I Can See Your Future! Analysing the Risk of Observable Callbacks in Android System

Chenkai Guo
College of Computer and Control Engineering
Nankai University
Tianjin, China
guochenkai88@gmail.com

Guangdong Bai
School of Computing
National University of Singapore
Singapore
baiguangdong@gmail.com

Jinsong Dong
School of Computing
National University of Singapore
Singapore
dcsdjs@nus.edu.sg

Jing Xu
College of Computer and Control Engineering
Nankai University
Tianjin, China
xujing@nankai.edu.cn

BalaBala
NASA Ames Research Center
Moffett Field
California 94035
fogartys@amesres.org

LabaLaba
Palmer Research Laboratories
8600 Datapoint Drive
San Antonio, Texas 78229
cpalmer@prl.com

ABSTRACT

The feature of event-driven acts as a key role that makes Android application differentiate from traditional PC software. For many of those events are hardly predicted and could not be observed by other applications, attackers are similarly impossible to engage corresponding attacks by finding the vulnerabilities of such an event-driven mechanism. However, of various kinds of events offered by either user or system, there are still events that can be received by more than one applications and further, could offer important basic resources to predict specific behaviours of targeted application.

PEC-threats(Public Events Callback) In this paper, we perform a systematic study of such "public events" and their callback functions, and analyse potential security threats inside them. By exploiting the prediction capability of such callbacks, we demonstrate typical kinds of proof-of-concept attack examples, including spoofing, phishing and privilege escalation. To evaluate the real influence of such threats, we implement a static analysis tool to detect a large scale of real world applications achieved from Google Play market. The detection result reveals such callback threats are pervasively existing in Android market applications. Since the threats involve in the fundamental design of the Android framework, it is very hard for ordinary developers and users to defense them. Given this challenge, we also proposed a migration strategy.

CCS Concepts

•**Computer systems organization** → **Embedded systems;** *Redundancy;* Robotics; •**Networks** → Network reliability;

Keywords

Android security; Information leaks; Permissions

WOODSTOCK '97 El Paso, Texas USA

DOI: 10.475/123_4

1. INTRODUCTION

Android platform has been already proven to be prevalent as a typical operating system installed in nowadays smartphone and tablet. Different from traditional PC program and software, applications within Android are event-driven and without a launching main function. That is to say, with different events, the program control flow would change correspondingly. Thus, call back mechanism that is used to response an event has played an important role in Android framework.

Callback mechanism is originated from PC programs, which provides an asynchronous way for a specific class to invoke another's method when a certain event happens. In Android, callback functions reside everywhere of program structure in a typical app, especially in UI handling. For instance, after user click a button, a new activity would be popped up, presenting corresponding functionalities expected by user. In this case, a typical callback mechanism is used to response this specific user-click event. That is to say, the new activity would not be activated until user event happens. Besides, the events from system, other apps or life-cycle have similar callback responding mechanism.

On the other side, Android sandbox is known as an effect mechanism to protect apps from other one's attack. Sandbox restricts the resources to be offered for certain independent entities, leaving only very few ways for them to communicate with others. Actually, through the usage of conventional so-called side-channels, certain indirect inferring gradually evolves into a typical sort of attacks. In general, this sort of attacks threat a broad range of apps causing side-channels(such as process status) exist in system level. In other words, most of developed apps would be unable to avoid the influence brought by features from system level. Nevertheless, the impact appears limited for the deviation posed by sophisticated algorithms from current side-channel

approaches. For instance, [9](UI state) treats several selected process statuses as side-channels, and utilizes them to infer a proper emergence time for the fake phishing Activity devised by attackers. However, the critical inference bases on a series of data collection and computations, leading to an inevitable derivation in term of device and targeted app itself. A typical example is that the typical Activity transition signal presence in [9] is quite week and the shape is also changed in our reproducing experiment. Thus, using the same procedures as proposed in [9] would hardly perform an ideal attack. The similar situation exists in other existing side-channel attacks aiming to Android apps as well [42][18].

Compared to conventional side-channel attacks, we introduce a new type of potential threats called public event callback(PEC) threats, which leverage the weakness of public event callback mechanism within Android apps. Generally, callback functions expect a typical event to trigger their execution, and correspondingly the event is specified to sent to these callbacks. For example, the explicit intent specifies the single address(package name) of the target recipient and sends the intent out through a broadcast event. The broadcast only can be received by the specified single recipient. The entire broadcast process is properly protected by such single receiver mechanism. It is hard to obtain any data of such intent by other apps from the application level. However, not all the cases emerge so ideally. There still exists a large spectrum of public events that can trigger the execution of callbacks from different components and even different apps. We call this kind of callbacks PECs(public event callbacks).Examples includes system event, implicit intent, service running status, etc., which are insensitive on surface but offer attack materials for other apps with evil purpose. In particular, as the source of the expected event is possible to be exposed to public, the invoking time of a typical callback function could be inferred by other apps. As a result, the inferring result evolves to a variety of corresponding attacks directed against the display-sink, such as spoofing, phishing, privilege escalation, etc.

However, the inferring process might not be as such simple as imaging. One challenge is that attackers have to fall into code level to gain necessary materials of victim apps overcoming the code confusion. Moreover, the attackers have to detect out at least one feasible inferring path across numerous control flow paths within targeted app. Such path has to contain a callback function that can be invoked by an exposed public event and a display-sink(e.g., startActivity or Toast.makeText) point with exposure features. The other challenge is that it should to be proven that there exists a determinable path from the start of such callback function to the display-sink, which involves more fine-grain analysis of program control flow. To achieve this goal, branch conditions and method invocation have to be properly handled to avoid the restriction of permission request and user-interaction. Aiming at above challenges, we implement a statical analysis prototype tool CSDroid with 4k lines python codes to automatically conduct this inferring process to 2375** apps collected from Googly Play. Of all the targeted apps, we find 783** such sensitive paths within 354** targeted apps, including some popular apps like **. Moreover, it's surprising that 69% ** of apps with service component suffer from such threats, which alerts us a high risk of such potential security threats within background components widely existing in generic apps.

Given such an app with certain PEC weaknesses, an adversary attacker could devise a rich variety of attacks. Since such attacks are mainly based on the display-sink, the attack types are somehow similar with traditional UI attacks. However, as the display-sink should be reasonably inferred from the PEC, adversary app needs to concern more details about time-delay and program flow conditions. Besides that, traditional UI attacks mainly rely on activity display of victim app; display-sink contains more types of displays such as dialog and notification, which allow attackers to flexibly craft more attacks contents. To validate the feasibility of such PEC weaknesses exploitation, we explore and implement a set of proof-of-concept attacks based on such threats, including not only traditional phishing and spoofing to sensitive pages, but also newly dialog and notification exploitation for privilege escalation and other privacy steal.

To migrate the impact of such PEC threats, we also discuss the defence strategy from two parts: Android system and developers.

To summarize, the main contribution of this paper are:

1. We systematically study vulnerabilities of Android app lead by abuse of public event callbacks as well as discover various four types of attack vectors leading to such PEC threats.

2. By a set of corresponding proof-of-concept attacks, we validate the feasibility of such PEC exploitation and illustrate the severity of such threats.

3. We introduce a tool implemented to automatically detect the PEC threats within targeted apps. By our study, a wide range of apps are affected by this kind of PEC attacks.

4. We provide a mitigation strategy to the PEC threats, under which a more security callback invoking mechanism can be established.

2. BACKGROUND

In order to better understand the mechanism of this kind of attacks, firstly some relative background knowledge needs to be introduced.

2.0.1 *Events and Callbacks*

Events and callbacks are the ways that app interacts with Android system and users. Here, we define an *event* as a situation that can trigger certain program logic to work. For instance, the explicit intent broadcast is such an event that can trigger a certain app receiving it and executing specific code after receiving. Again, we define a *public event* as situation that can trigger the programs logics from more than one app to work. Correspondingly, an implicit intent broadcast caters the definition of public event, which can be received by multiple apps. Apparently, the public event is the subset of the event. Furthermore, we define a *callback* as the code(e.g., function and program branch) that is executed after certain event(s) happen(s), and the execution is only related to such event(s). For example, an onClick method would only be executed after user clicks certain button.

2.0.2 *Events in Android*

The trigger-events originated by different sources are equipped with different trigger features. For instance, a *low-battery* event would happen only when the rest quantity of battery becomes as low as a pre-defined level, which is irrelevant to user's interaction. We categorize the events lied in

Android that can lead to certain invocation of callback as three groups, namely *life-cycle* event, *user-driven* event, and *system-driven* event.

Life-cycle event Android apps spontaneously arouse corresponding callback function according to the life-cycle step it resides. We call the event that can trigger life-cycle method to execute as life-cycle event. For instance, an Activity executes its *onCreate* function when it is initially loaded, and executes the *onPause* callback after it loses the screen focus. Life-cycle callbacks are somehow executed along with user's interactions and impacted by users. On surface, due to constraint of sandbox mechanism, the life-cycle phrases (Create,Pause,Destroy, etc.) of an given activity seem to be isolated from components belong to other apps. However, the event triggering certain life-cycle callbacks like onCreate and onDestroy enables other app to predict it in an indirect way. For example, the event that an activity is launching leads to the onCreate method in that activity executing. We conclude such life-cycle event that can expose the execution trace of certain app in Table 1.

User-driven event User-driven events interact with corresponding Android API in a direct way. User regularly are guided to perform diverse actions on screen, such as tapping, dropping up and down, click, double-click, etc., to operate a new app. Right after capturing an action from users, Android system would react following the corresponding callback function that reflects the functionality and logic designed by developer. Generally, the user-driven event trigger the code execution of the one getting screen focus, which is hard capture by other apps. However, there still exists some user-driven events unrelated to the screen(e.g., shake action) acting as public events.

System event System event normally come from the situation a given system status changes. If not specified, system event could be obtained by any app in theory so long as one is granted proper permissions. Generally, the permission granting could hardly attract user's attention for system event such as battery status is insensitive to user. To the end, system event provides an indirect entry for adversary app getting parts of the running process of other apps that implement system-driven callbacks interfaces. Table1 lists parts of system event instances.

2.0.3 Callback and Components

Android platform is a kind of open source operating system established by Linux core. Developers could design and develop their own app product on Android in a flexible way, causing Android freely provides a variety of open source libraries and frameworks for developers. Of which, four types of components plays the key role in Android app's construction, namely *Activity*, *Service*, *Broadcast Receiver* and *Content Provider*. *Activity* is designed as a graphical interface for interacting with users. Nevertheless, *Service* is mainly used for handling complex computing job in background without a graphical user interface. *Broadcast Receiver* provides a global listener that could trigger specific code logic when receive relative message. *Content Provider* providers a universal interface for data storage and management, and for sharing data between different components. As a side-effect of user-driven mechanism, Android provides numerous of callback interfaces to be implemented by generic developers. Most of them rely on certain component, which we give greater details in follows.

Callback in Activity Most of callbacks in Activity are invoked by user-driven and life-cycle events, causing activity is mainly used for providing a visible graphic interfaces so as to interact with users. As aforementioned, upon the protection of sandbox, these callback statuses are hardly obtained by other apps only if the event triggered by user is irrelevant to screen focus. For instance, a shake action from users would lead to *sensor* related events, which have no especial association with the screen. Besides, some system-driven callbacks are implemented in activity for catering the requirements of functionalities.

Callback in Service Handling program logic and computing tasks in background, Service has no direct interaction with users, which is typically different from Activity. By this means, callbacks from service only come from life-cycle and system statuses. Yet, user's action without screen focus could be an exception, which we will discuss later.

Service life-cycle consists of five callback functions(onCreate, onStartCommand, onBind, onUnbind and onDestroy) but has a simpler life-cycle procedure compared with Activity. Being the same as Activity, these life-cycle callbacks are isolated from components of other apps so that has little utility value for attacker.

Regularly, most of *Services* are designed to wait for a typical signal and then invoke corresponding callback functions to push their computing result to an output interface, such as a *dialog* or an *Activity*. In general, the signal could be any type of event, invocation from other component or a program condition. If the trigger event belongs to system events, it can be easily obtained by component from other apps. This makes possible for attackers to perceive the execution process of a target app.

It is also possible to trigger a callback through a user-driven event in service, which is irrelevant to visible screen (e.g., shaking the phone to trigger SensorChanged event). Without specified receiver, this kind of events could be easily obtained by other apps either.

Callback in Broadcast Receiver The *Broadcast* has a wide spectrum of sources, from system or other components and can be delivered across all the installed apps through a common medium *Intent*. Generally, callback functions in Broadcast Receiver only act as a channel that responses the coming Broadcast and then launches corresponding *service* or *activity* to handle it. However, once lacking of strong constraint, Broadcast Receiver could suffer from the threats of repeatedly receiving trash message from an adversary Broadcast sender. Similarly, it is also possible that a typical Broadcast is intercepted by a deliberately designed Receiver. Therefore, developers normally add some protection measures, such as setting the "export" property as false or using self-defined permission, to prevent from such attacks.

Callback in Content Provider Content Provider acts as a channel for operating database in Android system, so callbacks in Content Provider appear quite trivial. Similar with Service, the Content Provider is designed as background component and its lifecycle callback onCreate() is mainly used to initialize the provider object. However, the provider is initiated when invoked by the operation from ContentResolver related to user-driven and program logic. Therefore, attackers are hard to leverage the process causing the launching time of a certain provider is hard to be captured by other apps. Besides, developers seldom set security related logics within onCreate() and other callbacks of Con-

Table 1: Parts of events and callbacks in Android

Categories	Event	Callback	Permission	Public
life-cycle	app starting	onCreate/onStart(launching activity); statical broadcast receiver	N/A	Y
	activity losing focus	onPause	N/A	N
	service starting	onCreate/onBind	N/A	Y
user-driven	clicking button	onClick	N/A	N
	shaking	onSensorChanged	N/A	Y
system	Intent.ACTION_BATTERY_LOW	onReceive	N/A	Y
	Intent.ACTION_BOOT_COMPLETED	onReceive	RECEIVE_BOOT_COMPLETED	Y

tent Provider. Based on above consideration, we exclude the Content Provider in later PEC discussion.

3. EXPLOITING CALLBACKS

Android framework provides a variety of callback functions for developers conducting their apps a particular responding after a specific event happens. Yet, abusing these callbacks would incur undesirable consequence as well.

3.1 Motivating Example

We take the well-known app Wechat as a motivating example. Wechat is known as a popular IM(Instant Messenger) tool for users sharing information and connecting with each other. Of diverse functionalities it provides, "Shake" is rather welcomed, which enables users to randomly obtain an online chatting partner so long as slightly shaking the device several times.

Although harmony in surface, there exists potential crisis in the user shaking action which can be easily perceived by any other app through the SENSOR_SERVICE, a system service for managing inner device sensor. That is to say, only with the SystemSensorManager object instantiation and the arrangement of shaking action size , an adversary app could clearly receive the same action event signal as Wechat. Further more, associating with the running status of Wechat through AMS(Activity Manager Service), adversary app could naturally make a judgement about the accurate time when the Wechat responding page would be popped up after user shaking. Although there exists an exception that user could shake the device in other page of "Wechat",the possibility of the exception's emergence is scarcely small in practice because user shakes device in such wide margin normally with intensive intention. Figure 1 shows how the "Shake" event is "stolen" by others.

It seems that the PEC threats is easily ignored by developers and app markets. We see the callback time leakage severely because it offers a big change for attackers. A wily attacker could devise diverse attacks utilizing the PEC threats. For instants, attackers could deliberately construct a similar result page that links to a malicious third-party chatting tool, or even mimic a fake chatting page to pretend to chat with user for malicious purpose. We presents greater details about the attacks in section **.

3.2 PEC Model

In order to clearly illustrate the callback threats, we view the threats process as a kind of callback state model. Each PEC within a installed app can be described as (S,D,E,C), where "S" denotes current state of app, which can be observed by other apps; "D" denotes the display-sink contained by the callback function; "E" denotes the event that invokes executing of the callback function; "C" denotes the proper control flow conditional values towards the "F".

Component-state.

(s belongs to S)The component-state refers to if the components of an app are running, which provides basic materials to PEC. Similar to the Event, the component-state here also needs to be exposed to other apps.

Before Android 5.0 emerging, the execution state of component like Activity and Service can be easily observed by the getRunningTasks API in ActivityManager object. A more detailed observation needs to request the "GET_TASK" permission, which could somehow attract the attention of most of normal users. However, Android starts to constraint the usage of such permission request and the usage of related API from recent Android 5.0+ for the security consideration. The getRunningTasks and even getRunningAppProcess(the API for get all the running process in the device) are now deprecated and only return developer's own application process. A substitute solution is to request new "PACKAGE_USAGE_STATS" permission, and to disable warning that permission is system level and will not be granted to third-party apps. Again, this approach needs extra permission request. A decent solving method to get the foreground running app without any permission request is introduced in [19] through reading the /proc file where Linux preserves all its process information. For all, obtaining the foreground app is entirely a completable task in common Android system version.

Another good news is that the usage of "getRunningServices" are still available in the newest version and need not request any extra permission. Thus, we can judge if a typical service is running through the "getRunningServices" API without any permission request.

In addition, statically-registered BroadcastReceiver always in the receiving state when the app is running, so its component-state is also easy to ensure. On the other side, the component-state of dynamically-registered BroadcastReceiver follows the running state of its host component, e.g., activity and service, and the method where it is registered.

Display-sink.

(d belongs to D)Display-sink is based on the conception of leakage detection, which refers to a set of label functions that can expose data or information to outside. However, differ-

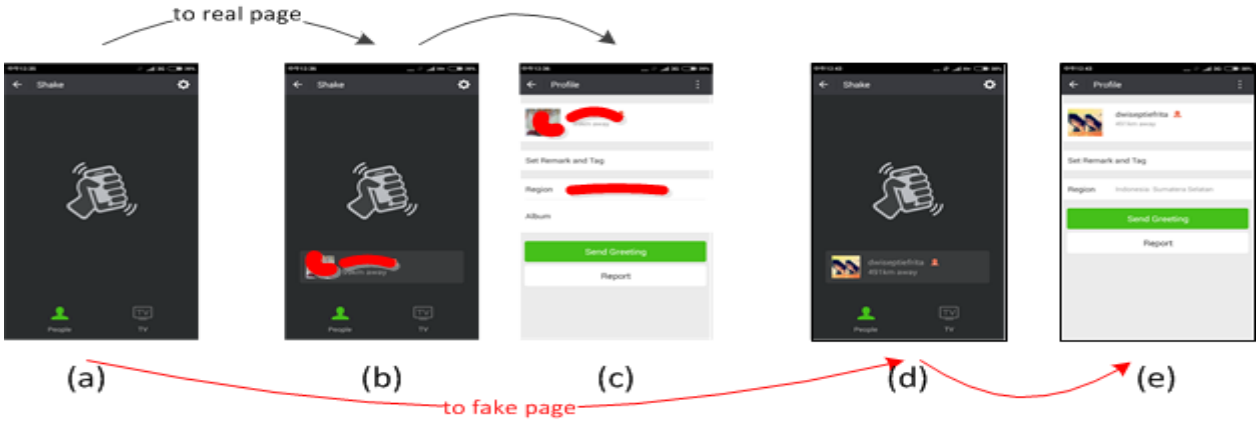

Figure 1: Motivating example about the PEC threats. In (a), a user prepare to try the "Shake"; (b) shows a normal case that a real result popped up after shake; (c) presents the further information of such real page after user clicks the result; (d) shows the case that the shake action is perceived by an adversary app, which pop up a fake "result" page after it receives the shake action; (e) presents the further fake information crafted by attacker, which is used to induce user to convince it.

ent from traditional conception, Display-sink here specifically refers to state output that could be perceived by users rather than information leakage, including UI page, dialog or notification, etc.

As to the Display-sink, only those output vectors which can be leveraged by attackers are considered. Most of which are related to UI activity, causing the UI is designed as the main channel to interact with users. Some of the UI vectors are mentioned in recent research[5](e.g., startActivity API, Toast message), yet we also collect some other UI related vectors, e.g., AlertDialog.Builder API, NotificationManager API. Generally, both of Dialog and Notification need response from users(e.g., click) and then engage corresponding reaction functionalities, which offers chance for attackers to interfere the users behaviours, which is insecure.

Dialog Generally speaking, it goes against Android design and UI guidelines that directly launching a Dialog from Service or BroadcastReceiver. Therefore, if Dialog is roughly created from a Service or BroadcastReceiver, there would pop up an error report causing Dialog is implemented relying on a particular Activity. However, it is pretty work that setting a system alert Dialog with the "SYSTEM_ALERT_WINDOW" permission grant. This kind of Dialog has global feature that can be invoked by a Service or a system event rather than an Activity.

Toast In common case, the Toast is also constructed based on an existing Activity rather than Service or BroadcastReceiver. The reason of that is one of Toast.makeToast parameters refers to the main UI context the one belongs to. Therefore, directly popping up from Service or BroadcastReceiver would not work causing the UI context is erroneously referred. However, a common-used tool class "Handler" can be used to insert the self-defined thread message into the main thread queue, so that the main thread UI context can be obtained by the Toast. Therefore, Toast can be launched not only in Activity but also in Service and BroadcastReceiver.

StartActivity The startActivity is known to be invoked by all the three components. However, some Activity is set self-defined permission to improve the security protection. For those low risk "protectionLevel", namely "normal" and "dangerous", we can still apply corresponding self-defined permission to access the target components. Nevertheless, for those high risk level, namely "signature" and "signatureOrSystem", currently there is no any effect way to access the target components by a third party app.

Notification The Notification can be launched by all the three components as well. Compare to Dialog and Toast, Notification is more likely used in service for notify the task progress or updated information.

Besides UI sink, some non-UI vectors are also contained, e.g., audio, vibrate related API, sendBroadcast, startService. Similar to UI related vectors, audio and vibrate related API also engage information output although as different information forms("voice and vibration"). The "sendBroadcast" and "startService" is able to lead new callbacks, which threatens the targeted app in a iterative way. Additionally, implicit intent loaded by "sendBroadcast" and "startService" also can be leveraged by attackers.

Event.

(e belongs to E) The event here refers to the public event since non-public one can hardly be exploited by other apps. As mentioned in section **, public events lie in all the three event categories(life-cycle, user-driven and system).

Control Flow.

(C)Control Flow is a collection of conditional values over the path from a typical State to a Display-sink. In theory, as for a callback threat vulnerable app, the "C" needs to guarantee an observable and feasible path that can reach the Display-sink. However, the "C" that strictly conforms above definition rarely emerges in real apps, so that we adopt a more flexible mechanism in practice which will be introduced in next section.

Attackers need to find a feasible path from the State to the Display-sink within the given operable source code of a target app. For judging the existing of such path, we first collect the condition set of the paths to the Display-sink. Then a signal used to refer to if it is visible for each value,

event and function emerging in the condition set is created. To describe more clearly, we denote the following concepts:

1. c: the condition value of each branch.
2. CS: the condition set collected from a program control flow path.
3. cv(a) : the condition value of variable a and a = variable | function | event
4. s(a): the signal value of variable a and a = variable | function | event, whose type is boolean.

s(a) represents if the variable a can be observed by components from other apps. s(a) = true represents yes, s(a) = false represents no.

5. CS = ∩ (cv(a) & s(a)) = (∩(c)) & (∩ s(a))

Above equation means the value of condition set of a particular path consists of two parts, 1. the value of each variable; 2. the signal value of it. A feasible path needs not only has a solvable condition set, but also observable for each variable. However, practically we normally use the second equation to compute the feasibility of a target path. First, computing out if it is feasible for each branch ; then , considering if it is able to be observed for each variable with the branch condition.

A traditional technique for above computation is symbolic execution. However, it needs unacceptable overhead in practice. To the end, we using an improved SMT approach to solve the constraint conditions. Normally there should be feasible from source to sink in a target app, otherwise there exists a designation shortage with in the target app. Under this prerequisite, we only need to consider s(a), namely if all the variables are observable.

Table 2 lists the main vulnerability vectors with the callback threats.

4. ATTACK

4.1 Exploiting Condition

App Description.

Timer.

4.2 Threat Model

In order to clearly illustrate the attack process, first the threat model need to be depicted. Suppose that we have a victim app installed in a given device, and it contains corresponding callback threats vulnerabilities. Besides, suppose that we have an adversary app installed in the same device and start its service running on the background stealthily. The goal of our adversary app is to engage diverse attacks through the victim app. In theory, the adversary app need not apply any permission when it is installed. However, in practice, it needs to apply some common-used permissions like "Internet" and insensitive permissions like "batter related permission" according to the attack type. The adversary needs to visually conceal its behaviours from users and also minimize its impact on the device performance.

The adversary is supposed having the capability of using public available resources and analysing them on the fly. Again, it should stay aware of the running states of apps and some particular services. As introduced in "Callback State Model", this capability can be obtained by particular methods within ActivityManager class.

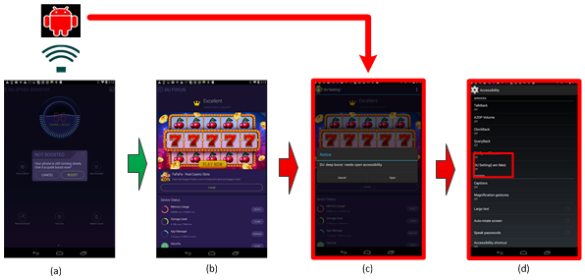

Figure 2: Example of Dialog Escalation with PEC threats

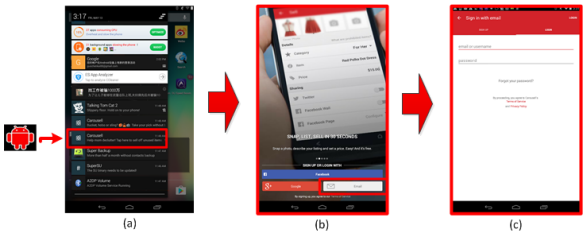

Figure 3: Example of Notification Forgery with PEC threats

4.3 Attack Overview

Following represents typical attacks forms that utilizes the PEC of victim app or PEC itself. According to the display-sink a PEC ends to, we representatively list several main types of attack examples, namely dialog & toast privilege escalation, notification forgery, shaking-display spoofing and activity phishing. Meanwhile, we also introduce other possible attack types.

4.4 Dialog & Toast Privilege Escalation

Dialog and Toast used to perform as the notice windows to notify users some intermediate results or alert information after apps successfully receive events. Once the received events are open to public, an adversary app would be easy to predict the time when a dialog or toast really emerges as long as the targeted app lacks of complex condition constrains in its control flow of the key callbacks. A common case comes from the notice related to those system resources like battery, memory, voice, etc. Normally, these kinds of notices are sent by pre-installed or system manager apps. Also, users are with fully trust for these apps in security part because they are professional and have numerous of users. However, this trust acts as a big opportunity that can be exploited to achieve the attack goal in attacker's eyes.

For example, when the memory rests as low as a given level, system manager apps, like DU Speed Booster, Avast Cleanup, etc., would pop up corresponding dialog with alarm information("Your phone is still running slowly. Give it a deep boost now?"), and if user clicks agree button, a new process launched by the manager app starts running. Then the user could be at ease with the memory trouble.

A typical implementation of above logic is represented as following:

The key point is the memory status(info.availMem) are public resources and can be obtained by any app installed in the device without any permission request. Moreover,

Table 2: Typical vectors of PEC

State	Display-sink	Event	Control Flow
1.Activity running state(Launching page)	1.startActivity	1.Broadcast from system: {Intent.ACTION_BATTERY_LOW}	1.branch or cycle condition analysis
2. Service running state	2. Toast message	2.Implicit Intent& pendingIntent	2.method invocation analysis
	3.AlertDialog. Builder	3.ServiceManager(need not permission grant): sensorManager, memoryManager	
	4.sendBroadcast (intent)	4.ServiceManager(need permission grant): batteryManager	
	5.AudioManager API 6.VibrateManager API 7.NotificationManager API		

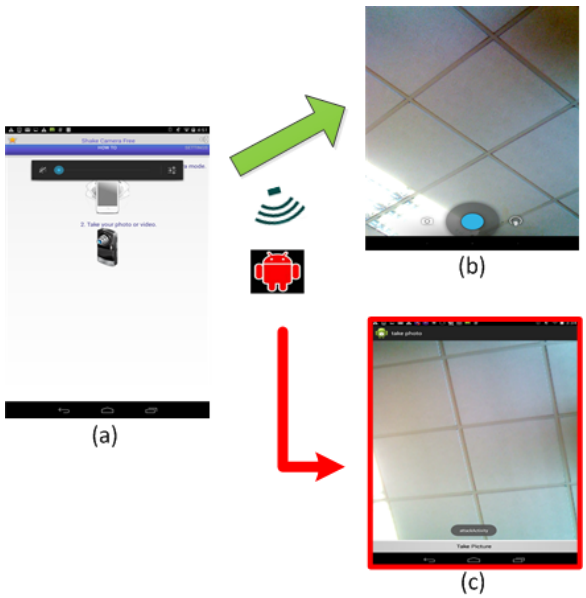

Figure 4: Example of Shaking Spoofing with PEC threats

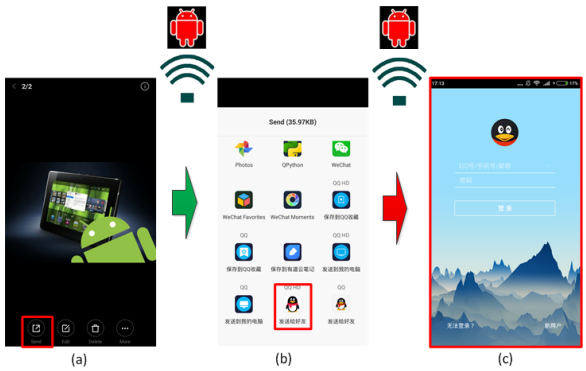

Figure 5: Example of Activity Phishing with PEC threats

```
private void displayBriefMemory() {
        final ActivityManager activityManager =
                (ActivityManager) getSystemService(ACTIVITY_SERVICE);
        ActivityManager.MemoryInfo info = new ActivityManager.MemoryInfo();
        activityManager.getMemoryInfo(info);
        if(info.availMem < threshold){
            AlertDialog.create()
    }
}
```

Figure 6: control-flow

the implementation control flow is designed so simple that the condition logic and variable "threshold" could be easily reconstructed in an adversary app. A experienced attacker could also obtain the these key logic and variables through reverse-engineering and binary analysis. As a result, the condition "info.availMem < threshold" as a public events makes the function "AlertDialog.create()" a PEC, and attackers are able to judge the creating time of the AlerDialog from its own reconstructed adversary app installed in the same device.

This time capture action offers a rich variety of opportunities to engage attacks escaping user's notice, of which a typical one is privilege escalation. Figure * presents an attack procedure for an attacker getting the "Accessibility Service" grant from user. The Accessibility Service is a special functionality offered to help disabled people to get the component status and handle the device operations more conveniently. Since Android 4.0, it supports richer interfaces for developers implementing their Accessibility functionalities like listening user actions. Undoubtedly, this functionality is so powerful that it is impossible to be ignored by attackers. However, all the implementations have to be engaged under the prerequisite that user needs to open the Accessibility Service in setting options for a specific app. It seems infeasible for attackers to get the grant directly. After all, it performs very suspicious if such a setting window suddenly opens on the foreground against user's intention.

By exploiting the PEC of such a trusted app, all the attack procedure would become more reasonable. First, the adversary app opens a service on the background to monitor if the PEC service is running through the getRunningServices() offered by the ActivityManager. Fortunately, the invoking of getRunningServices() needs not any permission

request and has not been constrained like the getRunningTasks()(mentioned in Section *) so far. Then, the adversary app construct the same logic as trusted app to wait for the time when the available memory size falls below the given threshold. Once it happens, the adversary app would predict the notice dialog will pop up in a few seconds(Figure *(a)). After that, it shows a dialog tip(Figure *(c)) to tell user the corresponding Accessibility Service have to be grant to complete the data clean task, and then opens the Accessibility Service setting page using "Setting.ACTION_ACCESSIBILITY_SETTINGS" intent like Figure *(d) did. In the setting page, an attack service with a confused name "DU deep boost" makes user fool about this attack. At last, the adversary app gets the grant and could engage further attacks.

4.5 Notification Forgery

From perspective of practise, the Notification works for showing the received message, notifying the updated information and representing the progress of running tasks, which is somehow similar as Dialog or Toast. Different from the Dialog, it is not necessary to handle a Notification immediately. Again, a Notification would not disappear in seconds as the Toast does until user handles it. The user could handle it at any time they like, which gives attackers enough time to forge a fake one escaping user's attention. Here we introduce two common used situations.

1. Push mechanism. Most of the market apps currently are based on CS(Client-Server) design, and the server needs to frequently push the updated data or content to the client in user's device. The client keeps running a service in the background receiving the update from server. Normally, this push-receive mechanism is implemented by specialized module with high-level security protection that makes it hard to get any status from the push process by other apps.

However, causing the status of client service for receiving is publicly opened, attacker also could exploit it to fool the user in an indirect way. For example,the popular goods trading platform app Carousell frequently uses the Notification to notify users to check its updated backup data, with services like PendingIntentCallbackService running on the background. The running status of these services could be easily observed by a third-party adversary app. Further more, the active Notification status also performs as a public event, which can be obtained by getActiveNotifications API offered by the NotificationManager. If the active Notification status contains one from Carousell, adversary app ensures that setting a Carousell notification wouldn't raise user's suspicions. At this time, a fake Notification would be constructed using the same resources and framework as original Carousell did(Figure * (a)). The difference is the fake Notification are connected to a carefully constructed "Carousell login" page that needs user input their Carousell account(even the account of third-party single sign-on Facebook and Google)(Figure * (b)&(c)). When the user open the Notification panel, there are several very similar Notifications listing there(as Figure * (a) shows). If the user chooses the fake one and tries to log in the Carousell account, unfortunately, the username and password will be delivered to attacker.

Notice. Another important usage of the Notification is to notify users update status from a specific app or the device system. For instance, the QPtyhon(An app for python debugging on Android) would perform a log Notification when the onDestroy() lifecycle function in the MainActivity happens. However, since the onDestroy() is invoked when the MainActivity is destroyed, a third-party app uses the getRunningAppProcesses()(offered by ActivityManager) function to judge it. Thus, the onDestroy() belongs to the PEC as well. Attackers could capture the time of such Notification emergence and even the time user triggers the Notification(when the targeted Notification is dismissed and corresponding app starts running.)by analysis from getActiveNotifications() offered by NotificationManager(after Android 4.3)

4.6 Shaking-display Spoofing

Shaking-display refers to the display(Dialog, Toast, Notification and Activity) led by a shaking action. A typical case comes from the motivation example we introduced in section 3.1**. The shaking action could be captured by the sensor in-set in the device. App developers normally use the sensorManager listener and its corresponding callbacks onSensorChanged() and onAccuracyChanged() to listen and handle the shaking action. Normally, the value of SensorEvent(parameter type of sensor callbacks) should be large enough to avoid the misrecognition of unexpected action without user's participating.

The shaking-display spoofing attack is based on the assumption that the shaking action always aims at the foreground activity. Since the shaking action always fits user's intention and in general cases, users only concern the foreground activity, the assumption is reasonable.

Consider the motivating example the Shake page pops up under two basic conditions. One is the WeChat is running on the foreground, and the other is shaking action event happens. As the assumption mentioned before, here we don't consider extreme situation that a device shaking happens accidentally in other page like the chatting page or setting page. Fortunately, both of the two conditions could be observed by public even without any of the permission request.

Then let's consider another shaking-display spoofing example with the targeted app named "Shake Camera Free". This app implements a novel usage that users could conveniently open the camera only needing slightly shaking the device several times. As analysis before, the handling function for responding the shaking event acts as a PEC. Therefore, an adversary app could capture when the camera is popped up by "Shake Camera Free". At that time, a fake camera is launched by the adversary background service and covers on the face of original benign camera. The user is hard to distinguish the two camera surfaces and he/she is very likely to choose the fake one because it is on the foreground. Then adversary could stealthily deliver the picture to the web or store it in a secret location for other usage. The entire attack procedure is hard perceived by user. The only challenge that the adversary app has to overcome is to request the "android.permission.CAMERA" permission. After all, the camera permission is not as sensitive as location or phone status for a user, so it is not hard for an attacker to get this non-sensitive permission.

4.7 Activity Phishing

Activity Phishing represents a more direct way to conduct the attack by the PEC. A traditional common used public event is the launching event of targeted app whose MainActivity contains private information input, e.g. login page and

sign up page. A background adversary service keeps monitoring if the targeted app process begins running. Once the targeted page is launched, a forgery page with the same outlook would cover on the top surface and phish user's private data.

Here, we introduce a more complex PEC example related to implicit Intent. The implicit Intent is known as a bridge to connect another component without specifying a single receiver. Instead, it limits a range of receivers through making a operation statement including the intent data and the intended action. If an app sends an implicit Intent request and there are more than one apps fitting the Intent statement, the Android system would pop up a system dialog listing all the fitting apps. Then the user could choose one of them to handle the Intent data.

There exists a sort of iterative PECs threats within the implicit Intent procedure. We still take the Gallery as an example. The Gallery providers a share feature that allows users to conveniently share the preserved picture for other chatters. When the user long-presses the picture, a system dialog listing all the apps that can handle this picture pops up. Supposing that the user chooses the QQ, which means the user wants to send this picture to one of his/her QQ friends. As a result, the QQ login page pops up and the user could complete his/her intent after logging in the QQ.

As the analysis before, the running status of foreground app acts as a public event, so that the intent sender("com.flayvr.flayvr" in this example) could be observed by an adversary app. Actually, the event that pops up the intent receiver QQ("com.tencent.minihd.qq") is also a public event and could be observed. Moreover, when the system dialog emerges for listing the receivers, the foreground process infers to the system dialog and the observable process name changes to "system:ui", which indicates the system dialog emerging is a public event as well. Therefore, there exists a PEC chain that contains sender app process, system dialog process and receiver process in sequence. By capturing the PEC chain, the adversary app could predict the emerging time of QQ login page and cover a phishing login page on the authentic one. Consequently, the adversary app could successfully bypass the unrelated pages(e.g., introduction page) and phish the login user name and password in an accurate way.

4.8 Other Attacks

The attacks discussed before are just typical attack types. There are still other attacks could be conducted using the PEC features. For instance, the Notification PEC also could be used to conduct the privilege escalation, only needing a reasonable spoofing page emerging when the user clicks the Notification icon. Again, shaking action could also be used to do phishing if the display is a sensitive login page. In addition, attackers even could try more flagrant attack forms, like ransoming user's critical data and ask for something they are interested for. To achieve this goal, attackers need to design a mal-block and pop it up once the targeted PEC happens. Actually, the attack is designed relying on the PEC type contained by victim app.

Another typical type of attacks follows a more direct and traditional way that utilizing PEC itself rather than a victim app. An adversary app could perform a customized response in terms of a targeted event. For example, adversary apps could be designed as a kind of spoofing pages, which would be popped up right after a particular event happens. The spoofing page content should be closely related to such event so that users have no doubt about it. Another example is to craft a fake phishing page for login that will pop up when the real app opens(the launching page is such a login page).

Table ** lists the possible associations of attack and the PEC.

5. DETECTION AND EVALUATION

This section we first analyse the detection details for the PEC and then provider an evaluation on a large scale of market apps.

5.1 The PEC Detection

To measure the PEC threats in real world apps, we designed a prototype tool CSDroid for detecting and analysing the PEC contained by market apps, which is implemented with 4k lines python code and based on the static analysing framework Androguard[11].

Our analysis includes four steps. First, we match the risk display with the source code of detected apps. We collect four types of risk display: Dialog, Toast, Notificaiton, and Activity, whose function features are listed in Table **.

Then, the CSDroid would recognize the PEC vectors within the source code, including some of lifecycle callbacks of Activity & Service, the callbacks of statically registered BroadcastReceiver and the callbacks defined as observable, e.g., onAccuracyChanged and onSensorChanged.

In the third step, the CSDroid backward traverses feasible method paths from risk displays to the PEC vectors. Generally, the callee methods invoked by the PECs are treated as observable, since these methods would be executed right after the execution of corresponding PECS according to control flow.

However, there are two challenges needed to overcome. First, it is inevitable that some apps contain iterative such method traces. For example, a background service is waiting for a public event and then would launch a new activity. Similarly, in the new activity, the execution of onCreate() lead to a notice dialog poping up. This case exists an iterative method path that the PEC of service first transmits to the activity, and then the activity PEC transmits to the dialog. We use redirecting PEC to handle this iterative cases. Considering the joints of an iterative path are always code-block for activity launching(code-block of method startActivity), such as the case introduced above, we model four types of intents used to launch a new activity and flag the target activity name within the intent. After that, in the rest of the method traces, if there is a PEC activity name being the same as the flagged one, the CSDroid associates the two methods, and as a result, the two method traces(one is ended by the startActivity referring to the flagged activity; the other is started by the flagged activity) are integrated as an iterative one.

The second challenge is that not all the extracted PEC method traces are observable for other apps. On the one hand, the adversary app is supposed to be installed without sensitive permission grant. In this way, the PEC path with a system API that needs permission grant is hard to be observed to an adversary app. On the other hand, features involving private resources of the targeted app are also forbidden to observe. For instance, the statical-registered BroadcastReceiver is regarded as a PEC. Yet, an explicit intent that specifies the single receiver's name, received by

the objective BroadcastReceiver, could not be observed by other app at all.

The CSDroid analysing such violation situations in a fine-grain way. To solve the permission API problem, considering that the key impact of such a permission grant API is able to confuse the direction of control flow, CSDroid focuses on handling the branch conditions containing such API, with three steps. First, to a method node N whose predecessor method node is denoted as M, CSDroid tells accurate position P in the body of M where N is invoked. Then, from position P, CSDroid conduct a backward traversing to detect whether P potentially depends on a branch condition(we denote such branch condition as B if it exists). If such branch condition exists, CSDroid would recognize whether it depends on a permission grant API through dependency flow analysis. Our permission grant API database comes from the project of PScout[4] which offers a complete mapping list for Android system permission and API. If a permission dependency branch condition is recognized, the CSDroid would remove it from existing method traces.

Compared to the permission API problem, CSDroid just easily build a whitelist to exclude such method traces with any of private resources feature.

handling of inner class

5.2 Market-scale Study

Our analysis is targeted on 2,723 real-world apps crawled by Google Play in 2016, covering all the 27 categories. The final detection results are listed in Table **.

5.3 Vulnerability Discuss

6. MIGRATION STRATEGY

The conditions that enable the attacks successful come from three parts. First, Android system allows the existence of such PECs (metioned in section **). Second, app developers ignore the PEC problem during their developing process. Third, users have little awareness about such PEC attacks. The violation of any condition would make the attacks fail. Therefore, to migration work should start with above three parts.

System Level. The observation to activity status is known to be prevented from other task stacks in Android 5 and newer versions. It brings substantial benefits for protection of app status from being stolen during runtime. However, the status of components like service, launching activity and broadcastreceiver still can be observed or predicted. Besides that, it is free to access some insensitive-in-surface system resources such as the sensor data. Again, system events can be received by more than one entities at the same time. All above system design makes the PEC attack possible. Therefore, Android system could conduct effective improvement aiming to above features. Compared with changing the event receiving mechanism which involves entire design of Android framework, adding reasonable access to obtain particular resources or observe specific components appears more feasible. However, it is a trade-off between security threats and existing functionalities after all. We expect Android has a concrete improvement for this concern in later versions.

Application Level. From our market survey of the PEC threats, numbers of real-world apps leave the possibility of being attacked by malware through such threats. One clear way to address such problem is to provide a reasonable guideline for developers to avoid this threats during their developing process. Some safety developing habits would be intensively suggested, such as don't set the log-in page in the launching activity, don't link a service to a sensitive activity or dialog, don't pop up a sensitive window right after a system event happens, etc. The developers should also clearly consider related factors based on such tradeoff between security and functionality before developing. However, it seems impossible for a vendor to constrain it's app products as such suggests, which appears overstrict for developers since only a small part of such "unqualified products" suffer from real attacks in practice. To address this challenge, users are expected to be presented a notification tip asking them to whether approve or not when a new display is built after a certain public event. Such notification guides users to be aware of the PEC threats, which is what we strive to in the next stage.

7. RELATED WORK

Android Event-driven Analysis Based on event-driven mechanism, Android application adopts a much flexible execution way, which makes it hard to be directly analysed like the one on conventional personal computer. Researchers has done much efforts on tackling this challenge. [39] fundamentally proposes a program representation built by context-sensitive static analysis of callback methods to model the app's GUI. [21] provides an event-driven analysis manner to measure the power dissipation. [8] centres around so-called Permission Event Graph built with static analysis and uses model checking to illegal interaction between an application and the Android event system. Again, [20] handle event-driven analysis to reach challenging source code for application testing. [27] presents Dynodroid to monitor the reaction of an app upon each event either from system or user. [16]presents a race detection tool named CAFA for event-driven mobile systems. Although prior works present different insights of event-driven analysis in Android system, there are few efforts concerning the security impact raised by public event abuse. Our work fills this gap via systematic study of Android system design and statical analysis for a large scale of real-world apps.

Android Defects Exploitation The exploration and exploitation of weaknesses within either Android system or apps are intensively focused by current related researchers. The study of memory side-channel analysis and exploitation[42, 9, 18]uncovers an indirect attack way aiming to the weak architecture of Android public data exposure. Several recent researches focus on the vulnerabilities raised by Android advertising libraries[36][35]. Besides that, [40] proposes data residue problem during app uninstallation. As a direct interaction channel with users, the Android UI also suffers from severity security threats, which is carefully discussed in [33][5][1][26][17][34][24]. Again,in the work of [29][38][28], the sensor data could also be regarded as a sort of attack resources predicting some isolated private information including user input,fingerprints. In addtion, [2] introduces a denial-of-service attack approach utilizing the AMS design defect. [37] and [6] present detailed studies about privilege escalation for malware through defects of Android os. As to Android permission mechanism, [14][4][41][13] reveal security threats arised by the abuse of permission requests and propose corresponding migration strategy. In contrast to ex-

isting work, this paper represents the impact of public event defect of Android that affects a much wider range of system components and services. The attacks relying on such public event callbacks are firstly studied systematically.

Android Vulnerability Detection As a critical security concerning on Android system, privacy data leakage incurs widely related research efforts. Of all the detection works, dynamic taint tracking [12][22][32][31] receives much eyesights for its sound accuracy. Another important complement for dynamic approach relies on static analysis[25][3][15], which can effectively reduce the false negative rate. Further works concern the impacts of ICC(inter-component communication) mechanism[7][30][23]. In addition, [10] measures implicit leak of user password using symbolic execution. Our work is devoted to uncover the public event threats within real market app, which can be easily exploited by attackers to build a wide spectrum of new malwares.

8. CONCLUSIONS

9. ACKNOWLEDGMENTS

10. ADDITIONAL AUTHORS

11. REFERENCES

[1] D. Akhawe, W. He, Z. Li, R. Moazzezi, and D. Song. Clickjacking revisited: a perceptual view of ui security. In *8th USENIX Workshop on Offensive Technologies (WOOT 14)*, 2014.

[2] A. Armando, A. Merlo, M. Migliardi, and L. Verderame. Would you mind forking this process? a denial of service attack on android (and some countermeasures). In *Information Security and Privacy Research*, pages 13–24. Springer, 2012.

[3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN Notices*, volume 49, pages 259–269. ACM, 2014.

[4] 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*, pages 217–228. ACM, 2012.

[5] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna. What the app is that? deception and countermeasures in the android user interface. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 931–948. IEEE, 2015.

[6] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on android. In *NDSS*, 2012.

[7] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen. Edgeminer: Automatically detecting implicit control flow transitions through the android framework. In *NDSS*, 2015.

[8] K. Z. Chen, N. M. Johnson, V. D'Silva, S. Dai, K. MacNamara, T. R. Magrino, E. X. Wu, M. Rinard, and D. X. Song. Contextual policy enforcement in android applications with permission event graphs. In *NDSS*, 2013.

[9] Q. A. Chen, Z. Qian, and Z. M. Mao. Peeking into your app without actually seeing it: Ui state inference and novel android attacks. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 1037–1052, 2014.

[10] L. P. Cox, P. Gilbert, G. Lawler, V. Pistol, A. Razeen, B. Wu, and S. Cheemalapati. Spandex: Secure password tracking for android. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 481–494, 2014.

[11] A. Desnos. Androguard: Reverse engineering, malware and goodware analysis of android applications... and more (ninja!).

[12] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.

[13] Z. Fang, W. Han, and Y. Li. Permission based android security: Issues and countermeasures. *computers & security*, 43:205–218, 2014.

[14] 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*, pages 627–638. ACM, 2011.

[15] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard. Information flow analysis of android applications in droidsafe. In *NDSS*, 2015.

[16] C.-H. Hsiao, J. Yu, S. Narayanasamy, Z. Kong, C. L. Pereira, G. A. Pokam, P. M. Chen, and J. Flinn. Race detection for event-driven mobile applications. In *ACM SIGPLAN Notices*, volume 49, pages 326–336. ACM, 2014.

[17] L.-S. Huang, A. Moshchuk, H. J. Wang, S. Schecter, and C. Jackson. Clickjacking: attacks and defenses. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 413–428, 2012.

[18] S. Jana and V. Shmatikov. Memento: Learning secrets from process footprints. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 143–157. IEEE, 2012.

[19] jaredrummler. Androidprocesses. https://github.com/jaredrummler/AndroidProcesses.

[20] C. S. Jensen, M. R. Prasad, and A. Møller. Automated testing with targeted event sequence generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 67–77. ACM, 2013.

[21] K. Kim, D. Shin, Q. Xie, Y. Wang, M. Pedram, and N. Chang. Fepma: Fine-grained event-driven power meter for android smartphones based on device driver layer event monitoring. In *Proceedings of the conference on Design, Automation & Test in Europe*, page 367. European Design and Automation Association, 2014.

[22] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer. Android taint flow analysis for app sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, pages 1–6. ACM, 2014.

[23] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 280–291. IEEE Press, 2015.

[24] C.-C. Lin, H. Li, X.-y. Zhou, and X. Wang. Screenmilker: How to milk your android screen for secrets. In *NDSS*, 2014.

[25] 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*, pages 229–240. ACM, 2012.

[26] T. Luo, X. Jin, A. Ananthanarayanan, and W. Du. Touchjacking attacks on web in android, ios, and windows phone. In *Foundations and Practice of Security*, pages 227–243. Springer, 2012.

[27] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 224–234. ACM, 2013.

[28] E. Miluzzo, A. Varshavsky, S. Balakrishnan, and R. R. Choudhury. Tapprints: your finger taps have fingerprints. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 323–336. ACM, 2012.

[29] A. Mylonas, V. Meletiadis, L. Mitrou, and D. Gritzalis. Smartphone sensor data as digital evidence. *Computers & Security*, 38:51–75, 2013.

[30] 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. *Effective Inter-Component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis*, 2013.

[31] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *NDSS*, volume 14, pages 23–26, 2014.

[32] V. Rastogi, Y. Chen, and W. Enck. Appsplayground: automatic security analysis of smartphone applications. In *Proceedings of the third ACM conference on Data and application security and privacy*, pages 209–220. ACM, 2013.

[33] C. Ren, Y. Zhang, H. Xue, T. Wei, and P. Liu. Towards discovering and understanding task hijacking in android. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 945–959, 2015.

[34] F. Roesner and T. Kohno. Securing embedded user interfaces: Android and beyond. In *USENIX Security*, pages 97–112, 2013.

[35] S. Sooel, K. Daehyeok, and S. Vitaly. What mobile ads know about mobile users. In *NDSS*, 2016.

[36] D. Soteris, M. Whitney, Y. Wei, Z. Aston, and A. Carl. Free for all! assessing user data exposure to advertising libraries on android. In *NDSS*, 2016.

[37] L. Xing, X. Pan, R. Wang, K. Yuan, and X. Wang. Upgrading your android, elevating my malware: Privilege escalation through mobile os updating. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 393–408. IEEE, 2014.

[38] Z. Xu, K. Bai, and S. Zhu. Taplogger: Inferring user inputs on smartphone touchscreens using on-board motion sensors. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, pages 113–124. ACM, 2012.

[39] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev. Static control-flow analysis of user-driven callbacks in android applications. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 89–99. IEEE Press, 2015.

[40] X. Zhang, K. Ying, Y. Aafer, Z. Qiu, and W. Du. Life after app uninstallation: Are the data still alive? data residue attacks on android. In *NDSS*, 2016.

[41] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 611–622. ACM, 2013.

[42] X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt. Identity, location, disease and more: Inferring your secrets from android public resources. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1017–1028. ACM, 2013.

APPENDIX

A. HEADINGS IN APPENDICES

A.1 References

B. MORE HELP FOR THE HARDY