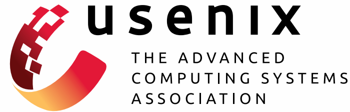

Towards Discovering and Understanding Task Hijacking in Android

Chuangang Ren, *The Pennsylvania State University;* Yulong Zhang, Hui Xue, and Tao Wei, *FireEye, Inc.;* Peng Liu, *The Pennsylvania State University*

https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/ren-chuangang

This paper is included in the Proceedings of the
24th USENIX Security Symposium

August 12–14, 2015 • Washington, D.C.

ISBN 978-1-931971-232

Open access to the Proceedings of
the 24th USENIX Security Symposium
is sponsored by USENIX

Towards Discovering and Understanding Task Hijacking in Android

Chuangang Ren[1], Yulong Zhang[2], Hui Xue[2], Tao Wei[2] and Peng Liu[1]

[1]Pennsylvania State University, State College
[2]Fireeye, Inc.

Abstract

Android multitasking provides rich features to enhance user experience and offers great flexibility for app developers to promote app personalization. However, the security implication of Android multitasking remains under-investigated. With a systematic study of the complex tasks dynamics, we find design flaws of Android multitasking which make all recent versions of Android vulnerable to *task hijacking* attacks. We demonstrate proof-of-concept examples utilizing the task hijacking attack surface to implement UI spoofing, denial-of-service and user monitoring attacks. Attackers may steal login credentials, implement ransomware and spy on user's activities. We have collected and analyzed over 6.8 million apps from various Android markets. Our analysis shows that the task hijacking risk is prevalent. Since many apps depend on the current multitasking design, defeating task hijacking is not easy. We have notified the Android team about these issues and we discuss possible mitigation techniques in this paper.

1 Introduction

In the PC world, computer multitasking means multiple processes are running at the same period of time. In Android systems, however, multitasking is a unique and very different concept, as defined in Android documentation: "A *task* is a collection of activities that users interact with when performing a certain job" [1]. In other words, a task contains activities [4] (UI components) that may belong to multiple apps, and each app can run in one or multiple processes. The unique design of Android multitasking helps users to organize the user sessions through tasks and provides rich features such as the handy application switching, background app state maintenance, smooth task history navigation using the "back" button, etc. By further exposing task control to app developers, Android tasks have substantially enhanced user experience of the system and promoted personalized features for app design.

Despite the merits, we find that the Android task management mechanism is plagued by severe security risks. When abused, these convenient multitasking features can backfire and trigger a wide spectrum of *task hijacking attacks*. For instance, whenever the user launches an app, the attacker can condition the system to display to the user a spoofed UI under attacker's control instead of the real UI from the original app, without user's awareness. All apps on the user's device are vulnerable, including the privileged system apps. In another attack, the malware can be crafted as one type of ransomware, which can effectively "lock" the tasks that any apps belong to on the device (including system apps or packages like "Settings" or "Package Installer"), i.e. restricting user access to the app UIs and thus disabling the functionality of the target apps; and there is no easy way for a normal user to remove the ransomware from the system. Moreover, Android multitasking features can also be abused to create a number of other attacks, such as phishing and spyware. These attacks can lead to real harms, such as sensitive information stolen, denial-of-service of the device, and user privacy infringement, etc.

The Android multitasking mechanism and the underlying feature provider, the Activity Manager Service (AMS), haven't been thoroughly studied before. In this paper, we take the first step to systematically investigate the security implications behind Android multitasking design and the AMS. At the heart of the problem, although the Android security model renders different apps sandboxed and isolated from one another, Android allows the UI components (i.e., activities) from different apps to co-reside in the same task. Given the complexity of task dynamics, as well as the vagaries of additional task controls available to developers, the attacker can play tricky maneuvers to let malware reside side by side with the victim apps in the same task and hijack the user sessions of the victim apps. We call this *task hijack-*

Attacks Types	Consequences	Vulnerable system & apps
Spoofing	Sensitive info stolen	all; all
Denial-of-service	Restriction of use access to apps on device	all; all
Monitoring	User privacy infringement	Android 5.0.x; all

Table 1: Types of task hijacking attacks presented in this paper (system versions considered - Android 3.x, 4.x, 5.0.x).

ing.

Given the security threats, it becomes important to fully study Android multitasking behaviors in a systematic way. We approach this topic by projecting the task behaviors into a state transition model and systematically study the security hazards originated from the discrepancies between the design assumptions and implementations of Android tasks. We find that there is a plethora of opportunities of task hijacking exploitable to create a wide spectrum of attacks. To showcase a subset of the attack scenarios and their consequences, we implement and present a set of proof-of-concept attacks as shown in Table 1.

We do vulnerability assessment to the task hijacking threats and discover that all recent Android versions, including Android 5, can be affected by these threats, and all apps (including all privileged system apps) are vulnerable to most of our proof-of-concept attacks on a vulnerable system. By investigating the employment of task control features by app developers based on 6.8 million apps in various Android markets, we find that despite the serious security risks, the "security-sensitive" task control features are popular with developers and users. We have reported our findings to the Android security team, who responded to take a serious look into the issue. We summarize our contributions below:

- To the best of our knowledge, we are the first to systematically study the security implications of Android multitasking and the Activity Manager Service design in depth.
- We discover a wide open attack surface in Android multitasking design that poses severe threats to the security of Android system and applications.
- Base on our vulnerability analysis over 6.8 million apps, we find that this problem is prevalent and can lead to a variety of serious security consequences.
- We provide mitigation suggestions towards a more secure Android multitasking sub-system.

2 Background

Android Application Sandbox: The Android security model treats third-party apps as untrusted and isolates them from one another. The underlying Linux kernel enforces the Linux-user based protection and process isolation, building a sandbox for each app. By default, the components of one app run in the same Linux process with an unique UID. Components from different apps run in separate processes. One exception is that different apps can run in one process only if they are from the same developer (same public key certificate), and the developer explicitly specifies the same process in the manifest file. The Linux sandbox provides the foundation for app security in Android. In addition, Android provides a permission model [12, 19] to extend app privileges based on user agreement, and offers an inter-component communication scheme guarded by permissions for inter-app communication.

Activity: *Activity* is a type of app component. An activity instance provides a graphic UI on screen. An app typically has more than one activities for different user interactions such as dialing phone numbers and reading a contact list. All activities must be defined in an app's *manifest file*.

Intent: To cross the process boundaries and enable communication between app components, Android provides an inter-component communication (ICC) scheme supported by an efficient underlying IPC mechanism called *binder*. To perform ICC with other components, an component use *intent*, an abstract description of the operations to be performed. An intent object is the message carrier object used to request an action from another component, e.g., starting an activity instance by calling `startActivity()` function. Intent comes in two flavors. *Explicit intent* specifies the component to start explicitly by name. *Implicit intent* instead encapsulates a general type of action, category or data for a component to take. The system will launch a component "capable" of handling this intent. If more than one target activities exist in the system, the user is prompted to choose a preferred one.

Activity Manager Service (AMS): AMS is an Android system service that supervises all the activity instances running in the system and controls their life cycles (creation, pause, resume, and destroy). The interaction and communication protocols between activities and the AMS are implemented by the Android framework code, which is transparent to app developers, leaving developers focusing on the app functionality. While Window Manager Service (WMS) manages all windows in the system and dispatches user inputs from the windows, AMS organizes all the activities in the system into tasks, and is responsible for managing the tasks and supporting the multitasking features as will be described in Section 3.

In addition, AMS is in charge of supervising service components, intent routing, broadcasting, content providers accesses, app process management, etc., making itself one of the most critical system services in the Android system.

3 Android Tasks State Transition Model

3.1 Task and Back Stack

In Android, a *task* [1] is a collection of activities that users have visited in a particular job. The activities in a task are kept in a stack, namely *back stack*, ordered by the time the activities are visited, such that clicking the "back" button would navigate the user back to the most recent activity in the current task. The activities in the back stack may be from the same or different apps.

The activity displayed on the screen is a *foreground activity* (on the top of the back stack) and the task associated with it is a *foreground task*. Therefore, there is only one foreground task at a time and all other tasks are *background tasks*. When switched to the background, all activities in a task stop, and remain intact in the back stack of the task, such that when the users return they can pick up from where they left off. This is the fundamental feature that Android multitasking offers to users.

3.2 A Tasks State Transition Model

The status of tasks in a system keeps changing as a result of user interaction or app program behaviors. To understand the complex task dynamics and its behind security implications, we view the task transitions through time as a state transition model. The model is described by $(S, E, \Lambda, \rightarrow)$, where S denotes a set of task states; E and Λ are sets of events and conditions respectively; and $\rightarrow$ indicates a set of feasible transactions allowed by the system under proper events and conditions.

1. **Task state** ($s \in S$): represents the state of all tasks (specifically, the back stacks) in the system and their foreground/background statuses. In other words, the tasks in the system remain in one state iff the activity entries and their orders in the back stacks stay the same, and the foreground task remains to be the same task.

2. **Event** ($e \in E$): denotes the event(s) it takes to trigger the state transition, for example, pressing the "back" button or calling `startActivity()` function.

3. **Condition** ($\lambda \in \Lambda$): the prerequisites or configurations (usually default) that enable a state transition under certain events. We denote $\lambda^{default}$ as the system default conditions in this paper.

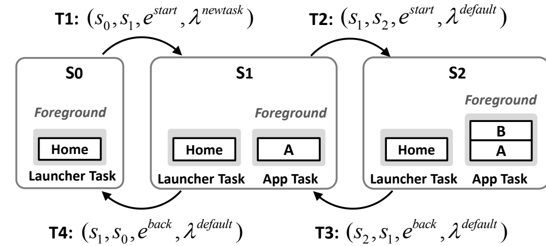

Figure 1: A simple task state transition example.

4. **Transition** ($\rightarrow$): stands for a feasible state transition. Not all task transitions are feasible, e.g., the order of activities in back stack cannot be changed arbitrarily (only push and pop are viable operations over the stack). A viable transaction is also represented as $s_1 \rightarrow s_2$, or (s_1, s_2, e, λ), where $s_1, s_2 \in S$.

3.3 A Task State Transition Example

Given the state transition model, we depict a simple task state transition example in Figure 1. The figure shows three task states, and the state transitions reflect the process in which the user first launches an app from the home screen ($s_0 \rightarrow s_1$), visits an additional activity UI in the app ($s_1 \rightarrow s_2$) and returns to the home screen by pressing the "back" button twice ($s_2 \rightarrow s_1 \rightarrow s_0$).

In each task state, we show all existing tasks and their back stacks. For example, s_0 is a task state in which no task, except the launcher task, is running in the system. The launcher task has only one activity in its back stack - the home screen from which users can launch other apps.

In $(s_0, s_1, e^{start}, \lambda^{newtask})$, a new app task is created and brought to the foreground in the resulting state s_1. e^{start} represents the event that `startActivity()` is called by the home activity in the launcher task. This event could happen when the user clicks the app's icon on the home screen. $\lambda^{newtask}$ specifies a special condition, i.e., the FLAG_ACTIVITY_NEW_TASK flag is set to the input intent object to `startActivity()` function. This flag notifies the AMS the intention of creating a new task to host the new activity. Note that in this example most state transitions are under default conditions, indicated by $\lambda^{default}$, while here $s_0 \rightarrow s_1$ is an exception because the launcher app customizes the condition ($\lambda^{newtask}$) for a valid design purpose: start the app in a brand new task when the user launches an new app. This is an example where app developers can customize certain configurable conditions to implement helpful app features. However, condition like $\lambda^{newtask}$ can be abused in a task hijacking attack, as discussed in Section 4.

Next, $(s_1, s_2, e^{start}, \lambda^{default})$ is triggered by event e^{start} again (this time called by activity A instead), yet under the default condition. By default, AMS pushes the new activity instance B on top of the current back stack

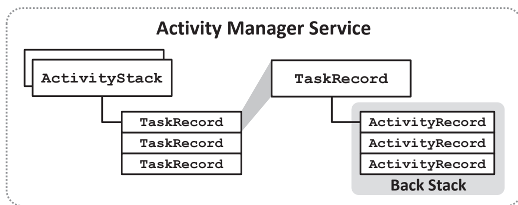

Figure 2: Data structures of tasks, activities and back stacks in the Activity Manager Service.

as shown in s_2. The previous activity A is stopped and its state is retained. In $(s_2, s_1, e^{back}, \lambda^{default})$, e^{back} represents the event of user pressing the "back" button. As expected by the user, the next activity A on stack is brought back to the screen, and its original state is resumed. Activity B is popped from the back stack and destroyed by the system. The initial state s_0 is finally restored through $(s_1, s_0, e^{back}, \lambda^{default})$ when the user presses "back" button again. The app's task is destroyed because when the popped activity is the last activity in the back stack, the activity is destroyed together with the "empty" task.

Note that activities from different apps can co-reside in the same task (e.g. activity A and B in this example). In other words, although activities from different apps are isolated and protected within their own process sandboxes, Android allows different apps to co-exist in a common task. This creates opportunities for malicious activities to interfere with other activities once they are placed in the same task, and the system passes the program control to the malicious activities.

In reality, the amount of possible task states in a system is big, and the state transitions can be complex, e.g., each state may again have numerous incoming and outgoing transitions connecting with other states. In Section 4, we discuss what may go wrong during the complex task state transitions.

3.4 Android Implementation

AMS maintains Android tasks and activities in a hierarchy shown in Figure 2. AMS uses `TaskRecord` and `ActivityRecord` objects to represent tasks and activities in the system respectively. A `TaskRecord` maintains a stack of `ActivityRecord` instances, which is the back stack of that task. Similar to the activities in a back stack, tasks are organized in a stack as well, maintained by a `ActivityStack` object, such that when a task is destroyed, the next task on stack is resumed and brought to the foreground. There are usually two `ActivityStack` containers in the system - one containing only the launcher's tasks and the other holding all remaining app tasks.

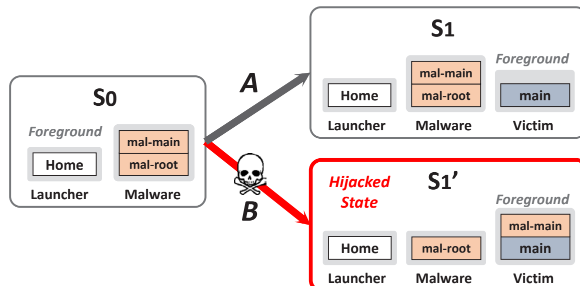

Figure 3: Task state transition of spoofing attack (A: task state transition by system-default. B: Hijacking state transition).

4 Task Hijacking in Android

In this section, we first discuss an example showing how an attacker could manipulate the task state transitions to his advantage, causing task hijacking attacks. We then explore the extent of different task hijacking methods and how they can be used for other various attack goals.

4.1 Motivating Example

Suppose attacker's goal is to launch an UI spoofing attack. Specifically, when the user launches a victim app from the home screen, a spoofing activity with an UI masquerading the victim app's main activity (e.g. the login screen of a bank app) shows up instead of the original activity.

Figure 3 shows the task state transitions of the UI spoofing attack. Initially in s_0, the home screen is displayed to the user while a malware task waits in the background. Like the task state transition example just shown in Section 3.3, when the user launches the victim app from the launcher, state transition A is supposed to occur by default, i.e. a new task is created and the app's main activity is displayed on screen. However, as shown in state transition B, the malware can manipulate the task state transition conditions such that the system instead displays the spoofing UI of activity "mal-main" by relocating "mal-main" from the background task to the top of victim app's back stack. The user has no way to detect the spoofing UI since the original activity UI is not shown on screen at all, and the "mal-main" activity appears to be part of the victim app's task (perceivable in recent task list). By this means, the victim task is smoothly hijacked by the malware activity from launch time, and all user behaviors within this task are now under malware's control.

In this example, the attacker successfully misleads the system and launches the spoofing UI by abusing some task state transition conditions, i.e. `taskAffinity` and `allowTaskReparenting`. We will introduce them together with other exploitable conditions/events in

Conditions	
Intent Flags (FLAG_ACTIVITY_*)	*Activity Attribute*
NEW_TASK	
SINGLE_TOP	launchMode
CLEAR_TOP	allowTaskReparenting
REORDER_TO_FRONT	taskAffinity
NO_HISTORY	allowTaskReparenting
CLEAR_TASK	documentLaunchMode (API 21)
NEW_DOCUMENT (API 21)	finishOnTaskLaunch
MULTIPLE_TASK	
Events	
Callback Function	*Framework APIs*
onBackPressed()	startActivity()
	startActivities()
	TaskStackBuilder class

Table 2: Task control knobs - configurable task state transition conditions and events provided by Android.

Section 4.5 and 4.6.

4.2 Adversary Model

We assume the user's Android device already has a malware installed (similar assumptions are made in [8, 25, 34, 38]). The malware pretends to seem harmless, requiring only a minimum set of widely-requested permissions such as INTERNET permission. The attacker's goal is clear: blend the malicious activities with the target app's activities in one task, and intercept the normal user operations to achieve malicious purposes.

4.3 Hijacking State Transition

A *hijacked task state* is a desirable state to attackers, in which at least one task in the system contains both malicious activities (from malware) and benign activities (from the victim app). The task state $s_1^{'}$ in the spoofing attack is an example of hijacked task state. A *hijacking state transition* (HST) is a state transition which turns the tasks in the system to a dangerous hijacked task state, e.g., the task state transition B in the previous example. Conceptually, there are two types of HSTs:

1. The malicious activity gets pushed onto the victim task's back stack (malware⇒victim);
2. The victim app activity is "tricked" by malware and pushed on the malware's back stack (victim⇒malware).

4.4 The Causes of HSTs

Android provides a rich set of task control features, i.e., task state transition conditions and events. We call these features as *task control knobs*. The task control knobs provide app developers with broad flexibility in controlling the launch of new activities, the relocation of existing activity to another task, "back" button behaviors, even the visibility of a task in the recent task list (a.k.a overview screen), etc. Table 2 lists such conditions and events in four categories: activity attribute, intent flags, call-back functions, and framework APIs. All these control flexibility further complicates task state transitions.

Due to HST's potential threats to app and system security, understanding the extent of HSTs in the complex task state transitions becomes important. To achieve this, we simulate the task state transitions in a Android system and try to capture all possible HSTs and hijacked task states that occur during the state transitions.

In theory, there are a huge number of possible task states (each app may have a number of activities, and an activity can be instantiated for multiple times). We confine the number of task states to more interesting cases by adding two constraints: (1) each app only has two activities - the main activity and another public exported activity (can be invoked by other apps), and (2) each activity can only be instantiated once. In the simulation, we specify three apps in the system - namely, Alice, Bob and Mallory (the malware) - as it covers most HST cases.

Given the task states, we generate the task state transition graph by connecting pairs of states with directed edges. For instance, state s_1 and s_2 are connected only if $\exists e \in E, \lambda \in \Lambda$, such that (s_1, s_2, e, λ) or (s_2, s_1, e, λ) are valid transitions, where E denotes all feasible events and Λ represents all possible conditions in Table 2. After constructing the task state transition graph, all hijacked states and HSTs are highlighted. We show a sub-graph of the resulting task state transition graph in Figure 4(a) and visualize the task states in Figure 4(b). For clarity of the presentation, we only show the interesting branches of the over-sized graph and have skipped many duplicated HST cases. Moreover, we zoom in each of the HSTs and show their detailed information in Table 3, including the conditions and events that trigger the HSTs. We manually verify all presented HSTs on real systems and these HSTs are proven to be exploitable to launch real attacks (indicated in the last column in Table 3).

We make two important observations from our result. First, once exploited, the hijacked states shown in Figure 4(a) could result in serious security hazards. For example, HST#3 is the task state transition of our example attack discussed earlier. As a result of this HST, the screen is under attacker's control in state s_{14}. As another example, in HST#2, the benign activity B2 is tricked to be placed in Mallory's task instead of Alice's task during start-up. This can also lead to spoofing attack or GUI confidentiality breaches.

Second, compared with the HST triggered by the system-default conditions and events (e.g., HST#1), more HST scenarios are produced under the configurable conditions and events (HST#2-6) . It means that, by abusing the flexible task control "knobs" readily offered

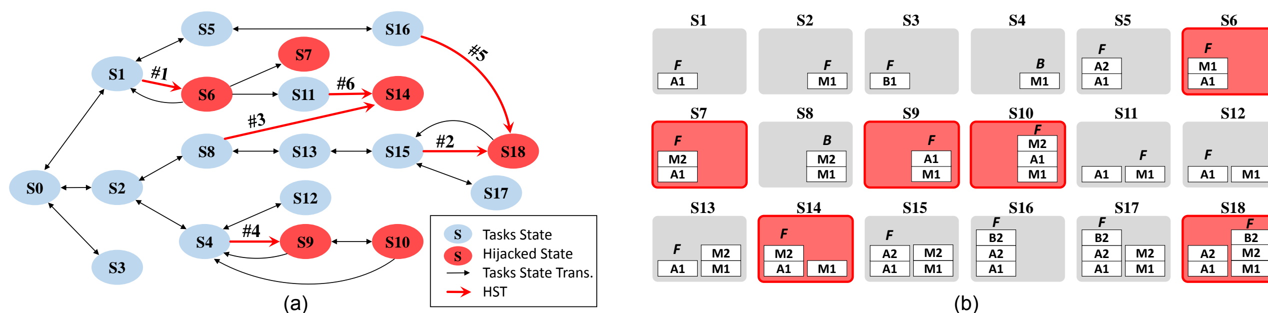

Figure 4: (a) A sub-graph of the over-sized task state transition graph for a simulated system with three apps. The sub-graph shows the typical cases of HSTs (red edges with HST indexes) and the resulting hijacked task states (red nodes). s_0 represents the initial state, i.e., no tasks except the launcher task exists in the system. (b) Visualization of task states of all nodes in figure (a). A, B and M represent the activities from Alice, Bob and Mallory (the malware) respectively. We skip showing the launcher task in the task states. Hijacked states are highlighted as red boxes. F and B denote foreground and background tasks respectively.

HST #	HST Type	Conditions	Events	Attacks in Section 5
1	malware⇒victim	Default	A1: startActivity(M1)	phishing I
2	victim⇒malware	M1:taskAffinity=B2 NEW_TASK intent flag set or B2:launchMode=”singleTask”	A2: startActivity(B2)	phishing II
3	malware⇒victim	M2:taskAffinity=A1; M2:allowTaskReparenting=”true” NEW TASK intent flag set	launcher: startActivity(A1)	spoofing
4	victim⇒malware	M1:taskAffinity=A1; NEW_TASK intent flag set	launcher: startActivity(A1)	denial-of-use; ransomware; spyware
5	victim⇒malware	M1:taskAffinity=B2; B2:allowTaskReparenting=”true”	startActivities([M1, M2]) or use TaskStackBuilder	phishing III
6	malware⇒victim	M2:taskAffinity=A1 NEW_TASK intent flag set or M2:launchMode=”singleTask”	M1: startActivity(M2)	-

Table 3: Detailed information of the HSTs (red edges with HST indexes in Figure 4). E.g., condition “M1:taskAffinity=B2” indicates that the taskAffinity attribute of activity M1 is set to that of B2; Event “launcher:startActivity(A1)” means that activity A1 is started by the launcher.

by the Android system, the attacker can actively create a plethora of HSTs that harm other apps. In Figure 4(a), we only show several typical HST cases, yet there are much more HST instances of these types in the complete state transition graph.

The HST cases and their conditions/events summarized in Table 3 may now look mysterious. We will demystify these conditions and events in the rest of this section.

4.5 Exploiting Conditions

In Table 3, HSTs #2, #4, #6 are similar with respect to their state transition conditions, i.e. all three HSTs occur by virtue of customized activity launch mode (by setting `launchMode` attribute or NEW_TASK intent flag). HSTs #3, #5 are similar as they both use `allowTaskReparenting` attribute to enable activity re-parenting.

4.5.1 Activity Attributes

One can define the attributes [2] of an activity in the `<activity>` element in manifest file. The attributes not explicitly defined are set to default values.

Task Affinity: Task affinity declares what task an activity prefers to join. It is a hard-coded string defined as `<android:taskAffinity="affinity">`, where `affinity` is the task affinity string that can be defined arbitrarily. By explicitly declaring a task affinity, an activity is able to actively “choose” a preferable task to join within its life cycle. If not explicitely specified in the manifest, the task affinity of an activity is the app package name, such that all activities in an app prefer to reside in the same task by default. The affinity of a task is determined by the task affinity of the task’s *root activity* (the activity on the bottom of back stack).

Task affinity is a crucial condition used in most of the HSTs in Table 3. There are two occasions in which an activity can “choose” its preferred host task: (1) when an activity attempts to be started as a new task (i.e., “`singleTask`” launch mode or NEW_TASK intent flag as in HST#2, #4, #6), and (2) if the `allowTaskReparenting` activity attribute is set to true, and another task with the same task affinity is brought to the foreground (as in HST#3, #5). We explain

the above two cases in detail in the following paragraphs.

Launch Mode: Activity launch mode defines how an activity should be started by the system. Based on the launch mode, the system determines: (1) if a new activity instance needs to be created, and (2) if yes, what task should the new instance be associated with. The launch mode can be either statically declared by specifying `<android:launchMode="value">` in the manifest file or dynamically defined using intent flags discussed in Section 4.5.2.

By default, `launchMode="standard"`. In this mode, the AMS would create a new activity instance and put it on top of the back stack on which it is started. It's possible to create multiple instances of the same activity and those instances may or may not belong to the same task. With `launchMode="singleTask"`, the decision-making of activity start-up is more complex. An investigation into Android source code reveals three major steps the AMS takes towards starting an activity. First, if the activity instance already exists, Android resumes the existing instance instead of creating a new one. It means that there is at most one activity instance in the system under this mode. Second, if creating a new activity instance is necessary, the AMS selects a task to host the newly created instance by finding a "matching" one in all existing tasks. An activity "matches" a task if they have the same *task affinity*. After finding such a "matching" task, the AMS puts the new instance into the "matching" task. This explains why in HST #2 and #6, the newly-started and foreground activities (B2 and M2) are put on other "matching" tasks (with the same task affinity) instead of the tasks who start them. Third, without finding a "matching" task, the AMS creates a new task and makes the new activity instance the root activity of the newly created task.

Task Re-parenting: By default, once an activity starts and gets associated with a task, such association persists for the activity's entire life cycle. However, setting `allowTaskReparenting` to true breaks this restriction, allowing an existing activity (residing on an "alien" task) to be re-parented to a newly created "native" task, i.e., a task having the same task affinity as the activity.

For example, in HST#3 resembles the spoofing attack example discussed in Section 4.1. M2 is supposed to stay on Mallory's task at all time. However, M2 has its `allowTaskReparenting` set to true, and `taskAffinity` set to Alice's package name, such that when Alice's task is started (A1 as the root activity) by the launcher, M2 is re-parented to Alice's new task and the user sees M2 on screen instead of A1. In this process, A1 is never brought to the screen at all. Likewise, HST #5 occurs due to similar reason, except that this time the benign activity B2 (with its `allowTaskReparenting` set to true) is re-parented to the malware task.

The above activity attributes offer attackers with great flexibility. The attackers can put their malicious activities to a preferred hosting tasks under certain events, e.g., `singleTask` launch mode during an activity start-up and `allowTaskReparenting` during a new task creation. Furthermore, an activity is free to choose any app as their preferred task owner (including the privileged system apps) by specifying the target app's package name as their task affinity. These conditions lead to a bulk of HSTs in the simulation, and these HSTs can be employed to launch powerful task hijacking attacks as we will see in Section 5.

4.5.2 Intent Flags

Before sending an intent to start an activity, one could set intent flags to control how the activity should be started and maintained in the system by calling `intent.setFlags(flags)`. `intent` is the intent object to be sent, and `flags` is an `int` value (each bit indicates a configuration flag to the AMS).

Noticeably, the `FLAG_ACTIVITY_NEW_TASK` intent flag, if set, lets an activity be started as if its `launchMode="singleTask"`, i.e. the system goes through the same procedures as explained in launch mode to find a "matching" task or create a new task for the new activity instance. This is the dynamic way of setting activity's launch mode. Launcher app always uses this flag to start an app in a new task as in HST#4.

4.6 Exploiting Events

4.6.1 Callback Function

Android framework provides a variety of callback functions for activities to customize their behaviors under particular events, e.g., activity life cycle events (start, pause, resume or stop), key pressing events, system events, etc.

`onBackPressed()` is a callback function defined in `Activity` class, and is invoked upon user pressing the "back" button. The default implementation in framework code simply stops and destroys the current activity, and it then resumes the next activity on top of the current back stack, as we have seen in Section 3.3. However, an attacker can override this callback function for its malicious activity and arbitrarily define a new behavior upon "back" button pressing, or simply disable the "back" button by providing an empty function. As a result, once the malicious activity is brought to the foreground, pressing the "back" button triggers the code of attacker's control.

Figure 5: The process of "back hijacking" phishing attack to a well-known bank app. (a) shows the main activity of the bank app. A new user taps on the tutorial video link in the bank app; In (b), a system dialog prompts the user to choose a video player available in the system; In (c), the video player activity is started, and the user later clicks "back" button, intending to "goes back" to the original main activity; In (d) and (e), the back button directs the user to the phishing UIs, which spoof the user and steal bank account credentials. The phishing activity then quits after user clicks "Sign On"; In (f), the original main activity is resumed, with a log-in failure toast message displayed by the quitting malware.

4.6.2 Framework API

Android framework provides APIs to create new tasks with established back stacks. For example, `TaskStackBuilder` is a utility class that allows an app developer to construct a back stack with specified activities, and to start the back stack as a brand new task in the system at a later time (e.g. using a PendingIntent). Similarly, `startActivities()` in `Activity` class achieves the same thing except that it builds and starts the tasks in one API function call. These framework APIs are helpful for attackers to build and launch new tasks containing designated back stacks without explicitly displaying all activities in the back stacks on screen.

5 Task Hijacking Attack Examples

In this section, we demonstrate more attack examples utilizing exploitable HSTs in Table 3. These attacks can breach the integrity, availability and confidentiality of victim apps' UIs respectively. We have tested these attacks on Android 3.x, 4.x and 5.0.x.

5.1 Breaching UI Integrity

The UI integrity here means the "origin/source integrity" of the victim app's activities, instead of the "data integrity". That is, instead of modifying the original activities of the victim app, attackers deceive the user by spoofing UIs, which can prevent the original UIs from being displayed on screen.

5.1.1 Spoofing Attack

As we have already seen in Section 4.1 and 4.5, by manipulating `allowTaskReparenting` and `taskAffinity`, an attacker can successfully hijack a new task with a spoofing activity. This attack affects all apps on device including the most privileged system apps (e.g., Settings). The attacker can even target multiple apps on user device at the same time, as long as the background malware tasks (targeting different task affinity) are started in advance.

Stealthiness: In order to make the spoofing attack more stealthy, the attacker could take advantage of other task transition conditions and events to achieve this. For example, the attacker can make its background malware tasks absent from the recent task list by setting the activity attribute `excludeFromRecents` to true. As another example, the user may accidentally resume the app's original activity (the root activity of victim app's task) by clicking the "back" button from the on-screen spoofing activity. To prevent users from observing this abnormal app behavior, the attacker can override `onBackPressed()` of the spoofing activity, bringing the home screen back to the foreground, such that it gives the user an illusion that it is in coherence with the system's default "back" behavior.

5.1.2 Phishing Attack - "Back Hijacking"

The back button is popular with users because it allows users to navigate back through the history of activities. However, attackers may abuse the back button to mislead the user into a phishing activity.

We devise three phishing attack methods that target the same banking app, and demonstrate two of them in this paper. Figure 5 shows the screen shots of the phishing attack process. The phishing UIs show up when the user returns from a third-party app activity, and the user unwittingly believes that he/she has returned to the original bank activity.

Figure 6 shows the state transition diagrams of two attack methods. The two attack methods differ in that,

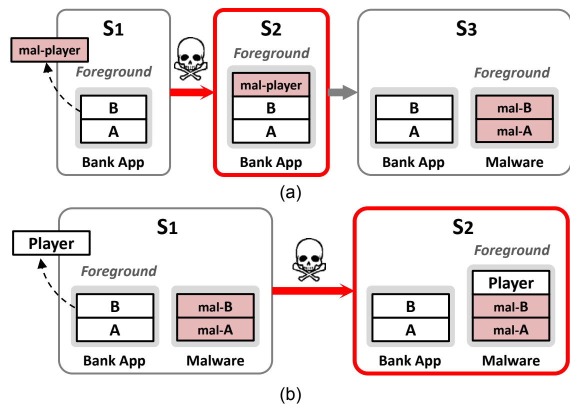

Figure 6: Tasks state transition diagrams of "back hijacking" attacks. Figure (a) and (b) shows method I and II respectively.

user chooses a malicious video player in the first attack, while in the second attack, even though the user chooses a benign player, the bank task can still be hijacked when the user launches the video player.

Method I: Figure 6(a) shows the state transition diagram of the first attack method. We skip the unrelated task(s) (e.g. launcher) in the system and only show tasks of interest. In s_1, the bank app task contains activities A and B, in which B is the login activity. The HST occurs in $s_1 \rightarrow s_2$, triggered by the event that the user clicks the tutorial video from the login UI, sending out a implicit intent to look for an exported activity in the system capable of playing the tutorial video. Unfortunately, the user selects the malicious video player activity "mal-player" from the system pop-up and this results in the hijacked state s_2. After user finishes watching the video, $s_2 \rightarrow s_3$ is triggered by user pressing the "back" button. However, the "back"-pressing event is modified by overriding `onBackPressed()` in the "mal-player" activity. As a result, instead of resuming activity B, a new malicious task is created (by using `TaskStackBuilder`) and brought to the front. As can be seen, the HST takes place under default conditions as in HST#1 (in Table 3).

The user session is hence hijacked to the malware task, which contains "mal-A" and the foreground "mal-B" phishing activities. Note that in this attack, the malware need to camouflage as a useful app (e.g. a video player in this case) that users are likely to use.

Method II: As shown in Figure 6(b), the same phishing attack can succeed even when the user selects a benign video player. In s_1, a malware task with two phishing activities lurks in the background. Similarly, HST occurs in $s_1 \rightarrow s_2$, when the user launches a benign video player. However, as shown in the resulting state s_2, instead of joining the banking task, the new video player activity is pushed in the malware task's back stack, such that pressing the "back" button after the video play resumes the phishing activity "mal-B".

This HST is similar to HST#2 (in Table 3) in that the benign video player attempts to be started as a new task, either because of the `NEW_TASK` flag set in the intent by the bank activity, or the "singleTask" launch mode set by the video player. Furthermore, the existing malware task has its `taskAffinity` maliciously set to the benign video player.

Stealthiness: We employ similar methods in the previous spoofing attack to ensure the stealthiness of the background malware tasks in both phishing attack methods. Moreover, we disable the animation of task switching, producing an illusion to the user that the screen transition is within the same task/app.

5.2 Breaching UI Availability

Task hijacking can also be leveraged to restrict the availability of an app's UI components, or in other words, to prohibit user access to part or all functionality of an victim app.

5.2.1 Preventing Apps from Being Uninstalled

In this example, the attacker is able to completely prevent apps from being uninstalled.

Ways to Uninstall An App: There are generally three ways for a user to uninstall an app from the device: (1) uninstall from the system Settings app; (2) dragging the app icon to the "trash bin" on home screen; or (3) uninstall with the help of a third-party app, e.g. an anti-virus app. In these scenarios, the Settings, Launcher, and the third-party apps will respectively generate an request to uninstall the app. Such a request eventually reaches the system package installer, which has the exclusive privilege to install/uninstall apps. Upon receiving the request, package installer pops up a dialog for the user to confirm. The dialog itself is an activity (namely uninstaller activity) from the system package installer and is pushed in the back stack of whoever is making the request (e.g. s_4 in Figure 7). No app can be uninstalled without user confirmation on the uninstaller activity.

Attack Method: The attacker can prevent app uninstallation by restricting user access to the uninstaller activity when it shows up on screen. In this attack, once the uninstaller is found to be in the foreground, a malicious activity is immediately pushed on top of the uninstaller activity in the same back stack, such that the uninstaller is "blocked" and becomes inaccessible to the user.

Figure 7 shows the state transition diagram of this attack targeting Settings app. Similar methods can be easily adopted to block app un-installation from the launcher or the anti-virus apps (e.g. when malware is detected).

In s_1, a task with only one root activity ("mal-root") from the malware is waiting in the background,

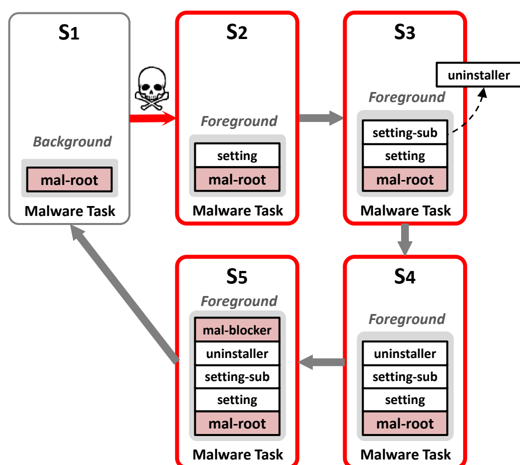

Figure 7: Tasks state transition diagram of application uninstall prevention attack.

with its `taskAffinity` set to the Settings app (com.android.settings). The HST occurs in $s_1 \rightarrow s_2$, triggered when the user opens up the Settings from the home screen (we skip Launcher task in the figure). In s_2, instead of hosting the newly-created "setting" activity in a new task, "setting" activity is pushed on top of the malware's back stack because the it is started by the launcher with a `NEW_TASK` flag. As a result, upon start-up, the privileged Settings app is unwittingly "sitting" on a task owned by the malware. This is similar to HST#4 in Table 3.

The user then goes through a few more sub-setting menus to find the app (as shown in s_3) and clicks the uninstall button, after which the uninstaller activity shows up for user confirmation (as shown in s_4). Once this happens, a malicious activity namely "mal-blocker" is immediately (even without user awareness of the uninstaller dialog) launched by a malicious background service, which keeps monitoring the foreground activity. The "mal-blocker" activity, started by a `NEW_TASK` flagged intent and with the same task affinity as the Settings app, is thus pushed in the same task, and effectively blocks the uninstaller as shown in s_5. The "mal-blocker" activity has its "back" button disabled, such that the user has no way to access the uninstaller activity right below it in the back stack whatsoever, and thus cannot confirm the app uninstalling operation.

In fact, the "back" button of "mal-blocker" is not only disabled, but is also augmented with a new event that triggers $s_5 \rightarrow s_6$: invoking (call `startActivity()`) the "mal-root" activity with an intent having `CLEAR_TOP` flag set, which results in the killing of the uninstaller and Settings activities in the task.

Preventing Un-installation from adb: An advanced user may resort to *Android Debug Bridge* (adb), a client-server program used to connect Android devices from a computer, and uninstall the malware from adb. However, in order to use adb, the user needs to first enable USB debugging in the Settings. The malware can block it in the Settings using similar technique and prevent the use of adb, as long as the USB debugging is not enabled before the attack (which is the case for most normal users).

5.2.2 Ransomware

Ransomware blackmails people for money in exchange of their data, and it has recently hit Android in a large scale [5]. The attackers may use UI hijacking to implement ransomware.

The malicious background service mentioned above takes the following two responsibilities and is difficult to be completely stopped. (1) Assure the malicious root activity ("mal-root") is alive: it re-creates a new root activity once the activity is found to be destroyed; and (2) monitor the foreground activity: if the target activity shows up, it immediately starts "mal-blocker" to block user access to the target activity, as we have seen in $s_4 \rightarrow s_5$. To prevent itself from being killed, the service registers itself in the system alarm service, who fires a pending intent in every given fixed time interval, relaunching the service if it is found to be killed.

By this mean, the ransomware is able to restrict user access to any target apps of attacker's choice, and can potentially render the Android device completely useless.

5.3 Breaching UI Confidentiality

The attack method in Section 5.2 can also be deployed to devise a new spyware, namely "TaskSpy" capable of monitoring the activities within any tasks in the newest Android 5.0.x systems (API 21), without requiring any permissions.

In Android, the system regards the owner of the root activity in a back stack to be the owner of the corresponding task. Android 5.0 allows an app to get the information of the caller app's own tasks (including the activities in the tasks) without requiring any permission. It means that, if a spyware can "own" the tasks of all the apps it intends to spy on, it is able to get the information of these tasks that in fact contain the victim apps' activities. Task hijacking is especially useful to "TaskSpy" in this case. In other words, "TaskSpy" can use the HST presented in Section 5.2 to "own" the tasks of any victim apps and thus stealthily spy on their activities without using any permission. Chen et. al. have achieved the same goal in their work [8] by monitoring and interpreting the shared VM information via public side channels. Compared with their attack, task hijacking can do this in a more direct and reliable way on Android 5.0.x.

Vul. app	Atk #	Vul. conditions	% of vul.	Tol. % of vul.
V	I	Send implicit intent for exported activities	93.9	93.9
	II	Send implicit intent for exported activities and use intent flag NEW_TASK	65.5	
S	II	Contains public exported activity and lauchMode="singleTask"	14.2	14.4
	III	Contains public exported activity and allowTaskReparenting="true"	1.4	

Table 4: Percentage of vulnerable victim apps (V) and "service" apps (S) to the "back hijacking" phishing attacks respectively, among 10,985 most popular Google Play apps.

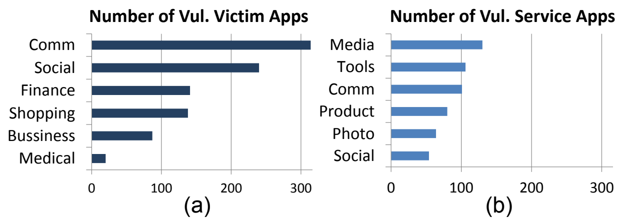

Figure 8: (a) Breakdown of vulnerable victim apps in security-sensitive app categories. (b) Breakdown of vulnerable "service" apps in the most widely useful app categories.

6 Evaluation

We first seek to understand the extent of vulnerable systems and apps to the attacks we have presented in Section 5. By doing large-scale app analysis across various markets, we then provide the current use status of the task control knobs in real implementations. Base on our insights from the result, we provide mitigation suggestions to defend against task hijacking threats in Section 7.

6.1 Vulnerability Analysis

Vulnerable Android Versions: We say an Android version is vulnerable to a particular attack if a malware can successfully launch the attack to a victim app on the system. Since the unique multitasking is part of Android design and most features have been introduced early in Android's evolution, we find that recent Android versions, including 3.x, 4.x and 5.0.x, are vulnerable to all our presented attacks, except the "TaskSpy" attack. As discussed in Section 5.3, "TaskSpy" relies on specific APIs introduced from API 21, and therefore, only affects the newest Android 5.0.x systems.

Apps Vulnerable to Task Hijacking Attacks: As summerized in Table 1, all the apps installed on a vulnerable Android system (including the privileged system apps) are vulnerable to all the attacks presented in this paper, except the "Back Hijacking" phishing attacks, which require certain prerequisites for an app to be vulnerable. Despite the prerequisites, the "Back Hijacking" phishing attacks are extremely stealthy, can be easily crafted and can cause serious consequnces. We try to further understand the scale of apps vulnerable to the "Back Hijacking" phishing attack by analyzing the most popular apps in Google Play.

Apps Vulnerable to "Back Hijacking": In a phishing attack, the attacker would be likely to target the most popular and valuable apps. Therefore, we focus our vulnerability analysis on the most popular 10,985 apps from Google Play, i.e., apps with over 1 million installs.

We indicate a vulnerable app in the phishing attacks to be of either one or both of the following two types: (1) victim app - the target victim app of the phishing attack (e.g. the bank app); and (2) "service" app - the benign app that provides publicly exported activities and is exploitable by the attacker to conduct user phishing on the victim apps (e.g. the benign video player). We do static analysis on the apps. Specifically, we perform inter-procedural analysis to identify all implicit intents (without permissions guarded) and the associated flags, and conduct manifest scan to find all activity attributes and public exported activities (excluding the main activities which are always exported). Table 4 lists the vulnerability conditions, and shows the percentages of both vulnerable victim apps and "service"apps to each and all the attack methods respectively.

As can be seen, 93.9% of the most popular apps in Google Play are vulnerable. This is partially because most apps would send out implicit intents (without permissions guided), which could potentially invoke a malware activity as in attack I. By taking a closer look at the results, among these apps, a majority (65% of apps) are vulnerable to attack II, i.e., they are vulnerable to phishing attack even if users launch trusted benign "service" apps from these apps. Moreover, 14.36% "service" apps can be exploited to "help" attack the apps who invoke these "service" apps, even if the apps being attacked may not be vulnerable by themselves.

The consequence and severity of a phishing attack depend on the content and sensitivity of the stolen information. To have a rough idea of the potential consequences caused by the "Back Hijacking" phishing attacks, we selectively show in Figure 8(a) the population of vulnerable victim apps in a few security-sensitive app categories. Noticeably, We observe that a significant number of security-sensitive apps are vulnerable, including the financial apps like banking and credit card payment (e.g., Citibank, Chase, Google Wallet), the most popular communication and social media apps (e.g. Google Hangouts, facebook), and shopping apps from the ma-

Activity Attribute	% of Apps
allowTaskReparenting="true"	0.80
launchMode="singleTask"	24.63
launchMode= other non-default modes	24.75
taskAffinity= own pck. name	2.36
taskAffinity= other	1.60
excludeFromRecents="true"	12.45
alwaysRetainTaskState="true"	2.03

Intent Flag	% of Apps
NEW_TASK	79.42
CLEAR_TOP	37.59
EXCLUDE_FROM_RECENTS	10.08
Events	
onBackPressed()	62.00
TaskStackBuilder	7.27
startActivities()	5.47

Table 5: Percentage of 6.8 million market apps that use each of the "security-sensitive " task control knobs.

Task Affinity	# of Apps
com.android.settings	492
com.android.camera	325
com.android.update	279
com.tencent.mm	273
com.gau.go.launcherex	237
com.fractalist	194
com.android.activity	158
com.xiaomi.payment	147

Table 6: Top package names specified as the task affinity by other apps

jor electronic commerce companies (e.g. Ebay, Amazon Shopping), etc. Similarly, in Figure 8(b), we show the statistics of a few app categories in which the vulnerable "service" apps and their functionality are most widely used, including the most famous photo editing tools, document editors, and file sharing services, etc.

6.2 Market-scale Study on the Use of Task Control Knobs

Due to the task hijacking threats, we have a pressing need for a defense strategy that can mitigate these threats while minimizing the side effects on Android multitasking features. To this end, it is important to first understand the current status about the use of Android multitasking features in real implementation, especially the use of "security-sensitive" task control knobs.

We analyzed 6.8 million Android apps from a variety of markets including Google Play and other 12 popular third-party app markets worldwide (e.g., from China). The analysis does not include duplicated apps (apps with same package name, public key certificate and app version number) distributed across multiple markets.

Table 5 shows the percentage of apps that use each of the task control knobs respectively. As shown in the table, a majority of the task control features are popular with app developers and users. For example, "singleTask" launch mode and NEW_TASK intent flag are used in a significant portion of apps to control the association of new activities with tasks. The flexibility of "back" button customization is widely adopted (as high as 62% apps). One reason is that the `onBackPressed()` callback function is heavily used by ad libs (which embed ads in app activities) for data clean-up before the activities are destroyed. In addition, a significant portion of activities can hide their associated tasks from the overview screen (by defining "excludeFromRecents" attribute or setting EXCLUDE_FROM_RECENTS intent flag).

Case Study - Task Affinity: Since task affinity can be abused in the most dreadful attacks, we are particularly interested in its use. 3.96% apps we studied explicitly declare task affinity. A considerable portion (1.6% of all apps) set their activities' taskAffinity string without containing their own package names. It means that, if there are task affinity conflicts, these 1.6% apps (totally 109 thousand apps) may interfere with the multitasking behaviors of one another. They may even affect other apps if the task affinity attributes are intentionally set to the package name of other apps (recall that the taskAffinity string can be set arbitrarily). We are especially interested in the latter case, and in our analysis, we find a total of 3293 apps of this kind. Table 6 lists the top package names designated as task affinity by these apps.

By reverse engineering a number of these apps, we find that intentionally setting the task affinity as another app is particularly useful in a class of "plug-in" apps, i.e. apps that provide complementary features to existing (and usually popular) apps just like a web browser's plug-ins (except that here the "plug-in" itself is implemented in a separate app). By being in the same task with the popular app, the "plug-in" app can change normal user experience and fulfill its feature functionality in the context of the app it serves. For example, an phone call recorder app namely FonTel can display an array of buttons on screen whenever there is a phone call, letting users to control phone call recording. The control buttons are contained in an mostly transparent activity. By setting the task affinity of the activity to `com.android.phone`, it can be pushed on top of the Android telephony task when a phone call occurs, such that users can access both the recording control buttons and telephony activity at the same time.

In summary, despite the security risks, Android multitasking features are popular with developers and even become indispensable to the normal functions of a significant number of apps that provide favorable features.

7 Defense Discussion

Given the pervasive use of the "security-sensative" task control features, simply disabling these features would greatly hurt app functions and user experience. Mitigating the task hijacking threats become a trade-off between

app security and multitasking features.

7.1 Detection in Application Review

Existing app vetting processes such as Bouncer [31] may conduct a inspection over the "sensitive" task control knobs, a light-weight defense strategy without significantly affecting existing multitasking features.

However, specifying a guideline balancing the security/feature trade-off is non-trivial. For example, a tentative guideline could be: `taskAffinity` attribute should be specified in a strict format, e.g., with app package name followed by developer-defined affinity name (now task affinity can be any string); and the task affinity should not contain any other app's package name, except that the two apps are from the same developer. This effectively eliminates a big portion of hijacking state transitions where a malicious activity specifies the victim app as its preferred affinity. However, this rule also restricts useful features and contradicts with an important principle of Android multitasking design - give an activity the freedom to live in its preferred task even though they are from different apps. This contradiction cannot be solved by app review alone in this case. We need system support together with app review to achieve a good balance of security/feature trade-off.

Moreover, detecting problematic events can be sometimes difficult for the app review. For instance, one could confine the behaviors in `onBackPressed()`, preventing it from generating potential hijacking transition event. However, discovering all possible program behaviors using static analysis is an undecidable problem. A skillful attacker can replace class methods (`onBackPressed()` method in `Activity` class) with another method by changing Dalvik internals using native code during runtime, and static analysis does not know this by simply looking at the original `onBackPressed()` method. Dynamic analysis is of little help as well since this behavior can be triggered only after passing the app review.

As a result, completely mitigating task hijacking risks and without affecting existing features in app review remains challenging.

7.2 Secure Task Management

An alternative approach involves security enhancement to the task management mechanism of Android system.

A more secure task management could introduce additional security guides or logic, which draws developers' awareness of the security risk and limits the attacker surface. Take the above task affinity for example, an additional boolean attribute can be introduced for each app to decide if it allows the activities from other apps to have the same affinity as the app. If the boolean is "false" (also by default), the system would not unconditionally relocate the "alien" activities to the app's task or vice versa, even though the "alien" activities declare to have the same task affinity as the app. Likewise, a finer-grained boolean attribute can be further employed for `allowParentReparenting` attribute - determining if to allow "alien" activities to be re-parented to the app's task (even though defining the same task affinity is permitted). For other "security-sensative" features, we suggest first consider the same approach. Considering the serious security hazards that can be prevented, it is well worth of making such changes. At the very least, enhanced security scheme like this has to be applied to assure the security of the most privileged system apps.

Completely defeating task hijacking is not easy. As we have discussed in the last section, it is difficult to identify the exact behavior of pressing "back" in an activity during app review phase. For these popular and security-sensitive features, more powerful runtime monitoring mechanism is required to fully mitigate task hijacking threats.

In summary, we advocate future support for security guidance and/or mechanism, which can protect Android apps from task hijacking threats and bring along a both secure and feature-rich multitasking environment for Android users and developers.

8 Related Work

GUI security : GUI security has been extensively studied in traditional desktop and browser environments [14, 29], e,g., UI spoofing [9], clickjacking [3, 17], etc. Android, on the other hand, is unique in the design of its GUI sub-systems. It has been shown that the GUI confidentiality in Android can be breached by stealthily taking screen shots due to adb flaws [22], via embedded malicious UIs [28, 24], or through side channels, e.g. shared-memory side channel [8] or reading device sensors information [25, 34]. In contrast to existing work, this paper focuses on the fundamental design flaws of the task management mechanism (supported by the AMS), the control center that organizes and manages all existing UI components in the Android system.

Android Vulnerability: The security threats in the inter-component communication (ICC) has been widely studied [13, 23, 10, 20, 32]. Moreover, there has been considerable prior work on emerging Android vulnerabilities and their mitigation measures in many aspects [38, 40, 18, 33, 27, 7, 30, 15, 21]. However, the critical Android multitasking mechanism and the feature provider, the AMS, have not been deeply studied before. This paper fills in this gap by systematically studying the An-

droid multitasking and the security implications of this design.

Android Malware: Many prior efforts focus on large-scale detection of malicious or high-risk Android apps [39], e.g., fingerprinting or heuristic-based methods [26, 41, 16], malware classification based on machine learning techniques [37, 6], and in-depth data flow analysis for app behaviors [11, 35, 36, 6]. The attack surface discovered in this paper can be easily employed by attackers to create a wide spectrum of new malwares, as discussed in Section 5. We report our threat assessment based on over 6 million market apps and provide defense suggestions in order to prevent the outburst of task hijacking threats in advance.

9 Conclusion

This paper systematically investigated the security implications of Android task design and task management mechanism. We discover a plethora of task hijacking opportunities for attackers to launch different attacks that may cause serious security consequences. We find that these security hazards can affect all recent versions of Android. Most of our proof-of-concept attacks are able to attack all installed apps including the most privileged system apps. We analyzed over 6.8 million apps and found task hijacking risk prevalent. We notified the Android team about these issues and we discussed possible mitigation techniques.

10 Acknowledgment

We would like to thank anonymous reviewers whose comments help us improve the quality of this paper. We thank Dr. Sencun Zhu and Dr. Dinghao Wu from Pennsylvania State University for providing valuable feedback. Chuangang Ren was supported in part by ARO W911NF-09-1-0525 (MURI). Peng Liu was supported by ARO W911NF-09-1-0525 (MURI) and ARO W911NF-13-1-0421 (MURI).

References

[1] Task and Back Stack. http://developer.android.com/guide/components/tasks-and-back-stack.html.

[2] App Manifest. http://developer.android.com/guide/topics/manifest/activity-element.html.

[3] AKHAWE, D., HE, W., LI, Z., MOAZZEZI, R., AND SONG, D. Clickjacking Revisited: A Perceptual View of UI Security. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)* (2014).

[4] Android Activity. http://developer.android.com/reference/android/app/Activity.html.

[5] Simplocker: First Confirmed File-Encrypting Ransomware for Android, 2014. http://www.symantec.com/connect/blogs/simplocker-first-confirmed.

[6] ARP, D., SPREITZENBARTH, M., HUBNER, M., GASCON, H., AND RIECK, K. Drebin: Effective and Explainable Detection of Android Malware in Your Pocket. In *Proceedings of Network and Distributed System Security Symposium (NDSS)* (2014).

[7] CHEN, E., PEI, Y., CHEN, S., TIAN, Y., KOTCHER, R., AND TAGUE, P. OAuth Demystified for Mobile Application Developers. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)* (2014).

[8] CHEN, Q. A., QIAN, Z., AND MAO, Z. M. Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks. In *Proceedings of the USENIX Security Symposium* (2014).

[9] CHEN, S., MESEGUER, J., SASSE, R., WANG, H., AND WANG, Y. A Systematic Approach to Uncover Security FLaws in GUI Logic. In *Proceedings of IEEE Symposium on Security and Privacy (S&P)* (2007).

[10] CHIN, E., FELT, A. P., GREENWOOD, K., AND WAGNER, D. Analyzing Inter-Application Communication in Android. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)* (2011).

[11] ENCK, W., GILBERT, P., CHUN, B., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2010).

[12] FELT, A. P., CHIN, E., HANNA, S., SONG, D., AND WAGNER, D. Android Permissions Demystified. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)* (2011).

[13] FELT, A. P., WANG, H. J., MOSHCHUK, A., HANNA, S., AND CHIN, E. Permission Re-delegation: Attacks and Defenses. In *Proceedings of the USENIX Security Symposium* (2011).

[14] FESKE, N., AND HELMUTH, C. A Nitpickers Guide to a Minimal-complexity Secure GUI. In *Proceedings of Annual Computer Security Applications Conference (ACSAC)* (2005).

[15] GRACE, M., ZHOU, Y., WANG, Z., AND JIANG, X. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *Proceedings of Network and Distributed System Security Symposium (NDSS)* (2012).

[16] GRACE, M., ZHOU, Y., ZHANG, Q., ZOU, S., AND JIANG, X. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)* (2012).

[17] HUANG, L., MOSHCHUK, A., WANG, H. J., SCHECHTER, S., AND JACKSON, C. Clickjacking: Attacks and Defenses. In *Proceedings of the USENIX Security Symposium* (2012).

[18] JIN, X., HU, X., YING, K., DU, W., AND YIN, H. Code Injection Attacks on HTML5-based Mobile Apps: Characterization, Detection and Mitigation. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)* (2014).

[19] K. W. Y. AU AND Y. ZHOU AND Z. HUANG AND D. LIE. PScout: Analyzing the Android Permission Specification. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)* (2012).

[20] KANTOLA, D., CHIN, E., HE, W., AND WAGNER, D. Reducing attack surfaces for intra-application communication in android. In *Proceedings of the ACM workshop on Security and privacy in smartphones and mobile devices (SPSM)* (2012).

[21] Li, T., Zhou, X., Xing, L., Lee, Y., Naveed, M., Wang, X., and Han, X. Mayhem in the Push Clouds: Understanding and Mitigating Security Hazards in Mobile Push-Messaging Services. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)* (2014).

[22] Lin, C., Li, H., Zhou, X., and Wang, X. Screenmilker: How to Milk Your Android Screen for Secrets. In *Proceedings of Network and Distributed System Security Symposium (NDSS)* (2014).

[23] Lu, L., Li, Z., Wu, Z., Lee, W., and Jiang, G. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)* (2012).

[24] Luo, T., Hao, H., Du, W., Wang, Y., and Yin, H. Attacks on WebView in the Android System. In *Proceedings of Annual Computer Security Applications Conference* (2011).

[25] Miluzzo, E., Varshavsky, A., and Balakrishnan, S. TapPrints: Your Finger Taps Have Fingerprints. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)* (2012).

[26] Peng, H., Gates, C., Sarma, B., Li, N., Qi, Y., Potharaju, R., Nita-Rotaru, C., and Milloy, I. Using Probabilistic Generative Models for Ranking Risks of Android Apps. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)* (2012).

[27] Poeplau, S., Fratantonio, Y., Bianchi, A., Kruegel, C., and Vigna, G. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *Proceedings of Network and Distributed System Security Symposium (NDSS)* (2014).

[28] Roesner, F., and Kohno, T. Securing Embedded User Interfaces: Android and Beyond. In *Proceedings of the USENIX Security Symposium* (2013).

[29] Shapiro, J., Vanderburgh, J., Northup, E., and Chizmadia, D. Design of the EROS Trusted Window System. In *Proceedings of the USENIX Security Symposium* (2004).

[30] Sounthiraraj, D., Sahs, J., Greenwood, G., Lin, Z., and Khan, L. SMV-HUNTER: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps. In *Proceedings of Network and Distributed System Security Symposium (NDSS)* (2014).

[31] Android and Security, 2012. `http://googlemobile.blogspot.com/2012/02/android-and-security.html`.

[32] Wei, F., Roy, S., Ou, X., and Robby. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)* (2014).

[33] Wu, L., Grace, M., Zhou, Y., Wu, C., and Jiang, X. The Impact of Vendor Customizations on Android Security. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)* (2013).

[34] Xu, Z., Bai, K., and Zhu, S. TapLogger: Inferring User Inputs on Smartphone Touchscreens Using On-board Motion Sensors. In *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Netowrks (WiSec)* (2012).

[35] Yang, W., Xiao, X., Andow, B., Li, S., Xie, T., and Enck, W. AppContext: Differentiating Malicious and Benign Mobile App Behavior under Contexts. In *Proceedings of International Conference on Software Engineering (ICSE)* (2015).

[36] Yang, Z., Yang, M., Zhang, Y., Gu, G., Ning, P., and Wang, X. S. AppIntent: Analyzing Sensitive Data Transmission in Android for Privacy Leakage Detection. In *Proc. CCS13*.

[37] Zhang, M., Duan, Y., Yin, H., and Zhao, Z. Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs. In *Proc. CCS14*.

[38] Zhou, X., Demetriou, S., He, D., Naveed, M., Pan, X., Wang, X., Gunter, C., and Nahrstedt, K. Identity, Location, Disease and More: Inferring Your Secrets from Android Public Resources. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)* (2013).

[39] Zhou, Y., and Jiang, X. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of IEEE Symposium on Security and Privacy (S&P)* (2012).

[40] Zhou, Y., and Jiang, X. Detecting Passive Content Leaks and Pollution in Android Applications. In *Proceedings of Network and Distributed System Security Symposium (NDSS)* (2013).

[41] Zhou, Y., Wang, Z., Zhou, W., and Jiang, X. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proceedings of Network and Distributed System Security Symposium (NDSS)* (2012).