A Framework for Detecting and Countering Android UI Attacks via Inspection of IPC Traffic

Joshua Kraunelis\textsuperscript{1}, Xinwen Fu\textsuperscript{2}, Wei Yu\textsuperscript{3}, and Wei Zhao\textsuperscript{4}

\textsuperscript{1}University of Massachusetts Lowell, jkraunel@cs.uml.edu
\textsuperscript{2}University of Central Florida, xinwenfu@ucf.edu
\textsuperscript{3}Towson University, wyu@towson.edu
\textsuperscript{4}University of Macau, China, weizhao@umac.mo

Abstract—Android represents an ever-increasing share of the worldwide smart device market. The platform's ubiquity and open nature make Android a prime target for malicious actors. Unfortunately, device fragmentation among manufacturers makes maintaining cyber security difficult, invoking the need for third party security software. We present a framework for detecting and countering deceptive user interface attacks on the Android platform via inspection and analysis of inter-process communication transactions in the operating system. We evaluate our proof of concept implementation on a known class of malware that exploits the Android display system, allowing a malicious application to control the screen and mimic any application launched by the user. We achieve 100\% detection rate of this malicious behavior with no false alarms.

I. INTRODUCTION

Over the past few years, Android devices have dominated the global smartphone market. As of March 2017, Android phones accounted for nearly 60\% of US sales with upwards of nearly 90\% of sales in China and Spain [1]. This trend looks to continue as other Android devices such as tablets, televisions, and wearables gain popularity. Along with the increased presence of these devices comes a disturbing rise in the volume and sophistication of Android malware and attack campaigns. Malware authors continue to be motivated by factors including financial gain, as evidenced by the WannaCry ransomware campaign, which took thousands of computers hostage in May of 2017.

Meanwhile, malicious applications have become harder to detect, analyze, and counter. Malware [9] that block incoming calls from a user’s bank, which would otherwise alert the user to fraudulent account activity, have been discovered. Some malware [11] has the capability to detect when it is being run inside an analysis environment and deactivate itself so to avoid revealing details of its behavior that may be used to counter it. Other malware [3], [8] exploit the graphical user interface, attempting to trick a user into interacting with a malicious application.

Designing an effective malware detector for the Android platform is difficult. Depending on the scope (and ambition) of the detector, a deep understanding of how the malware behaves may be required. The resource constraints of the mobile device also impose a challenge, since detection software must be wary of its power consumption, storage utilization, and mobile data usage (consider an approach which forwards data to a web server for analysis). These resource constraints must also be considered when designing an effective countermeasure.

There are many possible approaches to Android malware detection. One approach might involve signatures, much like in traditional antivirus software, to identify known malicious applications and prevent them from being installed. The issue with the signature-based approach is that malware authors can easily defeat the detectors by changing the malware slightly, thus altering the signature and avoiding detection. Another approach might involve static analysis of system calls, permission usage, or inter-component communication. Static analysis can be effective, but it may not capture more complex behaviors such as malicious code being retrieved after an app has been installed or malicious code being embedded in native APIs (in the case where the static analysis is only focused on Java, Android’s primary programming language API). Additionally, static analysis fails to capture temporal relationships between events, which may be crucial in detecting certain malicious behaviors. Yet another approach is dynamic analysis, where measurements are taken at runtime and used in determining when an app is acting malicious. Those measurements, or features, are typically supplied to a statistical model, which classifies the behavior as benign or malicious based on previously seen feature data. This approach has been shown to be effective for some cases, however, it is limited by the quality and availability of the data used to train the classification model.

Android’s interprocess communication (IPC) mechanism provides a rich source of data, which can be used in detecting abuse by malicious software. We approach the problem of detecting Android malware by inspecting IPC transactions. The main contributions of this work are:

- We demonstrate how Binder (Section II-B), Android’s unique interprocess communication (IPC) mechanism, can be used to detect a class of Android malware (Section II-A), known as a masquerade attack, which exploits a vulnerability in the user interface framework.
- We present a proof-of-concept software framework (Section III) for inspection and analysis of Binder transactions, and show how it can be used to detect and counter this masquerade attack.
- We evaluate the feasibility and effectiveness of our ap-
proach using real Android applications obtained from the Google Play Store, and use a physical Android device to determine the optimal parameters to our detection algorithm.

The rest of the paper is organized as follows. We introduce the background knowledge on the masquerade attack and binder in Section II. We present the IPC based malware detection framework in Section III. We evaluate the countermeasure against the masquerade attack in Section IV. Related work is summarized in Section V and we conclude this paper in Section VI.

II. BACKGROUND

Our malicious behavior detection and countermeasure approach leverages Android’s Binder mechanism. In order to evaluate this approach, we used a previously reported malware which combined a vulnerability in Android’s ActivityManager service with exploitation of the Android AccessibilityService to masquerade as a benign app and steal user credentials. This section presents the critical details of the malware example used in this work and the relevant features of the Binder framework.

A. Masquerade Attack

Our previous work [8] introduced a new type of malware. The foundation of this attack relied on the attacker implementing a custom AccessibilityService class, which could be used to receive information about the current state of the user interface in realtime. An attacker could leverage this capability to detect when the user launches an application and launch their own malicious application which masquerades as the intended application. The obvious threat is that the user may not realize they are interacting with a malicious application, since it may mimic the look and feel of the application they launched. Therefore, the user may input sensitive data such as a password, social security number, or credit card number that can be stolen by the malicious application.

The two key components of this attack are application launch detection and exploitation of the Activity stack logic. Application launch detection works as follows:

- A malicious application registers to receive AccessibilityEvent objects from the system
- The user launches an application by clicking the app icon from the home screen or app list
- The system generates an AccessibilityEvent containing information about the click event and delivers it to the malicious application
- The malicious application extracts the name of the application icon clicked by the user from the AccessibilityEvent

Android’s UI relies on several components, a key one being the ActivityManager. This component is responsible for managing the state of Android’s main graphical user interface component, Activities, and enforcing the Activity lifecycle (starting, pausing, stopping Activities). When an Activity is created, an entry for it is created and pushed onto an ActivityStack. The ActivityManager uses these stacks to maintain the current UI state, keep an ordered history of recently launched tasks, and determine which Activity is to be displayed when certain events occur. When the user clicks an application’s icon from the home screen or app list, a system application named Launcher calls the startActivity method, instructing the ActivityManager to display the main Activity associated with the clicked app. If a second call to startActivity is made within a small time window, the ActivityStack is reordered such that the Activity from the later call takes priority and is rendered to the screen. A more detailed description of this vulnerability may be found in [8].

Putting these two together, once the malicious app has detected that the user has just clicked an app icon, it can quickly launch its own fake Activity (via startActivity) designed to mimic the requested app’s main Activity and be guaranteed to be displayed on screen. The user never realizes they aren’t interacting with the requested application, assuming that the malware author has designed the fake Activity to look exactly like the main Activity of the real app. Another variant of this attack might take the form of denial of service or ransomware. For example, instead of the malicious app masquerading as a real one, it could simply launch a blank Activity whenever any real app was clicked by the user, effectively rendering the device unusable. In yet another variant, the malicious Activity could include instructions for delivering ransom payment in exchange for disabling this behavior.

Detecting and countering this malicious behavior using our approach is important because this attack exploits functionality that we believe is vital to the design of the Android operating system. This is apparent in that the AccessibilityService API is exposed to user applications, encouraging developers to design accessible applications. At this point, it is unclear to us whether the logic of reordering the tasks in the ActivityStack object is a bug or by design. The governing logic spans several classes and source code files1, and totals over 14,000 lines. In other words, this vulnerability can’t easily be patched, to the best of our knowledge.

B. Binder

It is well known that the Android platform is built on top of the Linux kernel. Upon installation, each app gets its own Linux user ID and directory in the filesystem. The operating system enforces standard Linux file permissions such that an app may not read or write to another app’s directory. Each Android application runs in its own process, but unlike traditional Linux distributions, only system applications have sufficient privileges to fully interact with the OS. To enable user applications to share data with other apps and interact with system services, Android uses a permission-based system

1ActivityManager.java, IActivityManager.java, ActivityManagerNative.java, & ActivityStack.java
(different than Linux filesystem permissions). An app must possess adequate permission, granted explicitly by the user, for a service in order to access that service. The inter-process communication (IPC) mechanism by which components within a single process or across separate processes communicate is called Binder. The Binder mechanism is ubiquitous in the Android platform and has implementations at the application, middleware, and kernel layers. Binder is highly optimized for performance on resource constrained mobile devices.

System services, such as the ActivityManager, typically follow a client-server pattern where a Java object residing in the client process space acts as a proxy to a native implementation of the service in another process. This is convenient for app developers, since they may interact with system services as if they were objects existing in their local process space, even though the interaction occurs with remote objects via Binder IPC. To achieve this, any data being transferred between the client and server must be marshalled to a Parcel object, a generic container that can be transported via IPC in a process known as a Binder transaction.

During a Binder transaction, Java Native Interface (JNI) code is invoked so that the Parcel and relevant metadata can be handled by a native shared library (libbinder.so), which has been loaded into the client’s process. The library transmits the Parcel and metadata by making a system call to a kernel object known as the Binder driver. Once in the kernel, Binder maps the Parcel object into the server’s process, where it is unmarshalled and interpreted by the remote service. After the requested work has been performed, the service sends a response back to the client by reversing the transaction process.

While this presentation barely scratches the surface of the complex Binder framework, it attempts to provide an adequate depiction of the inter-process information flow in the Android platform required to understand the malicious behavior detection and countermeasure presented in this work.

III. DETECTION AND COUNTERMEASURE OF MALICIOUS BEHAVIOR

In this section, we detail the approach to detecting and countering the masquerade attack described in Section II-A. The key to detecting this attack is measuring the timing between successive calls to the ActivityManager’s startActivity method, a natural starting point given that the attack exploits the short time window between two calls to startActivity. Recall that the malicious application calls startActivity after receiving notification that the user has clicked an app icon. This second startActivity call happens on a scale of milliseconds, far too quickly to have been initiated by a human clicking a second startActivity call happens on a scale of milliseconds, receiving notification that the user has clicked an app icon. This method, a natural starting point given that the attack exploits the complex Binder framework, it attempts to provide an adequate depiction of the inter-process information flow in the Android platform required to understand the malicious behavior detection and countermeasure presented in this work.

Listing 1 shows the signature of the transact method.

```java
int transact(int code, Parcel data, Parcel reply, int flags)
```

The two parameters of interest are code and data. In short, remote procedure call (RPC) in the Binder framework is done by encoding the number of the function to be invoked as an integer. The source code of each service that exposes additional duties, request additional permissions, and possess UI elements, all of which may not be sensible responsibilities for the ActivityManager service. Second, we believe that our solution demonstrates a general-purpose architecture that can be applied to other types of malware. Our implementation has three main aspects: the Binder library modifications (based on Artenstein et al. [2]), the user-level app, and a character device driver that is used for communication between the Binder library and user-level app.

A. Binder Library Modification

The first step to detection of this attack is recording the timestamp whenever a call is made to the startActivity method. To do this, our solution inspects all Binder transactions destined for the ActivityManager. Our modification takes place in the transact method of the IPCThreadState class, found in the /frameworks/native/libs/binder/ directory of the Android source code. This method belongs to libbinder.so, which is loaded by most applications, and is called whenever a Binder transaction is performed, such as the remote invocation of the ActivityManager’s startActivity method.

Listing 1 shows the signature of the transact method.

```java
int transact(int code, Parcel data, Parcel reply, int flags)
```

The two parameters of interest are code and data. In short, remote procedure call (RPC) in the Binder framework is done by encoding the number of the function to be invoked as an integer. The source code of each service that exposes methods to remote objects should enumerate those methods using integer codes. The ActivityManager class encodes the startActivity method as code 3. The data parameter is a Parcel object containing a string descriptor of the destination service and the marshalled data being delivered. In our case, we are merely concerned with transactions involving the ActivityManager.startActivity method, so checking that code == 3 and that the transaction is destined for the ActivityManager is sufficient.
the code parameter is 3 (for the `startActivity` method) and the destination, `android.app.IActivityManager`, is present in the first 64 bytes. These are the two key factors which signify a call to `startActivity`, thus the next step in our solution is to log the timestamp at which this transaction occurred and the UID of the calling app. We use the `clock_gettime` system call to get the timestamp in seconds since the epoch. Conveniently, the `IPCThreadState` class provides a method, `getCallingUid`, which returns the UID of the application initiating the transaction. Given that this modification is implemented in a shared library whose code can be run by multiple processes, we cannot simply store, as global variables, the timestamp and UID of the previous `startActivity` call for comparison to the current values. In Android, mapping shared memory isn’t as trivial as it is in desktop Linux distributions, and for this reason, we chose to write the timestamps to a custom character device file that is then read by our user-level app.

![Fig. 1. Log excerpt showing ASCII-rendered partial Binder transaction.](image)

### B. Character Device File

Implementation of the character device driver is relatively straightforward, however there are a few caveats worth discussing. First, we prefer to build the driver as a loadable kernel module, but the prebuilt kernel delivered with the Android source code does not support loadable modules. Therefore, the kernel source was obtained by cloning the git repository from android.googlesource.com/kernel, then configured and compiled with loadable module support. Second, Android enables enforcement of SELinux policies which prohibit access to unknown character device files by default. An attempt was made at modifying the policy to allow the device to be written to and read from by the required apps, but it was unsuccessful. In production, careful consideration would go into modifying the policy to enable this communication, but for this proof of concept software, we simply disabled SELinux enforcement to speed our progress toward detecting and countering the attack in focus. Otherwise, the implementation is analogous to a normal Linux character device driver. An inode for the device is created using the mknod utility, so that it can be opened and written to in the modified Binder library, and opened and read by the user-level app.

### C. User-level App

The final component in our detection and countermeasure solution is the user-level app. The purpose of this app is to receive the timestamp and UID of the calls to the `startActivity` method that were captured and written by the modified Binder library. This is an ordinary Android app consisting of a single background Service with a thread that continually reads the messages from the character device. Following Algorithm 1, each new timestamp is compared to the previous timestamp. If the difference is less than some threshold, \( t \), and the UID of the previous caller belongs to the Launcher, the last Activity to be displayed is considered malicious and action is taken to counter the malware. The countermeasure involves displaying a prominent notification to the user in the status bar and displaying the home screen so that user does not interact with the malicious application. To achieve this, we leverage the following Android APIs:

- `PackageManager`, which provides methods for querying packages and their UIDs
- `NotificationManager`, which provides methods for displaying notifications in the status bar
- `startActivity`, which can be used to display the home screen

#### Algorithm 1 Detecting Suspicious Activities

```python
lastTS ← 0.0
lastUID ← 0
launcherUID ← getLauncherUID()

loop
  curTS, lastUID ← read("/dev/cm")
  if curTS - lastTS ≤ t AND lastUID = launcherUID then
    notifyUser()
  end if
  lastTS ← curTS
  lastUID ← curUID
end loop
```

### IV. Evaluation

To evaluate our solution, we performed two sets of experiments. The first experiment was designed to prove the concept of the proposed malware detection/countermeasure framework on the Android emulator, while the second was designed to determine the optimal detection threshold, as described in Section III-C, on a real Android device.

#### A. Experiment 1: Verifying the Detection & Countermeasure Framework

For the first experiment, we built a custom Android ROM and kernel. The ROM included the modified Binder library and was built from the Android 6.0 source code, while the kernel was configured to support loadable modules and was built from
the goldfish-3.4 kernel source branch. As of October 2017, 82% of devices were running Android version 6.0 or below, according to Google\(^2\). We launched the emulator with our custom ROM and kernel, inserted our device character driver module and installed both the malicious app and our user-level detection/countermeasure app. Next, we downloaded and installed on the emulator ten of the top unpaid apps from Google Play using a webapp\(^3\). The downloaded apps were:

- MySprint
- Facebook Messenger
- Snapchat
- Wish
- Bitmoji
- Facebook
- Instagram
- Netflix
- Pandora
- Spotify

The purpose of this experiment was to test our solution, including the Binder modification, character device driver, and app for notifying the user when malicious behavior is detected, using a real malicious app and a variety of real world applications. The malicious app was configured to detect when the Calculator app had been launched and launch its own Activity instead of the Calculator’s main Activity. We manually launched the Calculator app 50 times, returning to the home screen after the impostor Activity was displayed. Next, we disabled the malicious app and employed monkey\(^7\), Android’s recommended tool for stress-testing applications. The monkey is designed to be run from a device’s command shell and can be configured to perform various actions such as generating sequences of random input events and switching between applications randomly. We configured the monkey tool to perform 100 application switch events by setting the \(--\text{pct-appswitch}\) parameter to 100, at a rate of one event every ten seconds. Note that the monkey was configured to generate 100 app switch events to ensure (in the case of failures or crashes) enough events were generated to match the number of results from the malicious app, therefore we truncated any events beyond the fiftieth.

The user notification of malicious activity was displayed in the status bar every time the malicious Activity was shown, and zero times during the monkey experimentation.

B. Experiment 2: Determining the Optimal Threshold Value for the Inter-process Activity Interval

While the emulator was a convenient and useful tool to prove the feasibility of the proposed framework, it falls short in terms of both performance and realism. The emulator UI is very sluggish, therefore we cannot reliably estimate an accurate threshold for malicious activity detection. Also, we wanted to test our solution under a variety of real life conditions, therefore we built a custom Android ROM with the modified Binder library from the Android 7.1.2 source code, targeting a volunteer’s personal Huawei Nexus 6P device. Due to difficulty flashing the custom kernel on this device (and risk of damaging the device), the full-fledged detection/countermeasure framework was not installed on this device, rather the Binder library was modified such that the transaction timestamp and caller UID information was written to the system log for later analysis. For future work, we plan to purchase Android smartphones, flash the full-fledged detection/countermeasure framework onto these phones and distribute them to volunteers in order to perform extensive long-term experiments. We flashed the ROM onto a volunteer’s Huawei Nexus 6P device and instructed the volunteer to use the device normally (e.g. make and receive phone calls/text messages/emails, use social media apps, browse the web) for three days. At the end of the three day period, we installed the malicious app on this device and repeated the process of generating the Binder transaction timestamp and caller UID logs by manually launching the Calculator app, as described in the first experiment.

The timing results from the Nexus 6P are summarized in Table I. The first column shows the statistics for the time intervals between `startActivity` calls from the volunteer’s three days of normal device use. For each timestamp/UID log event, we computed the time interval between that event and the previous event, and record the UID of the previous caller. The results were then filtered such that only the records containing a previous UID belonging to the Launcher\(^4\) app were kept. This reflects the main characteristic of the malware in question, in that we are not concerned with how frequently `startActivity` calls are made within the same app or between any other app, but only those calls occurring directly after a call was made by the Launcher. The second column shows statistics for the time intervals between successive calls to `startActivity` from the malicious service. That is, we are only concerned with the timing of the `startActivity` call performed by the malicious service, not by the experimenter clicking the app icon to launch the app.

Figure 2 shows density estimate plots for both the normal and malicious time intervals on a logarithmic scale. We can see that there is little variance in the intervals of the malicious calls, while there is a large variance for intervals of normal application calls. Furthermore, the histograms shown in Figure 3 make clear that there is no overlap between the intervals of malicious and normal calls. Given that there is an order of magnitude difference between the intervals for the malicious and normal calls, it is safe to choose a threshold value of \(0.4 s \leq t \leq 1.0 s\). We believe that this threshold is valid for other smartphones than Huawei Nexus 6P. Humans will not

---

\(^2\)https://developer.android.com/about/dashboards/index.html#Platform  
\(^3\)https://apps.evozi.com/apk-downloader/  
\(^4\)The UID of a given app can be found in `/data/system/packages.xml`
open or switch between a sequence of applications very fast. UI attacks like the masquerade attack discussed in Section II have to start an activity almost immediately after the victim activity from the victim app in order not to incur suspicion from the smartphone user. Given the CPU power of today’s smartphones, the interval between a victim activity of an app and a fake activity from the malware and the interval between sequential activities from different apps will be separate. Nevertheless, we plan to perform extensive experiments with more volunteers, collect more data and derive an optimal threshold as the future work.

Fig. 2. Density Estimation of Intervals Between Inter-process startActivity Calls for Malicious and Normal Activities.

V. RELATED WORK

Malicious Android applications are a frequent topic in computer security research. In particular, several papers related to Android UI attack techniques and their defenses have been published in recent years. We pay particular attention to Android UI attack research and behavioral malware detection techniques, especially those leveraging the Binder mechanism.

Bianchi et al. provide a useful categorization of known Android UI attacks, which they refer to as GUI Confusion Attacks in [3]. They describe four main categories of these attacks: Draw on top, App switch, Fullscreen, and Enhancing techniques; and list several attack vectors for each category. In addition to the categorization, they examine the startActivity API to uncover combinations of API calls which may lead to malicious usage of the UI. They describe results of malicious app classification via their static analysis tool, and present a framework for verifying UI applications using PKI.

In [9], Rasthofer et al. report their analysis of a novel Android malware application named Android/BadAccents. This malware targeted Korean users belonging to several popular banks. The malware used a UI redressing technique to trick the user into giving the malicious application enhanced permissions. Once the permissions were obtained, the malware would attempt to uninstall mobile antivirus applications and steal the user’s banking information. The specific UI redressing technique described in [9] is referred to as tapjacking. It involves displaying a fake UI which provokes a user to perform a particular sequence of touches, while forwarding the actual touch events to some underlying application. Using this technique, a user could be fooled into dialing a premium number or interacting with an Android settings window. A counter-measure to the tapjacking attack was developed and provided to the Android security team.

Chen et al. [4] describe a novel technique for Activity state inference using a shared memory side-channel. Their Activity state inference works by monitoring changes in shared virtual memory, as reported by the Android/Linux proc filesystem. They capture this information for a number of Activity transitions and use it to build a model that can infer the current UI state. Since proc filesystem memory reporting is an integral part of the operating system, there is no easy countermeasure for this UI state inference attack. However, the authors provide several suggestions on how to mitigate this attack by modifying the operating system and window manager design.

Fernandes et al. demonstrate their Trusted Visual I/O Paths (TIVO) prototype in [5]. TIVOs enable the user to set a secure image that is then displayed along with the current application name and icon when the user is providing input (e.g. entering credentials). If the user does not recognize the secure image, or if the secure image is correct but the expected
application name and/or icon is not correct, then they may assume the UI has been compromised. The TIVO proof of concept implementation requires modification to the Android OS and additional Android system services.

Artenstein et al. [2] present an overview of Android’s Binder mechanism, followed by a discussion on how a malware author might use Binder to violate the privacy of the user. They walk through three example attack scenarios, all of which leverage Binder to steal or modify data. The first example shows how Binder may be used to implement a keylogger, the second shows how Binder can be used to modify data being passed between two components of a hypothetical banking application, and the final example uses Binder to spy on SMS messages.

CopperDroid [10] is a system for malicious behavior reconstruction, which uses virtual machine introspection to monitor low-level system activity. From its system call and IPC analysis, it is able to extract high level semantic behaviors. One of the capabilities provided by CopperDroid is automatic unmarshalling of Binder transaction data. It achieves this by scanning available Android Interface Definition Language (AIDL) files to devise which unmarshalling routines to invoke on Binder transactions. CopperDroid also uses malware stimulation technique to ensure that events which trigger malicious behavior occur during analysis. The authors evaluated CopperDroid on over 2,900 malware samples, comparing the number of malware behaviors exhibited with and without their stimulation techniques, and reported a best average increase of 28.5% in additional malware behaviors observed with stimulation.

In [6], Fernandes et al. investigate a previously proposed solution to curbing Android UI deception attacks that displayed a notification to indicate the user was interacting with the correct app. They discover a side-channel in the Binder transaction log, where debug messages containing basic information about Binder transactions are printed by the Binder framework. They use this side-channel to estimate when the proposed UI deception defense performs its security checks, and launch their attack in between these checks to avoid detection. The side-channel is subject to noise, however the authors were able to perform a successful attack 92% of the time. The authors propose a novel solution to preventing UI deception attacks by adding a mutex into the ActivityManager, however they note that this approach restricts the way UI apps may interact with users.

VI. Conclusion

We proposed a technique for detecting and countering malware on the Android platform. The technique is built upon a software architecture containing three components: a modified IPC library, a character device driver, and a user application. The proposed technique has advantages over static analysis based malware detection in that it can model temporal relationships between events, is not subject to code obfuscation, and can analyze behaviors after an app has been installed. Our technique also improves on the current dynamic analysis based behavioral malware detection approaches, because it leverages a general purpose software architecture that can take active measures such as notifying the user when malicious activity is detected. We describe the proof of concept implementation of this framework and demonstrate its feasibility using a malicious application which exploits a vulnerability in Android’s Activity management logic. We evaluate our solution on a real Android device and show that it is effective for detecting 100% of the malicious activity with no false alarms.

REFERENCES