Bachelor Thesis Proposal on Hardening the Task Management in Android

Mirko Meinerzag
Saarland University
Saarbrücken, Germany
s8mimein@stud.uni-saarland.de

***Abstract*—Android's user interface has been frequently targeted by malware to perform attacks like phishing, denial-of-service, and more. These attacks often need little to no extra permissions but have devastating consequences for the user. One particular attack is called task hijacking. Task hijacking abuses the task management of Android to compromise the UI of benign applications. This can then be used to launch follow-up attacks that leak sensitive information or deny crucial services.**

In Android 12 a new API was introduced that prevents third-party overlays from being displayed over the UI. This way developers can secure their apps and UI against phishing attacks based on overlays. However, phishing by abusing task hijacking is not prevented with this API. This work proposes a similar approach against task hijacking. By extending the Android framework, App developers can decide, whether other (potentially malicious) apps can interfere with their tasks or not.

I. Introduction

Phishing attacks are a common threat in Android. There have been several phishing attacks that target the user interface(UI) to trick the user into leaking sensitive data, for example, credentials or credit card information. One way to perform such an attack is to abuse the SYSTEM_ALERT_WINDOW permission. When an app has this permission, it can overlay the UI of another app with arbitrary content. This opens many ways for malware to abuse it. For example, in the Cloak&Dagger attack[1] this permission is used to trick the user into granting an even more dangerous permission, giving access to accessibility features. This gives the app complete control of the UI feedback loop, potentially leaking passwords and accessing other sensitive resources. Attacks that abuse overlays have been tackled in Android 12 by giving app developers more control over when they want to allow overlays on their apps. However, this is not the only attack on the UI to perform phishing. Another attack vector is the task management of Android. Tasks are used to store UI components of apps. These components are called *activities*. Android enables app developers to decide on which task their activities should be stored. The plethora of features that can be used to manipulate the default task management opened ways for a new kind of attack: task hijacking.

The general idea of task hijacking is to infiltrate the task of a benign application with a malicious activity or to put a benign activity on a task controlled by the malicious application. Androids sandboxing mechanism isolates each app from another and creates a process-based boundary enforced by the kernel. However, the task management of Android allows activities from different apps to reside in the same task. This can be abused to launch several attacks including phishing, spoofing, denial-of-service, and more. For example, Mallory could create an activity that looks just like the Facebook login screen. She sets the task affinity of the activity to that of Facebook and allows the activity to be reparented by setting the allowTaskReparenting attribute. Now she only needs to start the activity in the background and wait. If the user starts Facebook, Mallory's activity will be reparented to the task of Facebook, resulting in a hijacked task and breaking the integrity of the UI. The unaware user will now put his credentials into the malicious activity, leaking his password to Mallory. The attack can be further disguised by hiding the activity from the recent screen or showing a toast that the login has failed and redirecting the user to the actual login activity.

The above example shows that there is a need for more secure task management. Similar to the approach by Android regarding overlay attacks, Chuangang Ren et al[2] proposed that developers should have more control over their own tasks and activities. The manifest could be extended to provide additional attributes for activities that determine if an activity is allowed to be reparented or to forbid other activities on the own task completely. Details on how to achieve this are provided in section IV. The next section looks into related work.

II. Related Work

The UI of Android has been the target for different types of malware. Click- or tapjacking is one of these threats. This kind of malware abuses the SYSTEM_ALERT_WINDOW permission to overlay the UI of other applications. Cloak&Dagger abuses overlays to trick the user into granting the BIND_ACCESSIBILITY_SERVICE permission. With this permission, the malware can then use accessibility features, which are intended to help disabled users. These features are exploited to capture user input and leak sensitive information.

To counter phishing and spoofing on the web, an indicator next to the URL ensures that the host of the web service corresponds to the one that is being depicted. This is done with HTTPS and certificate verification. A similar approach is taken by Bianchi et al[3]. They extended the navigation bar with an indicator to show the user which app he is currently interacting with. Furthermore, they provide a tool that can analyze an app and determine, if the app is using APIs which are used to hijack a task. However, their tool does not differentiate between benign or malicious use cases. Another approach to tackle task hijacking is to use dynamic instead of static analysis.

WindowGuard[4] performs dynamic analysis on a user session and warns the user if a potential task hijacking occurred.

III. Background Information

This section provides the necessary background information to better understand the Android multitasking mechanism. First, basic Android terms and mechanisms are covered. Then, we will look at the complex task management of Android. And last we will show how this opens ways for task hijacking and further investigate the threat it poses.

A. Android Basics

In Android, applications follow the *Principle of Least Privilege*. Each app is isolated in its sandbox. This means an app cannot directly access the resources of another app. The sandbox is enforced by the underlying Linux kernel, where every app runs in its process under a different Linux UID. Furthermore, an app can only perform sensitive operations, if it holds the necessary permission for these operations. If an app needs a permission, it must request it in its *manifest*. Some dangerous permissions also need User confirmation before being granted to the app. This is the foundation of app security in Android.

An Android app is defined by its components. There are four main components that can be used: *Activity*, *Service*, *Broadcast Receiver* and *Content Provider*. The component of most interest for us is the activity component. Activities provide a user interface(UI) the user can interact with. An app has typically more than one activity, each for a different purpose. For instance, an email app could have one activity to provide a login form, another to write emails, and one to view and read all received emails. All activities of an app must be declared in the app's manifest. The developer can set several attributes for an activity in the manifest. For example, one important attribute for this work is the *taskAffinity* attribute, which specifies the task an activity prefers to reside in. More attributes will be discussed further below. Activities can communicate with other components by using *intents*.

Intents are used by apps to describe an operation that needs to be performed, e.g starting activities or sending a broadcast. There are two different kinds of intents: explicit and implicit intents. An explicit intent exactly describes which application will be called to perform the operation, while an implicit expresses a general action that needs to be done. Another app can then declare an *intent filter* to show that it can handle a certain action. If there is more than one app that can handle an implicit intent, the system will prompt the user to choose one of the available apps to perform the operation. There are several *intent flags* that can be set to further specify the operation. For example, if the flag FLAG_ACTIVITY_NEW_TASK is set on an intent that wants to start an activity, the activity will be pushed onto a new task instead of the task it was started on. This is the case when an activity is started from the home screen. Next, we will look at how activities are started and how they are managed throughout their life cycles.

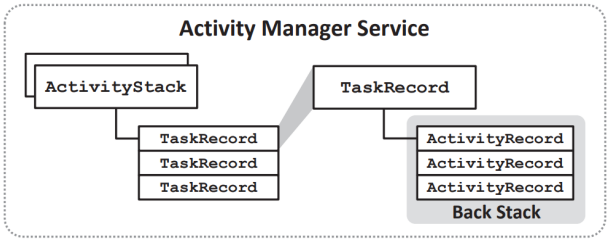

Figure 1.

B. Task Management in Android

The system service responsible for task management is called *Activity Manager Service(AMS)*. When starting an activity with the startActivity() API, the AMS will handle the launch of the activity based on the intent passed as an argument. The opened activity is then stored in a *task*. A task represents a collection of activities from one or more apps. Inside a task, activities are pushed on the *back stack* of that task. The activities in the back stack are ordered in the sequence of their launch. This enables transitioning of activities via the back button. A started activity is pushed onto the back stack of the activity that started it (if not specified otherwise by e.g. intent flags). When an activity is destroyed, it is popped off the corresponding back stack. The AMS is also responsible for managing other components like services, routing of intents, and more.

The AMS stores tasks and activities in the hierarchy seen in Figure 1. Several conditions can influence the default behavior of how an activity gets associated with a task. Conditions are either intent flags or activity attributes, and we will now look at the conditions that will become important when we talk about task hijacking.

The activity attributes most important for this work are taskAffinity, allowTaskReparenting, and launchMode[5]. Task affinity is a string that determines the task an activity prefers to reside in. When an activity is started with the NEW_TASK flag, the task with the same affinity is chosen over creating a new task. By default, this attribute is the package name of the app, but it can be any arbitrary string chosen by the developer. allowTaskReparenting is a boolean attribute, that indicates whether the activity can leave the task it was started on. Normally, an activity stays on the task it started on for its entire life cycle. However, when this attribute is set to true and a task comes to the foreground with the same task affinity as the activity, the activity will be reparented to that new task. The launchMode attribute determines how an activity should be started. Together with the FLAG_ACTIVITY_* intent flags, this attribute determines which task an activity gets associated with. It can have one of four values: "standard", "singleTop", "singeTask" and "singleInstance".

Ren et al[2] described the task management of Android in a formal model called the task state transition model. It consists of a tuple $(S,E,\Lambda,\rightarrow)$, where

1) S is a set of all task states. A task state contains all current tasks in the system, including their background status.
2) E represents the set of events that will change the

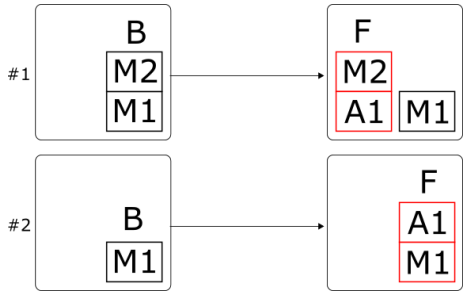

Figure 2. Examples demonstrating task hijacking with the task state transition model

task states, e.g. startActivity() or the back button is pressed.

3) Λ is a set of conditions.
4) And $\rightarrow$ is the set of state transitions. A state transition consists of the initial task state and the resulting task state, an event that started the transition, and a condition under which the transition happens.

We will use this model in the next part of this section to formalize what contributes to a successful task hijacking attack.

C. Task Hijacking

As already mentioned in the introduction, the goal of task hijacking is to get a task to contain both benign and malicious activities. The threat model is that the malicious app is already installed on the target device. It does not need any sensitive API for the attack, meaning no extra permissions are needed. However, to perform a follow-up attack like phishing, the attacker needs some basic knowledge about the target app. But information like the package name or activity layout is not hard to get, so that is not a real obstacle either. There are different ways to hijack a task and we will look at them now in detail.

There are two general types of task hijacking: either a benign activity gets pushed on an attacker-controlled task or a malicious activity infiltrates a benign task. We will now look at two examples of task hijacking to illustrate the different types and what conditions and events lead to the hijacked task. In both examples, two apps are installed in the environment: a benign app Alice and a malicious app Mallory. Mallory's goal is to hijack Alice's task and the examples show how she can pull that off. Both examples are illustrated in Figure 2.

Example #1: Mallory starts the attack by having her activities, M1 and M2, already started in the background. The conditions that will lead to a hijacked task are that M2's task affinity is set to that of A1 and M2 allows task reparenting. The event that causes the hijacking is that A1 is started from the launcher or that the NEW_TASK flag is set when startActivity() is called. When A1 is now started, M2 will be reparented to the task A1 now resides in, because it allows task reparenting and a new task came to the foreground with the same affinity as M2, resulting in a hijacked task. This example shows, how an activity infiltrates the benign task of Alice.

Example #2: Like in #1, Mallory already has M1 lurking in a background task. The condition here is that M1 has its task affinity set to that of A1. If A1 is now started from the launcher or with the NEW_TASK flag set, it will be pushed onto the task M1 resides in. This shows a hijacked task, where the benign activity is pushed onto an attacker-controlled task.

The two examples show how simple it is to hijack a task. The control over the UI can then be used to launch other attacks, including spoofing, phishing, and denial of service. We have now established how dangerous and easy to perform task hijacking is. The next section will cover the proposed solution to give the developer more tools to protect the integrity of their app's UI.

IV. Method

Completely preventing task hijacking is almost impossible. There are many benign use-cases for task transitioning and differentiating them from the malicious ones is quite troublesome. As mentioned in section II, WindowGuard uses dynamic analysis to detect potential task hijacking cases and warns the user about them. This approach would raise user awareness on the threat. However, it could affect usability if too many false positives are found. Moreover, having the user as the last line of defense is not always a good idea, because the everyday user has not enough context to decide whether the task transition is benign or malicious.

Since Google went with a similar technique against overlay-based attacks in Android 12, enabling developers to protect their tasks and activities against task hijacking seems like a good choice. The goal of this approach would be to give the developer the possibility to decide which activities should be treated as sensitive. Activities that are marked as sensitive are then handled differently by the AMS. They will not be pushed onto a task containing activities from another app and foreign activities cannot be pushed onto tasks where a sensitive activity is on top. Furthermore, the developer can decide if they want to completely seal their task. This means that only activities from the own app can be pushed on the task.

V. Evaluation

The proposed defense mechanism will be evaluated in three categories:

1) *Effectiveness*: To evaluate the effectiveness of the extended framework, we will use four proof-of-concept apps that implement several scenarios of task hijacking. Furthermore, real-world implementations of task hijacking like StrandHogg will be used to test the extension.
2) *Usability*: The extension should not prevent benign use-cases of task transitioning. This could be evaluated by writing apps that use task transition features to improve user experience and run them inside the extended frame work.
3) *Performance*: Performance is the least important category of the evaluation. However, the overhead should be small enough to not influence the user experience. The resulting overhead will be measured in contrast to a non-extended framework running the same operations.

VI. Schedule

The schedule of the thesis will have three phases:

1) Development
2) Evaluation
3) Writing the thesis

The first phase is the development phase. It is expected to range from 4 to 6 weeks. In this phase the extension to the Android framework and, if necessary, more test apps will be implemented. But before the actual implementation starts, a design will be created. This will take one to two weeks and consists of:

1) Extract the classes that need to be extended
2) Creating call sharts of relevant APIs
3) Class diagrams of extended classes

A good implementation depends on the design, so this part of phase one should not be neglected.

After the first iteration of the design is finished, the implementation begins. This part should take about a month to be ready for evaluation, which will be the next phase.

The evaluation phase will last from one to two weeks and as mentioned in V it consists of three parts.

And the last phase will be the writing of the thesis. This should take about three weeks. During phase one and two I already will gather content for the thesis and this phase should mostly consists of writing down this content in a scientific manner. There should be one to two weeks left for printing and proof reading the thesis. The dead löine for the first draft will be May the eleventh.

References

[1] Y. Fratantonio, C. Qian, S. P. Chung, and W. Lee, "Cloak and dagger: From two permissions to complete control of the ui feedback loop," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 1041–1057.

[2] C. Ren, Y. Zhang, H. Xue, T. Wei, and P. Liu, "Towards Discovering and Understanding Task Hijacking in Android," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 945–959. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/ren-chuangang

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

[4] C. Ren, P. Liu, and S. Zhu, "Windowguard: Systematic protection of gui security in android." in *NDSS*, 2017.

[5] "App Manifest Activity-Element." [Online]. Available: https://developer.android.com/guide/topics/manifest/activity-element