A B.Tech. Dissertation REPORT

on

"CLOAK AND DAGGER ATTACK USING ANDROID PERMISSION"

Submitted for the Partial Fulfilment of degree of
Bachelor of Technology
(COMPUTER SCIENCE AND ENGINEERING)

in

Department of Computer Science and Engineering
(2014-18)

Submitted By

Mannam Lokesh	2014UCP1489
Roushan Kumar	2014UCP1362

UNDER THE GUIDANCE OF

Dr. Vijay Laxmi

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
Malaviya National Institute of Technology
Jaipur - 302017

2014-2018

Department of Computer Science and Engineering

Malaviya National Institute of Technology, Jaipur

CERTIFICATE

This is to certify that the project entitled

" Cloak and Dagger Attack using Android Permission"

submitted by

Roushan kumar	2014UCP1362
Mannam Lokesh	2014UCP1489

is a record of bonafide work carried out by them, in the partial fulfilment of the requirement for the award of Degree of Bachelor of Technology (Computer Science and Engineering) at Malaviya National Institute of Technology, Jaipur under my supervision.

Date: / /

Dr. Vijay Laxmi
Associate Professor
Department of
Computer Science and Engineering
MNIT, Jaipur

DECLARATION

We, Roushan Kumar (2014UCP1362), Mannam Lokesh (2014UCP1489) declare the following regarding the work presented in this report titled Cloak and Dagger Attack using Android permission.

This work is done wholly and mainly for the final year major project assessment, where any part of this report has not been previously submitted for a degree or any other qualification at MNIT, Jaipur or any other institute this has been clearly stated. Where we have consulted the published work of others this is always clearly attributed. Whenever we have quoted from the work of other, the source is always given. We have acknowledged all sources of help and no part of the report is plagiarized and report does not suffer from any act of plagiarism.

Signature .....

Date .....

ACKNOWLEDGEMENTS

We would like to give sincere thanks and gratitude to our esteemed supervisor Dr. Vijay Laxmi for providing her valuable guidance and encouragement to learn new things and enhancing our knowledge in the field of Android Permission. His kind cooperation and suggestions throughout the course of this research guided us with an impetus to work, and successfully complete the project.

We would also like to express sincere gratitude to Prof. Udaykumar R Yaragatti, Director, Malaviya National Institute of Technology, Jaipur for all the facilities provided in the institute.

We extend thanks to Dr.Vijay Laxmi, Project Coordinator, Department of Computer Science and Engineering, for allocation of the desired project and timely assistance during various stages of the project.

Contents

List of Figures

Chapter 1

Introduction

1.1 Objective

Now a day's android applications are the fastest medium which delivers news and entertainment to users as compared to news paper and television. There are millions of applications available in Google play store. Android allows large number of users to create applications or post their views, ideas on any particular event. One of the key security mechanism for android is the permission system. For the permission system to actually improve security, the end-users and the community need to be aware of the security implications of different permissions being requested.

We are trying to analyse these permissions by developing applications and exploiting their usage. For this, we have implemented an application which uses overlays to grant a permission to the application and the event will not be specified to the user.

1.2 Motivation

The effectiveness of Android permission system fundamentally hinges on the user's correct understanding of the capabilities of the permissions being granted. In this project we show that both the end users and the security community have significantly underestimated the dangerous capabilities granted by the SYSTEM_ALERT_WINDOW and the BIND_ACCESSIBILITY _SERVICE permissions.

In particular we demonstrate how such an app can launch a variety of stealthy, powerful attacks ranging from stealing user's credentials and security pin, leaving the victim completely unsuspecting.

1.3 Report Structure

In the remaining report, Chapter-2 gives the overview of android permissions and tools used. Chapter-3 involves overview of System Alert Window permission. Chapter-4 describes in detail the Existing security mechanism. Chapter-5 shows the attacking of android UI feedback loop. Finally Chapter-6 provides various results for attacks using the app.

Chapter 2

Overview of Android Permissions and Tools Used

2.1 Android Permission

The purpose of a permission is to protect the privacy of Android user. Android apps must request permission to access sensitive user data(such as contacts and SMS),as well as certain system features(such as camera and internet).Depending on the feature ,the system might grant the permission automatically or might prompt the user to approve the request .An app must publicize the permissions it requires by including <users-permission> tags in app manifest.

2.2 Android Studio

Android studio is the official Integrated Development Environment(IDE) for app development, based on IntelliJ IDEA. On top of IntelliJ's powerful code editor and development tools, Android Studio offers even more feature that enhance your productivity when building Android apps

2.3 Language Used : Java

Chapter 3

The SYSTEM_ALERT_WINDOW permission

3.1 Introduction

An app having the SYSTEM_ALERT_WINDOW permission has the capability to draw arbitrary overlays on the top of every other app. According to the official documentation, “Very few apps should use this permission; these windows are intended for system-level interaction with the user.” Despite this warning, the SYSTEM_ALERT_WINDOW is used by very popular apps such as Facebook, Twitter, Skype. In particular, we found that about 10.2% (454 out of 4455) of top apps on Google Play Store require this permission. The most common usage of this permission is floating widget, such as the ones implemented by music players, weather notification apps, and Facebook Messenger.

Twitter Security apps such as app lockers, desk launchers, and password managers also use this permission to implement some of their key features.

3.2 Features

3.2.1 Android Version

It is important to mention that, starting from Android 6.0, this permission is treated differently from the others (such as the more traditional location related permission). In particular, in the general case, the user needs to manually enable this permission through a dedicated menu. Thus, the general belief is that it is quite challenging for an app to obtain this permission. However, we observed that if an app is installed through the latest version of the official Play Store app, the SYSTEM_ALERT_WINDOW

Permission is automatically granted. Moreover, if the app targets an SDK API higher or equal than 23 (an app's developer can freely select which API level to support through apps manifest), the Android framework will not show the list of required permissions at the installation time: in fact, modern versions of android ask the user to grant permissions at run-time. This means that, since the SYSTEM_ALERT_WINDOW permission is automatically granted, the user will not be notified at any point.

3.2.2 Intention

We note that this behavior seems to appear a deliberate decision by Google, and not an oversight. To the best of our understanding, Google's rationale behind this decision is that an explicit security prompt would interfere too much with the user experience, especially because it is requested by apps used by hundreds of millions of users.

3.2.3 Technical Aspects

Users On the technical side, the overlay's behavior is controlled by a series of flags. The Android framework defines a very high number of flags, the three most important being the following:

- FLAG_NOT_FOCUSABLE: if set, the overlay will not get focus, so the user cannot send key or button events to it, which means the UI events sent to the overlay will go through it and will be received by the window behind it.
- FLAG_NOT_TOUCH_MODAL: if set, pointer events outside of the overlay will be sent to the window behind it. Otherwise, the overlay will consume all pointer events, no matter whether they are inside of the overlay or not. Note that setting the previous flag implicitly sets this one as well.
- FLAG_WATCH_OUTSIDE_TOUCH: if set, the overlay can receive a single special MotionEvent with the action MotionEvent.Action_outside for touches that occur outside of its area.

If none of the above flags are specified, the overlay will receive all events related to the user interaction. However, in this case, these events cannot be propagated to the window below the overlay. There are several other flags and technical aspects that are relevant for our work: for clarity reasons, we postpone their discussion to later in the paper.

3.3 The Accessibility Service

The accessibility service is a mechanism that is designed to allow android apps to assist users with disabilities. In particular an app with this permission has several capabilities. In fact the app is notified (through a callback -based mechanism) of any event that affects the device. For example, the main onAccessibilityEvent() callback is invoked whenever the user clicks on a widget, whenever there is a “focus change,” or even when a notification is displayed.

The details about these events are stored in an Accessibility Event object, which contains the package name of the app that generated the event, the resource ID of the widget, and any text-based content stored by these widgets (note that the content of passwords-related widgets is not made accessible through this mechanism). This object thus contains enough information to reconstruct the full context during which the event has been generated.
Moreover, an accessibility service app can also access the full view tree and it can arbitrarily access any of the widgets in such tree, independently from which widget generated the initial accessibility event. An accessibility service can also perform so-called actions:

For example, it can programmatically perform a click or a scroll action on any widget that support such operations. It can also perform “global” actions, such as clicking on the back, the home, and the “recent” button of the navigating bar.

Google is aware of the security implications of this mechanism. Thus, the user needs to manually enable this permission through a dedicated menu in the Settings app.

Not surprisingly, this permission is less popular than the SYSTEM_ALERT_WINDOW permission: Among the 4455 apps on the play store, we find 24 apps use the accessibility service. It is worth nothing more that none of them are purely designed for people with disabilities. In fact, most of them are security apps such as password managers, app lockers, desk launchers, antivirus apps. Several of these apps are installed and used by more than one hundred of million users.

Chapter 4

Existing Security Mechanisms

The permission discussed thus far grants capabilities that are clearly security-related. However, although powerful, the actions that an app can perform must adhere to what specified in the documentation and to what is communicated to the user. Thus, the Android OS implements a number of security mechanisms to prevent abuse.

4.1 Security Mechanism 1:

The SYSTEM_ALERT_WINDOW permission allows an app to create custom views and widgets on top of any another app. However, the system is designed so that the following constraint always holds: if an overlay is marked as “pass through” (that is, it will not *capture* clicks), the app that created the overlay will not know when the user clicks on it (however, since the overlay is pass through, the click will possibly reach what is below); instead, if the overlay is created to be “clickable,” the app that created it will be notified when a click occurs, and it will also have access to its exact coordinates.

However, the system is designed so that an overlay cannot propagate the click to the underlying app. This is a very fundamental security mechanism: in fact, if an app could create an (invisible) overlay so that it could intercept the click *and* also propagate it to the app below, it would be trivial, for example, to record all user’s keystrokes: a malicious app could create several overlays on top of all keyboard’s button and monitor user’s actions. At the same time, since the overlays are invisible and since the clicks would reach the underlying keyboard, the user would not suspect anything.

An overlay can also be created with the FLAG_WATCH_OUTSIDE_TOUCH flag, such that the overlay will be notified of any clicks, even if they fall outside the app itself. However, once again for security reasons, the event's precise coordinates are set only if the click lands in the originating app, while they are set to (0,0) if the click lands on a different app. In this way, the attacker cannot infer where the user clicked by using the coordinates. This mechanism makes also difficult mounting practical clickjacking attacks: in fact, it prevents an app to lure the user to click on what's below and at the same time being notified exactly where the user clicked, and it is thus not trivial to infer whether the user has been successfully fooled. This, in turn, makes mounting multistage UI redress attacks challenging, since there is currently no reliable technique to know when to advance to the next stage.

4.2 Security Mechanism 2:

An accessibility service app has access, by design, to the content displayed on the screen by the apps the user is interacting with. Although the accessibility service does not have access to passwords, it does have privacy-related implications. Thus, in Android, the service needs to be manually enabled by the user: after pressing on the "enable" switch, the system shows to the user an informative popup and she needs to acknowledge it by pressing on the "OK" button. Thus, the receiving object (once again, the OK button in our case) can check whether this flag is set or not, and it can decide to discard the click or to take additional precautions to confirm the user's intent.

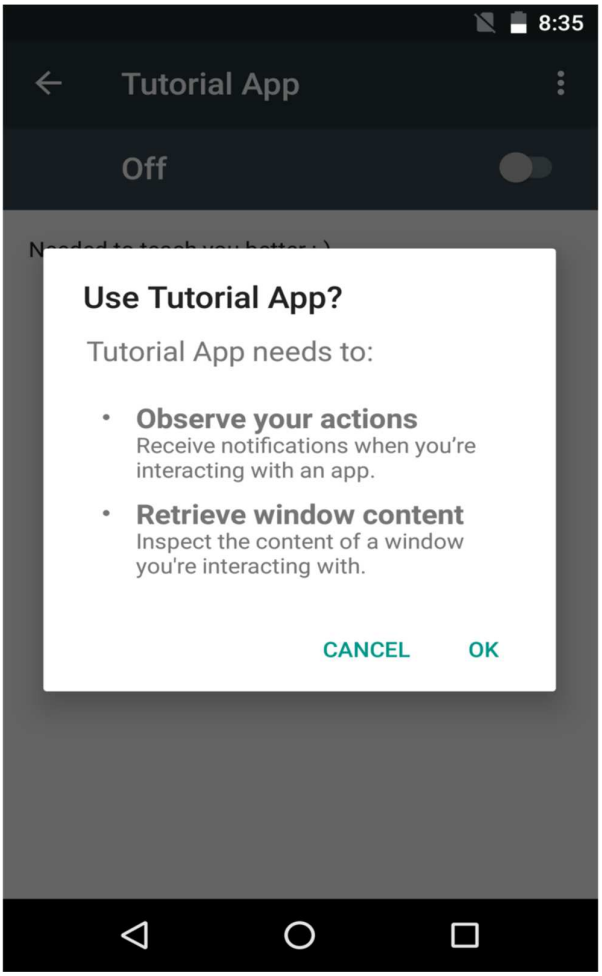

8:35
Tutorial App
Off
Use Tutorial App?
Tutorial App needs to:
Observe your actions
Receive notifications when you're interacting with an app.
Retrieve window content
Inspect the content of a window you're interacting with.
CANCEL
OK

Fig-1a

4.3 Security Mechanism 3:

Given the security implications of the accessibility service, the Android OS has a security mechanism in place that aims at guaranteeing that other apps cannot interfere during the approval process (i.e., when the user is clicking on the OK button). This defense has been introduced only recently, after a security researcher showed that it was possible to cover the OK button and the popup itself with an opaque, passthrough overlay: while the user is convinced to interact with the app-generated overlay, she is actually authorizing the accessibility service permission by unknowingly pressing OK.

The new security mechanism works in the following way. For each click, the receiving widget receives a MotionEvent object that stores the relevant information. Among these information, Google added the FLAG_WINDOW_IS_OBSCURED flag (*obscured* flag, in short). This flag is set to true if and only if the click event passed through a different overlay before reaching its final destination (e.g., the OK button). Thus, the receiving object (once again, the OK button in our case) can check whether this flag is set or not, and it can decide to discard the click or to take additional precautions to confirm the user's intent.

Figure 1b shows the message shown when the user clicks on the OK button while an overlay is drawn on top. We inspected the Android framework codebase and we found that this flag is used to protect the accessibility service, but also to protect the Switch widgets used to authorize each individual permission. Google is advising third-party developers to use a similar approach to protect security-sensitive applications.

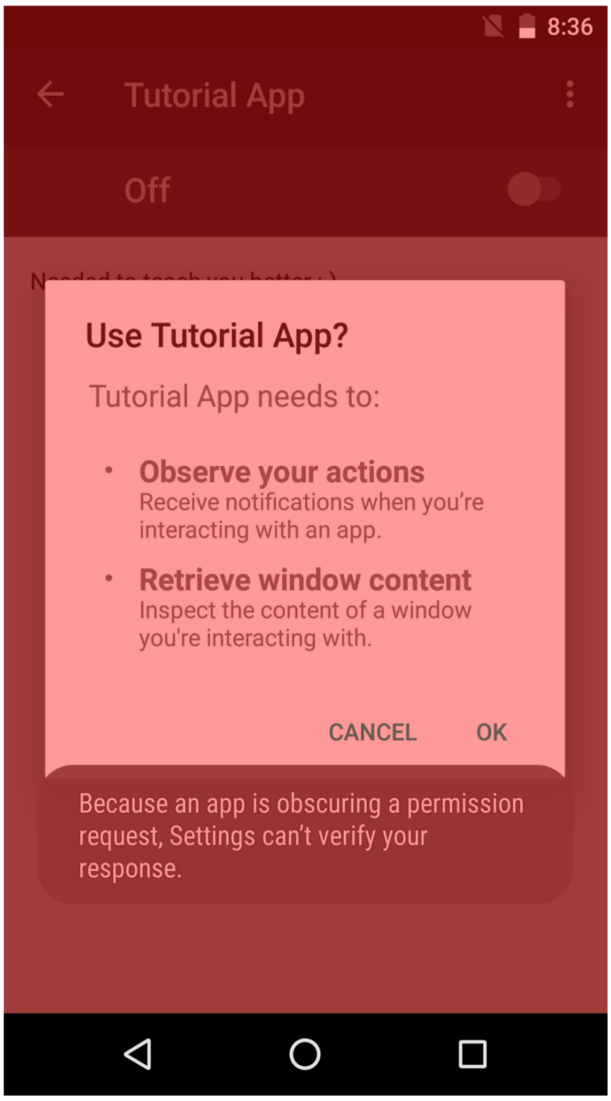

8:36
Tutorial App
Off
Use Tutorial App?
Tutorial App needs to:
Observe your actions
Receive notifications when you're interacting with an app.
Retrieve window content
Inspect the content of a window you're interacting with.
CANCEL
OK
Because an app is obscuring a permission request, Settings can't verify your response.

Fig-1b

Chapter 5

Attacking the UI feedback loop

5.1 Introduction

As we have mentioned in the introduction, the ultimate strength of “cloak and dagger” attacks lies in their complete control of the UI feedback loop between what users see on the screen, what they input, and how the screen reacts to that input. From a more conceptual point of view, the UI offers an I/O channel to communicate with the user. In turn, the two directions of the channel can be attacked in an active or a passive fashion. This leads to four distinct attack primitives, which we discuss next.

5.2.1 Primitive 1: Modify What The User Sees

An attacker may want to confuse or mislead the user by showing her something other than what is displayed on the screen. For example, in the context of clickjacking, the attacker may want to modify the prompt displayed by the system to trick the user into clicking “yes.” In other scenarios, instead, the attacker may want to hijack the user’s attention, for example by launching an attack while the user is distracted watching a video.

5.2.2 Primitive 2: Know What is Currently Displayed

Before we can properly modify what the user sees, we need to know what we are modifying. Continuing with the clickjacking example, our attack can only be successful if we know the system is displaying the targeted prompt: if we show our modified prompt when the target is not even on the screen, we will alert the user that something is wrong. As another example, to steal the user's password with a fake Facebook login, it only makes sense to show the fake UI when the real one is expected. In general, an attacker aims at determining which app is on top and which activity is displayed at any given time.

5.2.3 Primitive 3: User Input Injection.

To start with This primitive allows an attacker to control the user's device, while all the previous primitives provide proper "masking" for the effect of user input injection. In particular, to disable specific security features or to silently install an additional app, the attacker needs, for example, to inject clicks to the Android Settings app.

5.2.4 Primitive 4: Know What the User Inputs (and when).

The attacker may want to monitor relevant GUI events, such as user clicks. In some cases, this is necessary so the attacker can update the modified display in Primitive 1 to maintain the expected user experience. With the clickjacking example, it is necessary to know that the user has clicked on either "yes" or "no" to dismiss the fake message we are displaying. This primitive can also be used to leak the user's private information.

5.3 Implementation

The app we have developed always runs in the background of android system. Basic aim of the attack is to fooling the user and stealing his credentials without any suspicion to the user. For this, we select a target app, that is, which login credentials the attacker want to steal. After selecting the target app whenever the user opens the target app we obscure the main screen by launching our app over the target app and whenever the user enters his login credentials we have accomplished the task of stealing his credentials.

We make use of SYSTEM_ALERT_WINDOW permission to draw on top of other apps. Our app always runs in the background and checks whether the target app is actually running in the main screen.

But android does not allow an app to check what app is running in the foreground unless it is given the Usage Access permission. So, initially we must lure the user to grant the Usage Access permission as soon as the app is installed, we perform this attack by using overlays.

5.4 Services Overview

A Service is an application component that can perform long-running operations in the background, and it doesn't provide a user interface. Another application component can start a service, and it continues to run in the background even if the user switches to another application. Additionally, a component can bind to a service to interact with it and even perform interprocess communication (IPC). For example, a service can handle network transactions, play music, perform file I/O, or interact with a content provider, all from the background.

These are the three different types of services:

Foreground:

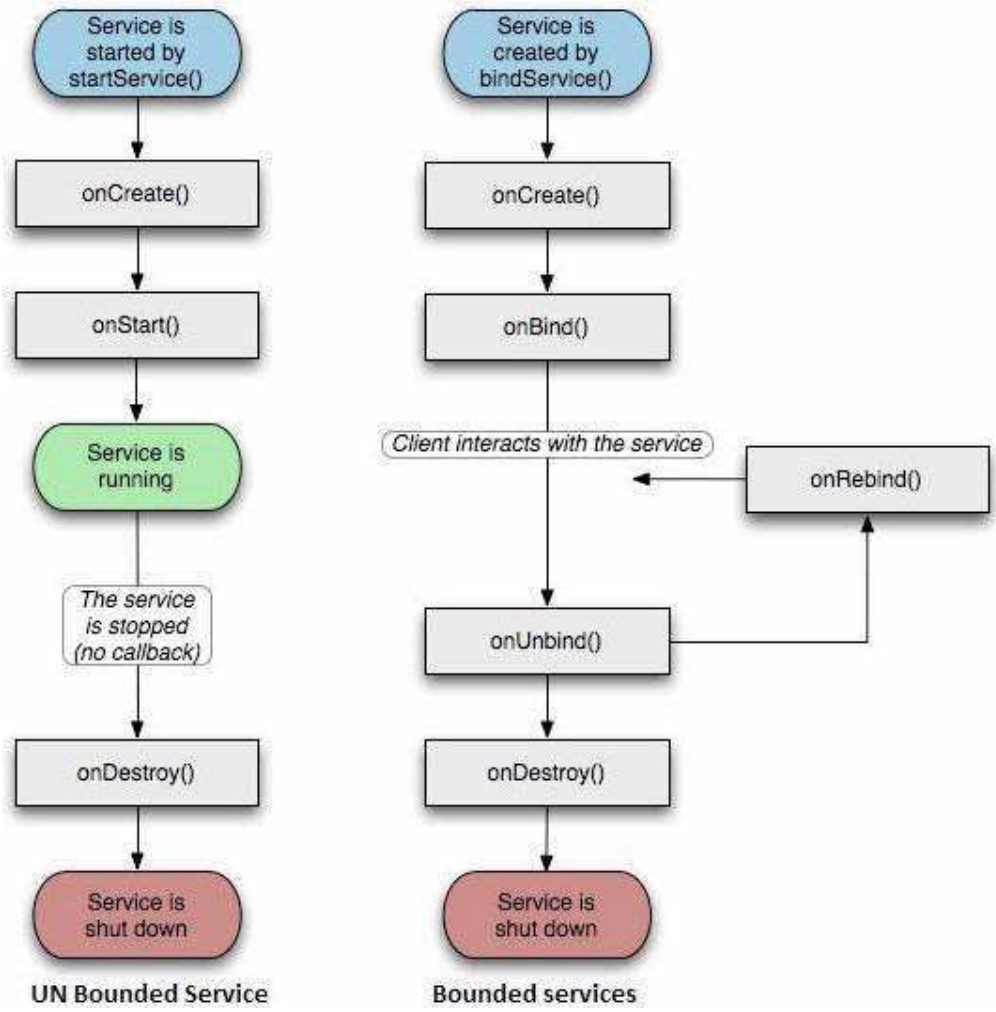
A foreground service performs some operation that is noticeable to the user. For example, an audio app would use a foreground service to play an audio track. Foreground services must display a notification. Foreground services continue running even when the user isn't interacting with the app.

Background:

A background service performs an operation that isn't directly noticed by the user. For example, if an app used a service to compact its storage, that would usually be a background service.

Bound:

A service is bound when an application component binds to it by calling bindService(). A bound service offers a client-server interface that allows components to interact with the service, send requests, receive results, and even do so across processes with interprocess communication (IPC). A bound service runs only as long as another application component is bound to it. Multiple components can bind to the service at once, but when all of them unbind, the service is destroyed.



Service Overview

OnStartCommand():

The system invokes this method by calling startService() when another component (such as an activity) requests that the service be started. When this method executes, the service is started and can run in the background indefinitely. If you implement this, it is your responsibility to stop the service when its work is complete by calling stopSelf() or stopService(). If you only want to provide binding, you don't need to implement this method..

OnBind():

The system invokes this method by calling bindService() when another component wants to bind with the service (such as to perform RPC). In your implementation of this method, you must provide an interface that clients use to communicate with the service by returning an IBinder. You must always implement this method; however, if you don't want to allow binding, you should return null.

OnCreate():

The system invokes this method to perform one-time setup procedures when the service is initially created (before it calls either onStartCommand() or onBind(). If the service is already running, this method is not called.

OnDestroy():

The system invokes this method when the service is no longer used and is being destroyed. Your service should implement this to clean up any resources such as threads, registered listeners, or receivers. This is the last call that the service receives.

5.5 Types of Attacks Possible

5.5.1 Context-aware Clickjacking:

One known attack in Android is the possibility to perform clickjacking attacks. These attacks work by spawning a security-sensitive app, which we call the target app, and by creating an on-top, opaque overlay that does not capture any of the user interaction: while the user believes she is interacting with the app she sees, she is in fact interacting with the target app in the background. In a security context, the target app would usually be the Android Settings app, or any other “privileged” app. The malware would then use social engineering techniques to lure the user to click on specific points on the screen.

The Clickjacking is relevant to our work because, very recently, a security researcher discovered that it is possible to perform clickjacking to lure the user to unknowingly enable the accessibility service [7]. In response to the researcher’s report, Google implemented the security mechanism based on the FLAG_WINDOW_IS_OBSCURED flag described in Security mechanism 3. The researcher subsequently discovered that the current implementation of this defense mechanism only checks that the portion of the clicked button was not covered by any overlay, and it does not guarantee that the OK button is visible in its entirety. According to Google, this issue does not pose any concrete and practical risks, and, reportedly, there are no plans to fix it.

The main limitation of current clickjacking techniques is that the malicious app does not know when and where the user clicked (this is ensured by Security Mechanism #1 described in Section III) and so it does not have precise control on when to move to the next step, making the attack less practical.

5.5.2 Context-Hiding :

Android OS features a security mechanism based on the obscured flag: an object receiving a click event can check whether the user click “passed through” an overlay. We also mentioned the security researcher determined how this mechanism is implemented in an insecure way: as long as the user clicks on a part that is not covered, the flag is not set. We argue that this defense mechanism would be insecure even if it were implemented correctly. In fact, we believe this mechanism is vulnerable by design: if the user can only see the OK button, how can she know what she is authorizing? Is she clicking the OK button on an innocuous game-related popup, or is she unknowingly enabling the accessibility service? By using context-aware clickjacking, it is easy to create overlays with a single hole in correspondence to the OK button, thus completely covering all security-relevant information. Thus, a malicious app can hide the real context from the user, and it can lure her into clicking on the OK button – even if the OK button is entirely visible.

5.5.3 Keystroke Inference :

This attack is based on a novel technique that attempts to circumvent Security Mechanism #1. In particular, we show how it is possible to use the well-intentioned obscured flag recently introduced by Google as a side channel to infer where the user clicked. The net result of this attack is that an app with just the SYSTEM ALERT WINDOW permission can record all keystrokes from the user, including private messages and passwords.

The attack works in several steps. We first create several small overlays, one on top of each key on the keyboard, as shown in Figure 2. Of course, during a real attack, these overlays would be completely transparent and thus invisible for the user. The overlays are created with the following flags: TYPE_SYSTEM_ALERT, FLAG_NOT_FOCUSABLE, FLAG_NOT_TOUCHABLE and FLAG_WATCH OUTSIDE_TOUCH. These flags make sure that each overlay does not intercept any click by the user (that is, when the user clicks on a keyboard’s key, the click will reach the keyboard, as the user would expect). However, note that, thanks to the FLAG_WATCH_OUTSIDE_TOUCH, each overlay receives a MotionEvent object for each click. these click events do not contain any information about where the user actually clicked.

However, we discovered that it is possible to use Google's obscured flag as a side channel to infer where the user actually clicked. This attack works thanks to the following two observations. First, the overlays are created in a very specific, attacker-known order, and they are organized as in a stack: overlay #0 (top left in Figure 2) is at the bottom of the stack, while overlay #42 (bottom right) is at the top: thus, each of these overlays has a different Z-level. The second observation is that, for each overlay, the obscured flag is set depending on whether or not the user clicked on an overlay that was on top of it. For example, if the user clicks on overlay #0, the MotionEvent events received by each overlay will have the obscured flag set to 0. However, if the user clicks on overlay #1, the event delivered to overlay #0 will have the obscured flag set to 1. More in general, we observed that if the user clicks on overlay #i, the events delivered to overlays #0*!* #(i-1) will have the obscured flag set to 1, while all the events delivered to the remaining overlays will have the flag set to 0.

Thus, by creating the overlays in a specific order and by collecting the events and their obscured flags, it is effectively possible to use the obscured flags as a side channel to determine on which overlay the user has clicked, thus breaking Security Mechanism #1. We were able to write a proof-of-concept that can infer all keystrokes in a completely deterministic manner

5.5.4 Keyboard app Hijacking:

The accessibility service is a powerful mechanism that has access to the content displayed by the apps the user is interacting with. However, for security reasons, it is designed so that it cannot get access to security sensitive information, such as passwords. In practice, this is implemented so that when an app attempts to read the content of an EditText widget containing a password, the getText() method always returns an empty string. This behavior is documented in android as any event fired in response to manipulation of a PASSWORD field does NOT CONTAIN the text of the password. It is possible to bypass this protection mechanism. In fact by specifying FLAG_RETRIEVE_INTERACTIVE_WINDOWS (according to the documentation, it indicates to the system that "the accessibility service wants to access content of all interactive windows"), the keyboard app itself (package name: com.google.android.inputmethod.latin) is treated as a normal, unprivileged app, and each of the key widget generates accessibility events through which it is easy to record all user's keystrokes, including passwords.

5.5.5 Password Stealer:

The attack works in several steps. First, the attacker uses the accessibility service to detect that, for example, the user just opened the Facebook app's login activity. At this point, the malicious app uses the overlay permission to draw an overlay that looks like the username and password EditText widgets. Note how, differently from the previous attacks, the widgets are actually visible: however, they match exactly the Facebook user interface, and the user does not have any chance to notice them. Thus, the unsuspecting user will interact with the malicious overlays and will type her credentials right into the malicious app.

To make the attack unnoticeable to the user, the malicious app would also create an overlay on top of the login button: when the user clicks on it, the malicious overlay would catch the click, fill in the real username and password widget in the Facebook app, and finally click on the real Facebook's login button. At the end of the attack, the user is logged in her real Facebook account, leaving her completely unsuspecting. Moreover, by checking whether the login attempt was successful, our attack can also confirm that the just-inserted credentials were correct.

We note that this technique is generic and it can be applied to attack any app (e.g., Bank of America app). Moreover, the malicious app would not need to contain any code of the legitimate app, thus making it more challenging to be detected by repackaging detection approaches. Our attack, in fact, replaces many of the use cases of repackaging-based attacks, making them superfluous.

5.5.6 Enabling all permissions:

Once the secondary malicious app is installed, it is trivial for the initial app to grant device admin privileges: we found that the “Enable” button is unprotected and it can be easily clicked through an accessibility event. However, the Switch widgets that the user needs to click are protected by the FLAG_WINDOW_IS_OBSCURED flag. It turns out that the current implementation of the mechanism that sets and propagates this security flag handles user-generated clicks and clicks generated by an accessibility service app in a different way. In particular, we observed that the flag is *not* set for any event generated by the accessibility service. Thus, we found it is possible to automatically click and enable all permissions while a full screen overlay is on top. At the end of this and the previous attack, the initial unwanted application was able to silently install a God-mode malicious app.

Chapter 6

Results

6.1 Enabling the permissions

We display a phishing page and when the user enters his login credentials we display an appropriate message such as incorrect details and redirect it to the original login page.

Initially we our app must have Usage Access permission to enable it to detect whether the target app is running in the foreground.
To enable this Usage Access Permission we create overlays to lure the user to enable the permission without any suspicion. In the screenshots mentioned below (figure 6a) the first “OK” is used to select the app which requires the Usage Access Permission and in the second “OK” (figure 6b) we enable the permission.

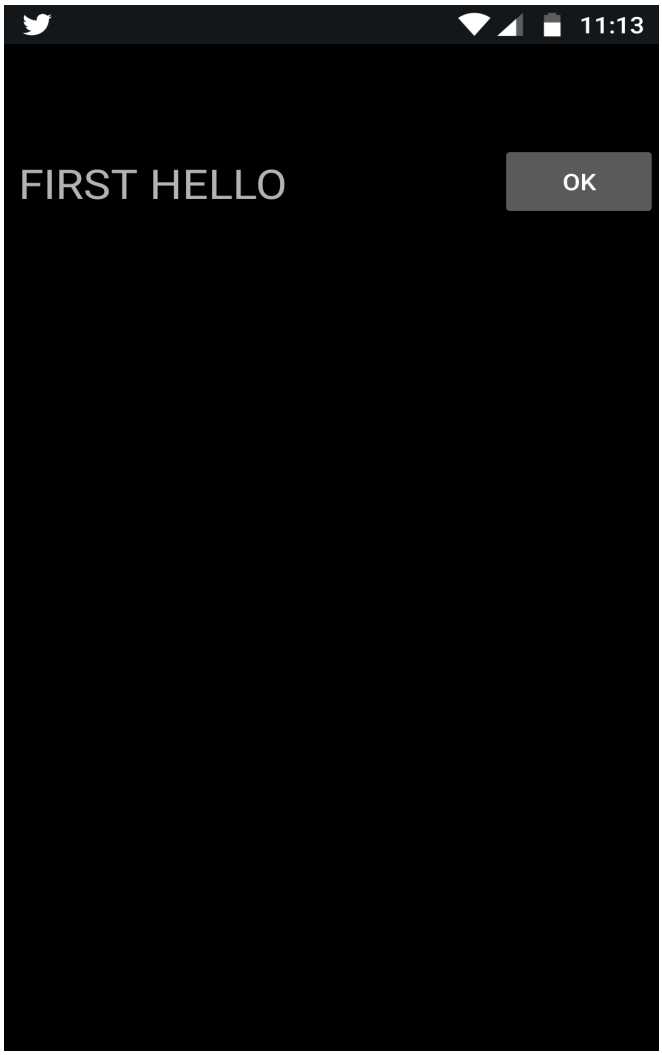


Fig-6a

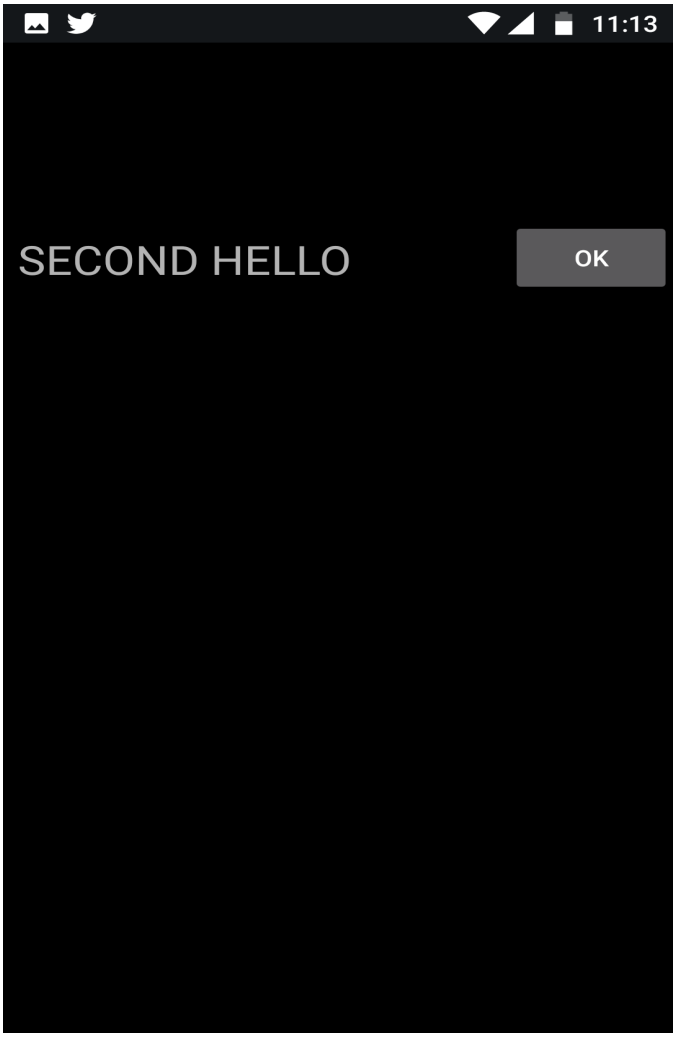


Fig-6b

6.2 Launching The Attack On Target App:

As soon as our malicious app is installed by the user it continuously checks if the Usage Access Permission is enabled or not. After the permission is enabled the app attains the power to check which app is running in the foreground and if it finds out that the target app is running in the foreground we use draw on top permission to launch our malicious window over the target app as soon as the user opens it. And if the user enters his login credentials we close our overlay and redirect him to original app by displaying an appropriate message.

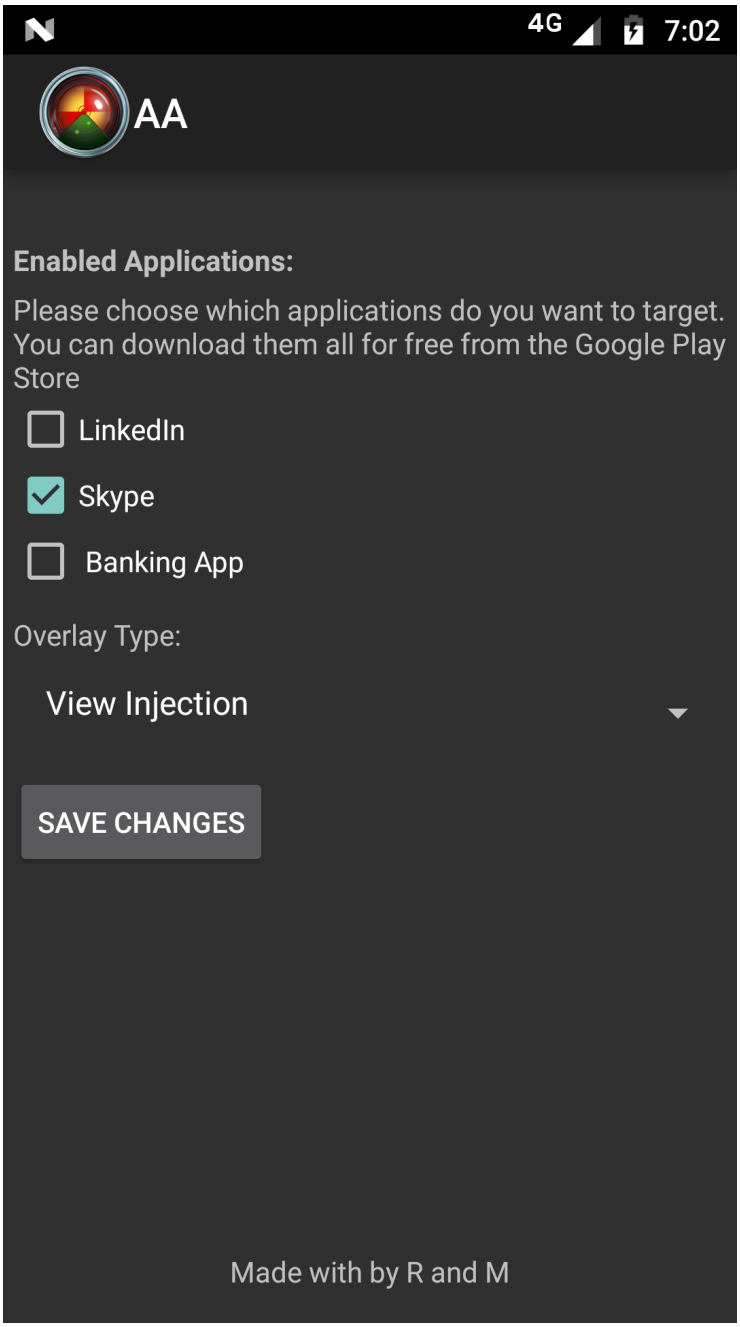


Figure -6c Choosing the Target App

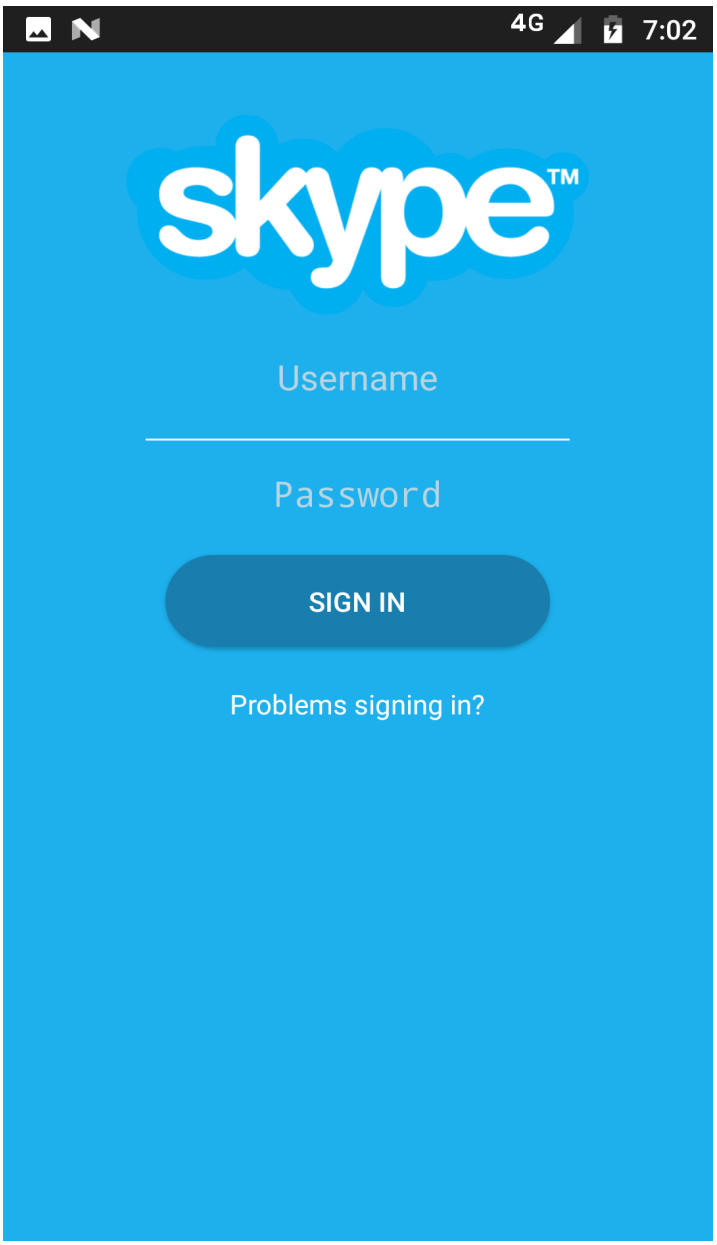


Fig-6d Displaying our overlay over target app.

References

[1] “Android studio”, developer.android.com, [Online]. Available: https://developer.android.com/

[2] https://www.kaspersky.com/blog/cloak-and-dagger-attack/16960/

[3] http://cloak-and-dagger.org/

[4] http://resources.infosecinstitute.com/understanding-cloak-dagger-attack-overview-tutorial/#gref

[5] https://securityzap.com/cloak-and-dagger-attacks-android/

[6] https://gbhackers.com/millions-of-android-phones-including-latest-versions-vulnerable-to-cloak-dagger-attack/

[7] https://www.tutorialspoint.com/android/android_services.htm

[8]http://iisp.gatech.edu/sites/default/files/documents/ieee_sp17_cloak_and_dagger_final.pdf