcastReceiver {
17   @Override
18   public void onReceive(Context context, Intent intent) {
19     // Broadcast Receiver on receiving intent matching it with the filter
20     if (intent.getAction().contentEquals("com.example.app.broadcast")) {
21       // Extracting vehicle properties and uploading it to the server.
22       String pName = intent.getStringExtra("propertyName");
23       String pValue = intent.getStringExtra("propertyValue");
24       uploadToServer(pName, pValue);}}}
```

4.7 Precise Location information

Location information can be extracted if the user allows the app for the permission. Location information can be used to track about user whereabouts, personal habits, to show location aware ads. On the wrong hand, it can also be used for physical attacks and harassment. Listing 7 shows the code to obtain location information. For automotive apps, accessing location information should be restricted to certain categories like: maps and navigation apps, where as music, podcast, weather and other type of apps should be allowed to access area location instead of precise location.

4.8 Insecure File Storage:

MODE_WORLD_WRITEABLE & MODE_WORLD_READABLE are flags which apps need to specify while storing and reading files and preferences to external storage. Storing files using these flags does not provide any security. Any other app can access these files stored on external storage. Since Android 7.0, these flags are deprecated and removed from Android SDK and hence, no longer valid for Android Auto and Automotive apps.

4.9 Adding Custom Permission at Run Time

In Android, each app needs to declare permissions in order to access certain features of the platform. For instance to access internet, an app needs to declare internet permission in its manifest file. Similarly, to share resources between a third party app and another app, and to restrict all other apps from accessing the same information, developers can define their own permissions which both apps will use to communicate in a secure manner. These permissions declared by developers are referred to as custom permissions. It can be declared in the manifest file or it can be declared at application run time. In order for this to work, apps declaring the custom permission need to be installed first then apps using this permission need to be installed later. As shown in listing 9, app_1 is declaring a custom permission in it's manifest file. To access resources under this permission app_2 can either declare it in the manifest file or can add it dynamically at run time. This pattern can be employed by malicious apps who want to communicate with each other securely without getting noticed. Declaring in the manifests file of both the apps have higher chances for security tools to detect this link. However, if these permissions are added dynamically then detecting this link between apps becomes more challenging.

Adding custom permissions at run time were tested both in Android Auto and Android Automotive emulators and it only worked for Android Auto. In Android Automotive, apps need MANAGE_USERS and CREATE_USERS permission in order to add custom permissions at run time. Both these MANAGE_USERS and CREATE_USERS permissions are system only. Hence, system apps can have this feature.

Listing 9: Custom permission vulnerability

```
// App 1: AndroidManifest.xml
<permission
  android:name="com.example.app.permission.READ"
  android:label="@string/read_permission"
  android:description="@string/read_permission_desc"
  android:permissionGroup="android.permission-group.EXAMPLE"
  android:protectionLevel="dangerous" />

// App 2: AndroidManifest.xml
<uses-permission
  android:name="com.example.app.permission.READ"
  android:maxSdkVersion="18" />

// JAVA Example of adding permission at runtime.
PermissionInfo permissionInfo = new PermissionInfo();
permissionInfo.name = "Custom Permission Name";
permissionInfo.labelRes = R.string.permission_label_string;
permissionInfo.protectionLevel = PermissionInfo.PROTECTION_DANGEROUS;
PackageManager packageManager = getApplicationContext().getPackageManager();
packageManager.addPermission(permissionInfo);
```

4.10 Cross Site Scripting Vulnerabilities

Android apps sometimes require the use of *WebView* to present websites within the apps. However, enabling JavaScript in the WebView can introduce Cross Site Scripting Vulnerabilities (XSSV) in the application. For security purposes, it's recommended that apps should only allow JavaScript to web pages which reside in the app package. Since, Android apps can also add JavaScript interfaces to WebView which allows the website to invoke native Android functionalities. If the app does not check which website it needs to add the JavaScript interface it could lead to severe XSSV.

Listing 10, shows an example code to enable JavaScript in a WebView and to add a JavaScript interface. XSSV exists in both Android Auto and Android Automotive.

Listing 10: Cross site scripting vulnerability

```
WebView webView = findViewById(R.id.webView);
webView.getSettings().setJavaScriptEnabled(true);
webView.loadUrl("file:///android_res/raw/index.html");
webView.addJavascriptInterface(new WebAppInterface(this), "Android");
```

4.11 WebView URL Location Monitoring

Browsing websites in WebView triggers *WebViewClient shouldOverrideUrlLoading* methods which can prevent or allow the WebView from visiting the URL. However, the same method can be used to track user visits to websites. This way an app can easily track user websites visits and can use it to show activity based ads, promotions, content, etc. This case is reproducible in both Android Auto and Android Automotive emulators.

Listing 11, shows an example of how an app can track user website visits. *shouldOverrideUrlLoading* method gets invoked every time user hits a different URL in the browser. From the request object a malicious app can get the URL and upload it to their server.

Listing 11: Webview URL monitoring vulnerability

```
webView.setWebViewClient(new WebViewClient() {
  @Override
  public boolean shouldOverrideUrlLoading(WebView view, WebResourceRequest
        request) {
    view.loadUrl(request.getUrl().toString());
    Log.d("WebViewActivity", String.format("URL: %s", request.getUrl().
          toString()));
    return false;}});
```

4.12 JavaScript interfacing Vulnerabilities

Adding JavaScript Interface to WebView enables JavaScript to perform native Android functionality, the severity depends on the type of functionality exposed. As for the best practices, an app should expose the JavaScript interface to local web pages instead of all web pages. As we experimented on it, adding JavaScript interface works in both Android Auto and Android Automotive emulators.

Listing 12, shows an example of how to define a JavaScript interface and listing 10 (line 4) shows how to inject it to the WebView.

Listing 12: Javascript interface definition example

```
private static class WebAppInterface {
   private Context context;
   WebAppInterface(Context context) {
     this.context = context;}
   /** Show a toast from the web page */
   @JavascriptInterface
   public void showToast(String toast) {
     Toast.makeText(context, toast, Toast.LENGTH_SHORT).show();}}
```

4.13 GPS Location Detection from WebView

By enabling JavaScript in Android's WebView, web pages can ask the app for user's location. Once a user explicitly allows the app permission to access location then any web page can access user's location without further permission from users. This case is reproducible in Android Auto. However, accessing location from Android Automotive's WebView is not reproducible.

Listing 13 and 14, shows example code for both Java and HTML page. On the web page, when the user clicks the button which invokes the *getLocation* function in the JavaScript side. It triggers the *WebChromeClient* listener on the Java side which checks for

the location permission. If it's already granted by the user it simply allows the web page to access user location. Otherwise, it prompts the user for permission. This way any web page requesting location inside the WebView of this app can track the user's location easily.

Listing 13: GPS location detection from WebView Java code

```
// JAVA
webView.setWebChromeClient(new WebChromeClient() {
  @Override
  public void onGeolocationPermissionsShowPrompt(String origin,
        GeolocationPermissions.Callback callback) {
    String permission = Manifest.permission.ACCESS_FINE_LOCATION;
    if (ContextCompat.checkSelfPermission(WebViewActivityJava.this, permission
        ) == PackageManager.PERMISSION_GRANTED) {
      callback.invoke(origin, true, false);
    } else {
      if (ActivityCompat.shouldShowRequestPermissionRationale(
          WebViewActivityJava.this, permission)) {
        // ask the user for permission
        String[] permissions = new String[] { permission };
        ActivityCompat.requestPermissions(WebViewActivityJava.this, permissions
            , 1);
        // we will use these when user responds
        geolocationOrigin = origin;
        geolocationCallback = callback;}} });
```

Listing 14: GPS location detection from WebView HTML code

```
<!DOCTYPE html>
<html>
  <body>
    <h1>Extracting Location</h1>
    <p id="location">Location: ...</p>
    <button onclick="getLocation()">Try It</button>
    <input type="button" value="Say hello" onClick="
        showAndroidToast('Hello Android!')" />
    <script>
      var x = document.getElementById("location");
      function getLocation() {
        if (navigator.geolocation) {
          navigator.geolocation.getCurrentPosition(showPosition);}
        else {
          x.innerHTML = "Geolocation is not supported by this browser."
              ;}}
      function showPosition(position) {
        alert(position)
        x.innerHTML = "Latitude: " + position.coords.latitude + "<br>Longitude:
            " + position.coords.longitude;}
      function showAndroidToast(toast) {
        Android.showToast(toast);}
    </script></body></html>
```

4.14 Information Leakage Vulnerabilities

Information leakage in Android app happens when an app broadcast an *Intent* object containing sensitive information to other apps and does not restrict the broadcast. So other apps can also listen to this broadcast. Listing 15 presents an example where an *Intent* is created on line 176. Sensitive car information is added to the *Intent* in (line 179-184). This information is then broadcasted to all apps (line 186). This example, demonstrates the case where any malicious app can eavesdrop on this *Intent* and send it's sensitive information to a malicious server. However, the problem can be mitigated if security best practices are implemented. Listing 15 (line 188 or 190) presents two possible solutions to mitigate the above leak. *LocalBroadcastManager* is responsible to broadcast *Intent* within the same app's components, i.e: activities, broadcast receivers and services. And *sendBroadcast* with custom permission on (line 190) ensures that the *Intent* will only be received by other applications that have the same custom permissions defined in their app's manifest file. Other applications which don't have that same custom permission defined will not be able to receive this intent.

Listing 15: Sources and Sink

```
175  private void broadcastIntent() {
176     Intent intent = createIntent(key, value);
177     intent.setAction("com.example.intent.broadcast");
178     CarInfoManager carInfo = (CarInfoManager) car.getCarManager(Car.
            INFO_SERVICE);
179     intent.putExtra("Battery Capacity", carInfo.getEvBatteryCapacity());
180     intent.putExtra("Connector Types", carInfo.getEvConnectorTypes());
181     intent.putExtra("Fuel Capacity", carInfo.getFuelCapacity());
182     intent.putExtra("Fuel Types", carInfo.getFuelTypes());
183     intent.putExtra("Manufacturer", carInfo.getManufacturer());
184     intent.putExtra("Model", carInfo.getModel());
185     // SINK
186     sendBroadcast(intent);
187     // Solution 1
188  LocalBroadcastManager.getInstance(this).sendBroadcast(v1);
189     // Solution 2
190  sendBroadcast(intent,"com.intent.broadcast.permission");}
```

5 RELATED WORK

Android Automotive is relatively new field as compared to Android. However, existing Android analysis tool can still look into common vulnerabilities. A survey paper by Keyur et al. [12] on Android analysis tool provides a list of Android tools. As shown in Table 3, We checked source code repository and reported whether the tools are publicly available and whether they are still under active development.

Static analysis tool like, CHEX [13] looks for component hijacking vulnerabilities in Android apps which includes permission re-delegation and leakage, intent spoofing, and private data leakage. It takes Android Package Kit (APK) file and perform analysis directly in Dalvik Bytecode. First it discovers all the entry point in the apps then it construct call graphs, data dependency graph and use that to look for taints, data flow of sensitive sources into sensitive sinks. It then reports component hijacking flows. It tracks 180 sources and sinks. This tool is not publicly available or hosted anywhere.

SCanDroid [10] checks the data flow on Android app by extracting security specifications from manifest file. It can check for ICC related vulnerabilities. SCanDroid does not support analysis on APK file, was not tested on real world applications and is not maintained. With all the changes in Android SDK and introduction of Automotive API, SCanDroid is simply not capable to detect tainted flows in Automotive apps.

QARK [18] decompiles Android APK files and then searches the API usages for multiple vulnerabilities and provides detailed explanation. It can produces proof of concept exploits for a given APK file, such as insecure Intent broadcast. However, It does not provide any taint analysis.

ComDroid [6] targets communication based vulnerabilities, however, the tool is not available for public use. Since ComDroid was published in 2011, this indicates that tool is outdated and might not work with Android Automotive apps.

Agrigento [7] provides a black box differential analysis technique for privacy leak detection. It works in two parts first it observes the network behaviour and then it modifies the sources of information and observe the changes in network results, as such, it monitors the network results to find information leakage in obfuscated and encrypted environment. SDLI [19], was built on top of a commercial static analyzer Julia, it generates XML reports for each inward and outward intent and compares XMLs reports for intent leakage, however, SDLI is not publically available. A recent work by Amit et al. [14] also explores the unprotected intent. The approach uses null fuzzing of intent to generate blank intents with no data to see which activities and services of apps became active due to it. Their approach helps in finding vulnerable communication component. As they report, due to intent fuzzing, many of the infotainment services became unresponsive and many apps crashed due to missing

Table 3: List of available tools for Android apps analysis.

Type	Name	Publicly Available	Last Updated
Static Analysis	CHEX	No	
	SCandroid	Yes	2012
	FlowDroid	Yes	2020
	Amandroid	Yes	2018
	QARK	Yes	2018
	ComDroid	No	
	Agrigento	Yes	2019
	HornDroid	Yes	2017
	LeakSemantic	No	
	MamaDroid	No	
	AndroBugs	Yes	2015
Dynamic Analysis	ARTDroid	Yes	2015
	VetDroid	No	
	DroidScope	Yes	2018
	CopperDroid	No	
Machine Learning	DroidScribe	No	
	StormDroid	No	
	RevealDroid	Yes	2017
Real-Time Monitoring	ARTist	No	
	TaintDroid	Yes	2016
	AppsPlayground	Yes	
	Andromaly	No	

information. Their approach helps us understand the importance of secure app to app communication specially in the automotive domain. Unresponsive infotainment systems can lead to driver distraction which impacts the safety, security of the driver as well as vehicles around it.

SIAT [11], a recent dynamic analysis tool that is based on TaintDroid [8], tests apps for information leakage. SIAT modifies Android framework by adding a monitor and analyzer to detect leaks. Monitors generates logs when information is read from sensitive sources and when is read by sinks. Whereas Analyzer observes these logs to find tainted flow from these logs. To use SIAT, users need to configure their devices with SIAT custom Android framework, then users need to write an automation script for each app. SIAT is only able to detect leaks when information leaves an app and is received by another. As monitor generates logs for each sources when it's read by an object and analyzer observes those logs to report leaks. In case of no communication it will not report that as a potential leak. So it has chances of missing some leaks if it does not find a suitable app pair. Another thing to consider with SIAT is that it was built on top of TaintDroid which was last updated on 2016 for Android 4.3 and same Android version supported by SIAT whereas Android Automotive is part of Android SDK 9.0. In that case, there tool is outdated.

FlowDroid performs context, object and flow sensitive analysis by statically analyzing .apk or .dex files. It depends on SuSi [20] to collect sources and sinks. It's a machine learning approach. It analyze Android SDK and searches for sensitive sources and sinks and generates sources and sinks files. Which we can use with FlowDroid to search for sources and sinks in an Android app. Also, to perform FlowDroid analysis, it requires Android SDK. Since, Android Automotive SDK is not available to third party developers. FlowDroid is not be able to detect Automotive sources and sinks.

AmanDroid is another context and flow sensitive taint analysis tool which takes inspiration from FlowDroid, It models Android environment for both control and data. it build inter-procedural control flow graphs (ICFG) which includes ICC communications, it treat ICC communications same as method calls and use it to build data dependency graph (DDG) and then it uses both graphs to report vulnerabilities. To track sensitive sources and sinks AmanDroid also requires a source and sink file whereas the one that comes with it does not contains Android Automotive sources and sinks.

6 CONCLUSION AND FUTURE WORK

In this study, we reviewed existing vulnerabilities of the Android platform and verified their applicability to Android Automotive apps, we also explore some use cases which are specific to Automotive apps. In total, we investigate fourteen security vulnerabilities and their applicability in Automotive apps. Our future work includes mapping these vulnerabilities with secure coding standards as they are developed for automotive platform. Carry out testing of other Android vulnerabilities and verifying their existence in Android Automotive platform. Research about tools which can look into these vulnerabilities and their applicability in Android Automotive platform.

REFERENCES

[1] Android Last accessed: November 30, 2021. Android. https://www.android.com/intl/en_ca/what-is-android/

[2] Android, Permission Last accessed: December 9, 2021. Types of Android permissions. https://developer.android.com/guide/topics/permissions/overview#types

[3] Automotive Architecture Last accessed: November 30, 2021. Android Automotive Architecture. https://source.android.com/devices/images/vehicle_hal_arch.png

[4] Broadcast Receivers Last accessed: December 9, 2021. Broadcast Receivers in Android. https://developer.android.com/guide/components/broadcasts#security-and-best-practices

[5] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, and Tadayoshi Kohno. 2011. Comprehensive Experimental Analyses of Automotive Attack Surfaces. In *SEC'11*.

[6] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. 2011. Analyzing Inter-Application Communication in Android. In *MobiSys '11*. 239–252.

[7] Andrea Continella, Yanick Fratantonio, Martina Lindorfer, Alessandro Puccetti, Ali Zand, Christopher Kruegel, and Giovanni Vigna. 2017. Obfuscation-Resilient Privacy Leak Detection for Mobile Apps Through Differential Analysis. In *NDSS*.

[8] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *OSDI'10*. 393–407.

[9] Benjamin Eriksson., Jonas Groth., and Andrei Sabelfeld. 2019. On the Road with Third-party Apps: Security Analysis of an In-vehicle App Platform. In *VEHITS,*. 64–75.

[10] A. Fuchs, A. Chaudhuri, and J. Foster. 2009. SCanDroid : Automated Security Certification of Android Applications.

[11] Yupeng Hu, Zhe Jin, Wenjia Li, Yang Xiang, and Jiliang Zhang. 2020. SIAT: A Systematic Inter-Component Communication Analysis Technology for Detecting Threats on Android. arXiv:2006.12831 [cs.CR]

[12] Keyur Kulkarni and Ahmad Y Javaid. 2018. Open Source Android Vulnerability Detection Tools: A Survey. arXiv:1807.11840 [cs.CR]

[13] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. 2012. CHEX: statically vetting Android apps for component hijacking vulnerabilities. In *CCS '12*. 229–240. https://doi.org/10.1145/2382196.2382223

[14] Amit Kr Mandal, Federica Panarotto, Agostino Cortesi, Pietro Ferrara, and Fausto Spoto. 2019. Static analysis of Android Auto infotainment and on-board diagnostics II apps. *Software: Practice and Experience* 49, 7 (2019), 1131–1161.

[15] Sahar Mazloom, Mohammad Rezaeirad, Aaron Hunter, and Damon McCoy. 2016. A Security Analysis of an In-Vehicle Infotainment and App Platform. (2016).

[16] Abdul Moiz and Manar Alalfi. 2020. An Approach for the Identification of Information Leakage in Automotive Infotainment systems. *SCAM* (2020), 110–114.

[17] OWASP Last accessed: November 30, 2021. OWASP Mobile Top 10 2016. https://owasp.org/www-project-mobile-top-10/

[18] QARK Last accessed: December 9, 2021. QARK by LinkedIn. https://github.com/linkedin/qark

[19] R. Salvia, P. Ferrara, F. Spoto, and A. Cortesi. 2018. SDLI: Static Detection of Leaks Across Intents. In *(TrustCom/BigDataSE)*. 1002–1007.

[20] SuSi Last accessed: Dec 16, 2021. SuSi - our tool to automatically discover and categorize sources and sinks in the Android framework. https://github.com/secure-software-engineering/SuSi

[21] Chris Valasek and Charlie Miller. 2015. Remote Exploitation of an Unaltered Passenger Vehicle. (2015), 93.

[8]