Security Threats Caused by Public Event Callback in Android Application

Chenkai Guo¹, Jingwen Zhu²*, Xiaoyu Yan¹ and Yan Li¹

¹ College of Computer Science, Nankai University, Tianjin, 300350, China
² College of Software, Nankai University, Tianjin, 300350, China
*Corresponding author’s e-mail: zhujingwennk@163.com

Abstract. The feature of event-driven acts as a key role that makes Android application differentiate from traditional PC software. Since 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 application and further, which could offer important basic resources to predict specific behaviours of targeted application. In this paper, we aim to analyse potential security threats inside them and demonstrate typical kinds of proof-of-concept attack examples. Apart from that, the critical mechanism-public event callback (PEC) that may cause the threat is firstly modelled and studied, where its four main parts are introduced in detail.

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 [1-4]. That is to say, with different events, the program control flow would change correspondingly. Thus, callback 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 [5,6]. 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 [7,8]. 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. 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 Activity transition
signal is quite weak in [9]. 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.

Compared to conventional side-channel attacks, we introduce a new type of potential threats called
public event callback (PEC) threats in this paper, 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. The PECs and related events are systematically studied in this paper.

2. Motivation
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.
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.

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.

3. State Model of Public State Event

In order to clearly illustrate the callback threats, we view the threats process as a kind of callback state model. Each PEC state model within an installed app contains four typical parts: “current state” of app, which can be observed by other apps; “final sink” contained by the callback function; “event” that invokes executing of the callback function; the proper “control flow” conditional values towards the final sink.

3.1. State

The State refers to current program state of an app, where the running location can be inferred. Similar to the Event, the 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 method in ActivityManager object [10]. A more detailed observation needs to request the "GET TASK" permission, which could somehow attract the attention of most of normal users. Unfortunately, Android starts to constraint the usage of this permission from recent Android 5.0 for the security consideration. The "REAL GET TASK" permission is only available to system apps; while normal third-party app is forbidden to request. 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.

3.2. Display-sink

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, different 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 [11]. Some of the UI vectors are mentioned in recent research(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 frame and Notification need response from users(e.g., click) and then engage corresponding reaction functionalities. This process offers chance for attackers to interfere the users behaviours, which is insecure. Besides, some non-UI vectors are also contained, e.g., audio, vibrate related API, sendBroadcast. Similar to UI related vectors, audio and vibrate related API also engage information output although as different information forms("voice and vibration"). The “SendBroadcast” is able to deliver information between different apps, breaking the constraint of sandbox. Therefore, the app that receives the broadcast could be operated by others.

3.2.1. For Dialog. Generally speaking, it goes against Android design and UI guidelines that directly launching a Dialog from Service. Therefore, if AlertDialog is roughly created from a Service there would pop up an error report causing Dialog is implemented relying on a particular Activity. However, still it would 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.

3.2.2. For Toast. Similar to Dialog, the Toast, in general, also should be constructed based on an existing Activity. The reason of that is one of the parameters needed by Toast function refers to the main UI context the one belongs to. Therefore, directly popping up from Service would not work causing the UI context is erroneously referred. In principle, the communication between main thread and sub thread mainly relies on the message queue and message cycle in Android, which follows the communication mechanism in traditional Windows.

3.2.3. For StartActivity. Some Activity or Service 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. This kind of cases also need to be taken enough attentions in real attacks.

3.3. Event
The event here refers to those can be observed by public. For example, normally user-click event does not belong to the set, causing it generally only can be preserved by local app components; other app is forbidden to obtain this event due to the isolation of sandbox. Following introduction aforementioned, Event here contains three categories.

1) Those Events originated from system are loaded by either Broadcast or serviceManagers. For serviceManagers ones, they normally need corresponding low-risk permission request(e.g., "android.permission.BATTERY STATS", "android.permission.VIBRATE ") to manipulate the event information. However, system broadcast need not any permission, leading to more threats.

2) Those Events originated from other apps are loaded by implicit Intent or pendingIntent (contains implicit Intent). For explicit Intent, the destination is clearly defined as a typical app so that it is impossible for other app to obtain this kind of Intents. However, implicit Intent and pendingIntent could be sent to a set of apps so as long they match the Intent features.

3) Those Events originated from users are loaded by serviceManager. These events generally are equipped with global characteristics that impacts not only single app. For example, user click event could only affect the activity that occupies focus; while sensor change event would impact all the apps installed. Some of serviceManagers like sensorManager,audioManager need not permission request.

3.4. Program Control Flow
Control Flow is a collection of conditional values over the path from a typical State to a Final Sink. Attackers need to find a feasible path from the State to the Final 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 Final 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 j function j 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).

A feasible path needs not only has a solvable condition set, but also observable for each variable. However, practically the second equation can be normally used 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.

4. Empirical Experiment
To verify the proposed PEC model in real world, we have download 113 applications from a common-used open source malware dataset AMD [12]. The open-source nature of the downloaded apps enables us easy to obtain their code structures, and analyse the callback, display-sink and control flow. From the investigated apps, we check 25 apps with the side-channel threats in manual, which indicates the ubiquitousness of that treats. Apart from that, the attack for the investigated apps covers all the four parts of PEC mode. The detailed result is shown as Table 1.

| Display-sink Type | Confirmed Number |
|-------------------|------------------|
| Dialog            | 12               |
| Toast             | 3                |
| Start-Activity    | 10               |
| Total Threats     | 25               |
| Non Threats       | 88               |

Among the checked threats, the dialog and start-activity type of display-sink account for 88%, which serves as dominant display-sink types within the threats. The main reason lies in that the complexity and scale of the two types are evidently larger than the Toast, which results in more potential vulnerabilities within the two types.

5. Conclusion
In this paper, we aims to a novel side-channel security threat in Android application. The threat is caused by the public event callback mechanism, where the public event can be detected by multiple applications with the device. We raise a typical example to demonstrate the threat, as well as analyse the essential model running behind such threat. The four parts of factors that closely related to the public event callback are systematically studied.
Acknowledgments
The work is supported by the Interdisciplinary Project of Nankai University 610-ZB195008, Industry-University Cooperation and Education Project of Chinese Ministry of Education F1030511, Tianjin Natural Science Foundation (Grant No. 19JCQNJC00300), and Fundamental Research Funds for the Central Universities of Nankai University (Grant No. 63191402).

References
[1] 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). USA.
[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. pp.13-24.
[3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. (2014)Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In ACM SIGPLAN Notices, 49: 259-269.
[4] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. (2012)Pscout: analyzing the android permission specification. In Proceedings of the 2012 ACM conference on Computer and communications security. pp.217-228.
[5] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna. (2015)What the app is that? deception and countermeasures in the android user interface. In Security and Privacy (SP), 2015 IEEE Symposium on. pp.931-948.
[6] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. (2012)Towards taming privilege-escalation attacks on android. In NDSS. San Diego.
[7] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen. Edgemin: (2015)Automatically detecting implicit control flow transitions through the android framework. In NDSS. San Diego.
[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. (2013)Contextual policy enforcement in android applications with permission event graphs. In NDSS. San Diego.
[9] Q. A. Chen, Z. Qian, and Z. M. Mao. (2014)Peeking into your app without actually seeing it: Ui state inference and novel android attacks. In 23rd USENIX Security Symposium (USENIX Security 14). pp. 1037-1052.
[10] L. P. Cox, P. Gilbert, G. Lawler, V. Pistol, A. Razeen, B. Wu, and S. Cheemalapati. (2014)Spandex: Secure password tracking for android. In 23rd USENIX Security Symposium (USENIX Security 14). pp. 481-494.
[11] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. (2014)Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Transactions on Computer Systems (TOCS). 32:2-5.
[12] Android Malware Dataset.(2019). http://amd.arguslab.org/.