Universität des Saarlandes
MI Fakultät für Mathematik und Informatik
Department of Computer Science

Bachelorthesis

Hardening Android's Task Management to Prevent Phishing

submitted by

Mirko Paul Meinerzag

on May 25, 2022

Reviewers

Dr.-Ing. Sven Bugiel

Dr. Giancarlo Pellegrino

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Saarbrücken, May 25, 2022,

(Mirko Paul Meinerzag)

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, May 25, 2022,

(Mirko Paul Meinerzag)

This thesis is dedicated to the memory of Christopher Ludwig Meinerzag.

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. The vulnerability can then be used to launch follow-up attacks that leak sensitive information or deny crucial services.

This thesis continues previous work on task hijacking. One proposed solution is to enhance task management so that developers can protect their UI from being hijacked. In this work, the proposed solution is implemented as a prototype on Android 10. This is done by modifying the Android Open Source Project such that developers can declare in the manifest of their app which parts of the UI should be treated as sensitive and need further protection by a security-enhanced task management.

The prototype is evaluated against several proof-of-concept apps to show its effectiveness, usability, and performance compared to an unmodified Android version. Furthermore, we performed a small-scale analysis of top apps from Google Play. This work also presents the results of the apk analysis and compares them to previous, similar studies on the topic.

Acknowledgements

First and foremost, I want to thank my supervisor Dr.-Ing. Sven Bugiel. He was always ready to answer my questions and made it possible for me to work on such an exciting topic.

I would also like to thank Yusra Elbitar for providing me with a huge dataset of APKs, which was very useful during the evaluation of my prototype.

And last but not least, I want to thank the friends who supported me during the thesis. Especially Julius Giloi, Joshua Barbie, and Andreas Hanuja. Their feedback and proofreading are really appreciated.

Contents

Chapter 1

Introduction

According to the annual report on internet crime by the federal bureau of investigation in 2021 [1], phishing is the most performed attack on the internet. It is a threat that abuses user interfaces (UIs) and performs social engineering to leak sensitive data, for example, the banking credentials of the victim, to the attacker. There are many different attack vectors for phishing attacks. Spoofed e-mails in companies [1], phishing SMS on smartphones [2], or malicious pop-ups in browsers [3] are just examples of the plethora of phishing attacks. Since these attacks plague users for quite some time now, it is no wonder that they found their way into Android's ecosystem.

Android's UI was and still is a prominent target for attacks like phishing. Simple attacks like SMS phishing are outshined by more sophisticated attacks that leverage flaws in Android's operating system to compromise the UI. Clickjacking [4], for example, abuses overlays to perform context hiding and tricks the user into clicking invisible UI elements. Cloak and Dagger [5] is a collection of attacks that abuse the accessibility features on Android to launch dangerous attacks and get complete control over the UI feedback loop. This thesis focuses on a specific vulnerability in Android called Task Hijacking.

Task Hijacking abuses flaws in Android's multitasking mechanism to compromise the UI of benign applications. Android's UI is made up of so-called activities. Activities represent a piece of UI and serve as the primary interaction point between users and applications. To make navigation between activities more coherent, they are stored inside a task, that is a stack of activities, when started. Each application has its task where its started activities are stored inside. However, Android allows activities from different apps to reside in the same task. Furthermore, developers can influence the task management to choose which task an activity gets associated with. This allows developers to set up tasks that provide the best possible user experience when switching between different

apps. These two facts, however, enable malware to inject malicious activities into the task of benign apps and thus compromise the UI, resulting in a hijacked task. Such a task can have devastating consequences for the user. For instance, if a malicious activity, that hijacked a victim task, mimics a benign activity and the user returns to that task, they might fall victim to phishing by the malicious activity.

The threat of Task Hijacking is now known for a relatively long time. Ren et al. [6] thoroughly analysed Android's task management and Task Hijacking. Moreover, they proposed two ideas on how to defend against the vulnerability. They realized the first idea in the form of a mechanism called "WindowGuard" [7], which we will look at in Chapter 3. Their second idea is to harden Android's task management and monitor specific activities and tasks such that attackers can no longer unconditionally hijack them. Unfortunately, Ren et al. never realized this approach and could show that a more secure task management is indeed feasible in practice.

This thesis aims to answer two questions: Is it possible to prevent Task Hijacking attacks by a hardened task management effectively and is such an extension feasible regarding performance and usability. To answer the first question, we provide an extension to the Android framework. This extension allows developers to protect their activities and tasks from Task Hijacking. The prototype is then evaluated by running proof-of-concept attacks and reviewing the resulting task states. The second question is answered by measuring the overhead with the extension in place. Furthermore, we performed a small-scale study of top apps that analyzes the features used by attacks that perform Task Hijacking. The study results are then compared to a previous, similar study. In summary, this thesis shows that a hardened task management is possible with almost no performance impact and great security benefits.

Chapter 2

Background

As mentioned in Chapter 1, Task Hijacking abuses flaws in Android's multitasking mechanism to compromise the UI of benign applications. Multitasking is used to run multiple processes concurrently. It is a helpful tool to leverage the full potential of the CPU without having one process block the processor. However, multitasking on traditional PCs is quite different from multitasking on Android smartphones. It is a rather complex mechanism that divides processes and tasks from each other and manages them separately.

This chapter provides the necessary background knowledge to understand Android's multitasking better. It will look at what constitutes an application, how the UI in Android is built up, what basic security principles are enforced by Android, and lastly, analyze the complex task management. With this foundation, we can then continue to investigate Task Hijacking and which flaws in the task management enable the vulnerability.

2.1 Android Basics

Applications (short: apps) are the core essence of smartphones. They enable users to customize their devices in many ways. When a user wants some specific functionality, for example, catch Pokémons or use social media, they can simply download the app that provides the functionality that suits their needs. Usually, the user installs third-party apps by using an app store (e.g., Google Play), where they can search for and download apps they would like on their smartphone. Another option is to directly drag or download the app's APK (Android application PacKage) onto the smartphone. The benefit of the former method is that apps are reviewed before they are uploaded to the store and are

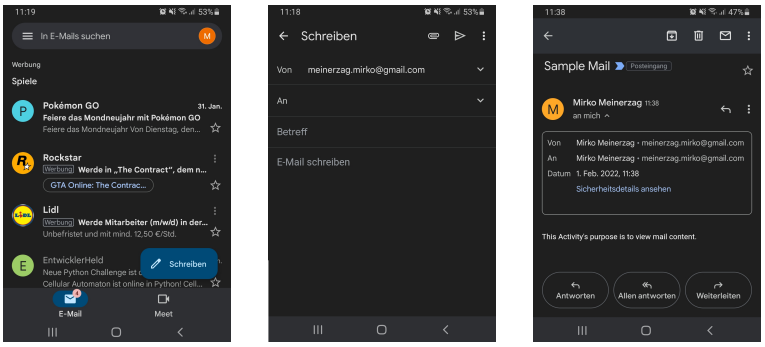


Figure 2.1: example of different activities in the Gmail app

less likely to contain malicious code. The APK contains all the files necessary to install and run the application. One of these files is the application manifest.

The *application manifest* is an XML file that contains essential information about the app [8]. These include the app's package name or what Android version the app supports, among other things. Another vital part of the manifest is the declaration of the app's components. On Android, there are four main components app developers can use to build their apps:

- **Activity**: An activity represents a piece of UI the user can interact with. It is a view that contains other UI elements, like buttons or text. Activities are used as the main method for user interaction and are arguably the most important component of app development.
- **Service**: Services are used for background processes that do not need user interaction.
- **Broadcast Receiver**: This component can be either registered in the manifest or programmatically at runtime. It allows the app to get notified when a broadcast is sent through the system. For example, when the device has booted, a broadcast tells the receivers exactly that.
- **Content Provider**: A content provider is used to share data between apps. The data management is done in a SQLite manner, using tables and typical SQL commands, like insert or delete. A URI is used to connect to a provider, specifying the scheme, the provider, and which data is requested.

Out of these components, activities are of the most interest for this thesis. They make up the UI of applications and are deeply intertwined with the task management. An app typically has multiple activities, each designed for a different purpose. Figure 2.1 shows an example of three different activities in the Gmail app. The first is designed

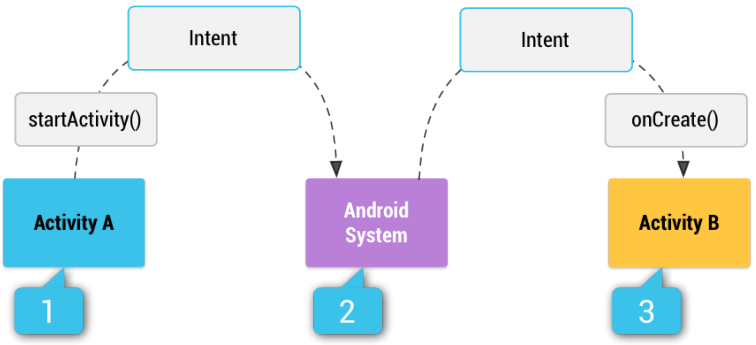


Figure 2.2: Illustration of implicit intent handling, taken from [9]

for viewing the inbox, and the other two are for writing and reading e-mails. Different components can communicate with each other by using so-called intent objects.

An *intent* is a data-carrier object which describes, as the name suggests, an intent to do something [9]. Examples of intents would be starting another component or sending a broadcast throughout the system. Intents come in one of two flavors: explicit or implicit. An *explicit intent* specifies exactly which component should be invoked. This is done by specifying the package and class name of the component. *Implicit intents*, on the other hand, specify a generic action that needs to be performed. For example, if the intent is to play a video, and the developer does not want a specific video player to start, they can create an implicit intent with the action `ACTION_VIEW`. To specify which intents an app can handle, developers can declare so-called intent filters.

Intent filters are declared in the manifest entry of the corresponding component. For example, when a component should handle the above describe `ACTION_VIEW` intent, it must use an intent filter that declares this action. When an implicit intent is sent, Android tries to find an app that can handle it. Figure 2.2 visualizes this process. If multiple apps can handle the intent, the user either chooses one of the available apps or has already set a default component that can handle the intent.

2.2 Security Principles in Android

In Android, apps are isolated from each other by a sandbox [10]. This sandbox is based on the underlying Linux UserID of the process, which is assigned to the app upon installation. This process boundary prevents apps from accessing each other's data and system resources. As long as the kernel is not compromised, the sandbox protects the app from potentially malicious apps. To access restricted or sensitive data, developers must request permission to do so.

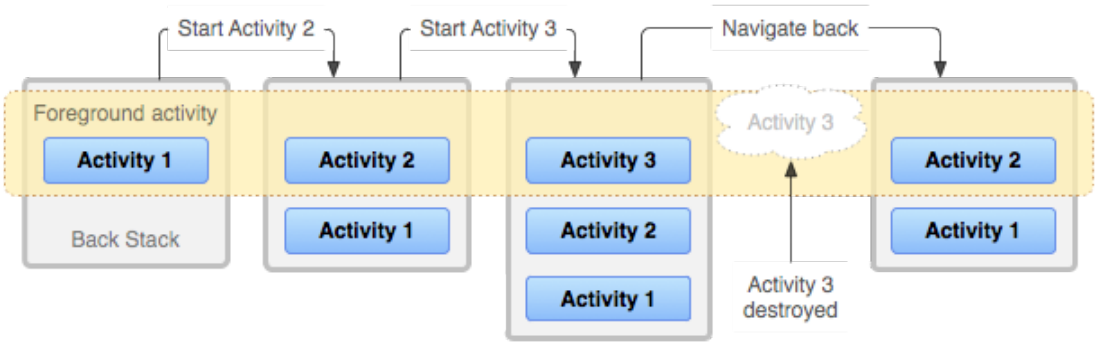


Figure 2.3: Illustration of back stack behaviour, taken from [11]

Android's permission system enforces the principle of least privilege. There are basically two different types of permissions:

- **Install-time permissions**: These permissions are granted to an app on installation. They do not heavily impact the user's privacy or the system in general. Sub-types include *Normal* and *Signature* permissions.
- **Run-time permissions**: Otherwise known as dangerous permissions, these permissions protect more sensitive APIs and resources. The app must request them during runtime and cannot access the protected resources if the user does not explicitly grant the permission.

To use resources protected by permissions, for example, GPS data, the developer must declare them in the application manifest. Whether the permission is granted depends on the type of permission and user confirmation.

2.3 Android's Task Management

Task management in Android is a rather complex mechanism that allows developers to alter its default behavior. This section provides insights into how the task management works and what features developers can use to manipulate it.

In Android, a task is a collection of activities that the user visited during a particular job [6]. Inside a task, activities are pushed onto the so-called back stack. This back stack is implemented in a last-in, first-out manner. When an activity is started, it is pushed onto the back stack, and when it finishes, it is popped off again. The back stack and tasks enable the user to smoothly transition between activities and apps through the back button or recents screen.

One particular task is the launcher task. The launcher task, or the app launcher, is the starting point for most applications. The user can start an app by pressing the corresponding icon in the app launcher. When an app is started from the launcher task, the system will either bring the corresponding task to the foreground or create a new task. The default behavior of the task management is as follows: when an activity gets launched from another activity, it is pushed onto the back stack it was started on. When an activity gets launched otherwise (e.g., from the launcher, broadcast receiver), the activity will be associated with a new task.

When an activity is started and associated with a new task, this task then becomes the new foreground task. There is always only one foreground task currently shown on the screen to the user. All other tasks are called background tasks. Through the recents screen, the user can switch between tasks such that another task becomes the foreground task. The user can also navigate through the back stack of a task with the back button. When pressed, the top activity of the task finishes and pops off the back stack. Figure 2.3 visualizes the back stack behavior when starting activities or using the back button.

Android allows developers to change this behavior of the task management by two means: manifest attributes and intent flags. Manifest attributes can be set on the corresponding activity entry in the application manifest. Intent flags can be set on the intent that starts the activity. The specific attributes and flags that can be used (and abused) are further elaborated in Chapter 4. Nonetheless, the intent behind these features was to enhance the user experience. The design of tasks and the back stack enable the user to transition between apps and activities smoothly. Furthermore, activities can be relocated into a more appropriate context. This is used, for instance, in most browser apps, such that when another app opens the browser app, the activity is opened in the task of the browser.

The system service responsible for starting and managing activities is called ActivityManagerService (AMS). When an activity is started, the intent is routed to the AMS. The AMS will then resolve the activity, evaluate flags and attributes, find the most fitting task for that activity, or create a new task entirely.

Chapter 3

Related Work

Android's task management is not the only entry point for attackers to perform phishing attacks or UI-Deception. Chapter 1 already mentioned other attacks on Android's UI, which this chapter further investigates. Moreover, it presents different defense mechanisms that try to tackle the threat of UI-Deception. And last but not least, a model to formalize the task management in Android is elaborated.

3.1 UI-Deception on Android

Clickjacking [4], or UI-Redressing, is an attack that abuses overlays to hide parts of or the entire user interface (UI) of benign applications. Android allowed overlays on any part of the UI, including UI that belongs to the system (e.g., the settings screen). The attacker can then redirect input from the user through the overlay to the invisible UI element beneath it. This can then be used to, for example, phish for more permissions. The malware simply needs to overlay the settings app and have a button on the overlay directly above the button that grants a dangerous permission. When the user now presses the button without having the correct context, the malware gains access to this permission.

In an attempt to prevent overlay attacks, Android introduced a new security flag for UI elements. The flag's purpose was to enable UI elements to determine when they are overlaid. However, this flag can be abused as a side-channel to launch a keystroke inference attack [5]. The inference is made by drawing an invisible grit of overlays on top of an input method editor. This grid is arranged in a specific order, such that when the user presses on one layer of the grid, all the layers below have the new flag set and all layers above do not. This arrangement allows the malware to precisely determine where

the user clicked on the grid, making it easy to phish for credentials. This side-channel shows how careful developers must be when introducing features to improve security.

Cloak and Dagger [5] is a collection of attacks that abuse two permissions to get complete control of the UI feedback loop. These permissions, `SYSTEM_ALERT_WINDOW` and `BIND_ACCESSIBILITY_SERVICE`, enable malware to draw overlays and to use accessibility(A11Y) features respectively. Clickjacking already demonstrated how dangerous it is to allow apps to draw overlays on top of other apps arbitrarily. However, combined with the A11Y service, it can lead to even more stealthy and impactful attacks. These services were intended to support users with disabilities. An app with permission to use them can alter what the user sees and make inputs on the user's behalf. Cloak and Dagger attacks abuse these features to hide context, fake the user input, or spy on the user.

3.2 Defenses against UI-Deception

UI-Deception or -Confusion is only possible when users cannot verify the integrity of the UI they are interacting with. Browsers approached this issue by introducing a security indicator next to the search bar. The indicator can then be used to verify the connection's identity quickly and if it can be assumed secure. The verification is achieved using the HTTPS protocol and EV (Extended Validation) certificates. Bianchi et al. [12] proposed a similar approach for smartphones. They introduced a security indicator to the navigation bar of the smartphone, such that users can always verify with which app they are currently interacting. In addition to that, they provided a tool to analyze applications statically. This tool can help in application vetting and make app stores more secure.

Ren et al. [6] proposed two defense ideas against Task Hijacking. They realized the first idea, a dynamic analysis of the task management, in their mechanism called WindowGuard [7]. Moreover, WindowGuard also analyzes window behavior and can successfully detect attacks based on overlays or Task Hijacking. When it detects such a breach in the UI, it warns the user. The user must then decide if the breach in the UI's integrity is of malicious or benign nature. The mechanism also enables users to whitelist specific apps to avoid getting spammed by false-positive warnings. This security extension shows how a more secure task and window management can prevent severe consequences on the UIs integrity and, thus, user privacy. Their second idea is to provide developers with additional security features, such that they can protect their apps against Task Hijacking. This thesis aims to implement and evaluate a prototype of this idea.

3.3 Task Transition Model

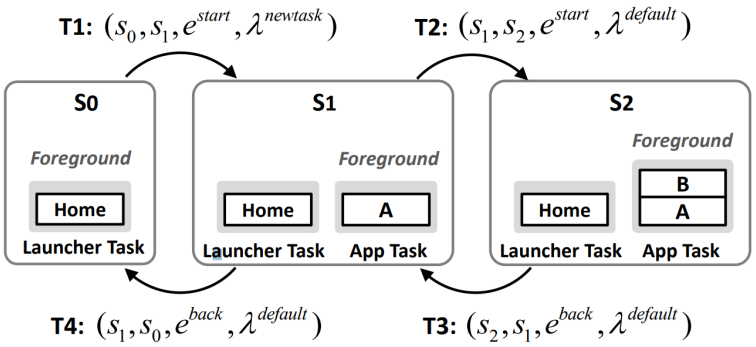


Figure 3.1: Example of task transition model, taken from [6]

To get a grasp on and better understand Android's task management, Ren et al. [6] proposed a model that formalizes it, namely the Task Transition Model. This model helps to visualize the transitioning between tasks and activities formally. It is a tuple of four sets $(\boldsymbol{S}, \boldsymbol{E}, \Lambda, \rightarrow)$, with $\boldsymbol{S}$ being the set of all task states, $\boldsymbol{E}$ being the set of events which trigger transitions, Λ being the set of conditions under which certain transitions take place, and $\rightarrow$ being the set of all transitions between task states. Figure 3.1 depicts a simple example for this model. It shows three task states and four transitions between them. Our work uses a simplified version of the Task Transitions Model to visualize the task states before and after a Task Hijacking attack.

Chapter 4

Investigating Task Hijacking

Task Hijacking abuses flaws in Android's task management, which allows activities from different apps to reside in the same task. Moreover, developers are encouraged to use task control features to manipulate the default task management. These two facts bring the UI of benign applications in danger. Attackers can relocate their malicious UI in the same context as the victim and thus can deceive the user into entering sensitive information into the malicious UI. This chapter will take a deep look at Task Hijacking. It provides a motivating example of an attack that uses Task Hijacking. Furthermore, the chapter investigates the previous work of Ren et al. [6] on the vulnerability and reviews if the attacks they describe still work in Android 10.

4.1 Example: StrandHogg

Strandhogg [13][14] is an attack that uses Task Hijacking to perform phishing attacks. Figure 4.1 shows the attack in action, depicting each step the user takes and how the malware reacts to it. The scenario is as follows: the user wants to use the Facebook app. They do not know that malware has infiltrated their device. The malware will try to steal the login credentials of the user's Facebook account. When the user now presses the launcher icon on the home screen, they expect Facebook to open. However, the malware has already placed itself on the victim task. The activity that now comes to the foreground is not part of the Facebook app. It is a phishing activity specifically crafted to imitate the Facebook login screen. The unknowing user now proceeds to enter their login credentials, and when they press the login button, the malware has successfully acquired the sensitive information. To further hide the attack, the malware now shows the actual Facebook activity that would have started if no attack had taken place.

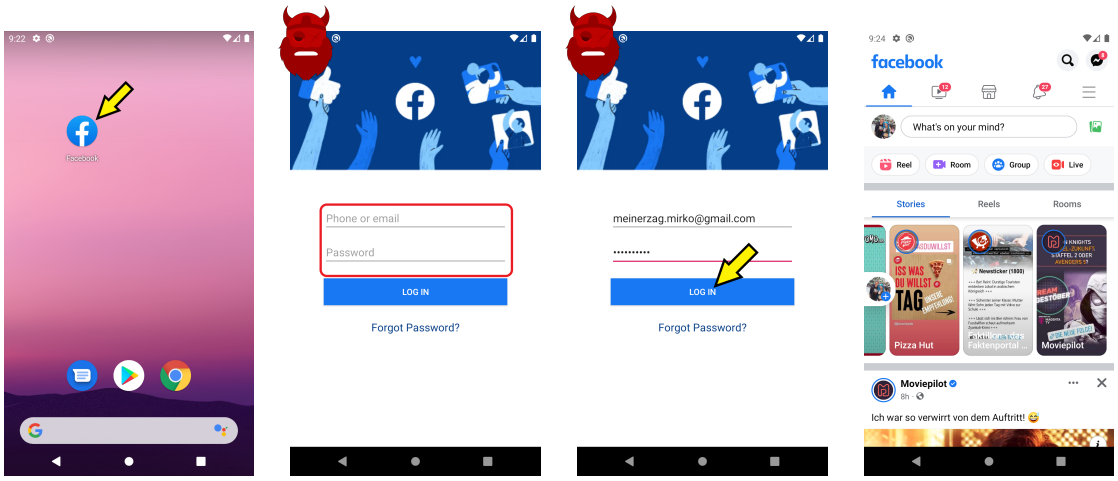


Figure 4.1: StrandHogg attack on Facebook

This example demonstrates the severity of Task Hijacking. Attackers can target almost any app and are not restricted by permissions or a sandbox. Sections 4.2 and 4.3 investigate the vulnerability and different types of Task Hijacking.

4.2 Threat Model and Attack Vectors

The threat model of Task Hijacking is that the malicious app is already installed on the device. It does not need any dangerous permissions and tries to hide its hostile behavior. The attack's goal is to infiltrate the task of the victim app without the user noticing. After successful infiltration, the malware tries to gain sensitive information by imitating the UI of the victim app.

The task control features mentioned in Chapter 2 are either manifest attributes or intent flags. The former can be set on the manifest entry of the corresponding component, and the latter are set on the intent that should start the activity. They allow developers to alter the default task management. This section describes which task control features are abused by Task Hijacking and how attackers use them to compromise the UI's integrity.

4.2.1 Manifest Attributes

The most important manifest attribute for a Task Hijacking attack is `taskAffinity`. This attribute can be set on activities. It is an arbitrary string that defaults to the application's package name. Developers can use this attribute to show which task the activity associates with. In a task, the task affinity of the root activity in the back stack

determines this task's affinity. Malware uses this attribute by setting it to the package name of the victim app. Combined with the following features, this can lead to a hijacked task.

When an activity is associated with a task, it stays on that task throughout its lifecycle. This behavior changes when the boolean `allowTaskReparenting` is set in the manifest. The corresponding activity can then leave its task if another task comes forth that has a better affinity than the current task [8][15]. This attribute can either be set for individual activities or the entire application. Task Hijacking abuses `allowTaskReparenting` by setting the `taskAffinity` to that of the victim. It can then relocate malicious activities onto the victim task when the user starts or maneuvers to the target app.

Android allows developers to customize how their activities are launched. This is done via the `launchMode` attribute [8]. There are four[1] values to which `launchMode` can be set:

- **Standard**: The default mode. Activities that define this `launchMode` get associated with the task they were started on. Multiple instances are allowed.
- **SingleTop**: When set, the AMS makes sure that there is only one instance of this activity at the top of the task. If there is already one, the start of the activity is aborted, and the intent is instead sent to the top activity.
- **SingleTask**: Activity will be started on a new task. If a task already exists and the root of that task is an instance of the starting activity, the intent will be forwarded to that root activity.
- **SingleInstance**: Like `singleTask`, except that the AMS will prevent other activities on a `singleInstance` task.

Another flag abused by Task Hijacking is the `clearTaskOnLaunch` attribute. When set, the task on which the activity is launched is cleared. Malware uses this flag to reset tasks. This allows them to be the root of the task, creating a fresh environment for the attack, such that the attacker can anticipate what the user should see next.

Since Task Hijacking attacks want to alert the user as little as possible, attackers try to hide the attack. The `autoRemoveFromRecents` flag is set when developers do not want the task to be shown in the recents screen. Typically, the recents screen shows all running tasks in the system, such that the user can navigate between them [16]. However, when malware is waiting to hijack a task, the user can see that the malicious UI lurks in

[1] In API level 31 a fifth launch mode, `singleInstancePerTask`, was added. However, it is essentially the same as `singleInstance` and this work focuses on Android 10, so it is only worth mentioning.

the background when using the recents screen. Since the recents screen shows which app the top activity in the task belongs to, the user could then realize that the mimicking UI is not part of the victim app. To circumvent this, attackers can use the flag to remove their malicious task from the recents screen, effectively hiding it from the user.

4.2.2 Intent Flags

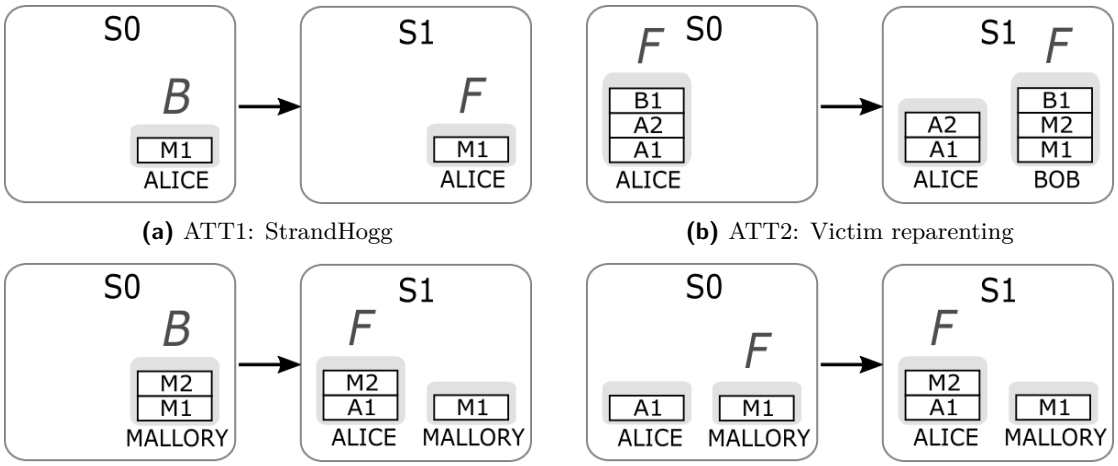
Intent flags can be set on the intent that starts the activity. The most prominent flag for Task Hijacking is the `FLAG_ACTIVITY_NEW_TASK`. The starting activity will be launched onto a new task when this flag is set. If there is a task with the same affinity, the activity will be associated with it. Otherwise, it will be pushed onto a newly created task. It is similar to the `singleTask` launch mode. When an activity is started from the launcher, this flag is set such that it gets its own task. Task Hijacking abuses this flag in combination with the `taskAffinity` to infiltrate the task of the victim. If the affinity is set to the package name of the victim and the activity is started with the `NEW_TASK` flag, the activity will then be pushed onto the task of the victim app since the affinities match.

Intent flags can also be abused to hide an attack. For example, the flag `NO_ANIMATION` disables the transition animation between activities. When an attacker wants to return the control flow to the victim app, this is useful to disguise the switch between applications.

4.2.3 Abusable APIs

Certain APIs can also be useful for Task Hijacking. `moveTaskToBack()` allows attackers to place their malicious task in the background, waiting to perform the attack. `overridePendingTransition()` is used to disable the sliding animations between activity transitioning. It is also possible for developers to override the functionality of the back button. This can be abused to give the control flow to another potentially malicious task instead of the expected activity on the back stack.

The attacker also wants to craft a malicious task with multiple activities in some situations. This can be useful if the malware precisely wants to mimic the victim app's task behavior. Android provides developers with the functionality to create entire tasks. The `startActivities()` API and `TaskStackBuilder` class enable developers to start multiple activities in a designated task at once. These APIs can be used in a Task Hijacking attack to create malicious tasks without displaying all started activities. The malicious task can then wait in the background for a victim activity to come to the foreground and then initiate the hijacking.



(a) ATT1: StrandHogg

(b) ATT2: Victim reparenting

(c) ATT3: Attacker reparenting

(d) ATT4: Task infiltration

Figure 4.2: Illustration of Task Hijacking attack strategies

4.3 Attack Strategies

Equipped with a better understanding of the task control features abused by Task Hijacking, this section describes several attack strategies. These strategies all aim to deceive the user and trick him into leaking sensitive information. Three apps are involved: Alice and Bob, the benign apps, and Mallory, the malicious app. Mallory will try to leak sensitive data of Alice, and in some scenarios, a third party, Bob, is involved. These examples are taken from Ren et al. [6] and a proof-of-concept StrandHogg repository [14].

Figure 4.2 visualizes the attack strategies as follows: Two task states are shown. The first state is the initial state before the attack, and the second is after the attack occurred. A task is visualized by its back stack, containing the currently running activities. Moreover, the foreground state is depicted above the task. If no foreground task is shown, the launcher task is assumed to be visible. Below the task, the app is shown to which the task belongs, indicating the affinity of that task.

StrandHogg[ATT1]: Figure 4.1 demonstrated a real-world attack named StrandHogg. The first attack strategy focuses on that vulnerability. The malware already pushed itself onto Alice's task. When the user now wants to open Alice through the home screen, the task management finds the hijacked task and brings it to the foreground. To further hide the attack, the malicious activity remembers which activity was started, and after it finishes, it starts the expected activity. This attack abuses the `taskAffinity` attribute and `NEW_TASK` intent flag. By setting the affinity to that of Alice, the AMS will push M1 directly onto Alice's task when started with the `NEW_TASK` flag, resulting in a hijacked task.

Victim reparenting[ATT2]: This attack involves an exported activity from Bob. The activity provides some functionality to other apps, for example, a video player. The exported activity set `allowTaskReparenting` to true. Mallory now starts a new task with its affinity set to Bob's. This can either be done through `startActivites()` API or by crafting a task with the `TaskStackBuilder`. When this task is started, the AMS will find that B1 has a better affinity to the malicious task and thus reparent B1 onto it, resulting in a hijacked task. When the user now presses the back button, the malicious UI of M2 will be seen instead of the expected UI of A2.

Attacker reparenting[ATT3]: For this strategy, the malicious task is already lurking in the background, either through a call to `startActivities()` or using `TaskStackBuilder`. M2 is the phishing activity. It has set its affinity to that of Alice and allows reparenting. When Alice is started, M2 is reparented on top of Alice's activity, resulting in a hijacked task and compromised UI.

Task infiltration[ATT4]: The last strategy involves action by Mallory. M1 starts M2 with the `NEW_TASK` flag, and M2's affinity equals that of Alice. The malware could disguise itself to provide some functionality on the victim app. The user then thinks they are forwarded to the victim app. However, M2 is then brought to the foreground on the task of Alice.

Chapter 5

Hardening the Task Management

Due to the complexity of Android's task management, preventing Task Hijacking is not an easy task. Chapter 3 presented some previous work on defense mechanisms against UI-deception. The security indicator proposed by Bianchi et al. [12] and the dynamic analysis of the UI by WindowGuard [7] could definitely help detect a breach in the UI's integrity. However, these mechanisms need the user to make the last call to determine if a breach happened. Several studies show that the average user does not have enough context to make such a security-aware decision [17].

Another solution would be application vetting in app stores. Static analysis of app components and the manifest could help detect malware that abuses Task Hijacking. Nonetheless, studies by Ren et al. [6] show that many apps benignly use these features. It is quite hard to differentiate malicious from benign use cases by only performing static analysis. The resulting task states and UI that occur through these features need to be reviewed to determine whether an attack took place or not. Many false positives could arise during app vetting, resulting in benign apps being removed from app stores, which is also not a viable option. Chapter 6 provides a similar study which shows that this is still the case for the most recently released apps.

The straightforward solution to the problem might be to disable all the task control features entirely. Android would then create a sandbox, similar to the process sandbox, effectively isolating the UI of different apps. However, since many apps use these features, this would also mean that the usability of these apps is at risk. Moreover, the user experience would likely suffer. For instance, the user could no longer use the back button to go back to other apps and would be forced to use the launcher or recents screen to switch between apps. All affected apps would need to be reviewed by their publisher if they still function correctly.

In Android 12, a new API [18] was added to improve the UI's integrity. The API's purpose is to give developers more control over who can overlay their UI. This essentially counters attacks like Clickjacking [4]. Ren et al. [6] proposed a similar solution against Task Hijacking. Developers should be able to protect their activities and task from malicious infiltration and reparenting. However, they did not show the feasibility of the approach by themselves.

This chapter expands on Ren et al. and proposes the general idea of such an extension. Moreover, a requirements analysis for the new tools is provided. This analysis shows, what is needed from the features to prevent Task Hijacking effectively, and thus phishing attacks that abuse the vulnerability. Furthermore, it presents a prototype that implements these features, and in Chapter 6, this prototype will be evaluated.

5.1 General idea

The idea proposed by Ren et al. is an extension of the application manifest. The extension introduces two additional flags: one coarse-grained, which protects the default task of the application, and a fine-grained flag which can be set on individual, sensitive activities.

The coarse-grained flag is called `allowForeignActivities`. *Foreign activities* are activities that do not come from the same package that defines this attribute. It defaults to true, and when set to false, foreign activities are no longer permitted on the default task of the application. The package name of the app determines the default task. This mechanism creates a sandbox for the own task, similar to the sandbox in which the app process resides. The sandbox would then deny any activity that is not from the same package from entering the task. However, it also comes with caveats which we will look at in Chapter 7.

The second additional attribute is called `isSensitiveActivity`. It can be set on individual activities and, in contrast to the first attribute, is meant to be a more fine-grained monitor mechanism. An activity that sets this boolean needs to be monitored throughout its entire life cycle. This flag does then prevent any Task Hijacking attack on the related activity. The next section provides a requirements analysis for both new attributes to define what the extension needs to be effective.

5.2 Requirements Analysis

The hardened task management needs to fulfill specific requirements to prove effective. This section formalizes these requirements. The requirement analysis will show the

capabilities of the new flags, such that they can successfully prevent Task Hijacking and, thus, phishing attacks.

5.2.1 allowForeignActivities

The requirements for `allowForeignActivities` are that no foreign activity can enter the app's task. Of course, this only holds for the default task affinity, i.e., the app's package name. Otherwise, the attribute could be used maliciously, as shown in denial-of-service attacks [6]. The requirements are as follows:

- **REQ1**: Foreign activities are not allowed as the root activity of a protected task.
- **REQ2**: Activities from another package cannot be pushed onto a protected task, neither through launch modes nor intent flags.
- **REQ3**: Foreign activities cannot be reparented onto a protected task.

REQ1 is needed to prevent malicious UI from becoming the root of a task. Attacks like StrandHogg abused this to come to the foreground when the victim app is started via the launcher. **REQ2** prevents activities from infiltrating an already existing task, and **REQ3** circumvents unconditional reparenting of malicious UI onto the victim task if it becomes the foreground task. When these requirements hold for the extension, foreign activities can no longer enter the task of an app that sets the `allowForeignActivities` flag to false. This essentially prevents Task Hijacking attacks that are based on infiltrating the victim task and isolating the task in its sandbox.

5.2.2 isSensitiveActivity

In contrast to `allowForeignActivities`, `isSensitiveActivity` is a more complex monitoring mechanism. Several requirements need to hold such that the control flow does not unexpectedly switch to malicious UI.

- **REQ4**: When a task contains a sensitive activity, foreign activity reparenting onto that task is disabled.
- **REQ5**: Reparenting of sensitive activities is only allowed if the task does not contain foreign activities. This must also hold for activities on a task that contains a sensitive activity.
- **REQ6**: When a task contains a sensitive activity, foreign activities cannot be pushed on that task.

- **REQ7**: When a task containing a sensitive activity comes to the foreground, the system must ensure that no foreign activity becomes visible.
- **REQ8**: When a sensitive activity is started, the system must ensure that the activity becomes visible.

The five requirements above should counter breaches in the UI's integrity. **REQ4** and **REQ5** monitor reparenting of activities. The target task and the activity must be checked before the task switch occurs. This is necessary, otherwise, attackers can abuse `allowTaskReparenting` to enter the victim task unconditionally or reparent a benign activity onto a malicious task. **REQ6** is necessary to prevent attacks based on infiltrating the task via `NEW_TASK` or launch mode `singleTask`. And last but not least, **REQ7** and **REQ8** ensure that the expected sensitive activity or rather the sensitive task comes to the foreground and not some attacker-controlled UI.

With these five requirements applied to the ActivityManagerService (AMS) and activities that set the attribute, the sensitive activity and tasks become secure in the presence of a Task Hijacking attack.

5.3 Implementation

The prototype for the extension is implemented for Android 10. Android 10 is the latest version which is vulnerable to Task Hijacking[1]. The monitoring of tasks and activities takes place in the AMS. This section provides insights into the implementation of the prototype and how the requirements from the last section apply.

5.3.1 Extending the Manifest

The application manifest supports an attribute called `<meta-data>`. This attribute enables developers to supply arbitrary data to the component that specifies it. The information can then be retrieved from the PackageManagerService. The manifest extension uses this attribute to simulate the additional flags. Listing 5.1 shows an example of what the extended manifest looks like. When parsing the manifest, the prototype searches for meta-data tags that specify one of the two additional attributes on the application entry and all activity entries. When a new flag is encountered, this information is then stored in the corresponding ApplicationInfo or ActivityInfo object, such that it can be retrieved at a later point.

[1] Android 11 changed something about tasks and the task affinity. More on that in Chapter 7

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    package="com.projects.alice">
    <application ...>
        <meta-data
            android:name="allowForeignActivities"
            android:value="false" />
        <activity android:name=".MainActivity">
            <meta-data
                android:name="isSensitiveActivity"
                android:value="true" />
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Listing 5.1: Example: Manifest extension

5.3.2 Sandbox for Tasks

The main reason why Task Hijacking works is due to the task affinity attribute. When an attacker sets this string to the victim's package name, they can unconditionally infiltrate its task. To fulfill the requirements for `allowForeignActivities` in Section 5.2, task affinities need to be monitored. If an app sets the flag to false, the AMS must prevent other activities from entering this task based on its affinity.

The prototype does this by monitoring the resolution of activities. When an activity is started, the resolved `ActivityInfo` object is used to create the corresponding `ActivityRecord` object. The `ActivityRecord` object stores information about the activity and to which task the activity belongs. During the resolution of the activity, it is now checked if the corresponding activity is allowed to have the affinity it defines, i.e. if the task affinity is the package name of an app that sets `allowForeignActivities` to false. If that is the case, an identifier for unsecure tasks is added to the `taskAffinity` of the `ActivityInfo`. For example, if the task affinity was `com.projects.alice` it now becomes `com.projects.alice_UNSECURE`. The AMS will no longer associate the activity with the victim task based on affinity. This check of task affinity before the creation of the `ActivityRecord` object fulfills almost all of the requirements. When an activity is started on a new task, but it is not allowed to have the protected affinity, this task's affinity is flagged as unsecure. The victim app will no longer interact with that task based on affinity.

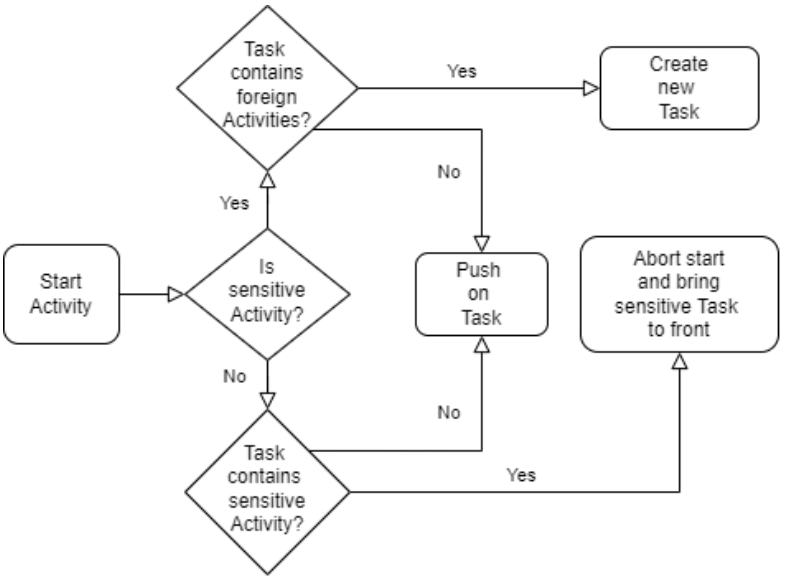


Figure 5.1: Illustration of monitoring when activity gets started.

One scenario not caught by the unsecure flag is when the protected app starts a foreign activity on its own task. The sandbox would also need to prevent this since no foreign activities are permitted on a protected task. This is achieved during the evaluation of the launch flags. When a foreign activity is about to be associated with a protected task, the `NEW_TASK` flag is set. The activity will be started outside the sandbox and not on the protected task.

5.3.3 Sensitive Activity Monitoring

The `isSensitiveActivity` attribute indicates that the activity needs to be treated differently by the AMS. It needs to be monitored such that during its life cycle, no foreign activity can infiltrate the task and thus compromise the UI.

REQ4 and **REQ5** are both needed to prevent malicious use of task reparenting. To fulfill the requirements, the `ActivityRecord` object needs to be monitored. When an `ActivityRecord` is about to be reparented, the prototype first checks whether the new task contains a sensitive activity and if the current activity is foreign to that task. If both conditions are true, then reparenting is aborted as it would otherwise violate **REQ4**. The second check verifies if **REQ5** holds. It does so by reviewing the activity and its current task. Again, reparenting is aborted if the former is a sensitive activity or the latter contains one.

REQ6 and **REQ7** ensure that an attacker cannot hijack an existing task with sensitive activities. This is achieved by imitating a `singleTop` launch mode. This launch mode only allows one activity instance at the top of the task. Otherwise, the intent is forwarded

to the existing activity. When the AMS identifies the target stack to be sensitive, it aborts the start of the foreign activity. Instead, it delivers the intent to the top activity of the sensitive task. This makes infiltrating a sensitive task impossible and ensures that the sensitive task still becomes visible.

The last requirement **REQ8** ensures that a sensitive activity gets started. When trying to find a suitable task for the activity, the AMS asks the `ActivityStack` to find such a task. The requirement is implemented by an additional check to this task-finding mechanism. If a task contains a foreign activity and the starting activity is sensitive, then this task is no longer suitable for that activity. Figure 5.1 illustrates the monitoring performed by the prototype when an activity get started.

Chapter 6

Evaluation

This chapter contains the evaluation of the security extension to Android. The evaluation is performed on three criteria:

- **Effectiveness**: Evaluating the effectiveness of the prototype is done by running Proof-of-Concept(PoC) apps on the extended Android version. These apps implement the attacks described in Chapter 4. The evaluation will show if the extensions described in the previous chapter successfully prevent the attacks.
- **Usability**: Since the prototype changes aspects of the task management, other apps that benignly use the features might be impacted. To evaluate the usability of the extension, a study performed on over 11.000 top apps from Google Play depicts how many apps use and depend on these features. Furthermore, an assessment of the impact on other apps is provided.
- **Performance**: The performance of the prototype is evaluated by measuring the runtime of different scenarios. These measurements are performed three times: once on an unmodified version of Android and twice for each new flag on the extended version. The results of the measurements are then compared to show if the prototype is feasible in practice.

6.1 Effectiveness

The effectiveness of the prototype is evaluated as follows: three PoC apps, Alice, Bob, and Mallory, implement the attack strategies described in Chapter 4. Table 6.1 shows the result of the executed scenarios and whether the requirements we made to the flags prevented the attacks or not. A red cross (✗) indicates that a benign task was hijacked,

	allowForeignActivities="false"	isSensitiveActivity="true"
ATT1	**REQ1** prevents M1 as root ✔	**REQ8** prevents malicious UI ✔
ATT2	Sandbox ineffective ✗	**REQ5** prevents reparenting ✔
ATT3	**REQ3** prevents reparenting ✔	**REQ4** prevents reparenting ✔
ATT4	**REQ2** prevents infiltration ✔	**REQ6**&**7** prevent malicious UI ✔

Table 6.1: Effectiveness of both additional flags

and malicious UI is on the screen. An orange check (✔) means that TASK HIJACKING was prevented; however, a malicious UI is still presented to the user. And last but not least, the green check (✔) shows that the prototype completely circumvented the attack. Each attack is evaluated separately and Figures 6.1 to 6.4 depict the task states before and after each attack.

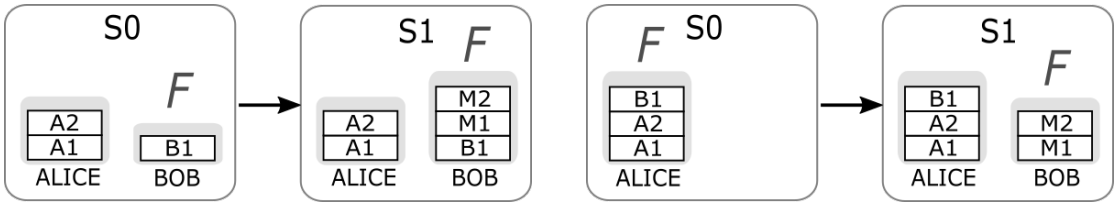
6.1.1 StrandHogg

(a) `allowForeignActivities = false` **(b)** `isSensitiveActivity = true`

Figure 6.1: Task states with new flags on **ATT1**: StrandHogg

ATT1 is prevented by both new flags. The malicious activity M1 tried to infiltrate Alice's task by setting its `taskAffinity`. `allowForeignActivities` prevents the malicious activity M1 as task root since its affinity got appended with the unsecure flag, indicated by the * in Figure 6.1. When the victim activity A1 is started, the AMS creates a new task for the activity instead of bringing the malicious task to the foreground. `isSensitiveActivity` also successfully prevents the attack. Instead of M1, A1 became the foreground activity. However, M1 still resides on a task with the affinity of Alice. This means that a non-sensitive activity of Alice could be associated with that task. Nonetheless, when reviewing this case, the task chosen by the AMS was the one that A1 occupied.

6.1.2 Victim reparenting

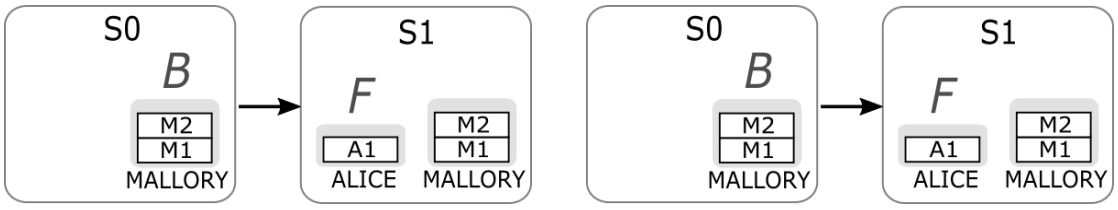


(a) `allowForeignActivities = false` **(b)** `isSensitiveActivity = true`

Figure 6.2: Task states with new flags on **ATT2**: Victim reparenting

ATT2 is only prevented by `isSensitiveActivity`. The attack tries to reparent the exported activity of Bob onto an attacker-controlled task. `allowForeignActivities` was designed to prevent infiltration by creating the sandbox, meaning this attack would be out of scope for this flag. Figure 6.2 shows that B1 is pushed onto a new task since it is not permitted inside the sandboxed task. Nonetheless, the second flag successfully prevents this attack. It needs to be mentioned, that `allowTaskReparenting` did not behave as described in the documentation or other work [6][15]. However, the requirements implemented into the reparent mechanism still hold. It uses the same checks as the other requirements and can thus be assumed to work accordingly. Since the victim task contains a sensitive activity, the AMS would successfully abort reparenting.

6.1.3 Attacker reparenting

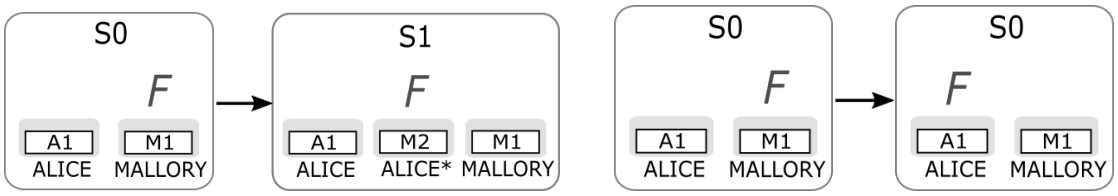


(a) `allowForeignActivities = false` **(b)** `isSensitiveActivity = true`

Figure 6.3: Task states with new flags on **ATT3**: Attacker reparenting

ATT3 is again prevented by both flags. The malicious activity M2 tries to infiltrate Alice's task by setting `allowTaskReparenting`. However, Since M2 does not have the same task affinity as A1 anymore, due to the appended unsecure flag, the reparent mechanism will not be called. For `isSensitiveActivity`, the AMS would abort reparenting of M2 since the task of Alice contains a sensitive activity, and that would otherwise contradict **REQ4**.

6.1.4 Task infiltration



(a) `allowForeignActivities = false` **(b)** `isSensitiveActivity = true`

Figure 6.4: Task states with new flags on **ATT4**: Task infiltration

And last but not least, **ATT4** also fails on both flags. Here, M2 wants to infiltrate the task of Alice by setting its `taskAffinity` accordingly and using the `NEW_TASK` flag. However, for `allowForeignActivities`, the malicious UI still comes to the foreground since only its task affinity has changed. Nonetheless, the task infiltration was prevented, and no requirement was disregarded. The second flag also prevents the malicious UI from taking over the screen. Due to **REQ6** and **REQ7**, M2 is not started, and instead, A1 is brought to the foreground.

The attacks marked with the orange check are only prevented partially. In case of a phishing attack, having the malicious UI in the foreground can still lead to devastating consequences. Chapter 7 will discuss those cases thoroughly.

6.2 Impact on benign Apps

The prototype extends Android's task management with additional checks on tasks and activities. The resulting changes in the behavior of tasks and activities might impact the usability of other apps that benignly use the features. Thus, an assessment of how the prototype might impact benign apps is provided. Furthermore, a study on top apps from Google Play is performed. This study analyzed over 11,000 apps on the features used to manipulate the task management. The results are then compared to a previous, similar study conducted by Ren et al. [6].

6.2.1 App interaction

For the coarse `allowForeignActivities` attribute, the flag would prevent any app, benign or malicious, from entering the task sandbox. However, their activities are still started and pushed onto a new task, which is not associated with the protected task at all. This means that apps that want to extend the functionality of the protected app can no longer do so. For instance, when the user clicks on a link that opens another app,

Activity Attribute	Apps(%)
allowTaskReparenting="true"	2.17
launchMode="singleTop"	48.39
launchMode="singleTask"	58.53
launchMode="singleInstance"	40.68
taskAffinity= default	0.42
taskAffinity= other	10.12
taskAffinity= empty	6.86
excludeFromRecents="true"	48.87
alwaysRetainTaskState="true"	2.60

(a) Results of study on 11.000 latest APKs

Activity Attribute	Apps(%)
allowTaskReparenting="true"	0.80
launchMode="singleTask"	24.63
launchMode= other non-default	24.75
-	-
taskAffinity= default	2.36
taskAffinity= other	1.60
-	-
excludeFromRecents="true"	12.45
alwaysRetainTaskState="true"	2.03

(b) Results of study performed by Ren et al.

Table 6.2: Analysis of activity attributes

this app will start in its own task, and hence pressing the back button will not bring them back to the original app. However, the goal of the flag is to give more control to the developer, so this drawback was anticipated and is a necessary sacrifice.

The second new flag is more volatile than the sandbox. In some cases, the start of an activity is prevented, as otherwise, **REQ6** or **REQ7** would be violated. This means that other benign apps can no longer launch activities onto the sensitive task. However, as the study in the following section will show, most of these apps do not intentionally use these features to enter another app's task. Hence, a sensitive Activity would not interrupt benign activities from using these features in their tasks. Moreover, if another task is not suitable for a sensitive activity, this task is not destroyed. A new task is then created, leaving the unfitting task as is.

6.2.2 Study on task control features

To view the usage of task control features that are used in Task Hijacking attacks, Ren et al. [6] performed a market-scale study on over 6.8 million applications. They analyzed the percentage of apps that use the features and which task affinities are used most often by apps. This study is now almost seven years old and was performed on apps running on Android 5. However, the threat is still available on Android 10, which is reason enough to conduct a similar study on over 11,000 recently released APKs of the most popular apps on Google Play.

Our study analyses the manifest attributes that are abused by Task Hijacking. The results of the study are illustrated in Table 6.2. For comparison, the results of Ren et al. are also presented. Task affinity is significantly used more, especially the empty task affinity. When the affinity is set to an empty string, the component tells Android it does not want to be associated with any task. Furthermore, more than 10% of the analyzed

Scenario	Vanilla Android	allowForeignActivities	isSensitiveActivity
Manifest Parsing	17.4 ms	21.4 ms	15.7 ms
ATT1	21.1 ms	16.8 ms	22.7 ms
ATT2	7.8 ms	8.0 ms	8.3 ms
ATT3	9.9 ms	12.7 ms	11.3 ms
ATT4	11.6 ms	9.6 ms	8.5 ms

Table 6.3: Performance evaluation results.

apps set the task affinity to a different string than their package name. This is almost 8% more than Ren et al. found, and it shows that even more apps use the feature of having different tasks than their default one. However, when analyzing which task affinities are used most often, we found that primarily affinities belonging to advertisement apps or arbitrary affinities that do not resemble package names of other applications are used.

The launch mode `singleTask` is also used more than twice as much as in the findings of Ren et al. This launch mode is used to indicate that an activity should be associated with a new task. However, since only around 10% of the apps set a `taskAffinity`, this means that most of the apps use this feature to enter a new task with the same affinity as their package name. Our prototype would not affect these use cases and thus still work as they would without the hardening.

This study shows that many apps benignly use the task control features. They allow apps to group their activities in a more appropriate context and enhance the user experience. Our prototype protects the task and sensitive activities of the app that sets the new flags. It does not interfere with other tasks, such that, for example, a task is cleared before a sensitive activity can join it. Activity launch is only prevented if an activity tries to infiltrate the task of the protected app. The study and assessment demonstrate that a hardened task management can be effective and usable simultaneously, giving developers more control over their tasks.

6.3 Performance

The performance of the extension will be evaluated by comparing the time of a vanilla Android version with an Android version that implements the prototype. The measurements were taken on different scenarios which are affected by the prototype. The new flags were evaluated separately. We ran each scenario ten times and calculated the average runtime. The results of the measurements are shown in Table 6.3.

Since the prototype extends the manifest by two additional flags, the parsing of the manifest had to be adjusted. The `isSensitveActivity` attribute only sets a boolean on

the corresponding `ActivityInfo` object, which should not impact the performance of app installation. `allowForeignActivities`, however, stores the package name in a list such that it can check at a later point if a task affinity is protected or not. It does so by communicating with the PackageManagerService. Nonetheless, Table 6.3 shows that the average parsing time with the flags in place does not produce massive overhead when parsing the manifest. Moreover, since the compilation overhead during APK installation ranges from seconds to tens of seconds, an overhead of tens of milliseconds would still be completely negligible.

The extension influences the launch of activities by performing several checks and applying the requirements from Chapter 5. The performance of starting activities is evaluated by running the PoC apps that implement the attack strategies described in Chapter 4 and measuring the average time an activity takes to start. The results in Table 6.3 show no noticeable difference between Vanilla Android and the prototype when launching activities. The highest delay is at around 3 ms. A delay below 100 ms is widely accepted as "reacting instantaneously" [19], and our measured delay is far below this threshold.

The performance evaluation shows that a hardened task management is feasible in practice. Furthermore, the measured overhead was anticipated and is still in an entirely negligible range with a maximum of 3 ms.

Chapter 7

Discussion

The evaluation of the prototype showed that it is indeed possible and feasible to harden the task management of Android. However, this chapter discusses the potential drawbacks of the extension. Moreover, other defense mechanisms against Task Hijacking are compared to our prototype. Furthermore, Android 11 introduced task management changes, which prevented our PoC apps from performing the attacks. This chapter also looks into these changes.

7.1 Drawbacks

A sandbox provides strong isolation. However, it also prevents benign apps from interacting with the sandboxed app. Since the `allowForeignActivities` essentially creates a task sandbox, it becomes isolated from all other apps in the system. Chapter 2 established that these features are intended to improve the user experience. Furthermore, the evaluation and Table 6.2 show that many apps use and depend on these features to work correctly. Nonetheless, developers should decide if they want other apps to interfere with their tasks or not. Additionally, our evaluation of the impact on other benign apps showed that many apps that use these features do not use them to infiltrate another app's task.

The effectiveness evaluation showed that in some cases Task Hijacking was prevented, but malicious UI still made it to the foreground. This is not desirable in the presence of a phishing attack. However, Task Hijacking is so dangerous because of its stealthiness. When hijacking the UI of a benign app, the attacker wants to raise as little suspicion as possible. As depicted in the task states, a somewhat unnatural control flow arises in the presence of a Task Hijacking attack. An observant user is more likely to detect the

UI-confusion and react accordingly. Nevertheless, we made it clear that the day-to-day user should not be the last defense for detecting phishing attacks.

When an activity sets the new `isSensitiveActivity` attribute, its task and itself become protected against Task Hijacking. However, this only holds during the life cycle of the activity. When the activity finishes, the protection is lifted off, and the task becomes open for hijacking attacks again. For example, the StrandHogg example in Chapter 4 showed that the activity Facebook wanted to launch was the feed of the user and not the login screen. More steps need to be taken, such that the hardened task management allows fine-grained monitoring without relying on one activity to be in the task.

7.2 Defenses

Defense mechanisms like WindowGuard [7], or the security indicator by Bianchi et al. [12] depend on the user to make the last call. The user must then decide if a breach in the UI's integrity is of malicious nature or not. Resorting to the user as a last line of defense is not always the best idea. Our prototype instead involves the developer. This way, the decision-making is not upon the user but instead on the developers and operating system.

The study on the top apps on Google Play in Chapter 6 demonstrated that, compared to the findings of Ren et al., even more apps use the task control features. This could make static analysis of apps even more troublesome. The more apps use these features, the harder it will get to differentiate benign from malicious use cases, resulting in more benign apps being removed from app stores.

Some other suggestions to prevent Task Hijacking is to use an empty `taskAffinity` or to set the launch mode to `singleInstance` [20][21]. This would effectively defeat all hijacking attempts. However, an empty affinity means that the affinity of that task cannot be used anymore to associate activities with it. This even holds for the owner of the task. Setting launch mode `singleInstance` also prevents any other activity from joining its task, making it a rather inconvenient solution. Our prototype does not come with caveats like this. The task owner is not restricted by the new features and can still interact with it normally.

7.3 Android 11

When running our proof-of-concept apps on Android 11, we found that most of the hijacking scenarios are dysfunctional. Further investigation showed that when Mallory

tried to infiltrate a task based on the `taskAffinity` attribute, it would be forced onto its own task. It essentially created a sandbox for tasks based on the process UID that started the activities. This behavior on Android 11 is documented nowhere. However, a report in the Common Vulnerabilities and Exposures (CVE) [22] states that the WindowManager could be abused to serve as a confused deputy, bringing unexpected apps to the foreground. In the security release notes of Android 11 [23], this vulnerability was addressed. Since nothing else was found in the documentation and release notes, this is our best guess on why Task Hijacking does not work on Android 11 anymore.

7.4 Other Attack Strategies

Phishing is not the only attack that can be performed through Task Hijacking. Ren et al. [6] described several other strategies that could lead to denial-of-service(DoS), such as preventing app uninstallation or ransomware that prevents the availability of certain apps until a ransom is paid. Furthermore, they showed how Task Hijacking could be used to spy on other tasks. This can be achieved by becoming the root of the task. Android treats the root of a task as its owner and thus allows the owner to access information on all activities inside a task. This could serve as a side-channel to gather information on the task state of the victim app.

Our prototype only addresses the issue of phishing attacks based on Task Hijacking. However, the `allowForeignActivities` attribute sandboxes a task, such that the attacks described above would also successfully be prevented. Ransomware and DoS can not infiltrate the task and disrupt the availability of apps and features. The sandbox could also be enforced on all UIs that are part of the system, such that preventing uninstallation would also be tackled.

Chapter 8

Future Work

The prototype and its evaluation demonstrate the effectiveness of a hardened task management. The current implementation performs runtime monitoring on the task states in the system before pushing or reparenting activities into another context. It enables developers to protect their sensitive UI from being hijacked by malware unconditionally. However, there is still room for improvement. This chapter looks at several ideas on how the effectiveness and usability of the prototype could be further enhanced.

8.1 Whitelisting

The current implementation of our prototype treats every activity outside the own package as foreign. This means that all benign apps are restricted from interacting with sensitive tasks or activities in addition to the malware. A solution to this would be to introduce a whitelist. All whitelisted apps could then still interact with the sensitive tasks. Developers could then add apps they trust to their whitelist and still benefit from Android's task management while preventing any hijacking from non-whitelisted apps. However, implementing a whitelist can be quite troublesome. Uniquely identifying an app on Android is not an easy task. For instance, the package name must be unique on one device; nonetheless, it is a string that the app developer can set arbitrarily. This means malware can forge package names such that a whitelist based on the package name could then be circumvented. In addition to the package name, the application's certificate (thumbprint) could be used to identify apps across devices uniquely.

8.2 Permissions

Another idea could be to integrate the permission system into the task management. An app that holds a newly introduced permission can then freely enter other tasks again, even if they are protected. However, this would then involve the user in the decision-making. We have already stated that the user might not be the best choice for enforcing security principles. Moreover, permissions introduce noise to the user. The user experience might get impacted if even more permission requests spam them.

8.3 UI comparison

Our prototype only takes the task states into account when making decisions. Phishing attacks often involve mimicking UI, such that the victim thinks they are interacting with the actual service and not the malware. Possemato et al. [24] proposed a defensive mechanism against phishing attacks based on overlays. This mechanism is called ClickShield, and it employs image analysis techniques to detect potential UI-confusion cases. Our prototype could be enhanced to perform a similar analysis on the UI when the task's state changes. If the new UI resembles the UI of the sensitive activity, then the AMS would treat that activity as dangerous. This would help in still allowing benign use cases of the task control features. However, it only defeats attacks that try to mimic the UI. For instance, another attack vector could lead the user to a malicious URL and phishing on the browser rather than in the app.

8.4 Multiple Sandboxes

The current implementation of `allowForeignActivities` only protects the default task of the app from infiltration. However, if the developer wants to use multiple tasks with different affinities, these new tasks would not be protected anymore. One idea could be to treat all affinities that start with the package name as secure. The developer could use multiple affinities that begin with the package name, and the hardened task management would protect all the resulting tasks. When another app would then try to use the protected package name as part of a `taskAffinity`, the monitoring would mark that affinity as unsecure and create a new task for it.

8.5 More extensive APK analysis

Our study in Chapter 6 has two shortcomings: a relatively small dataset and no code analysis. The former shortcoming could be solved by acquiring a more extensive set of APKs. An in-depth study of the activities needs to be performed to overcome the latter. In this analysis, additional, interesting information could be retrieved from the code, such as intent flags and task control APIs.

Chapter 9

Conclusion

Task Hijacking is a dangerous vulnerability that has devastating consequences if it goes unnoticed by the user. The goal of this thesis was to answer two questions: Can Task Hijacking be prevented through hardening the task management, and if such a hardening would be feasible regarding usability and performance?

We answered the first question by implementing a prototype on Android 10 and running proof-of-concept apps that implement different attack scenarios on top of the extension. Our prototype consists of two additional attribute flags in the manifest: `allowForeignActivities` and `isSensitiveActivity`. Developers can use these flags to indicate if foreign activities can join their task and if an activity should be monitored during its life cycle. No Task Hijacking attack can be performed on it. The evaluation showed that both flags proved effective. The first flag creates a sandbox of the task, such that other apps can no longer unconditionally enter the sandboxed task. This essentially defeats all attacks based on infiltrating the victim app's task. The second flag ensures monitoring of the sensitive activity. It monitors the task states at runtime such that no Task Hijacking can occur during the life cycle of a sensitive activity. Moreover, it ensures that the sensitive activity and task come to the foreground instead of malicious UI.

The second question is answered by evaluating the usability and performance of our prototype. We presented a study on over 11.000 apps. This study depicted how many apps use the task control features used by Task Hijacking. The results show that our extension does not heavily impact the usability of benign apps. These apps mostly use these features to relocate their activities into their own task or to create new tasks with the default affinity. Furthermore, the performance evaluation showed that it is indeed feasible to perform dynamic analysis on the task states. However, the produced overhead is negligible and was anticipated, as security extensions come at a price.

In conclusion, we showed that hardened task management could effectively prevent the threat of Task Hijacking. Furthermore, we provided a prototype for developers to protect themselves against the vulnerability and showed that it is feasible in practice.

Bibliography

[1] Federal Bureau of Investigation, "Internet crime report 2021." [Online]. Available: https://www.ic3.gov/media/PDF/AnnualReport/2021_IC3Report.pdf

[2] S. Mishra and D. Soni, "Sms phishing and mitigation approaches," in *2019 Twelfth International Conference on Contemporary Computing (IC3)*, 2019, pp. 1–5.

[3] C. Yue, "The devil is phishing: Rethinking web single Sign-On systems security," in *6th USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET 13)*. Washington, D.C.: USENIX Association, Aug. 2013. [Online]. Available: https://www.usenix.org/conference/leet13/workshop-program/presentation/yue

[4] L. Wu, B. Brandt, X. Du, and B. Ji, "Analysis of clickjacking attacks and an effective defense scheme for android devices," in *2016 IEEE Conference on Communications and Network Security (CNS)*, 2016, pp. 55–63.

[5] 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.

[6] 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

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

[8] Android Developer Guide, "App Manifest Activity-Element." [Online]. Available: https://developer.android.com/guide/topics/manifest/activity-element

[9] ——, "Intent and Intent-Filters." [Online]. Available: https://developer.android.com/guide/components/intents-filters

[10] Android Source, "Application Sandbox." [Online]. Available: https://source.android.com/security/app-sandbox

[11] Android Developer Guide, "Tasks and the back stack." [Online]. Available: https://developer.android.com/guide/components/activities/tasks-and-back-stack

[12] 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.

[13] T. Blegen, "The StrandHogg vulnerability." [Online]. Available: https://promon.co/security-news/the-strandhogg-vulnerability/

[14] "GitHub Repository: StrandHogg PoC." [Online]. Available: https://github.com/az0mb13/Task_Hijacking_Strandhogg/tree/main/AttackerApp/app/src/main

[15] Android Developer Guide, "App Manifest Application-Element." [Online]. Available: https://developer.android.com/guide/topics/manifest/application-element

[16] ——, "Recents Screen." [Online]. Available: https://developer.android.com/guide/components/activities/recents

[17] M. Alsharnouby, F. Alaca, and S. Chiasson, "Why phishing still works: User strategies for combating phishing attacks," *International Journal of Human-Computer Studies*, vol. 82, pp. 69–82, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1071581915000993

[18] "Hide Overlays API." [Online]. Available: https://developer.android.com/about/versions/12/features#hide-application-overlay-windows

[19] J. Nielsen, "Response Times: The 3 Important Limits." [Online]. Available: https://www.nngroup.com/articles/response-times-3-important-limits/

[20] "Android Task Hijacking." [Online]. Available: https://book.hacktricks.xyz/mobile-pentesting/android-app-pentesting/android-task-hijacking#preventing-task-hijacking

[21] "StackOverflow: Preventing Task Hijacking." [Online]. Available: https://stackoverflow.com/questions/68075544/is-there-a-fix-for-task-hijacking-on-android-10

[22] "CVE-2020-0267." [Online]. Available: https://www.opencve.io/cve/CVE-2020-0267

[23] "StackOverflow: Preventing Task Hijacking." [Online]. Available: https://source.android.com/security/bulletin/android-11

[24] A. Possemato, A. Lanzi, P. Chung, W. Lee, and Y. Fratantonio, "ClickShield: Are You Hiding Something? Towards Eradicating Clickjacking on Android," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Toronto, Canada, October 2018.