2017 IEEE Symposium on Privacy-Aware Computing

Privilege Leakage and Information Stealing through the Android Task Mechanism

Yinhao Xiao*, Guangdong Bai¶, Jian Mao‡, Zhenkai Liang† and Wei Cheng§
*Department of Computer Science, The George Washington University
¶Infocomm Technology Cluster, Singapore Institute of Technology
†School of Computing, National University of Singapore
‡School of Electronic and Information Engineering, Beihang University
§Department of Computer Science, Virginia Commonwealth University

***Abstract*—To facilitate apps to collaborate in finish complex jobs, Android allows isolated apps to communicate through explicit interfaces. However, the communication mechanisms often give additional privilege to apps, which can be exploited by attackers. The Android Task Structure is a widely-used mechanism to facilitate apps' collaboration. Recent research has identified attacks to the mechanism, allowing attackers to spoof UIs in Android. In this paper, we present an analysis on the security of Android task structure. In particular, we analyze the system/app conditions that can cause the task mechanism to leak privilege. Furthermore, we identify new end-to-end attacks that enable attackers to *actively* interfere with victim apps to steal sensitive information. Based on our findings, we also develop a task interference checking app for exploits to the Android task structure.**

I. Introduction

Android has dominated the smartphone market with a market share of 82.8% [1], with over 1,900,000 applications (apps) available on Google Play as of the first quarter of 2016 [2]. As smartphones are deeply integrated into our daily life, it is becoming an information and communication hub for tasks including communication, planning, banking, and health care. As a result, securing the apps and data on smartphones against malicious activities has become a top priority.

The Android system's security is based on several layers of security mechanisms. In particular, each app is assigned a set of permissions, and is only allowed to access system resources and services within the permissions given. In addition, to prevent apps from accessing information of others, each app is confined into its own partition. It is enforced using the process isolation and user-based protection mechanisms provided by Linux, where each app is assigned to a unique Linux user ID.

Strong isolation increases the bar for attackers to carry out malicious activities, but it also hinders benign apps from communicating and collaborating with one another. To facilitate apps collaboration in a complex task, Android allows isolated apps to communicate through explicit interfaces, such as the Intent mechanism. For example, Instagram uses intents to access the Single Sign-On (SSO) service of Facebook to authenticate users. Furthermore, Android provides the Android Task Structure mechanism to allow activities from different apps to be seamlessly integrated into a task, giving them the convenience when accessing common information. For example, when Instagram uses the Facebook API for the authentication service provided by Facebook, users can navigate through activities Instagram app and Facebook app as if they are the same app.

Though the mechanisms are designed for facilitating app communication and collaboration, relaxing the isolation provided by the Android system often causes over-permissive privilege to apps. As the task mechanism of Android is developed to facilitate inter-app collaboration, apps in a task may get additional privilege beyond what is allowed by the isolation-based Android security mechanism. Demonstrated by recent exploits [3], a malicious app can hijack the task mechanism for attacks such as spoofing and phishing. The privilege obtained by apps in the same task is well beyond that for collaboration, effectively making the Android task mechanism a form of authorization.

In this paper, we conduct an analysis of the security of the Android task mechanism. First, we analyze possible ways that an app can join a task and the privilege "leaked" to other apps in the same task. Specifically, to explore the ways Android controls tasks, we dynamically probe possible combinations of the flags and system states that can affect an app's task status. We also analyze the additional privilege that can be obtained by an app when it joins a task.

Second, built on the understanding of the task control mechanism and task privileges, we identify end-to-end attacks that steal information from other apps. In particular, we identified four proof-of-concept attacks based on exploiting the task mechanisms. The attacks include UI phishing, screenshot based password stealing, man-in-the-middle activity, and gallery stealing. All of them only require common permissions, e.g., INTERNET and READ_EXTERNAL_STROAGE. Compared to the attacks by existing exloits, we have identified new attack mechanisms that can *actively* interfere with benign apps. The short video demos can be found in [4].

Finally, to prevent attackers from misusing the task mechanism, we develop an efficient scanner that can help users to identify the risks related to Android tasks.

Contributions. In summary, we made the following contributions in this paper:

- *Identification of novel attacks exploiting the Android task*

978-1-5386-1027-5/17 $31.00 © 2017 IEEE
DOI 10.1109/PAC.2017.29

mechanism. We have conducted a security study on the Android task mechanism, in terms of both the reachability and impact of attacks. Beyond the known passive attacks reported in related work [3], [5], we have identified additional threat of privilege leakage through the task mechanism, which leads to new types of *active* attacks.
- *Task interference detection and prevention.* Based on our analysis, we have built an app to detect potential task interference and give warning to users when they install an app that may misuse the Android task mechanism.
- *Implementation and evaluation.* We presented four practical attacks targeting Android System 6.0.1 to demonstrate the effectiveness of our method and the severity of the identified vulnerability. To explore the feasibility of the attacks, we also have evaluated their overhead.

Paper Organization. The rest of this paper is organized as follows. Section II gives an overview of the target problem and our approach. Section III presents our analysis framework and introduces the key techniques and analysis results. Section IV summarizes the new attacks we identified. Solutions are discussed in Section V. Section VI presents the related work and Section VII concludes the paper.

II. Overview

In this section, we give an overview the Android task mechanism and the threat model, as well as an introduction to our approach.

A. Android Security Basis and Android Task Mechanism

Components in Android include *Activity*, *Service*, *Content Provider*, and *Broadcast Receiver*. An activity, representing a single screen with user interface, is the most basic elements in Android OS. A service in Android is a UI-less component running in the background. A content provider supplies data from one application to another through methods of the `ContentResolver` class with the ways of storing data in databases, in files, or over the network. A broadcast receiver responds to broadcast messages from other applications.

Android adopts several layers of isolation and sandboxing mechanism as its basic mechanism of security. In particular, it defines a set of permissions to control the access of apps. Apps can access specific resources only if they are granted with required permission. In addition, Android uses user-based protection of Linux to isolate apps. It allocates a unique Linux user ID to each app, which naturally isolates the app from others using the process-based isolation mechanism provided by the Linux kernel.

Android Task Mechanism

The Android task mechanism is designed for facilitate inter-app communication and for better support app collaborating under same tasks. It allows activities from different apps can reside in the same task to perform communications more conveniently [6]. As an example, when the user clicks a "feedback" button from an activity of a game app, Android starts the composer activity of an email app, and puts it onto the game app's activity. After the user finishes sending email, the composer activity is put off and the game app returns to the top. In such a way, two activities are organized to finish a task, while they are actually from different apps.

The Android task mechanism is affected by several flags of apps. The following are the key attributes affecting how apps are grouped.

launchMode: This is the attribute which decides how an activity will be launched. It has four values, i.e., *standard*, *singleTop*, *singleTask* and *singleInstance*. Activities with *standard* or *singleTop* can be instantiated multiple times while activities with *singleTask* or *singleInstance* can only begin a task and be the root of the task. Moreover, *singleInstance* does not permit other activities to be part of its task. An activity without *launchMode* specifically set is assumed to be *standard* by default.

taskAffinity: Activities with the same *taskAffinity*, normally the name of the package, conceptually are in the same task, but this is not always the case. We refer to Section III and Table I for more details. An activity without this attribute set is assumed to have the same *taskAffinity* as its own package name.

allowTaskReparenting: This is a boolean attribute indicating whether an activity can be moved to the task which has the same *taskAffinity* from the original task it is started. An activity without this attribute specifically set is assumed to be *allowTaskReparenting=false*.

For better demonstration, consider the example where an app has the functionalities of viewing contacts as well as sending emails to contacts. The app has two activities for these two functionalities, *SendEmailActivity* and *ViewContactActivity*. For better IPC and logic concerns, designers of the app set the *taskAffinity* of *SendEmailActivity* to be the same as the system email app, the *taskAffinity* of *ViewContactActivity* to be the same as system contact app and *allowTaskReparenting* to be true for both activities.

B. Threat Model

To study the security of the Android task mechanism, we consider a scenario as follows. There are two apps, i.e., AppB and AppM, installed in the same Android device, where AppB is benign and AppM is developed by attackers. We assume AppM does not have to require any permission to manipulate tasks. However, we assume AppM can be granted permissions for following-up behaviors, such as sending the retrieved information out or accessing local storage.

C. Approach Overview

The goal of our work is to comprehensively analyze the Android task mechanism to identify attacks, as well as creating solutions to prevent such attacks.

As shown in Figure 1, our research consists of four components: *Understanding Task Control*, *Understanding Task Privilege*, *Exploitation Analysis* and *Task Interference Checking*.

The first component is to analyze the Android task mechanism, identifying control conditions that can be leveraged

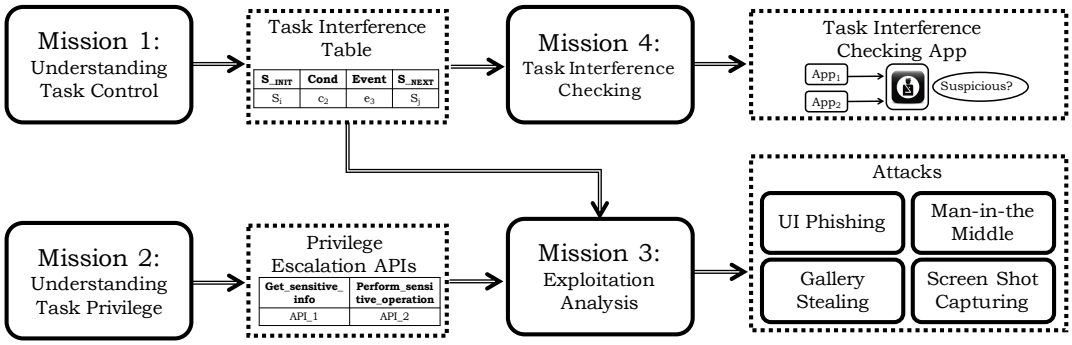


Fig. 1: Approach Overview

by attackers. We aim to find out the interference of tasks between two apps, and identify the dominating factors deciding the apps' property. Secondly, we focus on studying the additional privilege apps obtained when two apps are in the same task. We delicately test sensitive system APIs and compare the difference of the results before and after apps in the same task. Based on the understanding from the previous stages, to demonstrate the achievability and severity of the privilege escalation against Android task mechanism, we develop four light-weight real-world attacks that can steal the sensitive information successfully, most of which only requires `INTERNET` permission. At last, we design a task interference checking app to detect the task interference between users' important apps and other installed apps.

III. Security Analysis of Android Task Structure

In this section, we introduce our approach. We focus on two aspects of the Android task mechanism. First, We analyze the conditions that affect Android task control to identify different ways that can include an app into a task. Second, we explore the privilege an app can get when it is included into a task. These are two necessary components to identify new attacks.

A. Understanding Android Task Control Conditions

We explore the conditions and actions of Android task control through dynamic testing. To do this, we examine the Android documentation [6] to create test cases to drive the exploration. Our goal is to check the influence of the flags introduced in Section II on the task mechanism.

Testing Methodology. We implemented the two template apps introduced in Section II-B, AppB and AppM, as the inputs to drive the testing process. For each combination of the task-control-related flags, such as *launchMode* and *taskAffinity*, we set the corresponding value in app templates, create a pair of AppB and AppM, and test them with different sequences of launching events, e.g., using the Android Launcher to start AppB (denoted as Launcher→AppB) or using AppB to launch AppM (denoted as AppB→AppM). During our test, AppB's *taskAffinity* is set to "TaskB". We only test the conditions where AppM's *allowTaskReparenting* is set to "true", as a "false" value in this flag will not result in task interference.

The results are summarized in Table I. We are interested in cases with potential task interference, i.e., AppM ends up running as part of TaskB. We mark the cases for task interference, i.e., AppM running as part of TaskB, with an asterisk "*". The cases without an asterisk attached are considered to be safe. We list four identified dangerous cases below.

- **Case 2**. Under the conditions of this case, AppM is launched first by the Android Launcher, followed by AppB. Only AppM runs at the foreground, while AppB cannot be executed.
  In this case, AppM blocks AppB from execution, which is a case of denial-of-use to AppB.
- **Case 4**. Under the conditions of this case, AppM is launched by the Android Launcher, followed by AppB. AppB runs in the foreground, and AppM runs in the background, both in TaskB.
- **Case 9**. Under the conditions of this case, the Android Launcher starts AppM. AppM then starts AppB. AppB runs in the foreground, and AppM runs in the background, both in TaskB.
- **Case 10**. Under the conditions of this case, the Android Launcher starts AppB. AppB then starts AppM. AppM runs in the foreground, and AppB runs in the background, both in TaskB.

Whether two Apps are in the same task can be determined by viewing the *Recents* screen which renders all processes that were opened since last clearance [7]. *Recents* screen is rendered when a user presses the *Recents* button which is located at the third from left to right at the button bar followed by *Back* and *Home* buttons. If two Apps are in the same task, *Recents* screen will only show one process other than two when they do not reside in the same task. Note that according to our experiment, flags such as `FLAG_ACTIVITY_CLEAR_TOP` and `FLAG_ACTIVITY_REORDER_TO_FRONT` do not pose a difference than the `SIGLE_TOP` in our case. Therefore, we only include the `SINGLE_TOP` flag in our Table I.

B. Understanding Privilege Obtained in the Same Task

From the results in Table I, we can see that AppM has several ways to be included into the task of AppB, often without involving actions from other apps or the system. Next,

TABLE I: Task Interference Table. We assume AppB is running with the task "TaskB." In the events, the operation A→B stands for A launches B. In the resulting state's status, F stands for execution in foreground; B stands for execution in background; X stands for not-running.

Case #	Initial Conditions				Events	Resulting State		
	AppB	AppM				AppB	AppM	
	LaunchMode	LaunchMode	TaskAffinity	Reparenting		Status	Status	Task
1	standard or singleTop or flag(SINGLE_TOP)	standard or singleTop or flag(SINGLE_TOP)	TaskB	True	Launcher→AppB; Launcher→AppM	F	X	-
2 *	standard or singleTop or flag(SINGLE_TOP)	standard or singleTop or flag(SINGLE_TOP)	TaskB	True	Launcher→AppM; Launcher→AppB	X	F	TaskB
3	singleTask or flag(NEW_TASK)	standard or singleTop or flag(SINGLE_TOP)	TaskB	True	Launcher→AppB; Launcher→AppM	F	X	-
4 *	singleTask or flag(NEW_TASK)	standard or singleTop or flag(SINGLE_TOP)	TaskB	True	Launcher→AppM; Launcher→AppB	F	B	TaskB
5	singleTask or flag(NEW_TASK)	singleTask or flag(NEW_TASK)	TaskB	True	Launcher→AppB; Launcher→ AppM	B	F	TaskM
6	singleTask or flag(NEW_TASK)	singleTask or flag(NEW_TASK)	TaskB	True	Launcher→ AppM ; Launcher→AppB	F	B	TaskM
7	singleInstance or	any or	TaskB	True	Launcher→AppB; Launcher→AppM	F	X	-
8	singleInstance or	any or	TaskB	True	Launcher→AppM; Launcher→AppB	F	B	TaskM
9 *	standard or singleTop or flag(SINGLE_TOP)	standard or singleTop or flag(SINGLE_TOP)	TaskB	True	Launcher→AppM; AppM→AppB	F	B	TaskB
10 *	standard or singleTop or flag(SINGLE_TOP)	standard or singleTop or flag(SINGLE_TOP)	TaskB	True	Launcher→AppB; AppB→AppM	B	F	TaskB

we explore the privilege obtained through the Task mechanism, including privilege for retrieving other apps' information and privilege for changing other apps' states. Therefore, if the privilege given to apps in the same task allows them to carry out dangerous actions, it can be potentially misused by the malicious app.

1) Retrieving Information of Other Apps:

Figuring out the execution state of a victim app, such as whether it is running and which activity is in foreground, is often used as the first step in several attacks, such as UI hijacking [5]. Therefore, Android by default disallows one app from directly querying another app's runtime information from through the Android sandbox policy.

In older versions of Android, there were APIs allowing inter-app runtime information checking. For devices that are prior to *Android Lollipop (v5.0)*, directly calling the API `getRunningTasks(int maxNum)` will return the information of as many as `maxNum` running activities [8]. However, this function is deprecated after *Lollipop* since allowing third-party apps to invoke the function directly will cause information leakage in important apps.

For devices prior to *Android MarshMallow (v6.0)*, directly calling `getRunningAppProcesses()` returns a list of application processes that are running on the device [8]. This function returns a *RunningAppProcessInfo* object, which includes a member variable called *importance* that represents the importance level that the system places on the process [9]. It has one of these values: `IMPORTANCE_FOREGROUND`, `IMPORTANCE_VISIBLE`, `IMPORTANCE_SERVICE`, `IMPORTANCE_BACKGROUND` and `IMPORTANCE_EMPTY`. If *importance* is `IMPORTANCE_FOREGROUND`, the corresponding process is running in the foreground. This method, however, cannot accurately point out which activity running in the foreground since it operates on a process level and accesses only the package name. This method is also no longer supported for *MarshMallow* devices with API level 23 unless the third-party app who is making a call to this function has the same process ID as the target process.

Getting App Running Information in a Task: Although the Android API `getRunningTasks()` is deprecated for direct usage, we have found that it still works if the calling app and the target app are in the same task. The official documentation of `getRunningTasks()` does not explicitly point it out but only states that if it is called, this function only returns a small subsets of information, e.g., the information of the caller's own task and home task which is considered to be not sensitive [8].

2) Changing States of Other Apps:

UI Injection: Ideally, if an app is running in the foreground, other apps isolated from this app should not perform sensitive operations on it. Chen *et al.* [5] show two UI-injection methods that do not require any permissions: (1) starting an `Activity` by setting `lauchMode=singleInstance`. This is also how most system apps, such as the alarm app, pop up a window on the top of another app [10]; (2) starting an `Activity` from the Android broadcast receiver [11].

In our analysis, we find that if two apps are in the same task, the UI-injection attack becomes easier. The attacker can simply call the function `startActivity()` to achieve the same effect. Through understanding the Android SDK source code, `startActivity()` invokes `startActivityForResult()` with `requestCode=-1`. Later `startActivityForResult()` invokes

`execStartActivity()` in the Instrumentation class, a base class for implementing application instrumentation code [12]. `execStartActivity()` then checks the base package of the calling activity by invoking `getBasePackageName()`. If the base package of calling activity matches with that of target activity, the Android system launches the target activity.

Terminating UI: Android does not allow a third-party app to terminate a running app unless they are in the same process. Although an app is not in the foreground, the Android system still allows any third-party apps to make calls to `killBackgroundProcesses()` [13] to terminate any specific background app. There is no direct way to terminate a foreground-running activity.

Unlike `startActivity()`, even if two apps are in the same task, calling `finish()` or `finishActivity()` will not terminate the foreground activity, but will terminate the activity that makes the call. However, we found that as long as two apps are in the same task, calling the API `finishAndRemoveTask()` results in terminating the whole Task regardless of whether an activity is running in the foreground [14] or not. This provides the malicious app an interface to terminate other apps which are within the same task as it.

IV. Information Stealing Attacks

Based on analysis from Section III, we develop four proof-of-concept attacks: *UI Phishing*, *Activity-in-the-middle Attack*, *Gallery Stealing*, and *Screen Shot Capture*. These four attacks demonstrate the severity of the security problems we identified. In this section, the attacks are illustrated using two most popular social apps, Instagram and Facebook. Except Gallery Stealing, which requires `READ_EXTERNAL_STORAGE`, all other three attacks only require the `INTERNET` permission (in order to send out the information stolen).

A. UI Phishing

UI Phishing is a popular type of attacks to spoof users. The difficulty of phishing attack is to decide the timing when the spoofing interface should be prompted, in order to prevent the victim from noticing it. Ren *et al.* [3] introduced "Back Hijacking", which directs users to a spoofed bank *LoginActivity*. The key difference is that the attack method identified in our approach *actively* interact with the victim app using the privilege obtained through the approaches we discussed in Section III-B.

Our proof-of-concept UI Phishing attack is implemented against the scenario when a user logs in to Instagram using his/her Facebook account. In order to ease understanding the phishing attack against this scenario, we brief the Facebook SSO service. According to the Facebook Android developer documentation [15], Facebook SDK takes three ways for apps who require Facebook Login as part of functionality:

- **Native App Login.** If the Android device already has the Facebook app installed, pressing the Facebook login button directly opens the Facebook app where user can log in his/her Facebook account and grants permission to the third-party app to access his/her Facebook personal information.
- **Chrome Custom Tab Login.** If the Android device does not have the Facebook app installed, a third-party app can ask the Chrome browser to open Facebook login page by registering a scheme in the `Manifest` with the format `fb+facebook_app_id`.
- **WebView Login.** Finally if neither the Android device has Facebook app installed, nor does the third-party app register for Chrome Custom Tab Login, an embedded WebView will be launched dynamically rendering the content of Facebook login page.

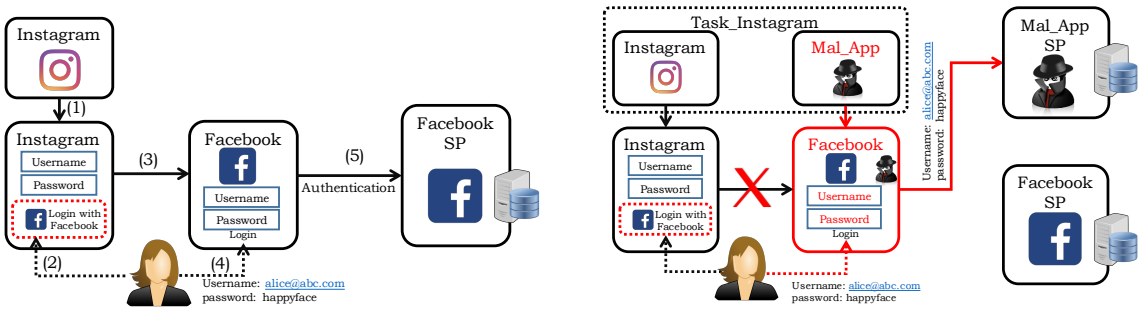
Our attack works when a user presses the "Log in with Facebook" button in Instagram. We assume that a user already has Facebook App installed in the device. Instagram includes Facebook Login as a part of its functionality. It does not register for Chrome Custom Tab Login. In other words, after pressing the "Log in with Facebook" button in Instagram, it will launch Facebook App directly. The package name of Facebook app is `"com.facebook.katana"`, the package name of self-build WebView is `"com.android.webview"` and the package name of Instagram is `"com.instagram.android"`. Shown in Figure 2, it works in the following steps.

(1) Attacking app declares the same task as Instagram. This step can be achieved by setting the attributes of *taskAffinity* and *allowTaskReparenting* as follows.

```
<activity
            android:name=".UIPhishingActivity"
            android:allowTaskReparenting="true"
            android:taskAffinity="com.instagram.
                android">
```

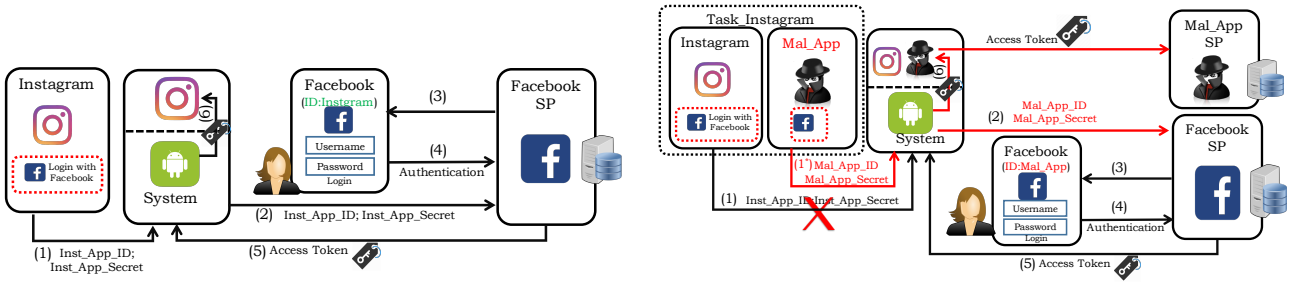
(2) The launcher first launches the attacking app, which launches Instagram immediately through invoking the task parent since Instagram is the parent of the current task. Before Instagram's `setContentView(R.layout.main)` function can be called, the attacking app launches Launcher (Home), meaning that Instagram cannot be displayed in order to achieve the stealthiness. The process can be completed in a fairly short time and will not be noticed by the victim with bare eyes. After that, our attacking app waits for victim to launch Instagram herself.

(3) The attacking app then creates a background thread which runs in a cycle of, we devise, every 100 milliseconds. In the meantime, it checks whether there is a change in the foreground package name, i.e., in our case, the package name changes from `"com.android.webview"` to `"com.facebook.katana"`. This can be achieved by simply making API call to `getRunningTasks()`. Once the foreground package has been changed to Facebook, our attack pops up our counterfeit Facebook *LoginActivity* overriding the real one. The spoofed UI collects the user's login information and send it to adversary's back-end server.



(a) Single-Sign-On Architecture (b) UI Phishing Attack

Fig. 2: UI Phishing. (a) When the user selects "login with Facebook" in Instagram, Instagram will launch the Facebook app. After the user inputs username, password and finishes the authentication with the Facebook service provider, Instagram gets the permission to access the user's Facebook resources. (b) The malicious app Mal_App declares same taskAffinity as Instagram. Mal_App creates a background thread and monitors the running states of Instagram. When the user selects Facebook login button and Instagram launches the Facebook activity, Mal_App will override Facebook with a fake login UI. The username/password input by the user will be sent to Mal_App's service provider.



(a) Single-Sign-On Permission Granting Architecture (b) Man-in-the-Middle Attack

Fig. 3: Man-in-the Middle Attack on SSO. (a) When a user wants to login with Facebook in the Instagram, Instagram sends system its application ID with application secret, Inst_App_ID and Inst_App_Secret. Android system forwards request with Inst_App_ID and Inst_App_Secret to Facebook service provider and retrieves Facebook Login Activity bound with Inst_App_ID. The user inputs her/his username/password for authentication. Facebook service provider returns the access token to Android system if the authentication passed. Android system sends the access token to Instagram according to the application ID that system received. (b) The malicious app Mal_App overrides the ownership of the Instagram task in the same way as UI-phishing. It monitors the Instagram running status and sends system its Facebook login request (including Mal_App_ID and Mal_App_Secret) before the Instagram's request arrives Android system. Android system sends Mal_App's request to Facebook service provider and drops the same login request arrived later. Facebook return Login Activity bound with Mal_App_ID. The user inputs her/his username/password for authentication. Facebook service provider returns the access token to Android system if the authentication passed. Android system sends the access token to Mal_App according to the application ID that system received.

B. Activity-in-the-middle Attack

The Facebook SSO process is based on OAuth 2.0 [16]. The main steps of this mechanism are: (1) When the user opens the relying party (RP) app, in our case, Instagram, it passes its `Facebook_app_id` and directed URL to the Android System. (2) The Android system then redirects user to the Service Provider (SP), in our case, Facebook, and passes it the `Facebook_app_id`. (3) If the user requests to grant the permission to the RP, SP will issue an access token to the Android system. (4) The Android system passes the access token to RP, by which RP is able to access the user's protected resource on SP.

Facebook has two types of access tokens, short-term and long-term [17]. Short-term access token lasts several hours and long-term lasts for 60 days. Unless specifically required, Facebook usually issues short-term access token. RPs use graph API provided by Facebook to retrieve protected resources hosted on Facebook server [18]. Graph API is an HTTP-based API, which is implemented as the following URL:

```
https://graph.facebook.com/me?fields=xxx&
    access_token=xxx
```

In other words, any party who has the access token can access the user's protected resources hosted on Facebook servers. In Android, redirecting to the Facebook app is done by

intent transition based on `startActivityForResult()` and `onActivityResult()`. We implement an MITM attacking app whose model runs in the following steps, shown in Figure 3.

(1) It follows the first three steps in the UI Phishing attack model.

(2) Instead of popping up a phishing login page like UI Phishing, the MITM attack pops up a transparent activity, which blocks the traffic that is supposed to be relayed to Facebook server from user's device system and passes our own traffic to Facebook. This step can be easily realized by creating an invisible Facebook Login Button and sending a button pressed event itself by invoking `mFacebookLoginBtn.performClick()`.

(3) After the user grants the permission, which he/she intends for Instagram, the MITM app retrieves the access token. Since Instagram runs in the background, once the foreground finishes, Instagram will be invoked.

(4) To finalize the process, the adversary needs to verify `APP_ID` and *App Secret* with Facebook. An adversary can either register its attacking app in Facebook Developer Website and use its own `APP_ID` and *App Secret* or steal other RP apps' ID and Secret. Facebook RP Apps will post an HTTP message when user system launches native Facebook App.

C. Gallery Stealing

Starting from Android 6.0 (API level 23), in order to access gallery, an app has to request for `READ_EXTERNAL_STORAGE` permission at runtime rather than at the installation time, which is classified as one of the dangerous permissions [19]. This mechanism provides more secure and flexible protection to user's photo gallery. However, this new security mechanism can be bypassed by exploits to the Android task mechanism, as shown in this section.

Timing. For Android devices before *marshmallow* (API level lower than 23), permissions are requested at the time when an app is being installed. Apps with suspicious permissions will easily trigger the user's attention and likely be denied access. However, with the new requesting-permissions-at-run-time mechanism, our attacking app can avoid requesting permission when being installed since it is requested at the run time. To figure out the suitable timing, it continuously monitors Instagram. Once the victim clicks the "Camera" button on Instagram which allows Instagram to access the gallery and camera, our attacking app instantly kills Instagram and pops up our requesting dialog for access photos in gallery. User mistakenly believes that he/she is granting the permission to Instagram. After the permission is granted, our attacking app retrieves all the photos from user's gallery and send them to the back-end server. In order to be stealthier, the attacking app pops up a dialog shows "System encounters errors" and finally kills itself.

Permission Dialog. Even though timing improves the naturalness of the attack, Android permission dialog shows the name of the app in bold who makes the request. We propose two ways to circumvent this problem.

(1) By employing the idea of social engineering, the attacker can name the attacking app using a name similar to the target app. In our case, e.g., "Instgram" or "Instagam". However, naming the app in this way can hardly pass the review of Google Play. Even if it does, careful users may still notice the difference. Therefore, in our attack, we do not use this method.

(2) Employing the tapjacking. The idea of tapjacking is putting message the attacker wishes to display on the top of the real system message by setting a window layout flag `TYPE_SYSTEM_OVERLAY`. Android realizes the potential threat and adopts the mechanism `MotionEvent.FLAG_WINDOW_IS_OBSCURED` which alerts the real dialog is being overlaid [20]. Unfortunately, Banaś [21] found that `MotionEvent.FLAG_WINDOW_IS_OBSCURED` is not triggered if the covered text does not cover the touch points, which are the buttons in the dialog. Android does not give a patch to the issue. Instead, it adopts an intent transition scheme to notify the user that the content is being overlaid starting from Android 6.0 (API level 23). We managed to circumvent the issue by implementing our main attacking app with `targetSdkVersion=23`, and in the target app we tricked the user to install a helper package, which is an activityless service. Only app which has the `targetSdkVersion=22` and has the overlaying functionality implemented. It seems tricking users to install additional package is not applicable, but it is in fact a very common situation for many apps such as those who require Android SQLite Manager.

The basic steps of the attack model are listed as follows.

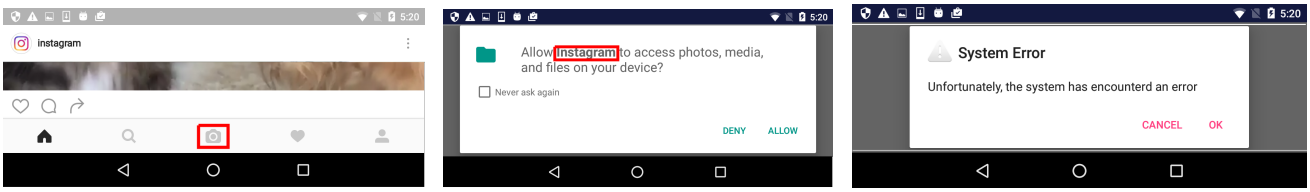
(1) The attack follows the first three steps of UI Phishing attack.

(2) Once a user presses the camera button of Instagram as shown in Figure 4(a), the foreground `Activity` will change to `"com.instagram.android.creation.activity.MediaCaptureActivity"`. Therefore, Instead of detecting whether the foreground package has changed to another app, we zoom in the design to focus on changing of the foreground `Activity`.

(3) Instead of popping up a counterfeit page like UI Phishing does, it pops up an `Activity` which immediately asks a user for `READ_EXTERNAL_STORAGE` so that the user thinks he/she is granting the permission to Instagram.

(4) Once the attack app gets the permission, it traverses all the pictures and transmits the image buffers to server.

(5) To achieve better stealthiness, we introduce the Tapjacking to assist our attack. Its basic idea is to cover the real texts with some fake texts. Then we are able to change the text of the permission dialog to "Allow Instagram to access ...?" instead of the real text which is "Allow GalleryStealing to access ...?", which is shown in Figure 4(b). Once user grants the permission, the app quickly sends all images as buffer to the server. Meanwhile, it fools the user by showing a "system



(a) (b) (c)

Fig. 4: Gallery Stealing Attack: first the user clicks the camera button highlighted in the first picture. Then the attacking app kills the real Instagram and pops up its `Activity` asking for user permission, the text of permission dialog is overlaid by the fake text as shown in the second picture. Once user grants the permission, the app quickly sends all images as buffer to the server, meanwhile, it fools the user by showing a "system" warning dialog telling the user the system encounter an error.

warning dialog" telling the user that the system encounters an error (shown in Figure 4(c)).

D. Screen Shot Capturing

With the ability of knowing which `Activity` is currently running in the foreground, we implement this screen shot capturing attack which starts taking screen shots while a user is entering username and password in the Facebook `LoginActivity` which is `com.facebook.katana.LoginActivity` in full. For password typing, every time a character is entered, the character will be shown for a short time before it turns into a star sign. Therefore, taking screen shot every 0.1 second should capture everything a user types in password box.

With respect to taking a screen shot programmatically, we summarize four possible ways:

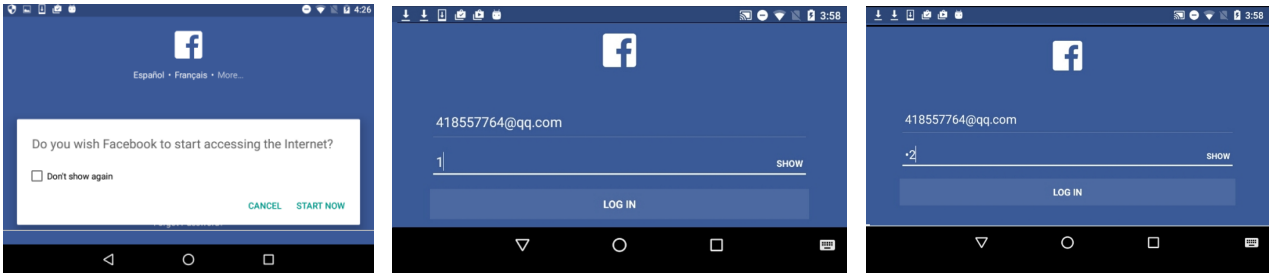
- **Using READ_FRAME_BUFFER Permission.** Declaring `READ_FRAME_BUFFER` permission in the `Manifest` allows an application to take screen shots by making calls to `ISurfaceComposer` [22]. However, this permission is not available to third-party application unless it has the same signature as the system does.
- **Using fb*.** Some Linux systems store frame buffers in `/dev/graphics/fb*` or `/dev/fb*`. `fb0` represents the first frame buffer, `fb1` represents the second frame buffer and so on. Using native C/C++ code to get access to these files and copy the buffer as a `GGLSurface` structure is theoretically possible. But there are two unsolvable obstacles of this method:
  - This method requires root permission.
  - It is likely that `fb*` does not even exist.
- **Using Backup Channel over USB.** Android system uses Android Debug Bridge (ADB) to listen to the debugging connections over USB [23]. ADB has slightly more privileges than normal apps. Bai *et al.* [24] manage to exploit Backup Channel through ADB to steal access tokens from other apps. Combining ADB with Dalvik Debug Monitor Server (DDMS) tool enables an app to get the screen shot from the device without any permission [25].
- **Using MediaProjection.** For devices beginning in Android 5.0 (API level 21), a class called `MediaProjection` was added to Android SDK which enables a third-party app to capture screen shot and record system audio [26]. While recording system audio requires `RECORD_AUDIO` permission, capturing screen shots does not.

We employed the fourth method to capture screen shots. The basic steps of the attack are listed as follows.
(1) It follows the first three steps of UI Phishing. What's different is we devise this attack to focus on Facebook App since we are hoping to steal user's Facebook username and password. Hence, we declared this attack to reside in the same task as Facebook.
(2) Once a user launches the Facebook app, a transparent Activity is popped up start taking screenshots. Although it does not require permission for taking screen shots, it uses intent transition to let the user decide whether or not an app can capture screen shots in a permission-like dialog. Again we employ the tapjacking in Gallery Stealing attack to cover the text to be "Allow Facebook start accessing Internet?"
(3) The attack starts capturing screens by calling `startActivityForResult()` which passes a screenshot as an intent from which we can extract an object of `MediaProjection` later, and finally use this `MediaProjection` object to pass the image to an object of `ImageReader` through its member function `createVirtualDisplay`. Since the screenshots are taken in the `RGBA_8888` format while bitmap takes `ARGB_8888`, we still need to do matrix transformations to get the image.

E. Performance

In order to study the feasibility of the four attacks, we implement them and evaluate them in two aspects, *time cost* and *memory usage*.

1) Time Cost: We conduct 6 - 8 rounds of tests for every attack to calculate the average running time cost. The testing results are summarized in Table II. As shown in Table II, our attacks are efficient since the time cost of most attacks is less than 8 seconds: the UI Phishing attack costs 7.1 seconds, which includes user interactions such as launching Instagram, entering username and passwords; The Activity-in-the-middle attack costs time 5.8 seconds; and the average time cost of the gallery-stealing attack is 4.6 seconds.



(a) (b) (c)

Fig. 5: Screenshot Capturing Attack: first the user launches Facebook, then the attacking app launches a transparent `Activity` which quickly asks for taking screen shot, the notification dialog is again overlaid by fake text as shown in the first picture. Once the user clicks start now, the attacking app begins to take screen shots as well as sending the screen shots to back end server. (b) and (c) are two of the screen shots received by server which expose the password plaintext.

The Screenshot Capturing attack is an exception, taking 38.1 seconds. This attack uses much longer time because screenshots in Android are passed in the format of RGBA, while we need to change to ARGB in order to convert it to common format such as JPEG and PNG. We perform the matrix transformation required for the conversion on the mobile device. But since the process is undertaken in the background, it will not trigger the suspicions of the user. In the real world attack, an adversary can leave the job of matrices transformation to the server.

TABLE II: Time Cost of Proof-of-Concept Attacks

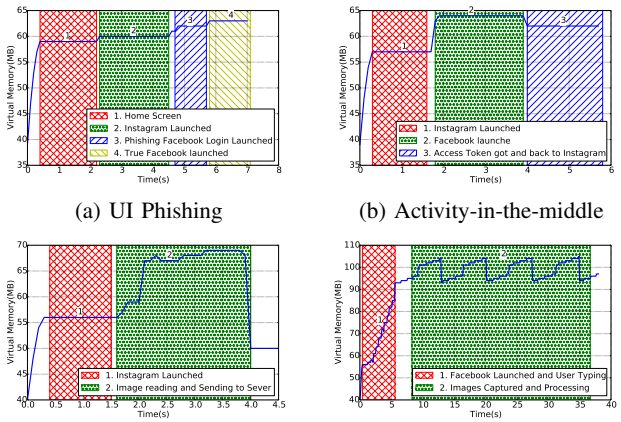
Attack	Time Cost (s)
UI Phishing	7.1
Man-in-the-Middle	5.8
Gallery Stealing	4.6
Screenshot Capturing	38.1

2) Memory Usage: We conduct 3-round experiments for each attacks to get the memory distractions of attacks and evaluate their average memory usage. The testing results are illustrated in Figure 6. It shows that the maximum memory usage of most attacks is less than $70MB$ except Gallery Stealing attack, whose memory usage is around $100MB$. Our testing results also disclose the memory usage distribution on different period, in which the major memory usage of each attack is to launch victim app (normal app), e.g., Instagram and Facebook. The memory usage differences caused by stealthy behaviors/operations are negligible.

In addition, we also conduct experiment to monitor the battery consuming status and evaluate the battery usage of our attacks. The results show that our attacks may not cause influence on battery aspect (the battery usage rates of most attacks are 0%). Above all, the experiment results demonstrate that our proof-of-concept attacks are light-weight with limited permission requirements.

V. Solutions to Eliminate Task Interference

In this section, we discuss solutions on mitigating Android task interference. Based on our study, we design and implement a task interference checking tool to detect the potential risk of task interference among apps. It protects apps specified by users from being manipulated by untrustworthy apps. In addition, we also propose some suggestions to limit the additional privilege achieved by the apps in the same task.



(a) UI Phishing (b) Activity-in-the-middle

(c) Gallery Stealing (d) Screenshot Capturing

Fig. 6: Memory Distribution Curves

A. Task Interference Checking

In this subsection, we present the design and implementation of our *Task Interference Checking* app named *TICK*.

Based on our understanding of the task interference problem, which includes the necessary conditions, events and their dependency, we design a method to check the task interference status among Android apps. Table I provides us with the basis of evaluating task interference checking.

Design. As shown in Figure 7, *TICK* consists of two basic modules, *Pre-processing* and *Interference Checking*, and two supporting databases, *Protected App Signature* and *Task Interference Table*.

- *Protected App Signature.* For the apps users who want to protect against attacks to Android tasks, this database includes the abstracted conditions of such attacks of each app, which is stored as signatures in this database.

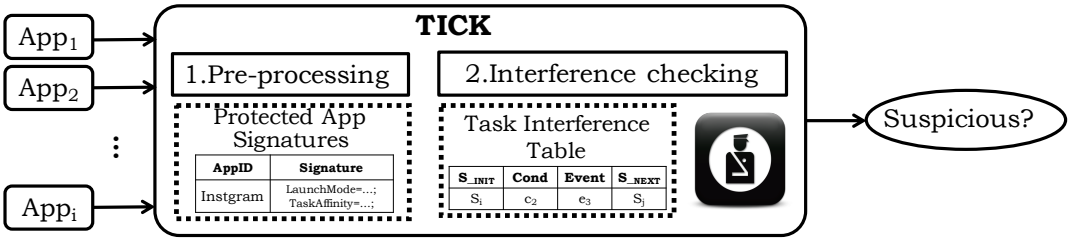


Fig. 7: Task Interference Checking Architecture

- *Task Interference Table.* This database is the output from our research in Section III, which includes the fundamental rules of task interference.
- *Pre-processing.* The module takes as inputs the manifest files of apps to be checked, abstracts and outputs their task interference features for checking.
- *Interference Checking.* The interference checking module takes inputs from *Pre-processing* module and *Protected App Signature* database, and checks the suspiciousness of the testing apps according to the rules specified in the Task Interference Table.

There are two application scenarios for deploying *TICK*.

C-1: Before a user installs an app, she/he can use TICK to detect the potential risk of task-related attacks from the app. TICK will parse the app meta data and check with our table and signature database to see if it interferes with an existing app. If the app is detected to cause interference, TICK will issue a warning to the user and suggest she/he not install the app.

C-2: Our checking app can do chronically scan from time to time, check if new packages are added to the device, audit if they have security concerns, and notify users if and suspicious package were installed.

Getting Meta Data of an Activity. Currently, our checking app only considers static declarations of task related attributes from the app package. Moreover, the key attributes related to Android Task cannot be altered during runtime, e.g., *taskAffinity* and *allowTaskReparenting*.

Android SDK provides standard APIs to get an app's Manifest meta data without requiring any permission. Given a package name and a flag, one can retrieve an object of `PackageInfo` through the `PackageManager`. Here, we set the flag to `GET_ACTIVITIES` and we retrieve a list of `ActivityInfo` by fetching the attribute `PackageInfo.activities`. ActivityInfo contains all the meta data described in the `Manifest` about every activity of a package, including those we care, e.g., `taskAffinity`, `FLAG_ALLOW_TASK_REPARENTING`, `LAUNCH_SINGLE_TASK` and so on. The key APIs of getting `packageManager` and `packageInfo` are respectively `getPackageManager()` and `getPackageInfo()`.

Effectiveness. To verify the effectiveness, we used Instagram and our UI-Phishing app as inputs to TICK. It parsed the meta data of each activity in both apps and found that most activities of Instagram with *taskAffinity* set to "com.instagram.android", while the rest has *taskAffinity* of "com.instagram.android.ShareHandlerActivity". Most of the activities in Instagram have *launchMode* set to "standard". A small portion of the activities have the *launchMode* set to "singleTop", e.g., the `LoginActivity`. Very few activities have *launchMode* set to "singleInstance". The UI-Phishing app's first activity has the following flags set: `taskAffinity = "com.instagram.android"`, `allowTaskReparenting = true` and `launchMode = standard`. This matches Case 9 and Case 10 in our task interference table. When we use TICK to scan the device, specifying the Instagram app as the one to be protected, TICK successfully warns users about the potential risk from the UI-Phishing app. The overhead of TICK can vary based on a number of factors such as device hardware and number of third-party Apps installed in the device. In our experiment, excluding all system-level Apps, there are totally 65 third-party Apps installed in our device. It rougly took 4-5 seconds to scan all Apps and identify suspicious ones.

B. Design Suggestions

From security issues we have demonstrated in the above attacks, it is clear that the privilege given to apps in the same task is well beyond what is expected for a mechanism that facilitates app collaboration and interaction. In fact, the task mechanism should be treated as a way of *authorization*, and the security mechanism around the task mechanism also should be designed accordingly.

In particular, when treating a task as a boundary for authorization, we need to be explicit about the ownership of a task and its authenticity. For example, if an app specifies the *taskAffinity* of an existing task, there needs to be a form of *authorization* before the app can be included into the task, and the authorization should be carried out by entities with privilege greater than the privilege given to the task. This is similar to the requirement made by the UNIX group mechanism. In addition, as the name of "task affinity" becomes an identifier for a security object, the system should avoid name conflicts. In case they occur, they need to be resolved with all involving entities to avoid unexpected privilege escalation.

VI. Related Work

IPC Security: IPC security is one of the top concerns while designing OSes. Some early studies have reported security threats in the Android IPC mechanism. More specifically,

Ren et al. [3] have proposed the first study of the security of Android task mechanisms and showed the possibilities of several enabled attacks, such as back-button hijacking and uninstalling-prevention attack. Other popular mobile systems like iOS are also not immune to the risks. Xing et al. [27] registered a counterfeit scheme which hijacks the real Facebook scheme in iOS and successfully stole a Facebook access token that was supposed to be passed onto the relying-party app. Besides problematic designs of IPC of OSes, mis-implementation of certain IPC-based protocols can also lead to security concerns. Chen et al. [16] have conducted an analysis on 149 mobile applications and showed 89 of them (59.7%) incorrectly implemented OAuth and thus are vulnerable to SSO-oriented attacks. Furthermore, IPC vulnerabilities were also documented on other platforms besides mobile OSes. Take browser platform for an example. Wang et al. [28] discovered 8 serious logic flaws among the traffics between high-profile ID provider and relying website through browser platform. Wang et al. [29] discovered logic flaws in several shopping websites and finally purchasing goods without or with little payment.

GUI Security: As for traditional desktop and browser environment, GUI security issues have been studied extensively [30], [31]. Niemietz et al [32] implement a UI redressing attacks on Android devices base on clickingjacking and tapjacking and the attack is feasible to be transferred from desktop to mobile and to browser, enabling the attack to be adapted to multiple platforms and functionalities. As mobile market begins to thrive, GUI security is more concerned in mobile platforms than ever before. Chen et al. [5] managed to impose Hidden Markov Model (HMM) on a public resource `shared_vm` combining a bundle of data to perform UI Inference Attack and successfully stole sensitive information from users such as user names, passwords and check images. Wang et al. [33] implemented a malicious app which circumvented the Apple Code Review system and successfully stole user secrets stealthily.

Defending against Malicious Behaviors: Defending malicious behaviors can be categorized into two branches, detection and prevention. In previous studies, various detection schemes have been introduced to prevent GUI-related attacks. Fu et al. [34] employ the Earth mover's distance (EMD) mechanism to detect possible malicious web page through measuring the similarity between two web pages by first converting web pages to images, and then grabbing and comparing the feature points through training data set. More generally, Chen et al! [35] introduce the concept of permission event graph (PEG) with model checking mechanisms to detect abnormal behaviours of Android apps. As for prevention, one idea is to prevent sensitive data from being leaked to the malicious server party. Hornyack et al. [36] develop a system for Android called AppFence, which can block the sensitive data from being transmitted, or substitute the fine-grained data to coarse-grained data if transmission is unpreventable. Ren et al. [37] develop WindowGuard which protect against GUI attacks by enforcing the Android Window Integrity (AWI).

VII. Conclusion

The Android task mechanism provides a mechanism for apps to closely collaborate under a task. It offers flexibility on top of the Android isolation mechanism. However, the flexibility results in unexpected privilege escalation in Android apps. In this paper, we study of the Android task mechanism. We explore the ways to reach a task and analyze the privilege obtained through the task mechanism. Based on our study, we developed four new *active* attacks, ranging from UI spoofing to gallery/screen stealing. We also develop a detection tool to detect apps that can potentially leads to privilege escalation and task interference.

Acknowledgment: This work was supported in part by the National Natural Science Foundation of China (No. 61402029), the National Natural Science Foundation of China (No. 61370190 and No. 61379002), Singapore Ministry of Education under NUS grant R-252-000-539-112.

References

[1] "Android market," http://www.idc.com/prodserv/smartphone-os-market-share.jsp, observed in July 2016.
[2] "Android applications," https://en.wikipedia.org/wiki/Google_Play, observed in July 2016.
[3] 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)*, 2015, pp. 945–959.
[4] "demo," 2016. [Online]. Available: https://www.youtube.com/watch?v=pY_0Sz3AA9Y
[5] 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)*, 2014, pp. 1037–1052.
[6] "Tasks and back stack," https://developer.android.com/guide/components/tasks-and-back-stack.html, observed in June 2016.
[7] "Recents screen," 2016. [Online]. Available: https://developer.android.com/guide/components/activities/recents.html
[8] "Activity manager getrunningtasks," https://developer.android.com/reference/android/app/ActivityManager.html#getRunningTasks(int), observed in June 2016.
[9] "Runningappprocessinfo importance," https://developer.android.com/reference/android/app/ActivityManager.RunningAppProcessInfo.html#importance, observed in June 2016.
[10] "Android launch mode," https://developer.android.com/guide/topics/manifest/activity-element.html#lmode, observed in June 2016.
[11] "Android broadcast receiver," https://developer.android.com/reference/android/content/BroadcastReceiver.html, observed in June 2016.
[12] "Instrumentation," https://developer.android.com/reference/android/app/Instrumentation.html, observed in June 2016.
[13] "killbackgroundprocesses," https://developer.android.com/reference/android/app/ActivityManager.html#killBackgroundProcesses(java.lang.String), observed in June 2016.
[14] "finishandremovetask," https://developer.android.com/reference/android/app/ActivityManager.AppTask.html#finishAndRemoveTask(), observed in June 2016.
[15] "Facebook login for android," https://developers.facebook.com/docs/facebook-login/android, observed in June 2016.
[16] E. Y. Chen, Y. Pei, S. Chen, Y. Tian, R. Kotcher, and P. Tague, "Oauth demystified for mobile application developers," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014, pp. 892–903.
[17] "Facebook access token," https://developers.facebook.com/docs/facebook-login/access-tokens#usertokens, observed in June 2016.
[18] "Facebook graph api," https://developers.facebook.com/docs/graph-api/overview, observed in June 2016.

[19] "Requesting permissions at run time," https://developer.android.com/training/permissions/requesting.html, observed in June 2016.
[20] "Flag_window_is_obscured," https://developer.android.com/reference/android/view/MotionEvent.html#FLAG_WINDOW_IS_OBSCURED, observed in June 2016.
[21] Iwo Banaś, "Android tapjacking," http://www.iwobanas.com/2015/07/android-tapjacking-fix/, observed in June 2016.
[22] "Read_frame_buffer," https://developer.android.com/reference/android/Manifest.permission.html#READ_FRAME_BUFFER, observed in June 2016.
[23] "Android debug bridge," https://developer.android.com/studio/command-line/adb.html, observed in June 2016.
[24] G. Bai, J. Sun, J. Wu, Q. Ye, L. Li, J. S. Dong, and S. Guo, "All your sessions are belong to us: Investigating authenticator leakage through backup channels on android," in *20th International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2015, pp. 60–69.
[25] "Dalvik debug monitor server," https://developer.android.com/studio/profile/ddms.html, observed in June 2016.
[26] "Mediaprojection," https://developer.android.com/reference/android/media/projection/MediaProjection.html, observed in June 2016.
[27] L. Xing, X. Bai, T. Li, X. Wang, K. Chen, X. Liao, S.-M. Hu, and X. Han, "Cracking app isolation on apple: Unauthorized cross-app resource access on mac os x and ios," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015, pp. 31–43.
[28] R. Wang, S. Chen, and X. Wang, "Signing me onto your accounts through facebook and google: A traffic-guided security study of commercially deployed single-sign-on web services," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy (S&P)*, 2012, pp. 365–379.
[29] R. Wang, S. Chen, X. Wang, and S. Qadeer, "How to shop for free online - security analysis of cashier-as-a-service based web stores," in *32nd IEEE Symposium on Security and Privacy (S&P)*, 2011, pp. 465–480.
[30] N. Feske and C. Helmuth, "A nitpicker's guide to a minimal-complexity secure GUI," in *21st Annual Computer Security Applications Conference (ACSAC)*, 2005, pp. 85–94.
[31] F. Roesner and T. Kohno, "Securing embedded user interfaces: Android and beyond," in *the 22nd USENIX Security Symposium (USENIX Security)*, 2013, pp. 97–112.
[32] M. Niemietz and J. Schwenk, "Ui redressing attacks on android devices," in *Blackhat Asia*, 2014.
[33] T. Wang, K. Lu, L. Lu, S. Chung, and W. Lee, "Jekyll on ios: When benign apps become evil," in *the 22nd USENIX Security Symposium (USENIX Security)*, 2013.
[34] A. Y. Fu, L. Wenyin, and X. Deng, "Detecting phishing web pages with visual similarity assessment based on earth mover's distance (emd)," *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 4, pp. 301–311, Oct 2006.
[35] 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 *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2013.
[36] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, "These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications," in *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, 2011, pp. 639–652.
[37] C. Ren, P. Liu, and S. Zhu, "Windowguard: Systematic protection of gui security in android," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.