Beyond the Surface: Uncovering the Unprotected Components of Android Against Overlay Attack

Hao Zhou[1*], Shuohan Wu[1*], Chenxiong Qian[2], Xiapu Luo[1§], Haipeng Cai[3] and Chao Zhang[4]
[1]The Hong Kong Polytechnic University, [2]University of Hong Kong, [3]Washington State University, [4]Tsinghua University

***Abstract*—Overlay is a notable user interface feature in the Android system, which allows an app to draw over other apps' windows. While overlay enhances user experience and allows concurrent app interaction, it has been extensively abused for malicious purposes, such as "tapjacking", leading to so-called overlay attacks. In order to combat this threat, Google introduced a dedicated window flag `SYSTEM_FLAG_HIDE_NON_SYSTEM_OVERLAY_WINDOWS` to protect critical system apps' windows against overlay attacks. Unfortunately, the adequacy of such protection in the Android system remains unstudied, with a noticeable absence of clear usage guidelines.**

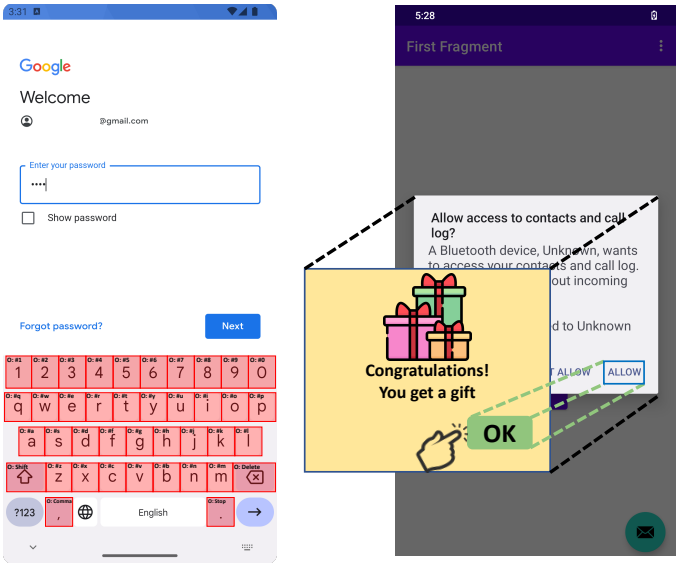
To bridge the gap, in this paper, we conduct the first systematic study on the unprotected windows of system apps against overlay attacks. We propose a comprehensive guideline and then design and develop a new tool named OverlayChecker to identify the missing protections in Android system apps. To verify the uncovered issues, we also design and create Proof-of-Concept apps. After applying OverlayChecker to 8 commercial Android systems on 4 recently released Android versions, we totally discovered 49 vulnerable system apps' windows. We reported our findings to the mobile vendors, including Google, Samsung, Vivo, Xiaomi, and Honor. At the time of writing, 15 of them have been confirmed. 5 CVEs have been assigned, and 3 of them are rated high severity. We also received bug bounty rewards from these mobile vendors.

I. Introduction

Overlay or floating window is one of the key UI features of Android. It allows an app to draw over other apps' windows for improving usability [73]. For example, an instant messaging app (e.g., Facebook Messenger) creates an overlay, appearing on top of other apps' windows, to let users conveniently access the received messages. Meanwhile, overlay allows users to interact with multiple apps at the same time. For example, a video streaming app (e.g., Youtube) creates an overlay to play videos while letting users interact with other apps simultaneously (see Fig. 11). Therefore, overlays are widely adopted by apps to improve user experience. A recent study [73] reported that around 35.4% of the top 500 popular apps on the Google Play Store create overlays to improve their usability.

Like one coin has two sides, overlays are also widely abused by malicious apps to launch attacks that compromise users' security and privacy. Numerous studies [48, 52, 53, 59, 75] have found that attackers can abuse overlays to steal users' private information by monitoring user input and lure users to allow user consent for performing sensitive operations by covering the sensitive widgets. For example, in Fig. 1a, a malicious app creates an overlay on top of the system's input method to eavesdrop on users' touch events to steal usernames and passwords. As another example, in Fig. 1b, the malware creates an overlay on top of the `Allow` button in the system's settings app's permission request window, deceiving users into granting the requested permissions to the malware.

Since system apps implement many security-sensitive functionalities (e.g., permission managing and debugging) to protect users' security and privacy, they usually explicitly ask for user consent before conducting sensitive operations. Hence, it is essential to protect system apps against overlay attacks. To achieve this purpose, recently released Android systems (Android 10.0 $\sim$ 13.0) provide a dedicated window flag namely `SYSTEM_FLAG_HIDE_NON_SYSTEM_OVERLAY_WINDOWS` (short for `HNSOW`) for system apps to defend against overlay attacks. Specifically, system apps' windows can enable `HNSOW` to prevent overlays created by third-party apps from drawing over them [38]. Thus, `HNSOW` can prevent malware from abusing overlays to intercept users' touch events and lure users to allow user consent, mitigating both attacks in Fig. 1.



(a) Keystroke inference [53]. (b) User consent bypass.

Fig. 1: Two cases of overlay attacks.

Although `HNSOW` can effectively protect system apps against overlay attacks, it is unknown whether this flag is adopted by the system apps whose windows need protection. Unfortunately,

*Co-first authors.
§The corresponding author.

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.24035
www.ndss-symposium.org

we find that Google is constantly applying patches to enable HNSOW in system apps' windows in recently released Android systems (Android 10.0 $\sim$ 13.0) [13–19, 21–23], indicating that missing protections against overlay attacks commonly exists in system apps. To make things even worse, we notice that the guideline for determining whether a window needs protection against overlay attacks is missing. Although Google outlines three cases of windows that need protection [39], we discover that they just cover very few protected windows of system apps. Lacking the guideline raises two problems. On one hand, in the official Android system developed by Google (i.e., AOSP [10]), a part of system apps' windows misses protection. On the other hand, since mobile vendors usually introduce additional system apps in their customized Android systems, a lack of protections commonly exists in these system apps. To enhance the security of system apps and enable them to defend against overlay attacks, there is an urgent need for a proper guideline to determine whether a system app window requires protection and a systematic approach for identifying unprotected windows.

In this paper, we systematically investigate the vulnerability of missing protections against overlay attacks in Android systems. First, we semi-automatically analyze the protected windows in the official Android system to build a guideline for determining whether a window of a system app requires protection (see §IV). More specifically, we automatically find all protected system apps' windows using static code analysis. Then, we conduct an extensive manual analysis on the found windows to gain insights into and summarize their common features in three aspects, including *startability*, *functionality*, and *interactivity*. Second, based on the guideline, we design and develop a new approach, named OverlayChecker, to uncover the windows that miss protections (see §V and §VI). In detail, OverlayChecker statically examines the bytecode code of system apps and Android framework to determine whether the features of a window are consistent with the guideline. If so, it further inspects whether HNSOW has been enabled in the window. If not, a window missing protection is uncovered. After uncovering a vulnerable window, we also craft a Proof-of-Concept (PoC) app to validate the vulnerability and study the potential security impact. To mitigate the manual effort, we generate the code for launching the targeting window by statically analyzing the system app (see §VII).

To evaluate the effectiveness of OverlayChecker, we apply it to find the system apps' windows that are vulnerable to overlay attacks in 8 commercial Android systems on 4 recently released Android versions (Android 10 $\sim$ 13) from 5 mainstream mobile vendors, including Google, Samsung, Vivo, Xiaomi, and Honor. In total, we uncover 49 unprotected windows, where attackers can abuse overlay to lure users to allow consent for conducting security-sensitive operations (e.g., privilege escalation), causing severe security and privacy problems to users and their devices. We have reported our findings to the corresponding mobile vendors. At the time of writing, 15 of them have been confirmed. 5 CVEs have been assigned, and 3 of them are rated high severity. We also received bug bounty rewards from Google, Samsung, and Vivo.

In summary, we make the following contributions:

- We are the first to systematically investigate the vulnerability of missing protection against overlay attacks in windows of Android system apps.
- We summarize the criteria for determining whether a system app's window needs protection against overlay attacks. Based on it, we design and implement a tool named OverlayChecker to uncover unprotected windows in Android systems.
- We apply OverlayChecker to 8 commercial Android systems on 4 recently released Android versions. OverlayChecker totally finds 49 unprotected windows, which are vulnerable to overlay attacks and can be compromised and lead to severe security and privacy problems.

II. Background

In this section, we present the knowledge about the Android app's user interface in §II-A and the Android system's architecture in §II-B. Then, we introduce overlay in Android in §II-C and the flag SYSTEM_FLAG_HIDE_NON_SYSTEM_OVERLAY_WINDOWS for preventing overlay attacks in §II-D.

A. User Interface of Android App

Activity. Activity represents a single screen in an app that incorporates various user interface (UI) components (e.g., Button and ImageView) [4]. It acts as an entry point for users to interact with an app. While an app typically consists of multiple activities, each one operates independently, collaboratively contributing to a cohesive user experience.

Window. Window serves as a top-level container that occupies a rectangular region on the device's screen, providing a canvas for rendering and manipulating UI content [40]. Each activity in an app has its own window, usually occupying the entire screen. However, it can also be smaller and float above other windows. It is worth mentioning that activities can create sub-windows (e.g. dialogs) to present UI content [26].

```
01<activity android:name="NfcImportVCardActivity"
02   android:exported="true"
03   android:enabled="true"
04   <intent-filter android:priority="1">
05     <action android:name="NDEF_DISCOVERED"/>
06     <data android:mimeType="text/vcard"/>
07     <category android:name=".category.DEFAULT" />
```

Fig. 2: A simplified example of the manifest file of app.

Intent. Intent is a messaging object designed for implementing inter-component communication in Android. An app component (i.e., activity, service, broadcast receiver, and content provider [30]) uses Intent to request operations from another component.

TABLE I: Composition of Intent.

Category	Sub-cat	Type	Manifest	Retrieving API
Basic	Action	String	<action >	getAction()
	Category	Set<String>	<category>	getCategories()
	Data	String	<data>	getIntent()
	Type	String	<data>	getType()
Extra	<k,v>	k: custom String v: multiple Types	-	getStringExtra() getFloatExtra() getParcelableExtra()

An Intent object can carry a set of data items, providing the necessary information for the requested operation. We outline the data items that Intent can supply in Table I. These data can be classified into two categories, namely the basic attribute and

the extra attribute. Basic attributes can be declared in intent-filters within the manifest file (Lines 4-7 in Fig. 2), while extra attributes are generally specified programmatically. The extra attribute is formatted as a key-value pair (<k,v>). The key is a string and the value could be of various types. Android offers APIs for the component receiving the Intent to retrieve the data. For instance, the `getAction` API can be called to obtain the intended `Action`. If the value of the extra attribute is of the `String` type, the API `getStringExtra` can be called to extract the string value.

B. Architecture of Android System

The Android system is built on a multi-layered architecture [35]. Fig. 3 shows the three core layers of the Android system, including the Library layer, the Android Framework layer, and the Application layer. System services (in the Android Framework layer) provide APIs for apps (in the Application layer) to interact with the Android system and perform sensitive operations. For example, the Telephony service provides APIs for apps to manage phone status and perform phone operations, and the LocationManager service provides APIs for apps to retrieve the sensitive location information of the device. Note that, some system services (e.g., the Camera service) rely on native code to access hardware and implement sensitive functionalities. To achieve this purpose, they will invoke functions in native libraries (in the Library layer).

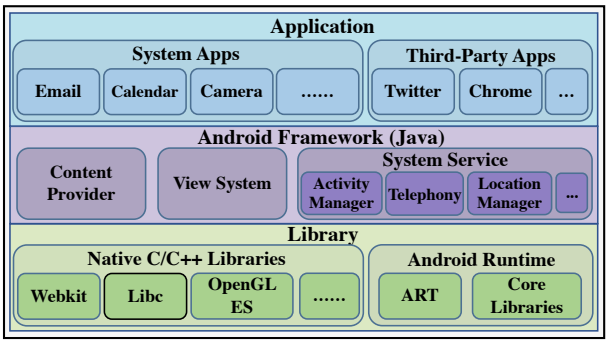


Fig. 3: Android system architecture.

C. Overlay

Overlay is a floating window that can be created by an app to overlap another app's windows. It is designed to enhance user experience by providing additional information or functionality without requiring navigation away from the current window. Overlay serves a range of purposes, such as the display of tooltips, temporary notifications, or offering interactive controls. Therefore, overlays are commonly used in apps. A recent study [73] surveyed the top 500 apps on the Google Play Store and found that 35.4% of them use overlays.

To create an overlay, an app needs to request the permission `SYSTEM_ALERT_WINDOW`. Prior to Android 7, this permission is automatically granted for apps installed from Google Play Store, without requiring the user's explicit consent [53]. However, such improper permission control leads to the notorious "Cloak & Dagger Attack" (or overlay attacks) [53], in which malicious apps can abuse overlay to monitor users' input events or lure users to approve user consent (see Fig. 1), resulting in significant security risks and privacy issues. To mitigate this problem, in the later versions of Android, apps need to declare the permission `SYSTEM_ALERT_WINDOW` in their manifest files, and explicitly ask the user to enable this feature within the system settings app (see Fig. 12a).

D. Hide Non-System Overlay Windows Flag

Since system apps usually implement security-sensitive functionalities, before conducting these operations, they ask for user consent. To protect system apps against overlay attacks, the flag `SYSTEM_FLAG_HIDE_NON_SYSTEM_OVERLAY_WINDOWS` (short for `HNSOW`) has been introduced in the recently released Android systems (since Android 10). Specifically, by enabling the `HNSOW` flag, system apps disallow non-system apps (i.e., third-party apps) to draw overlays on top of their windows.

For example, `GrantPermissionsActivity` [28] is an activity in the permission management system app. It will ask for user consent before performing the security-sensitive operation of granting runtime permissions to apps. Therefore, to defend against overlay attack, as depicted in the following code snippet, this activity calls the API `addSystemFlags` to enable `HNSOW` in Line 3-4.

```
1 public class GrantPermissionsActivity
2   protected void onCreate(Bundle b){
3     getWindow().addSystemFlags(
4       SYSTEM_FLAG_HIDE_NON_SYSTEM_OVERLAY_WINDOWS);}
```

Remark. It is important to note that the Android system enables `HNSOW` in a select portion of system app activities that require protection, rather than in all activities of system apps. This strategy strikes a balance between security concerns and user experience. For instance, as shown in Fig. 3, system apps (e.g., system launcher app, system email app, system messaging app, system settings app) provide various useful functionalities, and users frequently interact with them when using Android smartphones. If the windows of all system apps have `HNSOW` enabled, overlays become almost unusable. To minimize the impact on user experience, the system does not need to enable the `HNSOW` flag on the targeted window if a system app's window is not under potential threat from malicious overlays.

III. Motivation

Over recent years, as listed in Table II, a series of vulnerabilities of missing protections against overlay attacks in Android system apps have been exposed. For example, CVE-2021-0333 [16] discloses that the activity `BluetoothPermissionActivity` in the system settings app was left unprotected against overlay attack. As a consequence, malware can abuse overlay to obscure the activity for luring users to allow the consent to grant phonebook read permissions to a connected Bluetooth device. We consider that the root cause of these issues stems from the absence of a guideline, outlining which windows of system apps necessitate protections against overlay attack (i.e., enabling `HNSOW`). Through reviewing Google's documentation [39], we only derive a vague guidance, as presented below:

- The window for granting permission.
- The window for approving app installation.
- The window for showing a persistent sensor icon or equivalent privacy-sensitive notification.

Although the guidance covers three categories of system apps' windows where overlay could pose security problems, it misses a large portion of windows requiring protection. In particular, the guidance only covers three vulnerabilities (i.e., CVE-2021-0314, CVE-2021-0333, CVE-2021-1016) in Table II, while the remaining cases are non-compliant with the guidance.

TABLE II: Partial of CVEs about overlay attacks in Android.

CVE number	System App	Activity
CVE-2022-20212 [23]	com.android.settings	RequestToggleWifiActivity
CVE-2021-1016 [22]	com.android.systemui	UsbPermissionActivity
CVE-2021-0992 [21]	com.android.settings	PaymentDefaultDialog
CVE-2021-0538 [19]	com.android.phone	EmergencyCallbackModeExitDialog
CVE-2021-0523 [18]	com.android.settings	WifiScanModeActivity
CVE-2021-0391 [17]	android	ChooseTypeAndAccountActivity
CVE-2021-0333 [16]	com.android.settings	BluetoothPermissionActivity
CVE-2021-0314 [15]	com.android.packageinstaller	UninstallerActivity
CVE-2020-0394 [14]	com.android.settings	BluetoothPairingDialog
CVE-2020-0015 [13]	com.android.certinstaller	CertInstaller

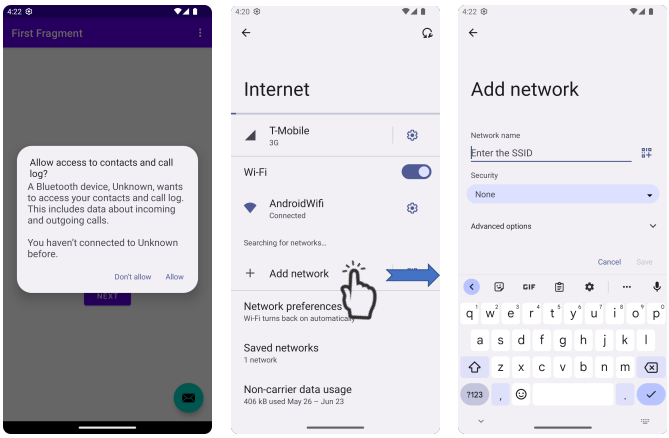
In light of the observations, a proper guideline for determining which windows of system apps require protection against overlay attacks is in urgent need. Additionally, a systematic approach to identifying unprotected windows is also crucial for bolstering security measures against potential overlay attacks.

IV. Guideline

To demystify why a system app's window requires protection against overlay attacks, we perform an in-depth analysis on the windows under protection in the recently released official Android system AOSP Android 12.

In order to complete the task, we propose a semi-automated approach, which includes three steps. First, we find the protected system apps' windows. Since windows call `addSystemFlags`, `addPrivateFlags`, or `setPrivateFlags` [5, 6, 37] and pass the value of `HNSOW` to these APIs to enforce the protection, we statically identify these APIs in bytecode of system apps and then perform data flow analysis on the parameters to identify the protected windows. Second, we study the features of these windows. Specifically, we manually inspect the source code of identified windows under protection to understand their functionalities. We also review the code modification history to find the potential reasons for system developers to enable `HNSOW`. Moreover, to observe their user-centric functionalities more intuitively, we use ADB [7] to launch them by constructing and sending appropriate intents. Third, we summarize the common features of the protected windows, which are treated as the guideline for determining whether a system app's window needs to be protected against overlay attacks.

To guarantee the reliability of our conclusions, we recruited four volunteers, two with over three years of Android development experience and two who have published Android related research papers in top-tier security conferences. In total, we found 66 windows that set `HNSOW`, and each volunteer devoted five days (about 30 hours) to analyzing them. They worked as a team and carried out extensive discussions on the features of each window, aiming to derive accurate and reliable conclusions. Based on the findings from manual analysis, we summarize the common features of the windows under protection in three aspects, including *startability*, *functionality*, and *interactivity*.



(a) One-step Launch. (b) Internet Window. (c) AddNetwork.

Fig. 4: Different cases of system apps' windows.

Startability. We discover that the windows under protection can be launched in one step.

The windows that can be directly launched (i.e., launched in one step) are more vulnerable to overlay attacks. For instance, the activity `BluetoothPermissionActivity` of the system settings app, which asks users to grant permissions to the connected Bluetooth devices (see Fig. 4a), can be directly launched by malware. Accordingly, while launching the activity, malware can draw an overlay on top of the activity to deceive users into clicking the "Allow" button to grant permissions. In this scenario, due to a lack of context information, users are unaware that they are interacting with `BluetoothPermissionActivity`. To prevent such overlay attacks, the Android system enables `HNSOW` in `BluetoothPermissionActivity`.

On the contrary, windows that cannot be directly launched, which indicates that users need to interact with other windows to launch them, are less vulnerable to overlay attacks because users will obtain more context information. For example, to launch the "AddNetwork" window in Fig. 4c, users need to first start the "Internet" window in Fig. 4b and then click the "Add network" button. With more interactions, users are more likely to recognize the interaction context. This contextual awareness makes overlay attacks challenging to execute. Hence, the system does not enable `HNSOW` in the "AddNetwork" window.

Functionality. We uncover that the windows under protection will perform sensitive operations.

Windows that implement sensitive functionalities, such as permission management, typically require user consent to perform such sensitive operations. Malicious apps can exploit overlays to deceive users into granting user consent, resulting in severe security consequences (e.g., privilege escalation). Therefore, these windows necessitate protection against overlay attacks. For instance, the activity `BluetoothPermissionActivity`, which implements the sensitive function of granting permissions to Bluetooth devices, must be defended against overlay attacks. As a result, system developers enable `HNSOW` in this activity.

Interactivity. We find that sensitive functionalities of protected windows can normally be executed with a limited number of user interactions, usually no more than two, which include one user interaction for launching the targeting window and another

TABLE III: Details about part of system apps' windows that are protected against overlay attacks in AOSP Android 13.

	App Name	Activity Name	One-step Launch	Sensitive Operations	Simplistic Interaction
1	com.android.settings	RequestToggleWiFiActivity	self.onCreate	WiFiManager.setWifiEnabled	press a button
2	com.android.systemui	UsbPermissionActivity	self.onCreate	UsbService.grantDevicePermission	press a button
3	com.android.settings	PaymentDefaultDialog	self.onCreate	Settings$Secure.putString	press a button
4	com.android.phone	EmergencyCallbackModeExitDialog	self.onCreate	Phone.exitEmergencyCallbackMode	press a button
5	com.android.settings	WifiScanModeActivity	self.onCreate	WifiManager.setScanAlwaysAvailable	press a button
6	android	ChooseTypeAndAccountActivity	self.onCreate	AccountManager.setAccountVisibility	press a button
7	com.android.settings	BluetoothPermissionActivity	self.onCreate	BluetoothDevice.setMessageAccessPermission	press a button
8	com.android.packageinstaller	UninstallerActivity	self.onCreate	PackageInstaller.uninstall	press a button
9	com.android.settings	BluetoothPairingDialog	self.onCreate	BluetoothDevice.setPhonebookAccessPermission	press a button
10	com.android.permissioncontroller	RequestRoleActivity	self.onCreate	RoleManager.addRoleHolderAsUser	press a button
11	com.android.nfc	ConfirmConnectActivity	BluetoothPeripheralHandover.onReceive	BluetoothHeadset.setConnectionPolicy	press a button
12	com.android.certinstaller	WiFiInstaller	CertInstallerMain.onCreate	WifiManager.addOrUpdatePasspointConfiguration	press a button

single user interaction for triggering the sensitive functionality of the window.

Windows requiring fewer user interactions to execute the sensitive functionalities are more vulnerable to overlay attacks. For example, after launching `BluetoothPermissionActivity`, it only needs one click event (i.e., clicking the "Allow" button) to perform the sensitive permission granting operation. Malware can easily launch the overlay attack by creating a deceptive overlay to lure users into clicking such a button. Since users have limited context information about the interaction, it is challenging for them to understand the consequences of such a simple click event.

Conversely, more user interactions generally imply a more complex chain of actions, which in turn reduces the likelihood of a user unintentionally performing these actions. As shown in Fig. 4c, adding a network requires four user interactions, including launching the activity, entering an SSID, selecting a security level, and clicking the save button. Although adding a network is a sensitive operation, it is less likely that an attacker can successfully deceive a user into performing multiple specific actions (excluding the operation for launching the activity) compared to just one. Multi-step interactions provide users with more opportunities to notice unusual behaviors, thereby raising their suspicions. It is worth noting that this aligns with the existing practices, as most mobile manufacturers, like Xiaomi, categorize the security risks that require more than two user interactions as negligible threats [31].

Based on these findings, we derive three criteria that serve as guidelines to determine whether a window of a system app requires protection. If a window simultaneously satisfies these three criteria, we consider that the window needs protection against overlay attack. Table III presents a partial list of system app windows that meet these criteria, all of which are protected against overlay attacks in the latest AOSP Android 13.

- **(C1)** *One-Step Launch*: The window can be directly launched.
- **(C2)** *Sensitive Operation*: The window implements security-sensitive functionalities.
- **(C3)** *Simplistic Interaction*: Sensitive operations of the window can be triggered via no more than one user interaction.

V. Overview of OverlayChecker

Fig. 5 illustrates the workflow of our approach OverlayChecker, which consists of two primary components. (1) The Discovery Module (see §VI) identifies the windows requiring protection against overlay attacks in a given system app via a three-step process. First, for each window in the system app, the module examines the app's manifest file and bytecode to determine if the window is one-step launchable (C1). Second, it identifies sensitive operations executed by the window, which originate from two primary sources (i.e., sensitive system APIs and sensitive content providers), thus determining if C2 is met. Third, the module inspects the event handlers involving sensitive operations to evaluate if they can be triggered through simple user interactions (C3). Once all windows requiring protection are identified, we verify whether they have enabled `HNSOW`. The windows that have not enabled `HNSOW` are labeled as "suspicious". (2) The PoC Creator Module (see §VII), constructs Proof-of-Concept (PoC) apps for these suspicious windows to practically confirm whether they are vulnerable to overlay attack. To reduce manual efforts in constructing PoC apps, this module generates the proper Intent objects, allowing us to launch the targeting windows. After launching, we manually analyze the windows to determine the regions for drawing overlays and the layouts on overlays. As a final step, we submit the PoC apps along with the bug reports to the mobile vendor.

VI. Discovery Module

A. Criteria 1: One-Step Launch

This step aims to identify all windows, including activities and dialogs, that can be directly launched (see C1 in §IV). To achieve this, we consider two types of windows. ❶) First, we examine windows that can be directly launched by external apps through Intent objects. ❷) Second, we also consider windows that can be launched by other components in the system (known as *preceding components*). In particular, we focus on instances where the preceding components are invisible, and the launch of the targeting window does not require any user interaction. The rationale is that visible and interactive preceding components will enhance users' awareness of their interaction context, thereby arousing suspicion. The second type can be further divided into two subclasses. The first subclass includes windows that can be started by system services, broadcast receivers, or services of system apps (all of which work in the background

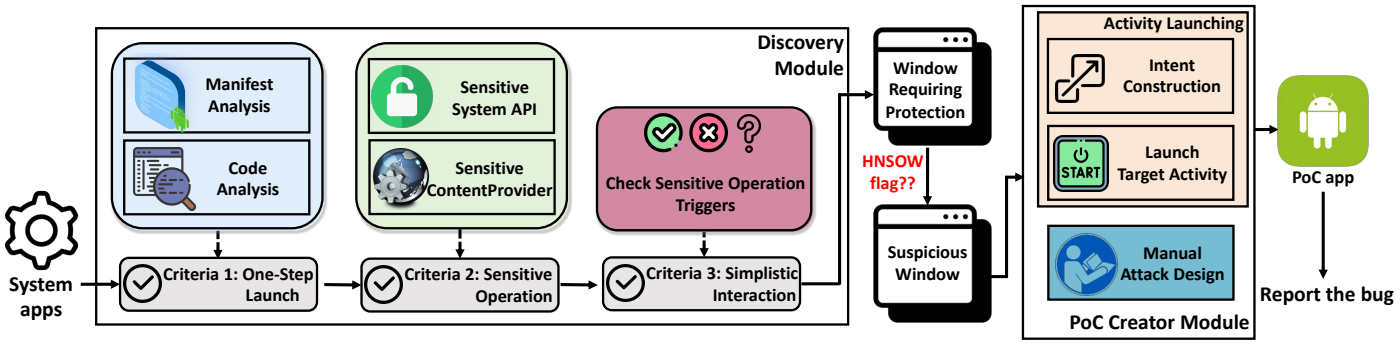


Fig. 5: Workflow of OverlayChecker.

without a user interface [8]). Note that, a service of a system app and a system service are two different concepts. The former is an application-level component for a specific app, while the latter represents an OS-level service that any app on the device can use to perform common tasks (see Fig. 3). The second subclass includes windows that are instantly launched during the startup of other windows (e.g., those launched within `onCreate` methods of activities).

In detail, given a system app, we first use `FlowDroid` [46] to parse the manifest file, which contains information about the app's components [11]. For each activity in the manifest file, we examine two attributes: "exported" and "enabled" (see Fig. 2). The "exported" attribute indicates whether the activity is accessible to other apps or components outside of its own app. The "enabled" attribute determines whether the activity can be instantiated. Therefore, for an activity, if both the "exported" and "enabled" attributes are set to "true", we consider it as type ❶, meeting C1. For activities that cannot be instantiated (i.e., "enabled" is set to "false"), we omit them because there is no possible way to launch them.

For each activity that is "enabled" but not "exported", we conduct a more in-depth examination to pinpoint the component responsible for launching it. First, we construct the call graph of system apps and then analyze the call graph to identify the system APIs (e.g., `startActivity`), which are called to launch the targeting activity (e.g., Line 12 in Fig. 8a). From there, we traverse the call graph backward to identify the caller. If the caller is found to be another activity's startup method (i.e., `onCreate`, `onStart`, or `onResume`), we further examine whether this activity is "enabled" and "exported" (i.e., type ❷). In other cases, if the caller is an intent handling method in broadcast receivers (e.g., `onReceive`) and system apps' services (e.g., `onCreate`, `onStartCommand`, `onHandleIntent`) or an interface of system service which can be called by apps, we also categorize the launched activity as type ❷.

In addition, we adopt a similar approach to identify the type ❷ dialog. More precisely, we first identify the system APIs (e.g., `Dialog.show`), which are called to launch the dialog, and then traverse the call graph backward to examine whether these APIs are called by an activity's startup method.

Moreover, it is common for preceding components to launch the windows using asynchronous mechanisms, such as Thread, Message Handler, and AsyncTask [47]. To account for this, we follow the existing practice [47] to analyze and recover the implicit function calls and then add the missing call graph edges. By doing so, from the call graph, we can find out whether the code, launching the targeting windows, can be triggered by preceding components.

B. Criteria 2: Sensitive Operation

Through our experience and a review of existing studies [66, 77, 78], we find that sensitive operations can be mainly conducted by system apps in two ways.

Invoking sensitive APIs. System apps can execute sensitive operations by calling sensitive APIs, such as those granting access to contacts or cameras [76]. Since sensitive APIs are implemented in system services [78], we first analyze the system services to identify the sensitive APIs and then inspect system apps to determine whether the identified sensitive APIs are called in system apps' windows.

In detail, since system services commonly adopt permission check [78] to enforce access control in sensitive APIs, we analyze the bytecode of system services to build the call graph for finding the interfaces that call permission check methods. These interfaces are treated as sensitive APIs. More specifically, we follow the existing approach [78] to first collect the remote interfaces of all system services, which are then used as the entry points for conducting call graph traversal to identify which interfaces call permission check methods.

With a complete collection of sensitive APIs, we analyze the control flow of the window's event handlers to determine if they perform sensitive operations by calling sensitive APIs in response to user interactions. Given that, we build the call graph and control flow graph for system apps and find whether the sensitive APIs are called by the event handlers (e.g., `onClick`) of the targeting windows (identified in §VI-A). We focus on event handlers because executing sensitive operations commonly requires user consent, such as a user interaction on the "ok" or "confirm" button.

Accessing Sensitive Content Providers. System apps, in some cases, can execute sensitive operations by accessing sensitive content providers (see §II-B), such as modifying system data or settings, which could permanently impact the user's device. Similarly, to ensure safety, Android enforces access restrictions (i.e., permissions) on these sensitive content providers. The permissions required by them are declared in their corresponding Manifest files. As such, if an event handler accesses a permission-protected content Provider, we consider there is a sensitive operation. Android provides a

unified interface (i.e., `ContentResolver`) for all apps to access content providers. When we detect the use of this interface within the control flow of an event handler, we follow the same steps as the previous work [76] to infer the content provider being accessed. Then, we examine the relevant Manifest file to check if any permission is required for access.

```
01 public class A-Activity{
02 protected void onCreate(Bundle bundle) {
03   Intent intent = new Intent(B-Activity.class);
04   startActivityForResult(intent);}

05 protected void onActivityResult(int resultCode,
                                   Intent result){
06   if(resultCode==RESULT_OK){
07     /**sensitive action**/;
08   }
09   sensitiveAPI(result);}
-------------------------------------------------
10 public class B-Activity{
11   public void onClick(){
12     Intent data = new Intent();
13     setResult(RESULT_OK,data);
14     finish()}
```

Fig. 6: Sensitive operation deferred to parent Activity.

Beyond the common scenario where the event handler directly executes the sensitive operation, we also consider a special case where the sensitive operation is deferred and executed in the parent activity. As shown in Fig. 6 the parent activity (A-Activity), starts another activity (B-Activity) in Line 4 using `startActivityForResult`. Within B-Activity, the user performs an action (i.e., click on a button), which produces a result and informs the parent activity via `setResult` in Line 13. Back in A-Activity, in `onActivityResult` that handles the returned data from previously started activity, it will perform sensitive operations based on the returned data. In this case, a single user interaction also activates a sensitive operation. To handle this, for each target Activity, we first scan all the `startActivityForResult` across the app, aiming to identify its parent activity. This involves conducting backward slicing on the arguments of `startActivityForResult` to confirm if the target activity is specified therein. Next, we inspect the parent activity's `onActivityResult` to identify if it contains any sensitive operations. If true, we further examine whether these sensitive operations are dependent on the returned data. Notably, both `resultCode` and `result` arguments of `onActivityResult` (Line 5) can store the returned data. We analyze both the data and control dependency between these two arguments and sensitive operations. We explore the data dependency because the returned data might be directly used in sensitive operations (Line 9), while control dependency is examined because the returned data could influence the execution of the sensitive operations (Line 6). If any such dependency is found, we classify the target activity as satisfying C2.

C. Criteria 3: Simplistic Interaction

To determine if an Activity meets C3 criteria, we investigate whether sensitive operations within it can be triggered through a single user interaction. As previously mentioned, multi-step interactions significantly reduce the probability of successful overlay attacks. To facilitate this, we aim to detect if sensitive operations within the event handler depend on other user inputs. This dependency can be established in two ways. First, as demonstrated in Line 6 of Fig. 7, the sensitive operation is protected by a global variable, which needs to be modified in other event handlers (Line 4) to satisfy the branch constraint. The second way is shown in Line 7, where a branch condition utilizes the text entered by the user. Although no global variable is used, the user still requires the correct interaction sequence to trigger the sensitive operation.

```
01 private String current;
02 private EditText editText;
03 public void onSwipeLeft(){
04   current = getSomething();}
05 public void onClick(){
06   if(current == null) {return;}
07   if(editText.getText() == null) {return;}
08   /**sensitive action**/;}
```

Fig. 7: Two ways that sensitive operations rely on user inputs.

Based on these observations, to complete the task, we first analyze and construct the control flow of the event handler containing the sensitive operations (identified in §VI-B). Then, we traverse the control flow to identify all branch statements preceding the sensitive operation. When a global variable is used in the branch condition, we further assess whether its default value can trigger sensitive operations. Specifically, we examine the target Activity's constructor (`<init>`) and initialization life-cycle functions (`onCreate`, `onStart`) to find the assignment statement of this global variable, thereby determining its default value. Subsequently, we use the concolic execution to verify if this default value can satisfy the branch condition and thus reach the sensitive operations. If so, we conclude that the sensitive operation can be triggered by a single user interaction, thus meeting C3. Otherwise, we consider the target Activity as not meeting C3. In addition, if the branch condition uses the return values from user-input-dependent methods (e.g., `getText`, `isChecked`), which require user input to yield results, we also regard the target Activity as not meeting C3.

VII. PoC Creator Module

After applying three criteria, we identify the windows within system apps that necessitate overlay protection. Next, we inspect these windows to see if they enable the `HNSOW` flag through control flow analysis, thus identifying suspicious windows that may miss overlay protection. To practically validate the potential discovered issues, this module is designed to build PoC apps for them. To facilitate this process and reduce manual efforts, we perform static analysis to automatically construct the window launching context. Once the suspicious window is launched, we manually design an overlay to be drawn over it within our PoC app.

A. Launching Target Activity

As introduced in §II-A, an Activity can be externally launched via an Intent object. When creating such an Intent to start an Activity, it is crucial to provide the proper context (e.g., "Action" and "View" in Table I). This ensures that the triggered Activity has the necessary resources and can exhibit the expected behaviors. Take the example in Fig. 8a, where the Activity, upon launch, first invokes its life-cycle method `onCreate`. To access the context during this initialization, it retrieves the Intent using the API `getIntent`. Subsequently,

```
01 public class NfcImportVCardActivity{
02 protected void onCreate(Bundle bundle) {
03   Intent intent = getIntent();
04   if(!NDEF_DISCOVERED.equals(intent.getAction())){
05      finish();
06      return;}
07   String type = intent.getType();
08   if(type==null || (!"text/x-vcard".equals(type)
09                  && !"text/vcard".equals(type))){
10      finish();
11      return;}
12   startActivityForResult(new Intent(this,
13   SelectAccountActivity.class), SELECT_ACCOUNT);}
```

(a) Basic Attribute.

```
01 protected void onCreate(Bundle bundle) {
02   Intent intent = getIntent();
03   String usrName=intent.getStringExtra(USER_NAME);
04   PersistableBundle actOptions =
       intent.getParcelableExtra(USER_ACCOUNT_OPTIONS);
$r0 = @this;
$r1 = virtualinvoke $r0.<.Activity:..Intent getIntent()>();
……
$r3 = virtualinvoke $r1.<..Intent: ..Parcelable
     getParcelableExtra(.lang.String)>("USER_ACCOUNT_OPTIONS");
$r4 = (android.os.PersistableBundle) $r3;
```

(b) Extra Attribute.

Fig. 8: Intent Construction.

the Activity extracts the attached data from the Intent through several APIs (e.g., `getAction`). The extracted data is then used for various purposes, such as determining branch paths.

To construct the correct Intent to launch the target Activity, we employ a three-step approach. First, for the basic attributes (see Table I) declared in the manifest file, we scan the manifest file to determine their values. For example, in Fig. 8a, for the `NfcImportVCardActivity`, its expected intent `Action` is `"NDEF_DISCOVERED"`. This value can be obtained by inspecting the <action> element within the Intent-filter (see Fig. 2). Among the basic attributes, the `Data` attribute stands out for its distinctive requirement. The value of the <data> element in the manifest cannot be directly used by the Intent. Instead, it specifies the type of data that the target Activity can handle. For example, in Fig. 2, `<data android:mimeType="text/vcard"/>` indicates that we need to input a VCard file (i.e., .vcf file) into the Intent as its `Data` attribute. To better handle `Data`, we establish a mapping for each specific type, e.g., we use random text for the type `text/plain`.

For extra attributes, which are key-value pairs, our approach aims to identify the correct key and the type of its value. Activities can use several specific system APIs, such as `getStringExtra` in Line 3 of Fig. 8b, to retrieve extra attributes using user-defined keys. Leveraging a list of APIs collected by prior research [71], which includes 29 APIs that can access extra attributes of different types, we locate the invocations of these APIs within the `onCreate` method of the target Activity. As a result, we can determine the key and the type of the extra attribute. For example, the key in line 3 is "user name" and the value is a string. Parcelable and Serializable are two special types of extra attributes, both of which are implemented as generic types for Android serialization. As shown in Line 4, when an object is received from an Intent using Parcelable, it needs to be cast to a specific type. To handle this, for APIs like `getParcelableExtra`, we track the data flow from its return value to find the cast statement (`r4`), allowing us to determine its types. It is worth noting that extra attributes are typically used for providing data. Therefore, after extracting their keys and types, we follow the existing practice [71] to assign random values based on their types.

For activities that need to be launched by other components (Activities, services, broadcast receivers), we launch them by sending Intents to their preceding components. Given that these components typically rely on the `Action` attribute within the Intent to determine the events to execute (e.g., launching an Activity) [12], we must determine the correct `Action` to set. To achieve this, we first trace the control flow path from the code point at which they receive the Intent to the target Activity's launch. Then, we adapt the approach in [71] to perform constraint solving on the `Action` attribute to determine its value. For other data attached within the Intent, we reuse the previous approach to identify the key and the type of its value. For Activities that cannot be launched through automatically constructed Intents (e.g., some Activities use extra attributes as condition constraints), we manually analyze their code to build their respective Intents. For third-party ROMs without source code, we use tools such as JEB [24] to analyze the decompiled code.

B. Manual Attack Design

After launching the suspicious window, we manually craft a PoC app tailored for it. Our goal is to cover the entire suspicious window's region with an overlay. We utilize uiautomator2 [9], an Android UI testing framework to obtain its coordinates and size. Once the suspicious window is launched, the PoC app swiftly draws an overlay of the corresponding position to cover it. We also make this overlay indistinguishable from the PoC app interface. By doing so, we effectively obscure the user's view of the underlying window, making it challenging for them to recognize their interactions with the system app due to the lack of contextual clues. Subsequently, we manually analyze the code of the suspicious window, to determine which widget (e.g., Button) triggers the sensitive operation. This knowledge helps us craft the deceptive widget within our overlay, which is placed at the exact coordinates of the sensitive widget in the actual Activity. It's noteworthy that our PoC app will request the SYSTEM_ALERT_WINDOW permission from the user, aligning with the threat model assumed for overlay attacks (see Appendix A).

VIII. Evaluation

In this section, we evaluate the performance and functionalities of `OverlayChecker` by answering the following four research questions (RQs).

RQ1: Are the guidelines reliable for determining whether a window of the system app requires protection?

RQ2: Can `OverlayChecker` effectively uncover the system apps' windows missing protection in Google AOSP?

RQ3: Can `OverlayChecker` effectively identify the system apps' windows lacking protection in third-party Android ROMs?

RQ4: What are security impacts of missing protection against overlay attacks in unprotected windows of Android systems?

We develop the `OverlayChecker` with more than 8k SLOC in Java and around 1k SLOC in Python. The prototype is

implemented based on several existing tools: `ADB` [7], a tool that helps dump app from Android devices; `Apktool` [3], a tool used to decompile Android apk; `Soot` [46], a static analysis framework we use to construct call graphs and perform control and data flow analyses; `Z3` [69], an SMT solver that we build our concolic testing engine; and `Fax` [71], a framework that assists in constructing the Intent objects.

TABLE IV: Details about Android systems under analysis.

	System	Version	Vendor	Date	#App	#Miss
1	AOSP	10	Google	01/2022	75	27
2	AOSP	11	Google	12/2021	102	16
3	AOSP	12	Google	11/2021	197	10
4	AOSP	13	Google	12/2022	212	7
5	OneUI	12	Samsung	01/2022	311	26
6	OriginOS	12	Vivo	02/2022	289	22
7	MIUI	12	Xiaomi	06/2022	245	22
8	MagicUI	12	Honor	06/2022	357	14

Data Set. To answer the research questions, `OverlayChecker` is applied to analyze 8 commercial Android systems. Table IV lists details about systems under evaluation, where `System`, `Version`, and `Vendor` provide the information about system name, Android version, mobile vendor that deploys the corresponding system on its mobile devices. In addition, `#App` provides the number of system apps. In detail, we choose 4 recently released versions of Android systems (Android 10 $\sim$ 13) deployed on the smartphones of popular mobile vendors [2] as our targets, including the official Android system AOSP [10] deployed on Google Pixel, and third-party Android systems OneUI [33] deployed on Samsung Galaxy smartphones, OriginOS [34] deployed on Vivo smartphones, MIUI [32] deployed on Xiaomi smartphones, and MagicUI [1] deployed on Honor smartphones. We extracted `JAR` files and `APK` files of Android framework and system apps of the systems under evaluation from the downloaded stock ROMs of Google Pixel 6, Samsung Galaxy S21, Vivo iQOO 8, Xiaomi 11, and Honor 60 respectively, between November 2021 and December 2022.

A. Reliability of Guidelines (RQ1)

To evaluate the reliability of the proposed guidelines, we follow the procedures in §IV to automatically find the windows protected against overlay attacks in each system app of AOSP Android 10 $\sim$ 13, and then we manually examined their source code to determine whether they satisfy the guidelines. More specifically, all (100%) of 27 protected windows found in AOSP Android 10 satisfy the guidelines. 42 (95.5%) of 44 protected windows found in AOSP Android 11 satisfy the guidelines. 56 (93.3%) of 60 protected windows found in AOSP Android 12 satisfy the guidelines. 60 (90.9%) of 66 protected windows found in AOSP Android 13 satisfy the guidelines. Since most (more than 90%) of protected windows in AOSP Android 10 $\sim$ 13 satisfy the guidelines, we are confident that the guidelines are reasonably reliable.

We further analyze the 6 protected windows that dissatisfy the guidelines, including `ContactsDumpActivity`, `WebActivity`, `UserConsentActivityDialog`, `CacheClearingActivity`, `UsbAccessoryUriActivity`, and `BugreportWarningActivity`. We find that they neither call sensitive APIs nor access sensitive content providers to perform sensitive operations (see Criteria 2 in §VI-B). Particularly, they conduct operations on files storing sensitive content (e.g., bugreport).

B. Unprotected Windows in Official Android Systems (RQ2)

Table V lists the details about the system apps' windows missing protection against overlay attacks in 2 recently released versions of AOSP, including Android 12 and 13. In total, 10 unprotected windows are identified by OverlayChecker. All of the vulnerable windows are found in Android 12, while 7 of them still exist in Android 13. To further determine whether the remaining 3 cases are false negatives, we manually examine their source code. We find that these windows have enabled `HNSOW` in Android 13, indicating that they become protected windows in Android 13. Therefore, they are not false negatives.

Responsible Disclosure. We are in the process of responsibly disclosing all our findings to Google. At the time of writing, 3 of the identified unprotected windows have been confirmed. We have been assigned 3 CVEs with high severity and got bug bounty rewards from Google.

Evolution of Defense Strategies. To understand the evolution of AOSP's overlay-related defense strategies, we also apply OverlayChecker to extra 2 versions of AOSP, including Android 10 and 11. Specifically, there are 27, 16, 10, and 7 unprotected windows found in AOSP Android 10 $\sim$ 13, respectively. Inspecting the detection results, we have the following findings. (1) Unprotected windows are gradually patched in AOSP. More specifically, 10 unprotected windows in AOSP Android 10 get protection in Android 11, 7 unprotected windows in AOSP Android 11 get protection in Android 12, and 3 unprotected windows in AOSP Android 12 get protection in Android 13. (2) New versions of AOSP introduce extra unprotected windows. Specifically, 2 and 1 unprotected windows are introduced in AOSP Android 11 and 12, respectively. (3) Unprotected windows still exist in the latest version of AOSP. For example, AOSP Android 13 still has 7 unprotected windows.

C. Unprotected Windows in Third-party Android Systems (RQ3)

Table VII presents the details about part of system apps' windows lacking protection against overlay attacks in Samsung OneUI Android 12, Vivo OriginOS Android 12, Xiaomi MIUI Android 12, and Honor MagicUI Android 12. In total, 39 unprotected windows are found by OverlayChecker. In detail, OverlayChecker uncovers 26 vulnerable windows in OneUI, 22 vulnerable windows in OriginOS, 22 vulnerable windows in MIUI, and 14 vulnerable windows in MagicUI. Note that, all unprotected windows in AOSP Android 12 also remain unprotected in OneUI, OriginOS, MIUI, and MagicUI. That is, compared with AOSP, more unprotected windows are found in the third-party commercial Android systems under evaluation.

From the results, we observe that mobile vendors fail to promptly apply Google's security patches to their customized systems. For instance, Google addressed an overlay missing protection issue in `RequestToggleWiFiActivity` in Android 12 (referring to CVE-2021-0837 [20]). This fix prevents overlay from rendering on top of the window that asks for user consent to enable WiFi. However, Samsung OneUI Android 12 still has this vulnerability (7^{th} case in Table VII), leaving opportunities for attackers to lure users to enable WiFi without consent.

TABLE V: A Summary of the identified unprotected windows in the recently released AOSP.

System	Version	ID	App	Window Name	One-step Launch[1]	Sensitive Operations	Simplistic Interaction
AOSP	12	1	com.android.settings	RequestManageCredentials	self.onCreate	KeyChain.setCredentialManagementApp	press a button
		2	android	HarmfulAppWarningActivity	ATMS.startActivity	PackageManager.deletePackage	press a button
		3	com.android.permissioncontroller	ReviewPermissionsActivity	self.onCreate	PermissionManager.startOneTimePermissionSession	press a button
	12 ∼ 13	4	com.android.settings	AppWidgetPickActivity	self.onCreate	AppWidgetManager.bindAppWidgetId	click on an AppWidget item
		5	com.android.managedprovisioning	PreProvisioningActivity	self.onCreate	DevicePolicyManager.wipeData	press a button
		6	com.android.settings	WifiNoInternetDialog	self.onCreate	ConnectivityManager.setAcceptUnvalidated	press a button
		7	com.android.phone	PhoneAccountSettingsActivity	self.onCreate	TelecomManager.setUserSelectedOutgoingPhoneAccount	select a phone account
		8	com.android.server.telecom	EnableAccountPreferenceActivity	self.onCreate	TelecomManager.enablePhoneAccount	select a phone account
		9	com.android.vpndialogs	ManageDialog	self.onCreate	VpnManager.prepareVpn	press a button
		10	com.android.captiveportallogin	CaptivePortalLoginActivity	self.onCreate	CaptivePortal.useNetwork	select a menu item

[1] ATMS: ActivityTaskManagerService

TABLE VI: Security implications of missing protection against overlay attacks on windows of AOSP system apps.

System	ID	A crafted overlay can lure users to ...
AOSP	1	set the credential manager app.
	2	launch harmful apps marked by Play Store.
	3	grant permissions to apps.
	4	manipulate App widgets on the home screen.
	5	set up work profile and agree to managed provisioning related tasks.
	6	connect to the network regardless of whether it is validated or not.
	7	select and adjust enabled phone accounts.
	8	adjust enabled phone accounts.
	9	set the VPN service to be controlled by new apps.
	10	use the current network even though it has an unsatisfied captive portal.

Responsible Disclosure. We are making responsible disclosures by reporting our findings to Samsung, Vivo, Xiaomi, and Honor. At the time of writing, 4 of the identified unprotected windows in OneUI have been confirmed by Samsung. We have been assigned 2 CVEs with moderate severity and got bug bounty rewards from Samsung. In addition, 8 of the uncovered vulnerable windows in OriginOS have been confirmed by Vivo. The Android security team of Vivo rates the confirmed cases as high severity, and we also got bug bounty rewards from Vivo.

D. False Alarms

To analyze false positives (FPs) of OverlayChecker, we manually inspect the source code of unprotected windows identified by OverlayChecker in AOSP Android 10 ∼ 13. We find that all of the unprotected windows satisfy the guidelines without enabling HNSOW. That is, no false positives are found.

Furthermore, to analyze false negatives (FNs) of OverlayChecker, we apply OverlayChecker to AOSP Android 10 ∼ 13 and compare reported unprotected windows with manually found protected windows in two adjacent versions of AOSP. For a window that is protected in the higher version but unprotected in the lower version, if OverlayChecker does not detect it in the lower version, it is an FN. In total, we find 3 FNs. Specifically, `ContactsDumpActivity` is protected in AOSP Android 11 but unprotected in Android 10. `BugreportWarningActivity` and `UserConsentActivityDialog` are protected in AOSP Android 12 but unprotected in Android 11. OverlayChecker cannot detect them because our guidelines do not cover sensitive operations performed in these windows.

IX. Case Study

To study the security impact of missing protection against overlay attacks in system apps of commercial Android systems (RQ4), in this section, we present two case studies to explore the details of identified vulnerabilities, assess their potential threats, and illustrate hypothetical scenarios in which these vulnerabilities could be exploited. Note that, we summarize the potential security implications of vulnerable windows identified by OverlayChecker in official Android systems and third-party Android ROMs in Table VI and Table VIII, respectively.

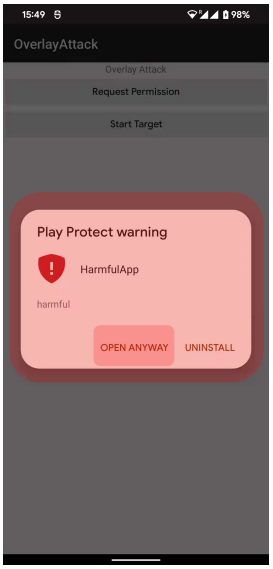


Fig. 9: The proof-of-concept for the case study of Google.

A. Google

Bug Overview. `HarmfulAppWarningActivity`, a system activity used to alert users when a potentially harmful app is installed or detected on their devices, is revealed to contain a bug that allows malicious apps to overlay this warning screen with a fake UI. Consequently, an unknowing user may be misled into trusting and opening the harmful app.

Proof-of-Concept. Our proof-of-concept demonstration unfolds in several steps. First, we create a malicious app and installed it on a Pixel 6 device. By executing the ADB command (i.e., ADB shell pm set-harmful-app-warning apkName harmful 0), we mark it as a harmful app. Then, we create a secondary app, which served as a launchpad for the malware. This app is installed on the same device and is granted with the permission system_alert_window. When the secondary app activates the malware, it also immediately draws an overlay to cover the

TABLE VII: A Summary of the identified unprotected windows in third-party Android systems.

System	Version	ID	App	Window Name	One-step Launch	Sensitive Operations	Simplistic Interaction
OneUI	12	1	com.samsung.android.settings	BluetoothScanDialog	self.onCreate	AdapterService.startDiscovery	click on an item
		2	com.android.permissioncontroller	ConfirmationActivity	ConfirmationReceiver.onReceive	IncidentManager.approveReport	press a button
		3	com.android.permissioncontroller	AppsPermissionsActivity	self.onCreate	AppOpsManager.setUidMode	press a button
		4	com.samsung.android.settings	WifiApAutoHotspotBlePairingDialog	self.onCreate	BluetoothDevice.setPairingConfirmation	press a button
		5	com.android.settings	SimDialogActivity	self.onCreate	TelephonyManager.setDataEnabled	press a button
		6	com.samsung.android.settings	WifiApWarning	self.onCreate	SemWifiManager.setWifiApEnabled	press a button
		7	com.android.settings	RequestToggleWiFiActivity	self.onCreate	WifiManager.setWifiEnabled	press a button
		8	com.samsung.android.settings	PaymentDefaultDialog	self.onCreate	Settings$Secure.putStringForUser	press a button
		9	com.android.systemui	SensorUseStartedActivity	self.onCreate	SensorPrivacyManager.setSensorPrivacyForProfileGroup	press a button
		10	com.android.systemui	TvUnblockSensorActivity	self.onCreate	SensorPrivacyManager.setSensorPrivacyForProfileGroup	press a button
		11	com.samsung.desktopsystemui	UsbConfirmActivity	self.onCreate	UsbManager.grantDevicePermission	press a button
		12	com.samsung.desktopsystemui	SensorUseStartedActivity	self.onCreate	SensorPrivacyManager.setSensorPrivacyForProfileGroup	press a button
		13	com.sec.android.app	WiFiStressTest	self.onCreate	WifiManager.setWifiEnabled	press a button
		14	com.android.nfc	BeamShareActivity	self.onCreate	NfcAdapter.enable	press a button
		15	com.android.nfc	ConfirmConnectToWifiNetworkActivity	self.onCreate	WifiManager.connect	press a button
		16	com.samsung.android.settings	WifiPickerDialog	self.onCreate	WifiManager.connect	click on an item
OriginOS	12	1	com.android.bluetooth	BluetoothOppTransferActivity	BluetoothOppReceiver.onReceive	BluetoothAdapter.enable	press a button
		2	com.android.permissioncontroller	ConfirmationActivity	ConfirmationReceiver.onReceive	IncidentManager.approveReport	press a button
		3	com.android.bluetoothsettings	BluetoothPermissionActivity	self.onCreate	BluetoothDevice.setAutoPlayAccessPermission	press a button
		4	com.android.wifisettings	WifiReselectApDialog	self.onCreate	TelephonyManager.setDataEnabled	press a button
		5	com.android.settings	SimDialogActivity	self.onCreate	SubscriptionManager.setDefaultDataSubId	press a button
		6	com.android.wifisettings	Settings$WifiSettingsActivity	self.onCreate	WifiManager.save	press a button
		7	com.android.wifisettings	Settings$AuxiliaryWifiActivity	self.onCreate	WifiManager.enableNetwork	press a button
		8	com.android.wifisettings	WifiNotifyDialog	self.onCreate	WifiManager.connect	press a button
		9	com.vivo.systemui	NotificationPermissionDialogActivity	self.onCreate	NotificationOpsManager.setNotificationEnabled	press a button
		10	com.android.systemui	SensorUseStartedActivity	self.onCreate	SensorPrivacyManager.setSensorPrivacyForProfileGroup	press a button
		11	com.vivo.permissionmanager	DefaultAppConfirmActivity	self.onCreate	Settings$Secure.putString	press a button
		12	com.android.wifisettings	Setting$WifiDisplaySettingsActivity	self.onCreate	DisplayManager.startWifiDisplayScan	click on an item
MIUI	12	1	com.android.bluetooth	BluetoothOppBtEnableActivity	BluetoothOppLauncherActivity.onCreate	BluetoothAdapter.enable	press a button
		2	com.android.systemui	UsbDebuggingActivity	self.onCreate	IAdbManager.allowDebugging	press a button
		3	com.android.systemui	NetworkOverLimitActivity	self.onCreate	INetworkPolicyManager.snoozeLimit	press a button
		4	com.android.settings	SimDialogActivity	self.onCreate	TelecomManager.setUserSelectedOutgoingPhoneAccount	select a phone account
		5	com.android.settings	MiuiHeadsetActivity	self.onCreate	BluetoothDevice.setPhonebookAccessPermission	press a button
		6	com.android.settings	BluetoothPairingDialog	self.onCreate	BluetoothDevice.setPairingConfirmation	press a button
		7	com.android.settings	WifiProvisionSettingsActivity	self.onCreate	WifiManager.connect	press a button
		8	com.android.settings	ManageApplicationsActivity	self.onCreate	NetworkPolicyManager.setUidPolicy	press a button
		9	com.android.settings	WifiAssistantDialog	self.onCreate	ConnectivityManager.setAcceptUnvalidated	press a button
		10	com.android.settings	MiuiSmsDefaultDialog	self.onCreate	SmsApplication.setDefaultApplication	press a button
		11	com.android.settings	PaymentDefaultDialog	self.onCreate	Settings$Secure.putString	press a button
		12	com.android.phone	MiuiPhoneAccountSettingsActivity	self.onCreate	TelecomManager.setUserSelectedOutgoingPhoneAccount	select a phone account
MagicUI	4	1	com.android.settings	BluetoothQuickDialogActivity	self.onCreate	BluetoothAdapter.startDiscovery	press a button
		2	com.android.settings	DialerDefaultDialog	self.onCreate	TelecomManager.setDefaultDialer	press a button
		3	com.android.settings	RequestIgnoreBatteryOptimizations	self.onCreate	IDeviceIdleController.addPowerSaveWhitelistApp	press a button
		4	com.android.systemui	HwUsbDebuggingActivity	self.onCreate	IAdbManager.allowDebugging	press a button

activity `HarmfulAppWarningActivity`'s window that pops up (as shown in Fig. 9). Thus, we can lure the user to open harmful apps by clicking on the "OPEN ANYWAY" button.

Potential Threats. This exploit could potentially lead to a multitude of severe threats. Given that the user is misled into trusting harmful apps, the malicious actors could gain unauthorized access to sensitive personal data, incur unwanted charges by making calls or sending messages, install additional malware leading to more serious security breaches or performance issues, or in worst-case scenarios, gain complete control over devices.

B. VIVO

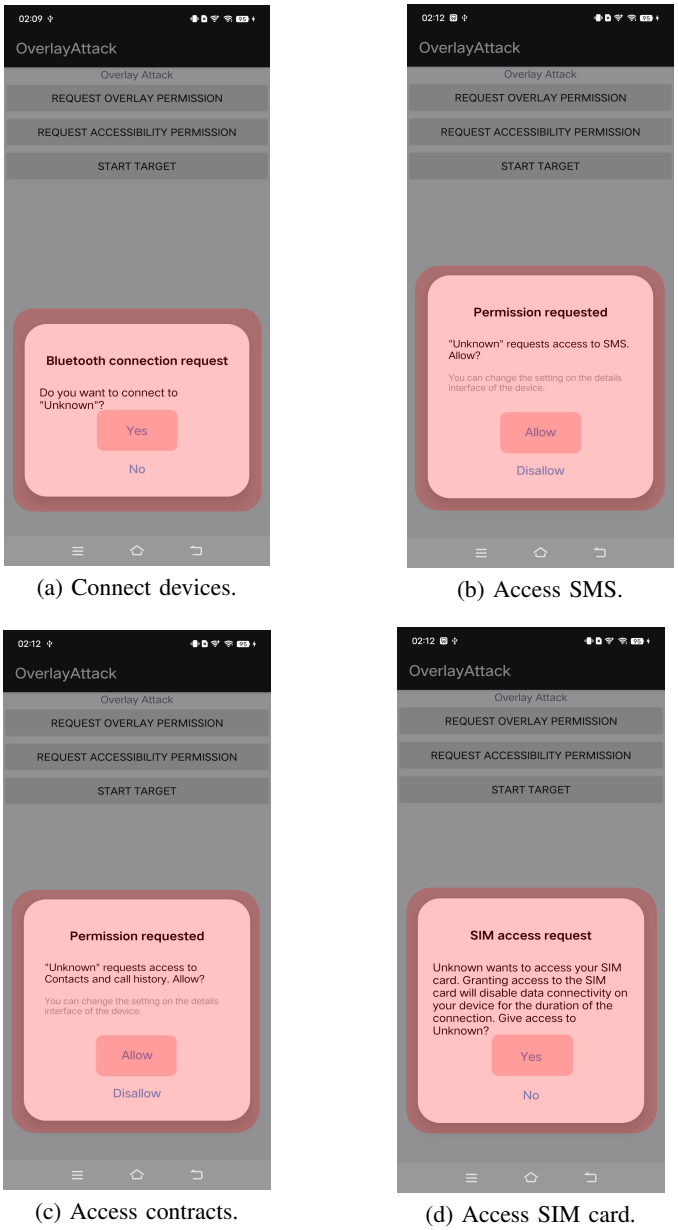
Bug Overview. `BluetoothPermissionActivity`, which is responsible for requesting and managing the permissions needed for Bluetooth operations, is found to lack overlay protection. Malware can create floating windows over its display, tricking users into granting permissions to connect Bluetooth devices, read contacts and call records, access text messages, and even engage with SIM card operations.

Proof-of-Concept. We create a malicious app and install it on a Vivo IQOO 8 device. Launching this malware, it requests Bluetooth operation permissions and simultaneously draws an overlay to cover the activity `BluetoothPermissionActivity` that pops up, thus executing an overlay attack as shown in Fig. 10a $\sim$ 10d. The malware carefully sequences the permission requests, leading users first to accept incoming Bluetooth connection requests from the attacker's device. Following that, permissions are granted for the connected Bluetooth device to read the contact list, text messages, and SIM card information. After acquiring the permissions, we could use the attacker's device to send Bluetooth protocols such as PBAP, thereby gaining access to the victim's sensitive data, such as contact lists and text messages.

Potential Threats. This vulnerability opens up a variety of potential threats. First, it allows an attacker to establish a Bluetooth connection without the user's awareness, creating a direct link for further attacks. For example, the attacker could exploit this connection to distribute malicious files or eavesdrop on other communications. Second, with granted permissions, the attacker could access sensitive data, leading to potential privacy breaches. Such breaches expose users to various forms of abuse, including but not limited to, identity theft, blackmail, or more sophisticated targeted attacks.

TABLE VIII: Security implications of missing protection on windows of system apps in third-party Android systems.

System	ID	A crafted overlay can lure users to ...
OneUI	1	scan and select unwanted Bluetooth devices.
	2	allow incident reports to be shared with apps.
	3	grant app-op permissions for specific UIDs.
	4	accept the Bluetooth device pairing requests.
	5	turn mobile data (i.e., cellular data service) on or off.
	6	enable Wi-Fi Soft AP (hotspot).
	7	enable or disable Wi-Fi.
	8	adjust the default NFC payment app.
	9	adjust the sensor privacy unblocked for camera/mic sensors.
	10	adjust the sensor privacy unblocked for camera/mic sensors.
	11	grant apps with permissions to access USB devices.
	12	adjust the sensor privacy unblocked for camera/mic sensors.
	13	enable Wi-Fi.
	14	enable NFC hardware.
	15	connect to Wi-Fi.
	16	connect to Wi-Fi.
OriginOS	1	enable Bluetooth.
	2	allow incident reports to be shared with apps.
	3	grant Bluetooth devices with the autoplay permission.
	4	turn mobile data (i.e., cellular data service) on or off.
	5	choose the SIM to use for calls, SMS, and data services.
	6	connect to or save networks.
	7	enable previously configured networks to access the internet.
	8	connect to Wi-Fi.
	9	enable or disable notifications for apps.
	10	adjust the sensor privacy unblocked for camera/mic sensors.
	11	adjust the default apps.
	12	start scanning for available Wi-Fi displays.
MIUI	1	enable Bluetooth.
	2	grant the permission to debug the Android device through ADB.
	3	ignore the restriction on the network data usage.
	4	select and adjust enabled phone accounts.
	5	grant Bluetooth devices with the phone book access permission.
	6	accept the Bluetooth device pairing requests.
	7	connect to Wi-Fi.
	8	set policies on network-related operations for specific UIDs.
	9	connect to the network regardless of whether it is validated or not.
	10	adjust the default SMS app.
	11	adjust the default payment app.
	12	select and adjust enabled phone accounts.
MagicUI	1	scan and select unwanted Bluetooth devices.
	2	adjust the default Dialer app.
	3	add apps to the power-saving whitelist.
	4	grant the permission to debug the Android device through ADB.



(a) Connect devices. (b) Access SMS.

(c) Access contracts. (d) Access SIM card.

Fig. 10: The proof-of-concept for the case study of Vivo.

X. Discussion

A. Limitations

Despite promising findings, our study has a few limitations.

Semi-automated Approach and Manual Effort. Since the complicated checks are enforced on data for launching windows, creating valid Intent objects for PoCs to launch 9 of 49 reported unprotected windows requires extra manual effort. Although the PoC Creator Module requires some manual effort, it offers the advantage of scalability for future research.

In future work, we plan to refine the Intent construction method. Currently, since most of the extra attributes are used for providing values, we fed them with random values of corresponding types. However, in certain scenarios, extra attributes could be used by branch constraints, causing unsuccessful launch of the target Activities by the constructed Intent objects. To address this issue, we will employ symbolic analysis to determine the valid values for extra parameters.

Experiment Scale. The scale of our experiments could be further expanded. Given the diversity of Android vendors on the market, our investigation into third-party Android systems is not exhaustive. In the future, we will include an expanded range of third-party Android ROMs.

B. Threats to Validity

The external validity of our analysis results could be affected by the following aspects.

Native Code. Some sensitive operations may be implemented in native code, hence some windows requiring protection might be undetected by our approach. Given the complexity of native code analysis in comparison to Java bytecode [42, 58], currently, there are no established methods for analyzing native code in system apps. In the future, we aim to develop new methods specifically to address this shortfall and extend our work.

Java Reflection. Besides native code, Java reflection can be leveraged to either enable HNSOW to enforce protection or implement sensitive operations. However, our methodology does not cater to Java reflection, which can potentially lead to false alarms. In future work, we plan to incorporate solutions from existing work, such as DINA [44] and DroidRA [57], to analyze Java reflection.

Guidelines. As observed in our experiments, some windows that require overlay protection, execute sensitive operations by accessing files containing sensitive data (e.g., bugreport), rather than calling sensitive APIs or content providers. In our future work, we will extend guidelines to cover sensitive operations on files. For example, we will follow existing studies, such as BigMAC [54] and PolyScope [56], to analyze SELinux policies in Android to determine sensitive files.

C. Other Future work

In this study, our primary focus was on Activities within system apps that require overlay protection, given their crucial role in device functionality and security which makes them high-value exploitation targets. However, in practice, certain Activities within third-party apps also necessitate such protections. In response to this, Google has introduced countermeasures for third-party apps, such as the `setFilterTouchesWhenObscured` method, which allows apps to disregard all touch events whenever the view's window is obscured by other windows [36] Additionally, the `FLAG_WINDOW_IS_OBSCURED` flag is incorporated into motion events, enabling apps to verify whether the window receiving the event is partially or fully obscured [38]. Despite these countermeasures, the lack of a clear guideline leaves developers puzzled about the correct implementation of overlay protection. Given the more complex and varied scenarios in third-party apps compared to system apps, our approach cannot be directly applied. For instance, many apps have a "Privacy Policy" window to inform users about the personal data the app collects, which should be protected. However, the method discussed in Section VI-B doesn't classify this as privacy-sensitive since it doesn't involve sensitive system APIs or Content-Providers. Therefore, in future work, we plan to thoroughly investigate popular third-party apps and establish clear criteria. Additionally, we will also propose a new framework that assists third-party app developers in identifying Activities within their apps that require overlay protection.

D. Recommendation

We provide several practical recommendations for Google, users, and mobile vendors in Appendix B, C, and D to mitigate the risks associated with overlay attacks.

XI. Related Work

Recently, there has been an increasing concern regarding the prevalence of overlay attacks, and many efforts have been made to investigate and address these threats.

Overlay-based Attacks. Many earlier research [48, 63, 75] used the overlay features to lure users to type passwords and grant permissions. To control the timing of overlay display, Chen et al. [50] discovered a side-channel vulnerability in the Android GUI framework, which could be exploited to infer the UI state of a target app without explicit permissions. Similarly, Yan et al. [72] revealed that power consumption traces could also be exploited for UI inference attacks. Deepening the security concern, Fratantonio et al. [53] demonstrated that an overlay could be used to trick users into unknowingly enabling the accessibility service, this action could potentially lead to the launch of a variety of powerful attacks, like granting arbitrary permissions to malicious apps. In a recent study [68], a novel overlay method was uncovered, where malicious overlays could be constructed by creating successive toasts, exploiting the fade-out animation of the toast so that transitions between two successive toasts cannot be observed. Most of those vulnerabilities were already fixed. For instance, Google has removed the OS-level side channels and the Toast overlay. Consequently, these attacks will no longer be effective on modern versions of Android.

Attack Defenses. Along with the overlay based attacks, there is also a large body of work proposing defense solutions. Possemato et al. [61] surveyed 60 apps to understand how real-world apps use overlays. The derived insights (e.g., overlays at the margins are typically not problematic) can be utilized to detect suspicious overlaps potentially exploited for clickjacking purposes. Yan et al. [73], similarly, studied the properties of suspicious overlays in malware apps and proposed an automated approach to detect overlay based malware at scale. Bianchi et al. [48], meanwhile, applied static analysis to apps, identifying code patterns that could be harnessed to initiate overlay attacks. They also suggested a security indicator to help users identify the app which app they are interacting with, and make sure that the inputs go to the app. However, this approach could unintentionally create side channels, as discovered by Fernandes et al [52]. They provided a more reliable solution by notifying users when a background non-system app creates an overlay on top of a foreground app, although this could disrupt legitimate apps that utilize overlays (e.g., Facebook). Addressing this, Ren et al. [64] propose the Android Window Integrity model, which restricts the use of overlay to only white-listed apps. Fratantonio et al. [53] suggested incorporate a new security flag in the OS, allowing developers to set it for any widgets or activities within their apps. This flag could prevent other apps from creating overlays on top of it. In countering UI inference, Possemato et al. [62] successfully pinpointed 18 vulnerable Android APIs that may disclose state-related information, thereby mitigating the impact of dependent overlay attacks.

Other GUI attacks. Besides overlay attacks, the research community has seen a surge in diverse GUI attacks on the Android system. Many of them [41, 43, 48, 49, 51, 59, 60, 75] primarily focused on tapjacking attacks, which display harmless-looking or transparent UI elements on top of sensitive ones. Ren et al. [65] unveiled design flaws in Android's multitasking, which can be abused to place a spoofing activity on top of other apps. Tuncay et al. [67] built false transparency attacks by moving a transparent app to the foreground to mimic a trustworthy one. Kalysch et al. [55] exploited a design flaw in Android accessibility to capture sensitive UI input. Aonzo et al. [45] discovered vulnerabilities in modern password manager apps and instant apps, which can be abused to facilitate phishing attacks. Xu et al. [70] presented how customizable notifications can be abused to lure users into interacting with malicious applications. In addition, GUI attacks can also be launched via WebView, as revealed by Yang et al. [74], untrusted

iframe/popup within WebView can serve as attack vectors.

XII. Conclusion

In this paper, we study the overlooked vulnerability within the Android system regarding missing protection against overlay attacks, bridging the existing knowledge gap. To automatically detect these vulnerabilities, we established a guideline defining which windows in system apps necessitate the protection. Based on the guideline, we design and develop a new tool named OverlayChecker, which can be applied to uncover the vulnerable windows in Android systems. By applying OverlayChecker to 8 commercial Android systems in 4 recently released Android versions, we totally uncover 49 unprotected windows, 15 of which have been confirmed by mobile vendors. 5 CVEs have been assigned, and 3 of them are rated high severity.

Acknowledgment

We thank the shepherd and the anonymous reviewers for their helpful comments. This research is partially supported by the HKPolyU Start-up Fund (A0048629), the Hong Kong RGC Project (No. PolyU15224121), the HKPolyU Grant (ZVG0), and the NSFC for Young Scientists of China (No. 62202400).

References

[1] "MagicUI," https://www.hihonor.com/cn/magic-ui/, 2022.
[2] "Mobile Vendor Market Share Worldwide," https://gs.statcounter.com/vendor-market-share/mobile, 2022.
[3] "A tool for reverse engineering Android apk files," https://ibotpeaches.github.io/Apktool/, 2023.
[4] "Activity," https://developer.android.com/guide/components/activities/intro-activities, 2023.
[5] "addSystemFlags," https://cs.android.com/android/platform/superproject/+/refs/heads/master:frameworks/base/core/java/android/view/Window.java;l=1247, 2023.
[6] "addSystemFlags," https://cs.android.com/android/platform/superproject/+/refs/heads/master:frameworks/base/core/java/android/view/Window.java;l=1231, 2023.
[7] "Android Debug Bridge (adb)," https://developer.android.com/tools/adb, 2023.
[8] "Android SDK: Common Android Components," https://code.tutsplus.com/tutorials/android-sdk-common-android-components--mobile-20873, 2023.
[9] "Android Uiautomator2 Python Wrapper," https://github.com/openatx/uiautomator2, 2023.
[10] "AOSP," https://developers.google.com/android/images, 2023.
[11] "App manifest overview," https://developer.android.com/guide/topics/manifest/manifest-intro/, 2023.
[12] "Common intents," https://developer.android.com/guide/components/intents-common, 2023.
[13] "CVE-2020-0015," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-0015, 2023.
[14] "CVE-2020-0394," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-0394, 2023.
[15] "CVE-2021-0314," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-0314, 2023.
[16] "CVE-2021-0333," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-0333, 2023.
[17] "CVE-2021-0391," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-0391, 2023.
[18] "CVE-2021-0523," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-0523, 2023.
[19] "CVE-2021-0538," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-0538, 2023.
[20] "CVE-2021-0837," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-0837, 2023.
[21] "CVE-2021-0992," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-0992, 2023.
[22] "CVE-2021-1016," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-1016, 2023.
[23] "CVE-2022-20212," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-20212, 2023.
[24] "Decompile and debug binary code and obfuscated apps." https://www.pnfsoftware.com/, 2023.
[25] "Developer Program Policy." https://support.google.com/googleplay/android-developer/answer/13438822, 2023.
[26] "Dialog," https://developer.android.com/develop/ui/views/components/dialogs, 2023.
[27] "Floating widget on top of other apps," https://developer.apple.com/forums/thread/78360, 2023.
[28] "GrantPermissionsActivity," https://cs.android.com/android/platform/superproject/+/master:packages/modules/Permission/PermissionController/src/com/android/permissioncontroller/permission/ui/GrantPermissionsActivity.java, 2023.
[29] "How to use Slide Over and Split View on iPad," https://www.imore.com/how-use-slide-over-and-split-view-ipad, 2023.
[30] "Intent," https://developer.android.com/reference/android/content/Intent, 2023.
[31] "MiSRC Vulnerability Reward Program Rules V6.1," https://cnbj1.fds.api.xiaomi.com/src/ppt/srcrule.pdf, 2023.
[32] "miui," https://home.miui.com/, 2023.
[33] "OneUI," https://developer.samsung.com/one-ui, 2023.
[34] "OriginOS," https://www.vivo.com/originos, 2023.
[35] "Platform architecture," https://developer.android.com/guide/platform, 2023.
[36] "setFilterTouchesWhenObscured," https://developer.android.com/reference/android/view/View#setFilterTouchesWhenObscured(boolean), 2023.
[37] "setPrivateFlags," https://cs.android.com/android/platform/superproject/+/refs/heads/master:frameworks/base/core/java/android/view/Window.java;l=1070, 2023.
[38] "SYSTEM_FLAG_HIDE_NON_SYSTEM_OVERLAY_WINDOWS," https://developer.android.com/reference/android/view/MotionEvent, 2023.
[39] "Tapjacking/overlay SYSTEM_ALERT_WINDOW vulnerability on a non-security-critical screen," https://bughunters.google.com/learn/invalid-reports/android-platform/5148417640366080/bugs-with-negligible-security-impact, 2023.
[40] "Window," https://developer.android.com/reference/android/view/Window, 2023.
[41] D. Akhawe, W. He, Z. Li, R. Moazzezi, and D. Song, "Clickjacking revisited: A perceptual view of UI security," in *Proc. WOOT*, 2014.
[42] S. Alam, Z. Qu, R. Riley, Y. Chen, and V. Rastogi, "DroidNative: Automating and optimizing detection of Android native code malware variants," *computers & security*, vol. 65, pp. 230–246, 2017.
[43] E. Alepis and C. Patsakis, "Trapped by the ui: The android case," in *Proc. RAID*, 2017.

[44] M. Alhanahnah, Q. Yan, H. Bagheri, H. Zhou, Y. Tsutano, W. Srisa-An, and X. Luo, "Detecting vulnerable android inter-app communication in dynamically loaded code," in *Proc. INFOCOM*, 2019.

[45] S. Aonzo, A. Merlo, G. Tavella, and Y. Fratantonio, "Phishing attacks on modern android," in *Proc. CCS*, 2018.

[46] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[47] M. Backes, S. Bugiel, E. Derr, P. D. McDaniel, D. Octeau, and S. Weisgerber, "On demystifying the Android application framework: re-visiting Android permission specification analysis," in *Proc. USENIX Security*, 2016.

[48] 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 *Proc. S&P*, 2015.

[49] D. Bove, "SoK: The Evolution of Trusted UI on Mobile," in *Proc. AsiaCCS*, 2022.

[50] Q. A. Chen, Z. Qian, and Z. M. Mao, "Peeking into your app without actually seeing it: UI state inference and novel android attacks," in *Proc. USENIX Security*, 2014.

[51] A. P. Felt and D. Wagner, "Phishing on mobile devices," 2011.

[52] E. Fernandes, Q. A. Chen, J. Paupore, G. Essl, J. A. Halderman, Z. M. Mao, and A. Prakash, "Android ui deception revisited: Attacks and defenses," in *Proc. FC*, 2017.

[53] 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 *Proc. S&P*, 2017.

[54] G. Hernandez, D. J. Tian, A. S. Yadav, B. J. Williams, and K. R. Butler, "BigMAC: Fine-Grained Policy Analysis of Android Firmware," in *Proc. USENIX Security*, 2020.

[55] A. Kalysch, D. Bove, and T. Muller, "How android's UI security is undermined by accessibility," in *Proceedings of the 2nd Reversing and Offensive-oriented Trends Symposium*, 2018.

[56] Y.-T. Lee, W. Enck, H. Chen, H. Vijayakumar, N. Li, Z. Qian, D. Wang, G. Petracca, and T. Jaeger, "PolyScope: Multi-Policy Access Control Analysis to Compute Authorized Attack Operations in Android Systems," in *Proc. USENIX Security*, 2021.

[57] L. Li, T. F. Bissyande, D. Octeau, and J. Klein, "Droidra: Taming reflection to support whole-program analysis of android apps," in *Proc. ISSTA*, 2016.

[58] L. Li, T. F. Bissyande, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and L. Traon, "Static analysis of android apps: A systematic literature review," *Information and Software Technology*, vol. 88, pp. 67–95, 2017.

[59] T. Luo, X. Jin, A. Ananthanarayanan, and W. Du, "Touchjacking attacks on web in android, ios, and windows phone," in *Proceedings of 5th International Symposium on Foundations and Practice of Security*, 2013.

[60] M. Niemietz and J. Schwenk, "Ui redressing attacks on android devices," *Black Hat Abu Dhabi*, 2012.

[61] A. Possemato, A. Lanzi, S. P. H. Chung, W. Lee, and Y. Fratantonio, "Clickshield: Are you hiding something? towards eradicating clickjacking on android," in *Proc. CCS*, 2018.

[62] A. Possemato, D. Nisi, and Y. Fratantonio, "Preventing and Detecting State Inference Attacks on Android," in *Proc. NDSS*, 2021.

[63] S. Rasthofer, I. Asrar, S. Huber, and E. Bodden, "An investigation of the android/badaccents malware which exploits a new android tapjacking attack," *Technical report, Technische Universitt Darmstadt*, 2015.

[64] C. Ren, P. Liu, and S. Zhu, "WindowGuard: Systematic Protection of GUI Security in Android," in *Proc. NDSS*, 2017.

[65] C. Ren, Y. Zhang, H. Xue, T. Wei, and P. Liu, "Towards discovering and understanding task hijacking in android," in *Proc. USENIX Security*, 2015.

[66] G. Tao, Z. Zheng, Z. Guo, and M. R. Lyu, "MalPat: Mining patterns of malicious and benign Android apps via permission-related APIs," *IEEE Transactions on Reliability*, vol. 67, no. 1, pp. 355–369, 2017.

[67] G. S. Tuncay, J. Qian, and C. A. Gunter, "See no evil: phishing for permissions with false transparency," in *Proc. USENIX Security*, 2020.

[68] S. Wang, Z. Ling, Y. Zhang, R. Liu, J. Kraunelis, K. Jia, B. Pearson, and X. Fu, "Implication of animation on Android security," in *Proc. ICDCS*, 2022.

[69] M. Y. Wong and D. Lie, "Intellidroid: a targeted input generator for the dynamic analysis of android malware," in *Proc. NDSS*, 2016.

[70] Z. Xu and S. Zhu, "Abusing Notification Services on Smartphones for Phishing and Spamming," in *Proc. WOOT*, 2012.

[71] J. Yan, H. Liu, L. Pan, J. Yan, J. Zhang, and B. Liang, "Multiple-entry testing of android applications by constructing activity launching contexts," in *Proc. ICSE*, 2020.

[72] L. Yan, Y. Guo, X. Chen, and H. Mei, "A study on power side channels on mobile devices," in *Proceedings of the 7th Asia-Pacific Symposium on Internetware*, 2015.

[73] Y. Yan, Z. Li, Q. A. Chen, C. Wilson, T. Xu, E. Zhai, Y. Li, and Y. Liu, "Understanding and detecting overlay-based android malware at market scales," in *Proc. MobiSys*, 2019.

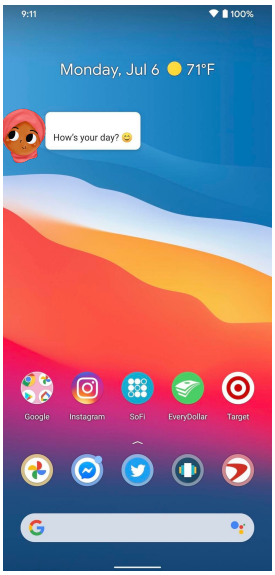
[74] G. Yang, J. Huang, and G. Gu, "Iframes/popups are dangerous in mobile webview: Studying and mitigating differential context vulnerabilities."

[75] L. Ying, Y. Cheng, Y. Lu, Y. Gu, P. Su, and D. Feng, "Attacks and defence on android free floating windows," in *Proc. AsiaCCS*, 2016.

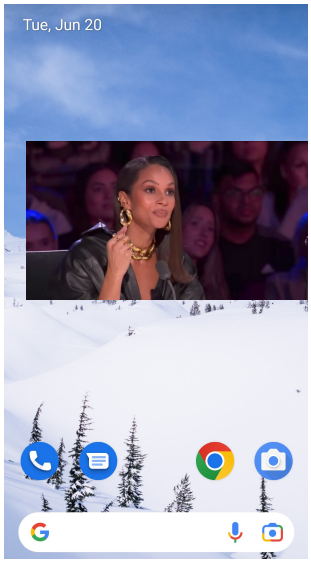
[76] L. Yu, X. Luo, J. Chen, H. Zhou, T. Zhang, H. Chang, and H. K. Leung, "Ppchecker: Towards accessing the trustworthiness of android apps' privacy policies," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 221–242, 2018.

[77] H. Zhou, X. Luo, H. Wang, and H. Cai, "Uncovering Intent based Leak of Sensitive Data in Android Framework," in *Proc.CCS*, 2022.

[78] H. Zhou, H. Wang, X. Luo, T. Chen, Y. Zhou, and T. Wang, "Uncovering Cross-Context Inconsistent Access Control Enforcement in Android," in *Proc. NDSS*, 2022.

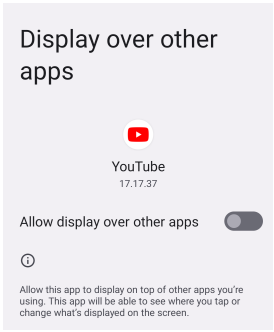


(a) Facebook messenger.

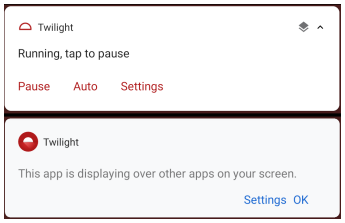


(b) Youtube.

Fig. 11: Android overlay window.



(a) Grant overlay permission to an app in Settings.



(b) Android system notification indicating overlay Use by an App.

Fig. 12: Illustration of overlay permission handling in Android.

APPENDIX

A. Threat Model

According to the details about overlay attack, we assume that attackers can trick victims into installing and launching a malicious app. This could be achieved through various methods, such as publishing the app on app market or through other distribution channels. This malicious app would then request the SYSTEM_ALERT_WINDOW permission from the user, requiring the user to grant this permission through the settings app. Given that many video-related and accessibility apps request this permission, it's very likely that users won't raise any suspicions during this process. Once this permission is granted, the malicious app, when interacted with by the user, can launch certain Activity in system apps and overlay a fake UI onto it.

B. Recommendations to Google

Establish Clear Guidelines. Google needs to provide clearer guidelines to instruct system developers accurately on which system app Activities necessitate overlay protection. Our research revealed that there are instances of missing overlay protection within AOSP, and some overlay protection patches lack clear justifications. This suggests that system developers may inadvertently introduce potential issues due to a lack of clear directives. A well-formulated guideline should strike a careful balance, ensuring adequate protection without hampering system flexibility. Because over-approximation of protection could restrict system functionality and flexibility. Conversely, under-approximation of protection might leave system security and user privacy exposed. Meanwhile, a clear guideline should not merely consist of instructions, they should be enriched with concrete examples and case studies. Such an approach can transform these guidelines into an intuitive manual that developers can readily understand and apply.

Expand User Notification. The Android system currently features a security alert in the notification drawer whenever an overlay is drawn (see Figure 12b). However, these alerts often go unnoticed due to their subtle presentation. To address this issue, Google could refine the alert system to make the notifications more conspicuous, thus increasing the chance of users noticing them (e.g. changing notification color, size, or adding a mandatory vibration). Additionally, the notification's text currently uses an ambiguous description, such as "[APP NAME] is Displaying Over Other Apps." Instead, the notification should provide more precise details, specifically indicating which app is currently overlaying which. Moreover, when users manually grant the SYSTEM_ALERT_WINDOW permission to apps, system should provide clear descriptions about the nature of the overlay and its potential security implications to raise users' awareness.

Strengthen PlayStore Vetting. Google could further enhance its vetting process for apps in the Play Store, particularly focusing on the apps that request the SYSTEM_ALERT_WINDOW permission. The vetting process could leverage some important features of overlays (e.g., Type, Flag, and Format), and incorporate machine learning techniques to detect suspicious overlay usage within apps.

Improve Policy Transparency. Google PlayStore should implement more strict requirements concerning the disclosure of app permissions in the privacy policies, especially for the SYSTEM_ALERT_WINDOW permission. This action would provide clarity for Play Store's vetting processes and enable the removal of apps that unnecessarily request this permission. As stated in the Google Play Developer Policy Center, developers are required to clearly state in their privacy policy why their application requests certain permissions [25]. However, our investigation reveals a concerning lack of adherence to this directive. We randomly downloaded 20 apps from the PlayStore that request the SYSTEM_ALERT_WINDOW permission. Unfortunately, none of these apps provided a clear explanation in their privacy policy outlining the specific purpose of this permission.

C. Recommendations to User

Update Regularly. Given Google's ongoing efforts to fix vulnerabilities related to overlay issues and release security patches to bolster system app protection, it is important for users to keep their Android devices updated. This practice not only mitigates weaknesses present in outdated versions but

also makes users benefit from the latest, most robust security mechanisms introduced by Google.

Permission Awareness. Users should exercise caution when granting permissions to apps, especially the SYSTEM_ALERT_WINDOW permission. Because this permission is quite powerful and many forms of Android malware and ransomware are known to exploit it [73]. According to Google's developer documentation, this permission should only be necessary and used under specific circumstances. Therefore, user should only granted this permission if it's essential to the app's functionality and if the app comes from a trusted source.

Download Carefully. Users should ensure to only download apps from trusted sources like the Google Play Store, which have protective measures in place to filter out potentially malicious apps. Extreme caution should be employed when downloading apps from unknown sources or clicking on links in unsolicited messages, especially when granting these applications permissions.

Security Alerts. Attention should be given to any security alerts on the device. Starting from Android 8.0, a security alert is displayed in the notification drawer whenever an overlay is drawn on the screen. If user see this alert and weren't expecting an overlay, immediate investigation is recommended. Meanwhile, users should avoid disabling the notification by changing the configuration of the Android System app, doing so might prevent these vital notifications from appearing.

D. Recommendations to Third-party ROMs

Update Security Patches Promptly. Update Security Patches Promptly. Google's team often releases security patches that address various types of vulnerabilities, including those that could be exploited via overlay attacks. Therefore, it is critical for custom ROM developers to promptly implement these security patches. Delaying these updates potentially exposes users to unnecessary risks.

Collaborate with security researchers Actively. Since custom ROMs are often developed by smaller teams with limited resources for testing and quality assurance, they may encounter more bugs and stability issues. Thus, it is highly recommended for custom ROM developers actively engage with security researchers and encourage broader participation in vulnerability disclosure programs. By engaging with the security community, ROM developers can stay updated on emerging threats and receive valuable feedback to improve their security.

E. Current Status of iOS Overlay Window

iOS, another leading mobile operating system, takes a fundamentally different approach. Unlike Android, it does not permit apps to draw floating windows over other apps, thereby closing off this security risk by not supporting screen overlays at all [27]. The only exception is found in the iPad, where specific APIs and frameworks provided by Apple, such as Slide Over and Split View, can be utilized [29]. A rational explanation for this is that the design philosophy of iOS puts a strong emphasis on user privacy. To protect user privacy and data security, iOS strictly limits inter-app interactions to prevent improper influences on other apps or system behavior. In contrast, Android, known for its more open nature, offers developers more extensive control over the operating system with fewer restrictions.