12-2016

Attacking Android smartphone systems without permissions

Mon Kywe SU
Singapore Management University, monkywe.su.2011@phdis.smu.edu.sg

Yingjiu LI
Singapore Management University, yjli@smu.edu.sg

Kunal PETAL
Samsung Research America

Michael GRACE
Samsung Research America

DOI: https://doi.org/10.1109/PST.2016.7906949

Attacking Android Smartphone Systems without Permissions

Su Mon Kywe[1], Yingjiu Li[1], Kunal Petal[2], Michael Grace[2]
{*monkywe.su.2011, yjli*}*@smu.edu.sg*[1]
School of Information Systems, Singapore Management University[1]
{*kunal.patel1, m1.grace*}*@samsung.com*[2]
Samsung Research America[2]

***Abstract*—Android requires third-party applications to request for permissions when they access critical mobile resources, such as users' personal information and system operations. In this paper, we present the attacks that can be launched without permissions. We first perform call graph analysis, component analysis and data-flow analysis on various parts of Android framework to retrieve unprotected APIs. Unprotected APIs provide a way of accessing resources without any permissions. We then exploit selected unprotected APIs and launch a number of attacks on Android phones. We discover that without requesting for any permissions, an attacker can access to device ID, phone service state, SIM card state, Wi-Fi and network information, as well as user setting information, such as airplane, location, NFC, USB and power modes of mobile devices. An attacker can also disturb Bluetooth discovery services, and block the incoming emails, calendar events, and Google documents. Moreover, an attacker can set volumes of devices and trigger alarm tones and ringtones that users personally set for their devices. An attacker can also launch camera, mail, music and phone applications even when the devices are locked. We compare our research on two Android versions, and discover that as platform providers incorporate more APIs, the number of unprotected APIs increases and new attacks become possible. We thus suggest platform providers to inspect Android frameworks systematically before releasing new versions.**

I. Introduction

Android adopts a permission system to protect users' security and privacy. To access resources that are out of application's sandbox, an application needs to request for permissions from users. In recent years, it has been reported that the Android permission system suffers from several flaws. For instance, unprivileged applications may leverage privileged applications to perform privileged tasks due to privilege escalation attacks [1] [2]. Several applications may collude to launch attacks with combined permissions from all of the applications [3] [4]. However, no rigorous study has been made on what potential attacks an application can launch on Android smartphone systems without requesting for any permissions. Therefore, in this paper, we question the coverage of the current protection mechanisms and investigate to what extent critical resources are exposed to malicious applications via APIs without any protection mechanisms. There are two steps in our study. In the first step, we analyze unprotected Application Programming Interfaces (APIs), which allow third-party applications without any permissions to interact with mobile system resources, such as GPS and camera, or to access users' personal information. In the second step, we demonstrate several attacks that can be easily launched by leveraging on the unprotected APIs obtained in the first step.

To retrieve unprotected APIs from Android framework, we perform the following source-code static analysis: (1) inter-procedural call graph analysis on system services for the discovery of all Android Interface Definition Language (AIDL) interfaces that are not protected by any permission checking or Linux ID checking mechanisms, (2) component analysis on system applications for identifying the exposed and unprotected broadcast receivers, activities and services, and (3) intra-procedural data-flow analysis for locating unprotected dynamically registered broadcasts in both system services and system applications. We apply our analysis on Android Open Source Project (AOSP) versions 5.1.0_r1 and 4.4.0_r1. On AOSP version 5.1.0_r1, we identify 735 unprotected APIs in system services. In system applications, we discover 612 unprotected components, where 156 are unprotected broadcast receivers, 423 are unprotected activities and 33 are unprotected services. Moreover, we discover 206 unprotected dynamically registered broadcasts, where 50 exist in system services and 156 exist in system applications. It is alarming that a high number of unprotected APIs is discovered in different parts of Android frameworks. We also compare our analysis results on versions 5.1.0_r1 and 4.4.0_r1. We discover that the number of unprotected APIs increases on the newer version due to the newly added functionalities. This is contrary to the common belief that the security of a new version should improve, since many security flaws in an old version are reported and fixed.

After obtaining unprotected APIs, we create an adversary third-party application without any permissions, which launches Java reflection attacks, broadcast injection attacks, broadcast hijacking attacks, malicious activity launch attacks, activity hijacking attacks, malicious service launch attacks, and service hijacking attacks. We discover that on Android version 4.4.0_r1, an attacker can block the synchronization of emails, calendar events, browser bookmarks, browsing history, browser extension, Google documents, and Google notes. In addition, an attacker can send notifications, set car mode, set night mode, wake up the device at certain time, and set screen-off time. We reported our attacks on AOSP version 4.4.1_r1 to

Google and some of the reported vulnerabilities are fixed on version 5.0.0_r1. Nonetheless, we still discovered more attacks on version 5.1.0_r1, which are also subsequently reported and fixed on version 5.1.1_r35 and version 6.0. This shows that while the platform providers make their effort in improving the security of Android framework, they need a powerful tool to win the "arms race".

On version 5.1.0_r1, we discover that an attacker can obtain country, Wi-Fi information, subscriber information, tether state, airplane mode, NFC state, GSM/ CDMA strength, location mode, USB state, power state and security setting for lock screens. Moreover, some resources, such as device ID and SIM card state, which should be accessed by permission-granted applications only, are accidentally made available to all applications via unprotected APIs. An attacker can arbitrarily set the volumes of Android phones and play users' incoming call ringtones, alarms, and notification sounds. An attacker can block Bluetooth discovery services, and launch camera, mail, music and phone system applications even when the targeted devices are locked. An attacker can also hijack various activities of system applications, including the interfaces for setting VPN (Virtual Private Network), Bluetooth and Wi-Fi, as well as the interfaces for adding device administrators and user accounts. These attacks show that the negligence in designing API-level permission enforcement causes various threats to users' security and privacy. We suggest platform providers to systematically analyze unprotected APIs before releasing new versions, so that similar attacks are prevented in the future.

The rest of the paper is organized as follows. In Section II, we define our adversary model. In Section III, we describe how we retrieve unprotected APIs. In Section IV, we show a set of proof-of-concept attacks on AOSP version 5.1.0_r1. In Section V, we compare our results with AOSP version 5.1.0_r1. In Section VI, we provide some discussions on our research. In Section VII, we summarize the related work. Finally, we conclude the paper in Section VIII.

II. Adversary Model

Our adversary is a third-party application without any privileges or permissions, which launches malicious operations using unprotected Android APIs. We refer to it as "an attacker" in this paper. We classify Android APIs into three categories: (1) normal APIs supported by *system services*, (2) loosely-coupled APIs supported by *system applications*, and (3) *dynamically registered broadcasts* in both system services and system applications. In the first category, API calls from applications are handled by system services, which provide main client-server interfaces between system-level processes and third-party application processes. For instance, to exercise a complete control over cameras, such as changing the zoom and flash light settings, an application may access `com.hardware.camera2` API, which in turn communicates with the system service, `android.hardware.ICameraService`. In the second category, loosely-coupled APIs are supported by system applications, which provide easy access to mobile phone functions. For instance, to take a photo or a video, an application may call the system application `Camera` using intents. Unlike normal APIs and loosely-coupled APIs, dynamically registered broadcasts in the third category are undocumented APIs. They are mainly used for internal communications among system services and system applications. All these types of APIs can be abused by an attacker when they are not properly protected.

A. System Services

Third-party applications access APIs of system services by calling the method `getSystemService(name)` of the `Context` class. The parameter `name` represents the name of the required system service. The returned object is then casted into the `Manager` class. For example, `AlarmManager` object can be retrieved by invoking the method with parameter "`alarm`". However, security checks performed inside the `Manager` class can be easily bypassed [5]. Moreover, APIs listed inside the `Manager` class are not complete; third-party applications can use Java reflection to invoke private APIs, which are marked with "@hide" annotations. Thus, we assume that an attacker may use Java reflection to interact with all unprotected APIs, including public and private APIs. Using Java reflection, an attacker invokes the `getService(name)` method inside the hidden `ServiceManager` class. Even though `ServiceManager` is a hidden class, it is unlikely to change, as the `android.jar` library relies on it to support normal APIs. The `getService(name)` method returns an `IBinder` object, which can be used to invoke any exposed methods inside the corresponding system services. Listing 1 shows an example attack on AOSP version 4.4.4_r1, where an attacker attempts to set the maximum screen-off time on mobile devices.

B. System Applications

System applications provide loosely coupled APIs by exposing their components in the applications' `AndroidManifest.xml` files. In each XML file, `<application>` is the parent element, which contains some sub-elements for the application's components, such as `<service>`, `<activity>`, and `<receiver>`. Several tags and attributes are used to protect the components of system applications from other applications. They include `intent-filter`, `exported`, `permission` and `enabled`. Each exported and enabled component without any permission protection represents an unprotected API. We consider the following types of attacks: broadcast theft, malicious broadcast injection, activity hijacking, malicious activity launch, service hijacking, and malicious service launch. Our paper is the first to consider these types of attacks on AOSP system applications and analyze them as a part of Android framework, although they have been applied on third-party applications and vendor-customized system applications by Chin et. al. [6] and Wu et al. [7] respectively. Note that we consider broadcast receivers of system applications or both broadcast theft and malicious

```
//Invoke ServiceManager.getService("power") method and obtain IBinder object of PowerManagerService
Class serviceManagerClass = Class.forName("android.os.ServiceManager");
Method getServiceMethod = serviceManagerClass.getDeclaredMethod("getService", String.class);
IBinder iBinder = (IBinder) getServiceMethod.invoke(null, "power");

//Get Stub object of IPowerManager by passing IBinder object to asInterface() method
Class stubClass = Class.forName("android.os.IPowerManager$Stub");
Method asInterfaceMethod = stubClass.getMethod("asInterface", new Class[]{IBinder.class});
Object IPowerManagerObj = asInterfaceMethod.invoke(null, iBinder);

//Invoke IPowerManager.setMaximumScreenOffTimeoutFromDeviceAdmin(0) method using Java reflection
Class IPowerManagerClass = Class.forName("android.os.IPowerManager");
Method setScreenOffTimeoutMethod = IPowerManagerClass.getDeclaredMethod
  ↪  ("setMaximumScreenOffTimeoutFromDeviceAdmin", Integer.TYPE);
System.out.print(setScreenOffTimeoutMethod.invoke(IPowerManagerObj, 0));
```

Listing 1: An Example Attack using an Unprotected API from a System Service

broadcast injection attacks, so that we can discover as many attacks as possible on AOSP framework.

In addition to `intent-filter`, `exported`, `permission` and `enabled` tags and attributes used in `AndroidManifest.xml` files, the concept of *protected broadcasts* is used for limiting broadcast injection. Protected broadcasts are broadcasts that can only be sent by applications running in system-level processes. Protected broadcasts are defined in the `AndroidManifest.xml` file of AOSP root source code. For example, if the file includes the `<protected-broadcast android:name = "android.intent.action.PACKAGE_INSTALL"/>` tag, Android system allows no applications except system-level applications to send broadcasts with the action string, `android.intent.action.PACKAGE_INSTALL`. Thus, when launching broadcast injection attacks, we exclude these protected broadcasts from a list of our discovered broadcasts.

C. Dynamically Registered Broadcasts

Both system services and system applications may register broadcast receivers dynamically using APIs, such as `registerReceiver(BroadcastReceiver,IntentFilter)` or send broadcasts using APIs, such as `sendBroadcast(Intent)`. For simplicity, we refer to these types of broadcast receivers and broadcasts as "dynamically registered broadcasts" or simply "broadcasts" under related sections. Using them, an attacker may launch broadcast theft and broadcast injection attacks.

III. Retrieving Unprotected APIs

Retrieving unprotected APIs is not trivial due to a wide variety of API types, vast presence of APIs in Android framework and different protection mechanisms enforced. In this section, we apply three types of analysis for retrieving unprotected APIs: (1) call graph analysis on APIs provided by system services, (2) component analysis on APIs supported by system applications ,and (3) data flow analysis on dynamically registered broadcasts. The result of our analysis provides a broad overview of unprotected APIs in Android framework. Our analysis is first applied to AOSP version 5.1.0_r1 with API level 22 (Lollipop). We perform call graph analysis and data flow analysis based on Soot [8] version 2.5, which is an existing Java source code analysis tool. We also develop our own tool written in Python for scanning and identifying necessary source code files, and for analyzing components of system applications. We use an LG Nexus 5 for testing.

A. Call Graph Analysis on System Services

Using Android Debug Bridge (ADB) command, we discover 97 system services in Android 5.1.0_r1 version, where 11 of them are listed without any interface names. These 11 system services are designed to communicate with other system-level processes only. We identify all the unprotected APIs of system services using call graph analysis. A call graph is a directed graph, where each node represents a method and each edge indicates the invocation of one method to another. Our call graph analysis involves three steps: (1) finding all available APIs from a system service, (2) finding all security checking methods protecting the APIs, and (3) finding whether there exists at least one method call chain from an exposed API to any security checking method.

Step 1: Finding Source Methods - The *source* methods are the public methods of system services that are exposed via AIDL interfaces. We apply Soot to load a list of system service classes, and loop through all their public methods. In this way, we discover 1,751 APIs that are exposed to third-party applications.

Step 2: Finding Sink Methods - The *sink* methods are the methods that perform security checks. In this paper, we consider permission and Linux ID checking mechanisms of system services. Some methods in Andriod framework are dedicated for permission checking [9], and we identify 35 of them, including 18 methods from the `ContextImpl` class, 2 methods from the `ActivityManager` class and 15 methods from the `PackageManagerService` class.

There is no specific method dedicated for Linux ID checking. System services normally perform the following steps for Linux ID checking. First, they obtain the UID or PID of the calling application or process using `getCallingPid()` and `getCallingUid()` methods from the `Binder` class. After that, they perform conditional check, such as "`callingUid`

`!= Process.SYSTEM_UID`", where `SYSTEM_UID` represents 1000. To locate ID checking methods, we first identify whether a method calls `getCallingPid()` and `getCallingUid()`. We then determine whether the returned variables are checked against any system-level Linux IDs in any `If` statements in the following source code. Note that the UIDs for system applications range from 0 to 9999. For instance, the UID for root user is 0, and the UID for telephony is 1001. The most commonly used UID is 1000, and it is used for running system server codes with certain privileges. As long as there exists at least one check against system-level IDs, we regard this method as a *sink* method.

Step 3: Building Call Graph - A context-insensitive interprocedural call graph is built using Soot. The set of publicly accessible methods (i.e. *source* methods) are marked as entry points of the call graphs. After building the call graph, we loop through the method calls, and check if each *source* method ends up with any *sink* methods. We then exclude the methods with any security checking. In this way, we discover a list of methods that are not protected by any security mechanisms. Our analysis discovers 735 unprotected APIs, which count for 41.98% of total public APIs of system services. Our call graph analysis shows that a large number of Android APIs are unprotected and accessible by any third-party applications without any privileges.

B. Component Analysis on System Applications

We apply component analysis to retrieve unprotected components of system applications. There are altogether 69 system applications. We extract the component information from the `AndroidManifest.xml` files of system applications. We discover altogether 110 broadcast receiver components, 414 activity components and 140 service components from 69 system applications. A single system application component may have multiple `<intent-filter>` tags with multiple action strings. To discover unprotected action strings of system components, we first analyze if the system applications implement any application-level permissions. We then explore the components of system applications that satisfy the following conditions: (1) the components' attributes contain `intent-filter`, (2) `permission` is set to `none`, and (3) `exported` is set to `none` or `true`. In our analysis, we do not consider the `enabled` attribute, since it can be changed dynamically. Table I shows the result of our component analysis on system applications. Activities represent the most common type of unprotected action strings, followed by broadcast receivers and services.

Unlike other components, broadcasts can be further protected by Android system. Such broadcasts are called *protected broadcasts*, which are defined with `<protected-broadcast>` tag. Only system-level processes are allowed to send protected broadcasts. We obtain a list of protected broadcasts from the manifest file located under directory `frameworks/base/core/res/AndroidManifest.xml`. In total, we discover 225 protected broadcasts in the manifest file. Among the 86 broadcast action strings exposed from system applications, 34 of them are protected system broadcasts. Thus, an attacker may launch broadcast injection attacks with the remaining 52 broadcasts.

	Broadcast Receivers	**Activities**	**Services**
No of unprotected action strings	156	423	33
No of unique unprotected action strings	86	189	23
No of unprotected system applications	30	45	9

TABLE I
ANALYSIS RESULT OF SYSTEM APPLICATIONS

C. Data Flow Analysis on Dynamically Registered Broadcasts

Both system services and system applications may register and send broadcasts dynamically. We first retrieve the source code files of system services and system applications. After that, we apply data flow analysis to obtain the broadcast action strings from the source code files.

1) Identifying Dynamically Registered Broadcasts: From the AIDL interfaces obtained from the ADB command, we retrieve the Java files of system services. For instance, we find `AlarmManagerService` file from `IAlarmManager` AIDL interface. To achieve this, we scan the entire framework, and obtain Java files that (1) extend AIDL interfaces, (2) create new Stub classes with AIDL interface names or (3) implement AIDL interfaces and later extend them. These are the different ways by which system services implement their AIDL interfaces. In total, we discover 80 files of system services. The remaining services are only exposed via native codes, and thus excluded from our analysis. To obtain the source code of system applications, we scan the AOSP source code directories, read in every Java file, and look for package names of system applications. In total, we collect 1,392 source code files for 69 system applications. From the identified source code files of system services and system applications, we search for broadcast registering methods, such as `registerReceiver(BroadcastReceiver, IntentFilter)`, and broadcast sending methods, such as `sendBroadcast(Intent)`, of `Context` class. We apply data flow analysis on these methods so as to obtain the action strings of dynamically registered broadcasts.

2) Data Flow Analysis: `IntentFilter` is a parameter of broadcast registering methods, and `Intent` is a parameter of broadcast sending methods. Both `IntentFilter` and `Intent` are defined using action strings. `IntentFilter` can be initialized with an action string using `new IntentFilter(String action)` method, or it can be initialized first using `new IntentFilter()` method and later defined using the `addAction(String action)` method. We perform a backward data flow analysis on these methods using Soot so as to identify the required action strings. In total, we discover 130 unique broadcast action strings (238 instances) from system services and 207 unique

action strings (424 instances) from system applications. After that, we determine whether the retrieved action strings are protected. By extracting protected broadcasts from our discovered broadcasts, we have 50 unprotected broadcasts in system services and 156 unprotected broadcasts in system applications, which can be abused by an attacker.

IV. Attacking without Permissions

Our static analysis provides a list of unprotected APIs from system services, a list of unprotected components from system applications, and a list of unprotected dynamically registered broadcasts. We confirm the attacks by exploiting them with a third-party application without any permissions. In particular, we show that Java reflection attacks can be launched on APIs supported by system services and that intent-based attacks can be launched by exploiting the APIs supported by system applications and dynamically registered broadcasts. The attacks are performed semi-automatically: many codes used in the attacks are generated automatically, while parameters required for some attacks are identified manually. For instance, we automatically generate the codes, such as `sendBroadcast(new Intent("actionString"));`, where `actionString` is replaced by the real action strings discovered from our static analysis. Note that we aim not to provide an exhaustive list of all possible attacks but to show how easily serious attacks can be launched to Android smartphone systems without requesting for any permissions.

A. System Services

We identify two main types of possible attacks via the unprotected APIs supported by system services. An attacker may control various audio functions of mobile devices and steal users' information.

1) Audio Control: A system serivce, `IAudioService`, provides several unprotected APIs for controlling the audio systems. Without requesting for any permissions, an attacker may trigger call ringtones and alarms that users personally set for their phones. An attacker may produce other special sound effects, such as notification, click, and keypress sounds. To do so, an attacker first uses the `getRingtonePlayer()` API to obtain an `IRingtonePlayer` object, and then invokes its `play()` method. Moreover, an attacker may arbitrarily set the volumes of call ringtone, alarm, notification, music, system and voice call sounds using the `setStreamVolume()` API. An attacker abusing both unprotected APIs can be dangerous. For instance, an attacker may set the devices to their highest volumes and start playing ringtone or alarm sounds continuously. In such cases, even when devices are set to silent mode, they start ringing, which may disturb users in various social situations, such as in meetings. The only way for users to stop such attacks is to shut down their phones. In similar attacks, an attacker may confuse users by playing notification sounds without sending any notifications. Alternatively, an attacker may set the volume to 0 so that users become unaware of any incoming calls or alarms.

2) Information Leakage: An attacker may obtain information about user's device ID, SIM card state, call state, ringer mode, input devices, country and copied data from clipboard. The unprotected APIs exploited for such attacks are shown in Table II. Interestingly, we discover that device ID and SIM card state are accidentally made available via unprotected APIs, although they are supposed to be protected by `android.permission.READ_PHONE_STATE` permission. An attacker, who tracks such information continuously, can easily identify individual users and infer users' behaviours, which violates users' privacy.

B. System Applications

By exploiting unprotected APIs of system applications, an attacker may launch the following attacks: broadcast theft, malicious broadcast injection, activity hijacking, malicious activity launch, service hijacking, and malicious service launch. Broadcast theft, activity hijacking, and service hijacking attacks occur when an attacker intercepts intents by registering intent filters with unprotected action strings in its `AndroidManifest.xml` file. Malicious broadcast injection, malicious activity launch, and malicious service launch attacks occur when an attacker sends intents with `sendBroadcast(Intent)`, `startActivity(Intent)` and `startService(Intent)` methods. Such intents are initialized with unprotected action strings identified in the previous section. The intent theft attacks normally result in information leakage and component hijacking, while the other attacks result in unintended changes of system state.

1) Broadcast Theft: We discover that an attacker is able to obtain network, alarm, and account related information by intercepting unprotected broadcast intents. From the broadcast with action string `android.net.conn.CONNECTIVITY_CHANGE`, an attack may obtain network name, network state (e.g. connected, disconnected, connecting), network type (e.g. Wi-Fi or mobile LTE), and roaming status. An attacker may receive `android.app.action.NEXT_ALARM_CLOCK_CHANGED` broadcast when a next alarm is set on the device. An attacker can also obtain `android.accounts.LOGIN_ACCOUNTS_ CHANGED` broadcast when user account information (e.g., Gmail, Facebook, Skype account) is changed. Although some information leakage seems benign, it becomes serious when combined with other information. For instance, by constantly retrieving device ID and network name, an attacker may identify an individual user and determine the user's home and work locations.

2) Malicious Broadcast Injection: Various attacks can be launched by sending broadcasts with unprotected action strings. An interesting finding is that the action string `android.bluetooth.intent.DISCOVERABLE_TIMEOUT` is unprotected. By continuously sending broadcasts with this action string, an attacker can block other Bluetooth phones from discovering the exploited device. This attack disables the scan mode of the device's Bluetooth adapter and thus, makes its Bluetooth service unusable.

Leaked Information	Description	Exploited Method	Exploited Class
Device ID	Unique device ID, such as IMEI for GSM and the MEID or ESN for CDMA phones	getDeviceId()	IPhoneSubInfo
SIM Card State	Whether SIM card is ready, absent or requires PIN to unlock	getSimStateForSubscriber()	iSub
Lock Setting	Whether user sets password or pattern lock	havePassword() and havePattern()	ILockSettings
Call state	Whether there is an incoming call, established telephony call, or established audio/video chat or VoIP call	getMode()	IAudioService
Ringtone mode	Whether ringer mode is silent and vibrate, silent and not vibrate or normal	getRingerModeInternal()	IAudioService
Input Device	External and internal input devices, such as joystick or keyboard type	getInputDevice()	IInputManager
Country	Current country of user	detectCountry()	ICountryDetector
Copied data	Copied data from clip board	addPrimaryClipChangedListener()	IClipboard

TABLE II
INFORMATION LEAKAGE FROM SYSTEM SERVICES

Another finding is that by exploiting the action strings, `android.btopp.intent.action.OPEN_RECEIVED_FILES` and `android.intent.action.DOWNLOAD_NOTIFICATION_CLICKED`, an attacker may open the folders where the mobile user receives files from Bluetooth transfer, and where the downloaded files exist. Finally, an attacker may launch an input method chooser for different languages using action string `android.settings.SHOW_INPUT_METHOD_PICKER`.

3) Activity Hijacking: During activity hijacking, an attacker launches its own applications when intents with unprotected action strings are triggered. From our analysis, we discover that `android.intent.action.DIAL` and `android.media.action.STILL_IMAGE_CAMERA_SECURE` action strings are not protected by any permissions. Thus, an attacker may hijack phone and camera applications, when users launch them from their lock screens. Another finding is that an attacker may hijack Bluetooth, Wi-Fi, account (e.g. Gmail account), and Virtual Private Network (VPN) setting pages, when they are launched from the Setting application. The exploited action strings include `android.settings.BLUETOOTH_SETTINGS`, `android.settings.WIFI_SETTINGS`, `android.settings.ADD_ACCOUNT_SETTINGS`, and `android.net.vpn.SETTINGS`. However, activity hijacking is hindered by the application chooser, which is launched, when there are conflicting applications handling the same intent. It is thus difficult for an attacker to launch these attacks without being noticed.

4) Malicious Activity Launch: We discover that an attacker may launch several unprotected activities from system applications. Since some activities are entry points of system applications, this attack leads to the launching of the corresponding applications. An attacker may launch lock screen, emergency dialer, camera, mail, and music applications in such attacks. Some attacks, such as launching lock screen and emergency dialer, may confuse users, while other attacks, such as launching camera, may drain device batteries. In the following, we provide more details about such attacks for different system applications.

Warnings: An attacker may launch activities for the following warning messages: "Network monitoring: A third party is capable of monitoring your network activity, including emails, apps, and secure websites. A trusted credential installed on your device is making this possible.", "'Attention. You need to set a lock screen PIN or password before you can use credential storage," "Attention. Remove all contents? Cancel or Ok," and "Oops! This device is already set up." A severe consequence of these attacks is that user's selection from the warning messages takes real effect on the state of the phone.

Setting UIs: 67 activities of the system setting application are exposed to third-party applications in our findings. These activities include setting User Interfaces (UIs) for security (lock screen, encryption, credential storage and device administration), privacy (factory reset, restore and backup), developer options, Bluetooth, Near Field Communication (NFC) payment, Wi-Fi, location, sound, USB, and system notification. An attacker may launch these interfaces at any time without requesting for any permissions.

Others: Several hidden features can be launched using unprotected action strings. An example is the colour correction setting. When this activity is launched, the exploited interface states that this colour correction feature is experimental and may affect the performance of phones. Another attack is to launch the mobile emergency alert setting page, which lists the threats to life and property (e.g., robbery) around the area. Other unprotected activities includes the Wi-Fi network choosing interface, the brightness setting interface, the wallpaper setting interface, the live wallpaper choosing interface, and the downloaded file interface.

5) Service Hijacking: From Android 5.0 and above, only explicit intents with clearly stated package names can be used for binding services. Consequently, an attacker cannot launch any service hijacking attacks by simply declaring similar services with the same action strings as those of system applications' services.

6) Malicious Service Launch: There are two steps involved in launching the malicious service launch. An attacker first binds the services exposed from system applications and then invokes the methods inside. We discover that an attacker can successfully bind 15 services of system applications, including 14 services from Bluetooth

Leaked Information	Description	Exploited Broadcast Action String
Network and Wi-Fi	`NetworkInfo` object - network name, network state (e.g. Connected, disconnected, connecting), network type (e.g. Wi-Fi or mobile LTE) `WifiInfo` object- SSID, BSSID, MAC address, link speed, frequency `LinkProperties` object - Interface name, link address, routes, DNS address, domains	android.net.conn.CONNECTIVITY_CHANGE android.net.wifi.STATE_CHANGE android.net.wifi.WIFI_STATE_CHANGED
Tether State	Which portable Wi-Fi hotspot is on and available	android.net.conn.TETHER_STATE_CHANGED
Airplane Mode	Whether airplane mode is on or off	android.intent.action.AIRPLANE_MODE
NFC State	Whether NFC is on or off	android.nfc.action.ADAPTER_STATE_ CHANGED
SIM Card State	Whether SIM card state, such as ready or absent, changes	android.intent.action.SIM_STATE_CHANGED
Phone Service State	Whether phone is in service, out of service, emergency only or power off	android.intent.action.SERVICE_STATE
Subscription State	Whether data, SMS or voice subscription changes	android.intent.action.ACTION_DEFAULT_SUBSCRIPTION_CHANGED android.intent.action.ACTION_DEFAULT_DATA_SUBSCRIPTION_CHANGED android.intent.action.ACTION_DEFAULT_SMS_SUBSCRIPTION_CHANGED android.intent.action.ACTION_DEFAULT_VOICE_SUBSCRIPTION_CHANGED
GSM/CDMA Strength	Various measurements including LteRsrp, LteRssbr, LteCqi, CdmaDbm, CdmaEcio, GsmSignalStrength, EvdoDbm, EvdoSnr, EvdoEcio, GsmBitErrorRate	android.intent.action.SIG_STR
Location Mode	Whether location mode, such as high accuracy (use GPS, Wi-Fi, cellular network to determine location), battery saving (use Wi-Fi and cellular network to determine location) or device only (Use GPS to determine location), changes	android.location.MODE_CHANGED android.location.PROVIDERS_CHANGED
Volume	Volum value and whether phone is muted	android.media.VOLUME_CHANGED_ACTION android.media.RINGER_MODE_CHANGED android.media.STREAM_MUTE_CHANGED_ACTION
USB State	Whether USB is connected, in ADB mode or configured	android.hardware.usb.action.USB_STATE
Power State	Whether power is connected or disconnected	android.intent.action.ACTION_POWER_CONNECTED android.intent.action.ACTION_POWER_DISCONNECTED

TABLE III
INFORMATION LEAKAGE FROM DYNAMICALLY REGISTERED BROADCASTS

system applications and one media service. An attacker may search for the class names of the exposed services in `AndroidManifest.xml` files of system applications, and use their class names for binding with explicit intents. After binding, however, an attacker cannot invoke any exposed methods from these services, because these methods are well-protected inside the source code of services. For instance, the methods from the Bluetooth system application are protected by the `android.permission.BLUETOOTH` and `android.permission.BLUETOOTH_ADMIN` permissions. Therefore, an attacker cannot launch any useful attacks by simply invoking these exposed methods.

C. Dynamically Registered Broadcasts

We show that several broadcast theft and malicious broadcast injection attacks can be launched by exploiting unprotected dynamically registered broadcasts.

1) Broadcast Theft: Similar to the broadcast theft attacks to system applications, an attacker may steal user information from unprotected dynamically registered broadcasts. We discover that an attacker is able to obtain network and Wi-Fi information, tether state, airplane mode, NFC state, SIM card state, phone service state, subscription state, GSM/CDMA strength, location mode, volume, USB state, and power state. A detailed description about the information leakage due to broadcast theft is given in Table III. Although some of the leaked information seems benign, much useful information can be inferred from it. For example, location information can be inferred from Wi-Fi data and GSM/CDMA strength [10]. Users' payment and travel behaviours may be inferred from NFC state and airplane mode. Users' sleeping patterns can be inferred from USB state and power state [11]. We discover that some information is available to an attack application without any permissions, even though it is stated in Android API documentation that such information must be protected by permissions. For instance, according to Android API documentation, network and Wi-Fi information should be protected by permission `android.permission.ACCESS_NETWORK_STATE`; the SIM card state, phone state, and GSM/CDMS signal strength information should be protected by permission `android.permission.READ_PHONE_STATE`. This shows that dynamically registered broadcasts leak a lot of information to third-party applications, and platform providers should take additional steps to protect these broadcasts.

2) Malicious Broadcast Injection: An attacker may broadcast false information via malicious broadcast injections. We discover that intended receivers of these unprotected broadcasts are third-party applications or vendor-customized system applications. However, they are excluded from our study, as we focus only on Android framework as the attack target. Thus, although we have confirmed that an attacker can send these broadcasts, further analysis is required to study the impact of the attacks to third-party applications and vendor-customized system applications.

We discover that an attacker may send false commands for music applications, such as "next", "pause", "previous", and "toggle pause". The exploited action strings in this attack include `com.android.music.musicservicecommand.next`, `com.android.music.musicservicecommand.pause`, `com.android.music.musicservicecommand.previous`, and `com.android.music.musicservicecommand.togglepause`. Moreover, an attacker may send malicious information about the status of currently running music, such as its metadata, play state, and queue state. The exploited action strings include `com.android.music.metachanged`, `com.android.music. playstatechanged`, and `com.android.music.queuechanged` broadcasts. We also discover that an attacker may send broadcast `android.security.STORAGE_CHANGED`. This broadcast is triggered when (i) a new Certificate Authority (CA) is added, (ii) an existing CA is removed or disabled, (iii) a disabled CA is enabled, or (iv) the trusted storage is reset. This attack may cause serious problems to the receiving applications that act according to the received broadcasts. Moreover, an attacker may maliciously broadcast user log-in account, NFC state, data connection state, and emergency callback mode changes. The exploited action strings in these cases are `android.accounts.LOGIN_ACCOUNTS_CHANGED`, `android.nfc.action.ADAPTER_STATE_CHANGED`, `android.intent.action.PRECISE_DATA_CONNECTION_STATE_CHANGED` and `android.intent.action.EMERGENCY_CALLBACK_MODE_CHANGED`.

V. Attacking a Different Version

We apply our study to AOSP version 4.4.4_r1, and compare to what we have discovered on AOSP version 5.1.0_r1. We discover the differences in terms of unprotected APIs and viable attacks on these two versions. The attacks on version 4.4.4_r1 have been reported to the Google's security team, and most of them have been mitigated in version 5.1.0_r1. Even so, we still discover more unprotected APIs and new attacks in version 5.1.0_r1, which have also been reported to Google. This implies that the ad-hoc effort in mitigating the reported attacks is not sufficient, and systematic analysis would be helpful for platform developers to analyze unprotected APIs and improve the security of new AOSP versions.

A. Retrieving Unprotected APIs

We discover 79 system services and 69 system applications on AOSP version 4.4.4_r1. Compared to AOSP version 5.1.0_r1, we have 10 less system services, and the same number of system applications. Some system services, such as the fingerprint service and the web-view update service are not included in AOSP 4.4.4_r1. Our analysis reveals 557 unprotected APIs from AIDL interfaces of system services, which count for 34.77% of all 1,602 public methods on AOSP version 4.4.4_r1. There are 88 unprotected unique broadcast action strings (150 instances), 165 unprotected activity action strings (394 instances) and 18 unprotected service action strings (30 instances) from system applications in our results. It is also discovered that 47 out of total 114 dynamically registered broadcasts in the source code of system services are unprotected, and 124 out of 171 dynamically registered broadcasts in source code of system applications are unprotected. Compared to AOSP 5.1.0_r1, the number of unprotected APIs is smaller. This result is alarming, since it indicates that more unprotected APIs are introduced to the framework as new APIs are added in the later version.

B. Attacking without Permissions

Our attacks on AOSP previous version 4.4.4_r1 can be summarized as follows.

1) Denial-of-Service Attacks: By exploiting a single unprotected API, `Content.cancelSync()`, an attacker may launch denial of service attacks on the synchronization of all content providers. This synchronization API is used for transferring data between an Android device and web servers. We discover that on Nexus 5, an attacker may block the synchronization of Gmail, Google Calendar, Google Drive, Google Note, Chrome and etc. By doing so, the attacker can prevent users from receiving new emails, even when users manually click on "refresh" in email apps. An attacker may also prevent synchronizing new calendar events with users' desktop computers, receiving newly shared google drive documents, and synchronizing Google notes, synchronizing Chrome's bookmarks, history, tabs, passwords, extensions and many browser-related information. Moreover, we discover that other popular applications, including Dropbox, Twitter, Facebook, Skype and Mozilla Firefox, also use the synchronization API `Content.cancelSync()`. For instance, Skype uses the API for synchronizing contact information, while Firefox uses it for synchronizing bookmarks, history, tab and password information. Thus, their synchronization functions can be deferred by an attacker.

2) Other Attacks: An attacker may send notifications to users, set car mode (which is used to open speaker directly from calls), set night mode (which allows the OS to intelligently change the color theme depending on the time of day), wake up the device at certain time (without the wake-lock permission), and set the screen-off time. Moreover, an attacker may obtain a variety of valuable information from users' devices, including what password salts are used, whether users set security for lock screen, whether users use passwords, pins

or pattern locks for log-in, whether the lock screen is on or off, and whether the screen is turned on. Moreover, a false system notification can be sent to show that devices have entered into the emergency callback mode.

VI. Discussions

Our research reveals many attacks that can be launched by applications with no permissions. This discovery is important, because a significant portion of Android security research focuses on applications that have permissions, and no one has looked into the unprotected resources, which are easily accessible without permissions. Many of our attacks, after being reported twice for two versions, have been acknowledged and fixed by the platform provider. This shows that our analysis on unprotected APIs is necessary in improving the security of Android framework. Note that we do not suggest to reclassify and protect all the corresponding resources that are attacked in this paper. The reason is that protecting all resources may degrade the usability of the framework. For example, usability researchers state that too many permission requests may cause users to grant permissions without careful considerations [12]. Therefore, while we highlight the security flaws of unprotected APIs in this paper, we believe that an optimal defense mechanism should consider not only the security and privacy aspects but also the flexibility and usability aspects of the framework. Coming up with an optimal solution for this problem is not trivial and requires involvement from both research and industry communities. Thus, we leave it as future work to find various ways of protecting the currently unprotected resources without degrading other aspects of the framework. In the meantime, we suggest platform providers to systematically analyze unprotected APIs before releasing new versions, so that similar attacks are prevented in the future.

We identify three ways in which our analysis of Android APIs can be improved and used as a commercial vulnerability analysis tool. First, our paper focuses only on detecting unprotected APIs and exploiting them for attacks. Thus, a natural step forward is to determine whether an unprotected API is indeed vulnerable by analyzing the nature of the API source code. Second, platform providers may consider analyzing other types of unprotected APIs, such as callback methods, listeners and class fields. Callback methods and listeners provide alternative ways of inter-process communication, and they may expose some vulnerabilities from Android APIs. Third, platform providers may consider a more advanced adversary, which possesses certain privileges or permissions. Such adversary may be categorized according to its permission level, such as `normal`, `dangerous`, `signature`, and `signatureOrSystem`, and/or according to the types of Linux users associated to privileges, such as `root`, `system`, `keystore`, `media`, `nobody`, `wifi`, and `u0_a86`. A more advanced adversary would lead to more serious attacks.

VII. Related Work

The mapping between API calls and permission checks on Android has been investigated in prior research, including Stowaway by Felt et al. [5], COPES by Bartel et al. [13] and PScout by Au et al. [14]. Our work is different from these works in that they focus on permission usage, while our paper focuses on unprotected APIs and potential exploits. Besides permission checking, we also consider Linux ID checking in our analysis.

The topic of system-level vulnerabilities in Android framework has been studied before. DexDiff by Mitchell et al. [15] investigates the vulnerabilities in vendor- customized frameworks by comparing them with the official Android systems in binary analysis. ADDICTED by Zhou et al. [16] performs the analysis of vendor-customized components. In particular, it identifies critical Linux files and compares their protection levels in terms of Linux file permissions between customized framework and AOSP. If a file is less protected on customized framework, then it is more likely to be attacked. This line of works focuses on the vendor-modified components of Android frameworks, while we focus on unprotected APIs and their exploits in Android frameworks. Similarly, Wu et al. [7] study vendor customizations of system applications. They discover that the vendor-customized applications are vulnerable to permission re-delegation attacks, confused deputy attacks, passive content leak attacks, and content pollution attacks. Another way of Android vulnerability analysis is performed by Yang et al. in IntentFuzzer [17] and Ye et al. in DroidFuzzer [18]. They automatically construct intents and use brute force to discover vulnerabilities. However, as stated in IntentFuzzer, their research does not penetrate deep into application logic nor uncover interesting bugs for launching serious attacks. Kratos [19] also uses call graph analysis to find vulnerabilities in Android framework. However, it focuses on the inconsistencies of security checking, while our paper focuses on APIs without any security checking.

Vulnerable components of third-party applications have been investigated before. ComDroid by Chin et. al. [6] analyze the inter-application communications among third-party applications so as to identify vulnerable components of third-party applications. We use a similar threat model in our analysis and apply it on system applications. Wu et al. [20] use the reachability analysis to identify and categorize such vulnerabilities. Another work, EPICC, by Octeau et al. [21] show that over 93% of third-party applications contain vulnerable components. To exploit the vulnerable components, Li et al. [22] propose an approach which can automatically generates an attack application. In comparison to these works, part of analysis in our research focuses on intent-based vulnerabilities of system services and system applications, rather than third party applications. On the other hand, a line of work focuses on how to prevent attacks from exploiting vulnerable components of third-party applications. For example, CHEX by Lu et al. [23] mitigates such attacks by statically vetting third-party applications. AppSealer by Zhang and Yin [24] focuses on how to generate security patches automatically so as to prevent the intent-based attacks to vulnerable components of third-party applications. Oh et al. [25] propose a solution to address the denial of service attacks on ordered broadcasts.

VIII. Conclusions

In this paper, we show how an attacker, which is a third-party application without any permissions, can attack Android smartphone systems by exploiting various unprotected Android APIs, including unprotected AIDL interfaces of system services, unprotected components of system applications, and unprotected dynamically registered broadcasts. The attacks we discover include blocking Bluetooth and email services, controlling audio functions, stealing valuable device information, and hijacking system activities and broadcasts. The result of this paper suggests that Android platform providers should carefully analyze the exposed APIs, and mitigate any identified attacks. We envision that with more features added to Android devices, larger source code sizes of Android frameworks, and faster paced releases of Android versions, such analysis and mitigations are much needed to achieve better security in Android system development.

References

[1] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Privilege escalation attacks on android," in *Proceedings of the 13th International Conference on Information Security*, ser. ISC'10. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 346–360. [Online]. Available: http://dl.acm.org/citation.cfm?id=1949317.1949356

[2] P. P. Chan, L. C. Hui, and S. M. Yiu, "Droidchecker: Analyzing android applications for capability leak," in *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, ser. WISEC '12. New York, NY, USA: ACM, 2012, pp. 125–136. [Online]. Available: http://doi.acm.org/10.1145/2185448.2185466

[3] A. P. Felt, S. Hanna, E. Chin, H. J. Wang, and E. Moshchuk, "Permission re-delegation: Attacks and defenses," in *In 20th Usenix Security Symposium*, 2011.

[4] R. Schlegel, K. Zhang, X. yong Zhou, M. Intwala, A. Kapadia, and X. Wang, "Soundcomber: A stealthy and context-aware sound trojan for smartphones." in *NDSS*. The Internet Society, 2011. [Online]. Available: http://dblp.uni-trier.de/db/conf/ndss/ndss2011.html#SchlegelZZIKW11

[5] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 627–638. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046779

[6] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '11. New York, NY, USA: ACM, 2011, pp. 239–252. [Online]. Available: http://doi.acm.org/10.1145/1999995.2000018

[7] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang, "The impact of vendor customizations on android security," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '13. New York, NY, USA: ACM, 2013, pp. 623–634. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516728

[8] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The soot framework for java program analysis: a retrospective," *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, 2011.

[9] Android, "Enforcing permissions in androidmanifest.xml," http://developer.android.com/guide/topics/security/ permissions.html, [Online; accessed 9-October-2014].

[10] L. T. Nguyen, Y. Tian, S. Cho, W. Kwak, S. Parab, Y. S. Kim, P. Tague, and J. Zhang, "Unlocin: Unauthorized location inference on smartphones without being caught," in *2013 International Conference on Privacy and Security in Mobile Systems, PRISMS 2013, Atlantic City, NJ, USA, June 24-27, 2013*, 2013, pp. 1–8. [Online]. Available: http://dx.doi.org/10.1109/PRISMS.2013.6927176

[11] T. Hao, G. Xing, and G. Zhou, "isleep: Unobtrusive sleep quality monitoring using smartphones," in *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, ser. SenSys '13. New York, NY, USA: ACM, 2013, pp. 4:1–4:14. [Online]. Available: http://doi.acm.org/10.1145/2517351.2517359

[12] A. P. Felt, S. Egelman, M. Finifter, D. Akhawe, and D. Wagner, "How to ask for permission," in *Proceedings of the 7th USENIX Conference on Hot Topics in Security*, ser. HotSec'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 7–7. [Online]. Available: http://dl.acm.org/citation.cfm?id=2372387.2372394

[13] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus, "Automatically securing permission-based software by reducing the attack surface: An application to android," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2012. New York, NY, USA: ACM, 2012, pp. 274–277. [Online]. Available: http://doi.acm.org/10.1145/2351676.2351722

[14] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: Analyzing the android permission specification," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 217–228. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382222

[15] M. Mitchell, G. Tian, and Z. Wang, "Systematic audit of third-party android phones," in *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '14. New York, NY, USA: ACM, 2014, pp. 175–186. [Online]. Available: http://doi.acm.org/10.1145/2557547.2557557

[16] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang, "The peril of fragmentation: Security hazards in android device driver customizations," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, ser. SP '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 409–423. [Online]. Available: http://dx.doi.org/10.1109/SP.2014.33

[17] K. Yang, J. Zhuge, Y. Wang, L. Zhou, and H. Duan, "Intentfuzzer: Detecting capability leaks of android applications," in *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '14. New York, NY, USA: ACM, 2014, pp. 531–536. [Online]. Available: http://doi.acm.org/10.1145/2590296.2590316

[18] H. Ye, S. Cheng, L. Zhang, and F. Jiang, "Droidfuzzer: Fuzzing the android apps with intent-filter tag," in *Proceedings of International Conference on Advances in Mobile Computing and Multimedia*, ser. MoMM '13. New York, NY, USA: ACM, 2013, pp. 68:68–68:74. [Online]. Available: http://doi.acm.org/10.1145/2536853.2536881

[19] Y. Shao, J. Ott, Q. A. Chen, Z. Qian, and Z. M. Mao, "Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework," in *Proceedings of the 23rd Network and Distributed System Security Symposium (NDSS'16)*, San Diego, CA, February 2016.

[20] D. Wu, X. Luo, and R. K. Chang, "A sink-driven approach to detecting exposed component vulnerabilities in android apps," *arXiv preprint arXiv:1405.6282*, 2014.

[21] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon, "Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis," in *Proceedings of the 22Nd USENIX Conference on Security*, ser. SEC'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 543–558. [Online]. Available: http://dl.acm.org/citation.cfm?id=2534766.2534813

[22] L. Li, A. Bartel, J. Klein, and Y. Le Traon, "Automatically exploiting potential component leaks in android applications," in *Proceedings of the 13th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom 2014)*. IEEE, 2014.

[23] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: Statically vetting android apps for component hijacking vulnerabilities," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 229–240. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382223

[24] M. Zhang and H. Yin, "Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications," *Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS'14)*, 2014.

[25] J.-S. Oh, M.-W. Park, and T.-M. Chung, "The solution of denial of service attack on ordered broadcast intent," in *Advanced Communication Technology (ICACT), 2014 16th International Conference on*, Feb 2014, pp. 397–400.