Unleashing the Walking Dead: Understanding Cross-App Remote Infections on Mobile WebViews

Tongxin Li[1,2,*], Xueqiang Wang[2], Mingming Zha[3,4], Kai Chen[3,4], XiaoFeng Wang[2], Luyi Xing[2], Xiaolong Bai[5], Nan Zhang[2], Xinhui Han[1]
[1]Peking University [2] Indiana University Bloomington
[3]SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences
[4]School of Cyber Security, University of Chinese Academy of Sciences
[5]Tsinghua University
{litongxin,hanxinhui}@pku.edu.cn,{xw48,xw7,luyixing,nz3}@indiana.edu
{zhamingming,chenkai}@iie.ac.cn,{bxl12}@mails.tsinghua.edu.cn

ABSTRACT

As a critical feature for enhancing user experience, cross-app URL invocation has been reported to cause unauthorized execution of app components. Although protection has already been put in place, little has been done to understand the security risks of navigating an app's WebView through an URL, a legitimate need for displaying the app's UI during cross-app interactions. In our research, we found that the current design of such cross-WebView navigation actually opens the door to a *cross-app remote infection*, allowing a remote adversary to spread malicious web content across different apps' WebView instances and acquire stealthy and persistent control of these apps. This new threat, dubbed *Cross-App WebView Infection* (XAWI), enables a series of multi-app, colluding attacks never thought before, with significant real world impacts. Particularly, we found that the remote adversary can collectively utilize multiple infected apps' individual capabilities to escalate his privileges on a mobile device or orchestrate a highly realistic remote Phishing attack (e.g., running a malicious script in Chrome to stealthily change Twitter's WebView to fake Twitter's own login UI). We show that the adversary can easily find such attack "building blocks" (popular apps whose WebViews can be redirected by another app) through an automatic fuzz, and discovered about 7.4% of the most popular apps subject to the XAWI attacks, including Facebook, Twitter, Amazon and others. Our study reveals the contention between the demand for convenient cross-WebView communication and the need for security control on the channel, and makes the first step toward building OS-level protection to safeguard this fast-growing technology.

CCS CONCEPTS

• **Security and privacy → Mobile platform security**;

KEYWORDS

Android; cross-app WebView infection; remote deep phishing; remote privilege escalation; fuzzing tool; OS-level mitigation

1 INTRODUCTION

Clicking on "vnd.youtube://862r3XS2YB0" in your mobile Chrome, you will see the YouTube app popping up to play the video. Here Chrome hands over control to YouTube since the latter is better suited for the task. This is an example of *integrated service*, which leverages apps with different capabilities (video playing, social networking, payment, etc.) to provide best possible user experiences. This idea is rooted in the designs of Android and iOS, and can be seen in the implementations of most apps today. Such URL based, web-to-app communication, however, could also expose a mobile system to security hazards: it is reported that vulnerable implementations within Opera and Chrome allowed a web page to access browsers' local resources by sending an Intent scheme to their private activities [37]; also Samsung KNOX's MDM app was found to expose critical services (e.g., app installation) to the Intent scheme from other apps [27]. In response, protection is now in place to guard sensitive app components, e.g., through closing the channel used by the Intent scheme or limiting the access of these components only to the app with a proper permission. A problem is that such protection does not directly apply to *WebView*, a key user-interface (UI) component that often needs to be triggered by URLs from a different app: e.g., using the URL in Chrome to launch another app's UI, which runs in the app's WebView.

Cross-app WebView navigation. More specifically, once the web content (e.g., a script) inside the Chrome WebView triggers a URL fb://webview/?url=[web.page.url], immediately Chrome sends to Facebook an Intent containing web.page.url; upon receiving the Intent, Facebook automatically redirects its WebView to web.page.url, loading web content from the link. Such a collaboration, which we call *cross-app WebView navigation* (*XAWN*), is commonplace in mobile app designs, for the purpose of enabling a seamless transition between different apps' UIs, for example, from the YouTube page opened in Chrome to the YouTube app. It is built on top of the aforementioned URL-base cross-app channel: in the example, if the Facebook activity is registered with an Intent filter containing a scheme, the WebView can be directly invoked through an Intent-based or custom scheme; otherwise, if the activity is set *exported* or registered with a normal Intent filter, the script running in Chrome

* The work was done during the first author's visit at Indiana University Bloomington.

CCS '17, October 30-November 3, 2017, Dallas, TX, USA

ACM ISBN 978-1-4503-4946-8/17/10…$15.00
https://doi.org/10.1145/3133956.3134021

needs to trigger a *deep link* (Section 2.3) to directly activate the Facebook WebView.

With its pervasive use in today's mobile apps, once abused, XAWN can also become a security nightmare, even bigger than other types of URL-based invocations (e.g., triggering a service) mentioned above, since this new channel allows malicious content to *propagate* across apps. In the above example, once a malicious website is visited by Chrome, an attack page in the browser can redirect Facebook's WebView also to the site, through XAWN; such propagation can go beyond these two apps on the target device and continue to affect other apps like Twitter, when the attack content in the Facebook's Webview issues a navigation request through another scheme to redirect the Twitter's WebView also to the attack site. This spread of malicious web content across different apps' WebView instances can proceed like an infectious disease, enabling a *remote* adversary to gain partial control on multiple apps through their WebViews (loading all from a malicious website). We call this threat *Cross-App WebView Infection* or simply *XAWI*.

Cross-app WebView infection. The fundamental cause of XAWI is the cross-app WebView navigation weakness, which allows the web content loaded in one app's WebView to issue navigation requests (e.g, URL scheme) and launch another app's WebView to visit a malicious website. In our research, we systematically studied this previously unknown XAWN weakness and its security implications, particularly the complicated XAWI attacks that can be constructed to exploit the weakness and their consequences. Our research shows that in a XAWI attack, the adversary can *maintain* persistent and stealthy control on infected apps by running their WebViews in the background, and can further discover other *vulnerable* apps on the same device: that is, those whose WebViews can also be redirected through a scheme or a deep link. As a result, the remote adversary can collect a set of infected apps on a device, and turns these "zombies" into the bolts and nuts of a complicated *colluding* attack. Such an attack consolidates the individual capabilities of their infected WebView instances (e.g., rendering UI of a Phishing page, infecting other apps through deep link) into a powerful attacking force.

As an example, an infected Chrome can acquire the privilege of silent app installation by first contaminating a WebView of Amazon Shopping, and later utilizing the Shopping app to spread the infection to Amazon AppStore through its *deep linking* capability (Section 3.3). Note that in the example, Chrome itself *cannot* directly infect Amazon AppStore (i.e., invoking its WebView), since AppStore's WebView is not receiving any broadcasted Intent (produced by an Intent URL), but this becomes possible through the *stepping stone* (Amazon Shopping) capable of sending Intents to a specific activity. In our research, we found that high-profile apps like Facebook, Chrome, Twitter, Amazon Shopping, Amazon Appstore, etc. can all serve as building blocks for such complicated, multi-step, cross-app attacks, enabling a *remote* adversary to acquire critical system privileges such as sending unauthorized messages, silently installing apps, making unauthorized changes to a device (Section 3.3).

Also importantly, we show that given the pervasiveness of exposed WebViews across popular apps, even those *without* JS interfaces can be turned into effective attack weapons. Particularly, we found a series of *remote deep Phishing* never thought before. For example, an infected Chrome can stealthily navigate a WebView of Twitter to the attack content and then switch the app to background; after this, whenever the user clicks on Twitter, she will be greeted with the Twitter's infected WebView, which can display a fake Twitter login view to get the user's credential. Also we found that an infected app (e.g., Facebook) can actively invoke the infected WebView of another app (e.g., Twitter) to cover its UI (Section 3.2). This trick becomes useful when some apps' UIs are less suitable for Phishing than others: e.g., including URL bars. Through XAWI, however, the adversary can remotely select right components from those infected to build a complicated and highly realistic Phising attack. The video demo of our attacks can be found online [1]. Our research shows that our remote attack is much stealthier than local Phishing (which requires a malicious apps to be installed on a device) and can easily defeat all existing defense, including the most recent UI integrity protection [4, 30].

Most alarming here is that such a powerful attack can be systematically constructed. In our research, we developed a tool, called *ViewFinder*, to automatically analyze popular apps to discover exposed WebView interfaces. Our approach strategically fuzzes the apps using the URLs automatically generated from the "clues" recovered from these apps' code and meta-data. After running ViewFinder on 5,000 top-ranked Google Play apps, our study leads to the discovery of 372 apps exposed to XAWI. Our findings provide evidence that the threat of XAWI is general, realistic and significant.

Mitigation and understanding. We have reported all the apps involved in confirmed attacks to their vendors, including Facebook, Google, Amazon, Baidu and others, who all acknowledged the novelty and importance of this new type of threats. So far, we have received over $10,000 from Facebook and Twitter for the discovery of remote privilege escalation and remote deep Phishing, and also Amazon tells us that they have deployed fixes [1]. Due to the generality of the problem and pervasiveness of vulnerable apps in the wild, we designed and implemented a new OS-level solution to protect Android users. Our solution notifies the user of cross-app web navigation when the request has not been triggered by her activities, which effectively mitigates the attacks we discovered with a low overhead and a limited user impact (Section 4.3). On the other hand, our findings show that the elimination of the threat relies on resolving the contention between the strong demand for smooth web-to-app interactions and the need for security control on such channels, which certainly requires rethinking how they should be designed.

Contributions The contributions of the paper are summarized as follows:

• *New attacks*. We conducted the first study on the security implications of cross-WebView navigation, and discovered a new type of pervasive, high-impact remote attacks on Android. Through propagating malicious content across WebView, a remote adversary can gain persistent control of multiple apps and use them as building blocks to construct a complicated, coordinated attack. These attacks leverage infected apps' individual capabilities to acquire unexpected privileges and perform realistic Phishing attacks, which are all beyond existing defense, with a significant impact on today's Android ecosystem.

• *New findings*. Our research further demonstrates the pervasiveness of the threat: about 7.4% of leading Android apps (> 16,907,555,000 total downloads) contain exposed WebView instances that can be picked up by the remote adversary to compose the coordinated attacks. The findings highlight the need for more disciplined security designs for the web-to-app interaction channels.

• *New techniques*. We developed a new technique for identifying exposed WebView interfaces in apps, which helps better understand the scope and magnitude of this new threat. Further, we implemented an OS-level mitigation and demonstrate its preliminary success.

2 BACKGROUND

2.1 Activity and Task

On Android, a WebView instance is attached to an *activity*. Activity is an app component that provides a UI for users to interact with the app (e.g., phone call, photo taking, email management, etc.). A typical activity is described by the ⟨`activity`⟩ tag in an app's Manifest file and served by a Java class that acts as its controller. Following we briefly introduce how activities are triggered and managed.

Activity launch mode. An activity can be launched in four different modes [12], which affects the running status of its WebView instance. Activities with the "standard" mode or the "singleTop" mode can be instantiated multiple times. For example, a system setting activity can be launched by different apps, and each instance of the activity can have its own status. On the other hand, activities in the "singleTask" mode or "singleInstance" mode can only have one instance at a time (only one in a task). Google officially refers to the first two modes as "normal launches for most activities", while calls the other two "specialized launches" and does not recommend them for general use ("not appropriate for most applications") [14]. Therefore, most activities belong to the first two modes, which opens an avenue for hiding infected WebViews, as elaborated in Section 3.1.

Task and back stack. It is very common for an activity to invoke other activities on the same device. For example, an app listing emails in an activity can start a new activity (which could come from a different app) to show the attachment of a given mail. When a new activity is launched, the foreground activity will be brought to the background and covered by the newly started activity. When more activities have been fired, the background activities begin to stack up, with the foreground activity always on the top. To link these activities to a series of related operations (e.g., email listing and checking), Android associates them to a *task* and puts them all in the *back stack* of the task, which helps the user conveniently navigate back to the prior activity when an operation is finished and its activity is closed, or when the Android back button (aka., *return button*) is clicked. When an app is launched, the activity on the top of its task is displayed, which can be another app's activity. Prior research shows that the stack can be rearranged through setting special properties in the manifest, to make the backward navigation different from the user's expectation [31]. Our work, however, shows that this *task hijacking* can be done completely remotely, through scripts running in apps' WebViews, and through a collusion among multiple infected apps.

2.2 WebView Security

Resource-access mechanisms. Most mobile apps contains WebViews, which utilize web content to enrich their functionalities [25]. To serve this purpose, seamless use of device resources (through the apps' privileges on the device) from the web is often desired (e.g., getting a device's geo-location for displaying local news). On Android, three mechanisms are provided to enable such web-device interactions, including JavaScript interfaces, HTML5 and event handlers.

• *JavaScript interfaces*. JS interface is a mechanism that exposes an app's Java objects to the JavaScript code running inside the app's WebView instance. Through the mechanism, the app developer can register a Java object using an API `addJavascriptInterface()`, which enables the script to invoke all public methods annotated with *@JavascriptInterface* of the object.

• *HTML5*. HTML5 provides a set of built-in APIs as interfaces for web content to remotely access an app's local resources, which can be customized by the developer to control the access.

• *Event handlers*. WebView reports the web event it observes, which can be handled through a set of callback functions in its hosting app. A special callback is `shouldOverrideUrlLoading()`, a function that allows a developer to control the URLs allowed to be loaded into a WebView instance.

WebView protection. Given the importance of local resources exposed through these mechanisms, access control should certainly be in place to prevent them from being abused by untrusted domains. Android offers a set of APIs for controlling the domains a WebView can visit, including `shouldOverrideUrlLoading()`, `onPageStarted()` and `shouldInterceptRequest()`. Using these APIs for domain control, however, is highly complicated. WebView can visit untrusted domains under different circumstances: for example, when its hosting app is activated to load a page directly through `loadUrl()`, when it is asked to load another page in an iframe, when it is redirected by user interactions or when it loads another page due to a post request. Under each of these situations, a different set of callbacks are triggered and security checks therefore need to be performed at various program locations based upon the unique properties of the callbacks. Given the complexity of WebView integration within an app, complete mediation of its navigation is difficult. Once such an attempt falls short, which happens frequently in practice, some smart tricks can be played to bypass the protection, as discovered in our study (Section 3.3).

On the other hand, app developers today often do not have incentive to put too much restriction on the domains their apps are allowed to visit, due to the need to retain their customers as long as possible, a feature critical for their apps' commercial values [18, 34] (for advertising, in-app purchase, etc.). So app design today is leaning more toward "soft protection" of WebView instances. Specifically, many apps do not apply any restrictions to the instances that do not include any JavaScript interfaces, since these instances are considered to be of "low risk". A more common approach, as observed in our study (Section 4.2), seems to just limit the app UIs (e.g., not providing any URL bar) to prevent the user from inadvertently

directing WebView to untrusted domains, but has little constraints on the navigation requests from other apps. Such protection turns out to be insufficient and can be easily defeated by an XAWI attack, as discovered in our research (Section 3).

2.3 Remote App Linking

Intent and Intent-filter. To invoke an app's activity from the web content, the WebView asks its hosting app to construct an Android Intent and send it through the `StartActivity` API. When the Intent carries the recipient's package name and the activity name, the OS directly locates the target component. Otherwise, the system needs to utilize the action, category and data URI within the Intent to find the target. For this purpose, the target activity first needs to register an *Intent filter* with the OS to specify the attributes of the Intents it expects to receive. For example, "example://" matches the attribute "<data android:scheme="example"/>" specified in an Intent filter. In the presence of multiple activities expecting the same Intent, the OS prompts a dialog to ask the user to choose. In our attack, to avoid this user interaction, we utilize the URL scheme channel capable of generating the Intent with a package name whenever possible, unless the recipient's Intent filter has not been registered by another app.

URL scheme. URL schemes are the standard support for remote app invocation. On Android, when the user clicks a link, the system will send an Intent to its target. There are two types of schemes supported by Android, implicit (or broadcast) scheme and explicit (or Intent) scheme. An implicit scheme does not name a specific app but provides data attributes for locating the target, through its Intent. An explicit (or Intent) scheme, starting with intent:, includes not only the data URI but also the target's package name. For example, "intent://example.com/path#Intent;package=com.example.app;scheme=http;end" will be parsed to the Intent with data URI "http://example.com/path" and package name "`com.example.app`".

Deep linking scheme. Unlike web pages available on the Internet, content within apps cannot be searched and shared. To solve this problem, deep linking has been proposed to connect the content within mobile apps with a single link, which enables the invocation from web pages to the activities inside apps. Unlike URL schemes, deep linking supports are provided by individual app vendors and incorporated into apps through SDKs. To use the mechanism, an app developer includes her own `WebViewClient` to handle callbacks from WebView (thereby disabling both the implicit and explicit schemes), which contains the customized program logics to implement individual vendors' own deep-linking protocols. Since there is no standard for this technology right now, we consider any customized scheme or web content capable of specifying both package and activity names to be a deep linking approach, as it can directly reach the activity, which the standard schemes cannot do. An example is Facebook's *applink*[16] (see Figure 1).

Security guards. URL-based app invocation has not been extensively guarded by mobile OS. On Android, the protection is built almost entirely on Intent permissions and filter. Alternatively, one can "hide" an activity by registering no scheme in its Intent filter, so neither the implicit or explicit scheme can trigger the activity. However, this protection becomes completely ineffective in the presence of deep linking, which enables specification of

```
<html>
<head>
  <meta property="al:android:url" content="example://" />
  <meta property="al:android:class" content="
      WebViewActivity" />
  <meta property="al:android:package" content="com.
      example.app" />
</head>
</html>
```

Figure 1: An example of Facebook's applink

activity name and therefore can reach such a "hidden" activity (note here, "'hidden" means an activity could not be accessed by remote party through scheme, and it may still be exported to local apps). Further, the `WebViewClient` object provides an interface (e.g.shouldOverrideUrlLoading()), for the activity to determine how a URL in a web page should be handled, which can be used to control this app/component invocation and can even completely shut down the channel. However, our study shows that such protection can be circumvented even in popular apps, due to their problematic implementation (Section 3.1).

3 INFECTION ACROSS WEBVIEWS

In this section, we elaborate the XAWI attacks, starting with preliminaries for the attacks and then explicating the techniques we used to conduct remote deep Phishing and escalate the adversary's capabilities. These attacks exploit high-profile apps (e.g., Facebook, Twitter, Baidu, Amazon, etc.), posing realistic threats to a large number of popular apps (at least 7.4% found in our research). Their video demos are posted online [1].

3.1 XAWI Basics

Overview and threat model. The root cause of XAWI attack is the XAWN (cross-app WebView navigation) weakness, which allows the malicious content in one WebView to send a navigation request through a URL scheme to another WebView in a different app, redirecting the latter to an attack website, so as to gain a partial control of its hosting app. In this way, the attack web content (e.g., a script) can spread across multiple apps on the same device like an infectious disease, making it possible for the remote adversary to utilize these infected zombies to launch a *colluding* attack. In our study, we demonstrate the feasibility of such an attack. Most importantly, we found that infected WebViews can be used collectively to amplify the effectiveness of the attack, enabling the remote adversary to perform the activities that cannot be done through a single app.

Unlike most prior studies [2, 4, 7, 8, 20, 23, 24, 31], we do *not* assume the presence of a malicious app on the victim's device. What needs to make the attack work is just having malicious content (e.g., JavaScript) loaded into the WebView in at least one of the victim's apps. This happens when the user inadvertently visits some malicious, compromised or other less secure domains through her app. Actually, we believe that a main entry point for such an attack is a mobile browser, such as Chrome, even though its WebViews only have limited capabilities (no JS interfaces) and therefore need stepping stones to gain more privileges.

Target and channel. The goal of the XAWI adversary is to gain privileges and control apps, which is served by aggressively infecting other apps' WebViews, particularly those with JS interfaces. Our study on top 5,000 Google-Play apps shows that 7.4% of them expose at least one of their WebView instances, providing the materials for constructing exploits that reach these targets. Particularly, 38.4% of these vulnerable targets have JS interfaces capabilities, supporting location, device private and network state information collection. Since almost all the WebViews with certain capabilities also have different levels of domain control, the key of the attack is to bypass such protection. A challenge here is the channel for such attacks, since an infected app, such as Chrome, may not have the capability to access the targeted resources and needs the help from other apps to do that.

Cross-app channels are those URL based inter-process communication (IPC) mechanisms, including the implicit and explicit schemes and deep links (Section 2.3). The app's in-WebView infection may only utilize the channel the app supports to reach out to other apps. Therefore, in the case that the target WebView cannot be directly invoked (e.g., not registering any Intent filter), the adversary needs to strategically infect another app having that channel (e.g., deep link) to attack the target. On the other hand, for the WebView not having any cross-app channels (i.e., not allowed to make any IPC call), its infection apparently cannot go beyond its hosting app. Interestingly, however, we discovered that this limit can actually be broken sometimes, which enables a WebView not having the IPC privilege to issue navigation requests to other WebViews, as elaborated below.

• *Exploiting a race condition in popular apps*. We found that in popular apps like WeChat and Pinterest, there exists a race-condition when a WebView is about to be closed, which once exploited, temporarily grants the WebView the privilege to send out implicit or even explicit schemes, even though the WebView is not supposed to have this channel. Specifically, when a WebView instance is to be destroyed, these apps will set its `WebViewClient` object (for controlling URL navigation) to NULL. This actually turns the object to the default one with the capability to send out schemes. As a result, the malicious content within the WebView can issue navigation requests to others before it is closed by the OS. Note that for the popular apps with this problem, oftentimes, the attack page within an WebView instance can programmatically close the instance through commands, thereby actively triggering this process to produce scheme requests before the WebView stops running: e.g., we can load "weixin://webview/close/" into WeChat's WebView or "market://" into Pinterest, which will cause the app to set `WebViewClient` to NULL, so the attack script's navigation requests can be sent out before its hosting WebView is closed.

Persistent control and reconnaissance. Serving the purpose of strategic infection spreading are two key capabilities: stealthy and persistent control on the infected app, and reconnaissance for finding other vulnerable apps on the same device. In our research, we found that by default, a WebView can operate in background, continuously receiving and executing the commands (e.g., monitoring other apps and changing their states) from the remote adversary. Among all the vulnerable apps we examined (> 16,907,555,000 total installs), 81.6% of them can respond to remote commands while running in the background. Further, the activity hosting WebView can be launched in a standard mode, under which each invocation of the activity creates a new instance. Our research shows that many apps are running in this mode (e.g., Taobao, Baidu Appstore, Twitter, etc.). Leveraging these features, as soon as an infected app (i.e., the attacker) loads attack content to a victim app's WebView, the content (e.g., a script) in the WebView first launches another activity of the same app to cover the WebView and then the attacker triggers another app so as to move the victim to the background (see Figure 2). This transition can be done within a very short period of time, barely noticed by the user (see the online demo [1]). Most importantly, the background WebView infection can continuously command and control the whole infected device behind the scene, even when the infected app is launched by the user (only the top activity displayed) and even when the exact same activity is called again (a different instance of the activity displayed). Also, as long as some of the infected apps (called *commander*) can operate persistently, the adversary can maintain a firm control of the device, since other apps, even after their infected WebViews are closed, can be easily reinfected by the commander.

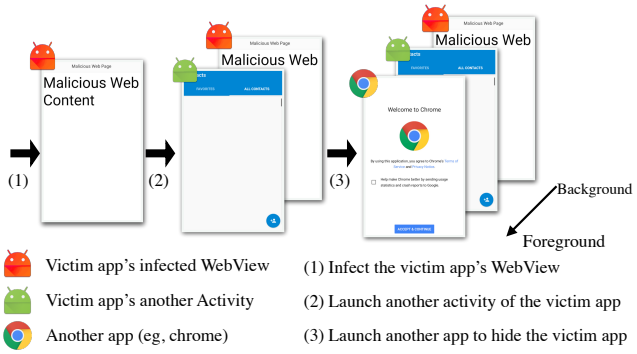

Figure 2: An infected WebView in the background

The background running commander also needs to identify and infect other co-located vulnerable apps to serve a XAWI attack. It can simply send navigation requests to the popular apps likely already installed on the target device: if the recipient is indeed there, the web content loaded to its WebView will notify the remote adversary. Note that with the adversary's persistent control, this can be done over a long period of time. Alternatively, we can leverage some apps' JS interfaces. For example, the Baidu app lets its WebViews query the presence of a specific app; also the widely-deployed AdMob library (a leading mobile advertising platform) tests the presence of a given package by trying to open it, and then informs the script running in its WebView once succeeds. To use this platform, we successfully delivered an attack advertisement (ad) through AdMob to our app using the library. The ad can discover vulnerable apps through AdMob and infect them using the WebView navigation.

Entry points and triggers. A XAWI infection starts from an *entry-point* app, whose WebView is the first one stuck by the attack web content on a device. Browsers, social-networking apps and mobile ad platforms are clearly more likely to become the entry points than other apps. For example, Chrome can be turned into the "source of transmission" once it visits an attack site. A problem here, however, is that unlike the WebViews within many other apps, in which a

script can automatically issue navigation requests, Chrome is only allowed to do so in the presence of a user click. However, we found that the browser is not good at linking the click to the URL scheme to be triggered: you can click on an image, a button and even a link opening a new page to trigger the delivery of a scheme unknown to the user. Also, at the moment a new page is loaded (e.g., when the browser is launched by a navigation request from another app), Chrome is allowed to send out an Intent scheme to other apps, without the user's click.

3.2 Remote Deep Phishing

With such supporting techniques, a XAWI adversary's capability to infect and control multiple apps from the remote becomes a game changer for mobile Phishing. No longer do we need a malicious app to be installed on the victim's device, as assumed by all prior work [4, 7, 31]. The new attack through XAWI can happen completely from the remote, through the scripts running in infected WebViews. Also we are talking about a *coordinated, multi-app* attack, which can do a lot more than the conventional, single-app attack can possibly achieve. Most importantly, such an attack is *practical*, only relying on the WebViews *without* JS interfaces, which are less protected and often need to be available for integrated services (discovered in 7.4% of popular apps). We call this new attack *remote deep Phishing* or *RDP*. The importance of RDP has been acknowledged by both Facebook and Twitter, which awarded us for the discovery of this new type of attacks [1].

More specifically, our research shows that in an RDP, the adversary can stealthily change a legitimate app's state and the relations between infected apps. For example, we can use one app's WebView to fake its own login UI, so when the user launches the app, she will unsuspiciously expose her credentials to the remote adversary. Further, an app with in-WebView infection can invite another app to impersonate some of its own UIs, when the latter includes an activity more suitable for the task. Since all these attacks happen with the cooperation from the "victim" app, the one impersonated or hijacked, they cannot be captured by existing defense. Following we elaborate three examples of such attacks, on Twitter and Facebook apps.

Evil twin from within. We found that a remote adversary with scripts running in Chrome can stealthily change the state of the Twitter app, using its WebView to impersonate its own login view. This attack renders all existing protection useless, since the Phishing content comes from the Twitter app itself. To make it happen, the infected Chrome first sends a navigation request to Twitter through a scheme invocation. Twitter has a public activity `UrlInterpreterActivity` that handles all the `StartActivity` requests from the browser and other apps (Figure 10 in Appendix illustrates the Intent filter registered by the activity and the URL that can be used to trigger the activity). Upon receiving the URL, the activity launches another activity and navigates the latter's WebView to attacker's website, which grants the control to the adversary. During this process, to avoid the `http` scheme that triggers a dialog window asking the user to choose the handling app, our attack utilizes an Intent scheme with Twitter's package name.

Twitter's WebView activity does not contain any title bar and other UI widget, and therefore can be easily converted into a fake login page. This activity is placed at the top of Twitter's task stack, so once the app is launched again, the login page will first be displayed. To hide this state change, as soon as the attack content is loaded into the Twitter WebView, the script running there immediately sends out a navigation request through the scheme googlechrome:// (reserved by Chrome) to Chrome, bringing its WebView to the foreground. A problem here is that a WebView in Twitter will be automatically closed after it issues a scheme. Therefore, the attack web page in the WebView needs to invoke another Twitter WebView instance with the attack link, together with the navigation request for Chrome, before it is terminated. In our attack, actually, the infected WebView opens Twitter's scheme multiple times before triggering Chrome's scheme. In this way, several Phishing pages will be put on Twitter's task stack before the foreground is handed over to Chrome. Once the user later launches the Twitter, not only will she see the Phishing page, but she will continue to be presented the same one if she touches the back button.

The RDP happens when the Twitter is in the login state. As a result, after the user enters her user ID and password to the fake login page, the remote adversary immediately instructs the infected WebView to switch to Twitter's main activity. The whole process, therefore, becomes indistinguishable from a real login. All the view switching in the attack happens almost instantly and is hard to notice by humans, as demonstrated in our online video [1]. We summarize the whole attack in Figure 3.

Faking my UI. Unlike Twitter, Facebook has a URL bar on its activity, which discloses the source of the web content in its WebView and therefore cannot be used to serve a Phishing page. The remote adversary, therefore, needs to find an accomplice app that can work with the infected Facebook to fake its login page. Obviously, the Twitter app can serve this purpose. In our research, we built an RDP in which Chrome infects the Facebook app, and whenever Facebook is launched, it further triggers Twitter to cover its interface. Through this coordinated attack, the remote adversary again can show to the user a realistically-looking attack page.

Specifically, a Facebook WebView can be invoked by the URL with the scheme fb://. For example, the link fb://webview/?url=http://www.attack.com in Chrome, once clicked by the user, brings Facebook to www.attack.com. Once infected, Facebook sends a URL scheme googlechrome:// to switch back to Chrome's WebView, without being noticed by the user. What we want to do here is that whenever Facebook is launched again, it instantly infects and invokes a Twitter WebView to display a fake Facebook login UI. To this end, the script dispatched to the Facebook WebView runs a loop, making continuous effort to trigger the infected Twitter activity. Actually, a Facebook WebView is suspended in the background and therefore the Intent scheme it tries to deliver to Twitter is blocked. However, immediately after it gets to the foreground (after the user invokes the app from the launcher or "recent apps"), the scheme is delivered, causing the Twitter Phishing page to show up. Further, if the user clicks on Android's back button, the system rolls back to the infected Facebook WebView, which again fires Twitter to impersonate its official login view. Also similar to the Twitter attack, after the user enters her password, the infected Twitter WebView

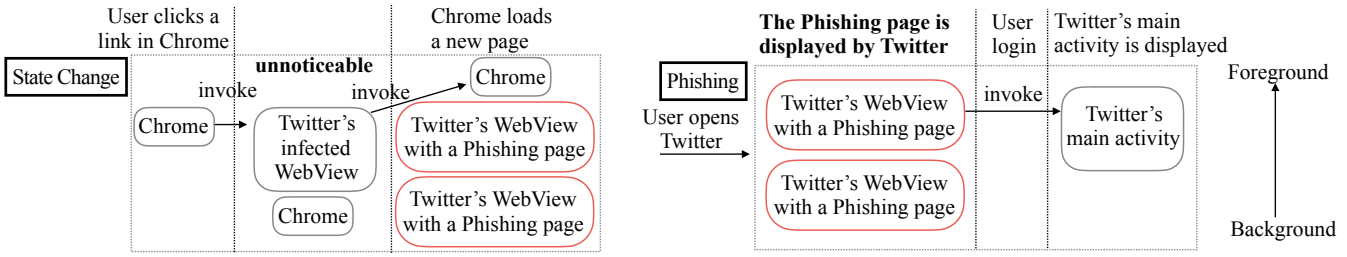

Figure 3: The Phishing attack on Twitter. ***Note that there are multiple instances of Twitter's WebView during Phishing, so once the back button is clicked, another instance of Twitter's WebView with a Phishing page is displayed.***

launches Facebook's main activity, which presents the user's account information when the attack takes place in the logged-in state. This attack is found to work smoothly, as summarized in Figure 4 and shown in our demo [1].

Inviting for hijacking. Actually, on the target device if there are apps with activities running in the *standard mode*, UI impersonation could become easier. Specifically, for the Facebook app, another popular app that can become its accomplice is PicsArt, whose activity operates in the standard mode. Such an activity, once launched by Facebook, will be automatically added to its task stack. So later on, when Facebook is opened, PicsArt's activity always shows on the top.

In our research, again, we run Chrome to infect Facebook's WebView, which then sends a scheme picsart:// to PicsArt, invoking its WebView and most importantly adding the related activity `WebViewActivity` to the Facebook's task stack. Then PicsArt can invoke Chrome to hide itself. After that, PicsArt hijacks Facebook's task and always shows on top of its UI. Further, the infected PicsArt can also gain control on Android's back button. Specifically, the app overrides the `onBackPressed` method and launches the most recent page once the button is clicked. This feature is then leveraged in our attack, which loads the attack page http://attacker.com/phishing.html that redirects the WebView to http://attacker.com/phishing.html#123. Once the button is pushed, PicsArt moves the WebView to phishing.html, which automatically goes back to phishing.html#123. In the meantime, after the user inputs her Facebook login credentials, the Phishing page will launch Facebook's main activity. The attack is summarized in Figure 5.

Against known defense. Compared with today's mobile Phishing attacks, RDP is unique in its complete reliance on the web content to control local apps and the cross-app coordination it can orchestrate. These features make existing defense less effective. Specifically, a prominent solution proposed in the prior research [4] utilizes a indicator in the system navigation bar to inform users which app they are interacting with. This protection is meant to defeat the UI overlay attack [19, 26, 29, 36] (the legitimate app's UI covered by an attack activity). However, it does not work on the RDP in which the infected WebView impersonates the UI in the *same* app. In our Twitter attack, all the user can see from the indicator is that the Twitter app is running on the top, which actually convinces her of the authenticity of the UI she provides login credentials.

Most recently, a technique called WindowGuard [30] has been proposed to enforce an Android Window Integrity (AWI) model, which defines the legitimacy of GUI system states in the user's interactions with apps. Particularly, it prompts a dialog and raises an alarm whenever a new activity is not initiated by the foreground app, and notifies the user whether the order of activities in the background has been rearranged after a new activity is launched. In our RDP attacks, however, all the new activities are launched by a foreground app and the order of the background activities will not change. Fundamentally, our new attacks are caused by the collusion between the app being impersonated and the perpetrator, since they are all infected by the attack web content and turned into the same remote master's zombies. This makes our attack completely different from what has been seen today, rendering WindowGuard ineffective.

3.3 Remote Privilege Escalation

In addition to remote deep Phishing, powerful XAWI attacks can be built to escalate the adversary's privilege on a device. Here, we elaborate two prominent examples in which the remote adversary acquires the capabilities to silently install apps and send out messages without the user's consent. An additional example is presented in Appendix A, in which the remote adversary can stealthily gather device information (e.g., app installed), monitor how the phone is used and change the device state (such as adding calendars) and even automatically install apps.

Unauthorized app install. We found that the Amazon Appstore app can be exploited by the remote adversary to silently install any third-party app on a mobile device without its owner's consent. The attack leverages the Appstore's powerful WebView, whose JS interface provides the object `IntentBridge` for app installation. However, the WebView is closely guarded and does not expose any UI for the user to navigate to non-Amazon domains. Also, through analyzing its code, we found that the app forcefully affixes the domain https://mas-ssr.amazon.com to any URL its WebView is asked to visit, thereby confining the app just to the Amazon domain. Another challenge is that the activity hosting the WebView has not registered any Intent filter and thus *cannot* be triggered by an Intent scheme.

In our research, we come up with a coordinated attack that starts from a Chrome browser running attack web content. The browser propagates the infection to the Amazon Shopping app through navigating its WebView to the attack domain, and further acquires the control of Amazon Appstore's WebView through the Shopping app (see our demo [1]). Here, Amazon Shopping serves as a stepping stone since the attack content hosted by Chrome can only issue an implicit or Intent scheme, not the deep link capable of invoking Amazon Appstore's unregistered activity. The Shopping app, however, allows its WebView to issue a deep link, that is, converting the

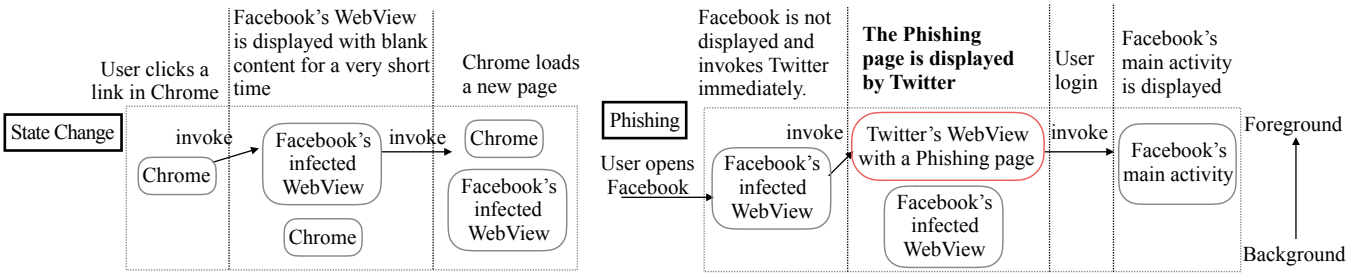

Figure 4: The Phishing attack on Facebook. ***Note that once the back button is clicked when a Phishing page is displayed, Facebook's WebView is resumed and will immediately invoke Twitter's WebView to display a Phishing page again.***

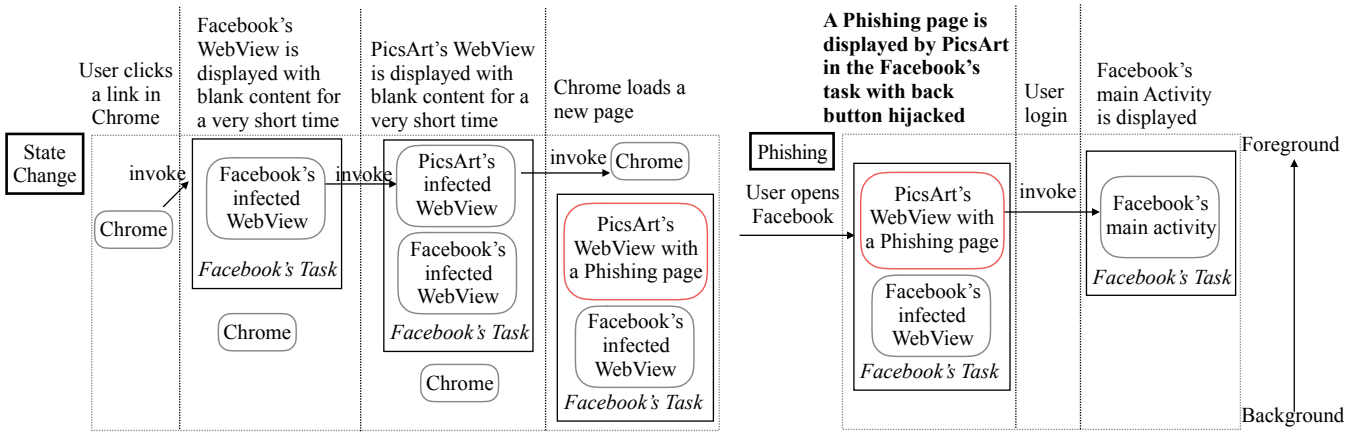

Figure 5: Hijack Facebook's Task. ***Note that the Phishing page can leverage the capability of Picsart's WebView to hijack the back button.***

URI like intent:.attacker.com#Intent;package=com.amazon.venezia;component=com.amazon.venezia/com.amazon.venezia.Venezia;end; into an explicit Intent for the package `com.amazon.venezia` and the activity `com.amazon.venezia.Venezia`. Also Amazon Shopping registers the scheme URI com.amazon.mobile.shopping.web://domain/path, which Chrome can use to navigate the app's WebView to the adversary's domain attack.com. During each attack step, a newly infected WebView is always switched to the background, as mentioned earlier (Section 3.1)

A complexity, however, comes from Amazon Shopping's domain control: the app verifies every URL to be loaded into its WebView and only proceeds with those from "amazon.com". In our research, we carefully studied this protection and found that the app uses Android API `Uri.getHost` to get the domain name of a URL. However, this API does not handle complicated URLs well[1]: for example, the domain of the URL https://a:a@test.amazon.com:a@attack.com is reported as test.amazon.com by the API, while when it is parsed in WebView, its domain is considered to be attack.com. Exploiting this discrepancy, our infected Chrome was able to load attack.com into Amazon Shopping, making it an accomplice of the attack. This newly discovered vulnerability was reported to Amazon.

Once the Shopping app is infected, its WebView can trigger the deep link to navigate Amazon Appstore's WebView. To move

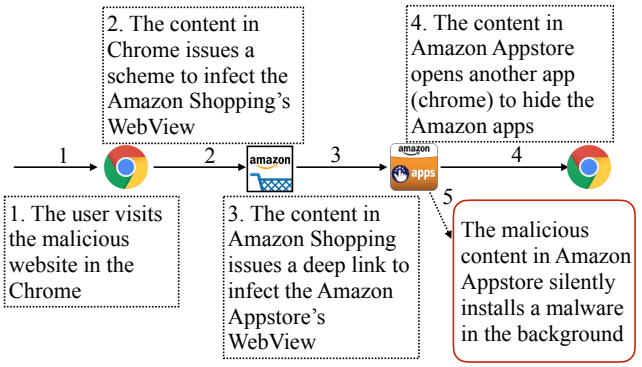

Figure 6: Unauthorized app install

the WebView to attack.com, we found that the adversary can simply create a sub-domain mas-ssr.amazon.com.attack.com. The protection on the Appstore side fails to append the URL affix https://mas-ssr.amazon.com with '/' and therefore can be circumvented by a carefully crafted navigation request: here, the request is to navigate to .attack.com, which is issued by the infected WebView in Amazon Shopping. As a result, attack.com gains control of all three apps and the privilege of silent app install. The process is summarized in Figure 6.

Stealthy messaging. In addition to directly escalating the privilege of a malicious website, XAWI can also help the *remote* adversary to exploit a vulnerability that originally can only be attacked *locally*, in the presence of a malicious app installed on the target device. A

[1]Another researcher reported this vulnerability in `Uri.getHost` to Google earlier than us. We independently discovered it and reported it to Google when the vulnerability was not fixed.

simple example is the Intent Spoofing attack [9], which requires that on-device malware sends a crafted Intent to unprotected components (e.g, broadcast receivers, activities and services). Using XAWI, the adversary can now utilize a malicious website to infect the WebView of a different app on the same target device and then command it to send that Intent to the vulnerable app. Specifically, in our research, we found that Facebook is one such app, which exposes an interface that can be attacked by a local adversary to cause it to send unauthorized messages through Facebook Messenger. The challenge here, however, is to execute this attack remotely, without running any malicious code on the target. Here we explain how this is done. Our attack has been acknowledged by Facebook, which awarded us $7500 for our findings.

Specifically, Facebook Messenger has an activity `SecureIntentHandlerActivity` (see Figure 9 in Appendix), which upon receiving an Intent with the scheme fb-messenger-secure:// will send out a message. However, this activity is protected by a permission `FB_APP_COMMUNICATION`, a signature one only given to Facebook's products. We found that the authorized Facebook app can serve as a stepping stone to deliver the message-sending Intent to Facebook Messenger. Facebook has a unique interface (activity `IntentUriHandler`) to interpret a Facebook deep link (called *applink* [16]) and generate an Intent to trigger the Messenger app's protected activity. This interface can be easily exploited by a local adversary, which can send an Intent to activate `IntentUriHandler`. The content of the Intent will then be used by Facebook to generate the scheme fb-messenger-secure:// to the Messenger. As a result, a message will be issued upon the local adversary's request.

However, exploiting this vulnerability *remotely* is much more difficult. A trouble here is that `IntentUriHandler` does not register any Intent filter for the applink scheme fbrpc://. As a result, it cannot be accessed by both implicit and explicit (Intent) schemes supported by Chrome. Further, after the vulnerability is exploited, the chatting UI of Facebook Messenger will show up in the foreground, exposing the attack to the user. Therefore to make the attack stealthy, the chatting UI should be switched to background after an unauthorized message is sent out.

Our technique, again, is to find a stepping-stone app with the capability to issue a deep link and run in the background. An example for such an app is Amazon Shopping. In our research, we utilized a Chrome WebView running attack scripts to spread the infection to a WebView instance in Amazon Shopping, which then issues a deep link directly to `IntentUriHandler`, like what happens in the app install attack, with an applink fbrpc:// in its data field. This applink causes the Facebook app to send an Intent to protected Facebook Messenger, leading to unauthorized messaging. During the attack, Amazon Appstore acts as the commander, automatically switching Chrome to the foreground as soon as it triggers `IntentUriHandler`.

Alternatively, we exploited a *selector Intent* weakness in `IntentUriHandler` to let Chrome directly talk to `IntentUriHandler`. Specifically, we found that `IntentUriHandler` registers an Intent filter for the scheme fb://. This allows us to construct a selector Intent scheme, which is a combination of two schemes, with fb:// in the selector field for determining the recipient activity and fbrpc://[payload] in the data field (Figure 7). This scheme, once triggered, causes Chrome to fire an Intent to `IntentUriHandler`

```
intent://[payload]#Intent;scheme=fbrpc;action=android.
    intent.action.VIEW;SEL;scheme=fb;action=android.intent
    .action.VIEW;end;
```

Figure 7: Selector Intent scheme

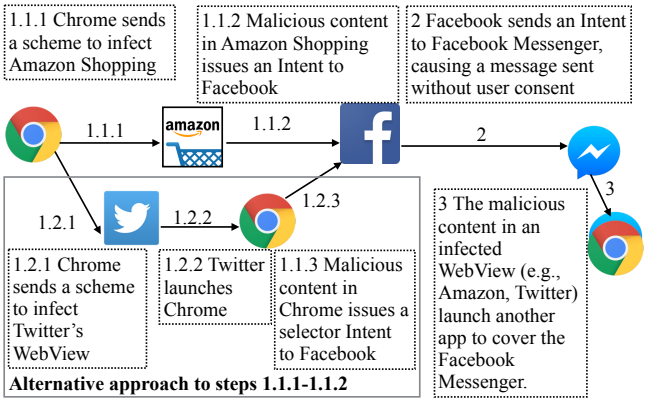

Figure 8: Attack Facebook Messenger

based upon fb://. When interpreting the Intent, however, the activity will receive a fbrpc:// URL, in the format of an applink, from the OS. This triggers the operations within `IntentUriHandler` to convert the URL into the Intent for Facebook Messenger, causing an unauthorized message to be sent out. A trouble here, however, is that Chrome's WebView cannot operate in the background and we need a commander to control app switching in the background, so as to hide the execution of Facebook Messenger. To this end, we utilized Twitter's WebView to coordinate the whole attack. Specifically, the malicious web content in Chrome's WebView first infects Twitter's WebView, which then brings Chrome to the foreground to trigger the Facebook's vulnerability. After the unauthorized message is sent out, Twitter's WebView in the background again invokes Chrome (waiting for 2 seconds after it is navigated to the attack site) to cover the Messenger app. The attack is summarized in Figure 8.

4 TARGET FINDING AND PROTECTION

In this section, we present *ViewFinder*, a technique for automatic discovering vulnerable apps. Also, we present an OS-level solution to mitigate the threat, through controlling navigation requests across apps.

4.1 Automatic XAWI Analysis

Key to the identification of a XAWI-susceptible app is to determine whether any of its WebView instances is exposed to the public and can further be invoked remotely through a URL (implicit, explicit schemes or deep links). Although such public activities can be easily found from an Android app's manifest, it is hard to be certain whether they can be navigated to a domain given by the adversary. Static analysis alone does not provide a solution. Data flow analysis tools [3] could help determine whether input data is propagated to a WebView, but they usually fail to provide any clue about the input that exploits the target vulnerability. Symbolic execution could be used to analyze all the constraints between app's entry point and the WebView before resolving them to generate the input, which,

however, is a process known to be complicated, expensive and easy to fail.

In our research, we went down the dynamic path and developed a simple fuzzing system, *ViewFinder*, which scans apps for remotely-controllable WebView instances. This approach is efficient and returns confirmed results: whatever we found will certainly be an opportunity for a XAWI adversary. The challenge, however, is how to find the right test cases to trigger the weaknesses. Our idea is based upon the observations that most clues for constructing the URL that can pass the app's checks are out there in its code and manifest. Actually, we found that even a simple yet systematic analysis of URL-related strings discovered from the app already leads to the discovery of a large number of confirmed vulnerable apps, 7.4% among all popular apps we studied. Following we elaborate on this technique.

Design. More specifically, the idea behind ViewFinder is to find partial URLs or strings similar to URL components from the Intent filter and the code of a public activity, for generating the test cases (that is, Intents) most likely to navigate the activity's WebView to risky targets. This purpose is served by an ADB-based [13] fuzzer, a simple app analyzer and a runtime monitor that instruments Android APIs. The fuzzer receives from the analyzer *clues* gathered from app data related to individual activities. These clues are converted into Intents by the fuzzer to test the activities. As mentioned earlier, all web-to-app invocations go through Intents: both explicit and implicit schemes are translated to the Intents without the target activity name, which rely on Intent filters to locate their recipients; a deep link, however, provides an activity name used by an Intent to directly trigger a specific activity. Our fuzzer directly generates these two types of Intents, with or without activity names, to test each app. This test is further helped by our instrumentation of APIs, which enables the monitor to inject content into the calls for extracting data from a test Intent. Also, to find out whether an input successfully navigate a WebView, the monitor watches the operations that load URLs to the instance.

URL-guided fuzz. For each app under the test, the app analyzer first inspects its manifest file to identify public activities (e.g.,"*exported=true*"). Each of them are then evaluated by the fuzzer, through the Intents with or without the activity name, depending on whether the activity claims an Intent filter. The Intent filter has a data field, including `scheme`, `host`, `path` and other attributes. These attributes are set for capturing an Intent with a navigation request (`StartActivity`) from a remote URL (scheme IPC or deep link), when they are found in the data URI field in the Intent.

To fuzz an app, most importantly here is to construct the right URI field. The field carries a URL, with a scheme (standard HTTP "http(s)://" or customized one "fb://"), a domain, a path and parameters (e.g., "?URL="). This field is automatically built by the fuzzer based upon the clues collected from Intent filters and the app code, as follows.

• *Activity with Intent filter*. For the activity opened through the standard Android scheme IPC, it needs to claim an Intent filter. To fuzz such an activity, the analyzer first attempts to pick up data pieces from its Intent filter. Specifically, in the case that the activity expects HTTP links, it will claim domain and path in the filter, which the fuzzer can directly use to create a link (for the data URI field), together with the target URL (e.g., "www.attack.com") to be loaded into the WebView. As an example, for the activity receiving a URL (through an Intent) with the scheme "http://", host "www.amazon.com/" and the path "abc", our fuzzer generates a link "http://www.amazon.com/abc?url=www.attack.com" for the test. If the monitor sees "http://www.attack.com" opened by the target activity, ViewFinder reports that it is vulnerable.

More complicated is when an activity claims a customized scheme (e.g., "fb://"), since the scheme can directly locate the activity and therefore the OS does not need the domain and path information in the Intent filter, and can leave the format checking to the app. To generate the URI string for a test Intent, the fuzzer uses the following strategies. It tries the test cases with the target URL directly attached to the scheme (e.g., "fb://www.attack.com"), and the domain-like string discovered from the manifest (from "host" field in the Intent filter), together with the standard redirection parameter like '?url=" (e.g., "fb://www.facebook.com/?url=www.attack.com"). Also, it leverages the discoveries made by the analyzer from the app code. Specifically, the analyzer disassembles the app (through apktool[22] in our implementation), collects all the strings from the activity and identifies the URL components from them, particularly the strings containing navigation parameters such as "?url=", "?redirection=", "?uri=", etc. These selected strings are then used by the fuzzer to generate other test cases, together with the domains found from the manifest, e.g., "pinterest://www.pinterest.com/offsite/check?url=www.attack.com".

• *Activity without Intent filter*. For the activity does not claim any Intent filter (which is often reserved for use by local apps only), it needs to be triggered by the Intent carrying its class name, together with the right data URI. To find such a URI, the analyzer identifies all URL-like strings from the app code, and picks out those not using the HTTP scheme but having the navigation parameter fields like "?url=" and "?uri=". These strings are then used to fill the URI field in a test Intent, with the navigation fields set to the target domain (e.g., attack.com). Using the Intent generated in this way, the fuzzer evaluates every public activity through ADB to find those manipulatable from the remote.

Another test performed by the fuzzer is whether an activity directly reads from the data URI field or the *extra* field of an Intent a URL for navigating its WebView. To this end, the fuzz sets the URI to the target domain. The extra field, however, is more difficult to handle: the field is a collection of customized key-value pairs. Without knowing the right key, we cannot put the target URL at a right place. Our solution is to hook the Android system function `Intent.getStringExtra()` for getting the values from the extra field for the app under the test. The idea is that when the app queries through the function, the monitor returns the target URL (such as attack.com) and watches whether the URL redirects the app's WebView. To avoid the performance impact introduced by frequent injections, we label each test Intent by adding a tag to its extra field. During the fuzz, only when the monitor finds `Intent.getStringExtra()` operating on the labeled Intent, will it change the return value.

Our approach also utilizes known vulnerabilities to generate test cases. For example, when the monitor observes that test URLs (e.g., amazon.com) are loaded but the redirection through parameters

like ?url= fails, ViewFinder automatically generates another sample using a:a@amazon.com:a@attack.com, based upon the inconsistency problem (between `Uri.getHost()` and WebView) discussed in Section 3.3. This strategy helps identify the apps with common vulnerabilities.

Runtime monitor. In our implementation, we built the monitor (for finding whether a test URL is loaded by a WebView instance) on top of an open-source tool called Xposed [32]. To inspect URL loading, ViewFinder hooks the API `WebView.loadUrl()` to intercept the navigation operation. Also instrumented in our implementation is `Intent.getStringExtra()`, through which ViewFinder changes the return values for the queries on the extra field in an Intent.

Discussion. As mentioned earlier, ViewFinder does not introduce any false positive: any flagged app is confirmed to be indeed problematic. On the other hand, as many other dynamic analyzers, there is no guarantee whatsoever that we can identify all vulnerable apps or all vulnerable activities in individual apps. Nevertheless, our study shows that even this simple technique can easily find many high-value targets for the remote adversary, making the case that remote infections, cross-app collusion are not a fantasy but a real threat. Further running the tool over thousands of most popular apps, we demonstrate that the threat is pervasive and significant, even based upon the low-end estimate made by this imperfect tool.

4.2 Findings

Setup. we collected 5,000 apps receiving URL schemes or Intents from other apps (with at least one Intent-filter for schemes or the attribute “android:exported" set to “true”) from Google Play top-ranked apps, in October, 2016, covering 36 categories like “Social”, “Communication” and “Tools”. Running ViewFinder to analyze all these apps took 7 days on 3 Nexus 5. To validate the results, we manually checked each of the detected apps, using the generated schemes as inputs to confirm that the app can indeed be navigated to the site under our control. No false positive was found. In the meantime, due to the challenges in unguided manual analysis of these complicated apps (16.7 MB on average), we did not have the ground truth to understand the coverage of the scan. So, all the findings reported here should only be considered as a lower limit for the impact of the XAWI threat.

Landscape. Among the 5,000 apps, 372 of them (7.4%) were found to contain the WebViews subject to remote infections. Besides Facebook and Twitter (Section 3), other popular apps include TripAdvisor, Google Drive and Yelp. Table 1 in Appendix presents the top 50 XAWI-susceptible apps, together with their Google-Play install counts. As we can see here, each app has 46,195,505 installs on average, which may affect hundreds of millions of users around the world. Also, we found that most of these apps are newly updated: 84.2% apps are updated in year 2016. This indicates that the security risk of XAWI has not yet come to the app vendors’ attention.

Attack opportunities. Our scan also brought to light the potential attack opportunities exposed by these apps (Table 1). Particularly, 81.6% popular apps (e.g., Best Buy, WPS Office and Cymera) can respond to remote commands while running in the background, which enables the remote adversary to maintain a persistent control on these apps, once their WebViews are contaminated. Also JS interfaces, HTML5 supports and callbacks are found in Pinterest, KaKaoTalk, Hola Launcher, etc. Further discovered in our study are the apps that provide ideal materials for an RDP: 287 apps have at least one vulnerable WebWiew without any address bar, 151 without any title and 80 apps can show a webpage in full screen. As soon as these apps or their co-located apps are infected by XAWI, they could be turned into building blocks for the RDP attack, for displaying the fake UIs to impersonate the critical views of other apps or their own. Examples of these apps including TouchPal Keyboard, iQiyi and mjweather (see Table 1 in Appendix). Among these apps, the WebViews in 162 of them can be triggered by HTTP schemes, while the others need the activity names to invoke.

Taking a close look at the vulnerable apps, our studies brought to light a few surprising findings. For example, we found that some WebViews without JS interfaces and callbacks can still leak out device information to a remote adversary. For instance, iQiyi, a famous video-sharing app, a counterpart of YouTube in China, exposes such information as DeviceID and locations by appending them to any URL given by the remote adversary through an infected WebView (e.g., https://attack.com/?deviceID=[deviceid]&platform=[platform]&...&location=[location]). Also discovered is the vulnerable WebView inside shared libraries. As an example, KaKao SDK, a popular OAuth library in Koera, includes exposed WebViews, making all the apps integrating it vulnerable. Examples include `com.kakao.taxi`, `com.ileon.melon` and `com.kalao.page`, each of which has 10,000,000 ∼ 50,000,000 installs. Other examples of the new attack opportunities we found are presented in Appendix.

4.3 Mitigation

Mitigating the XAWI risk is challenging, due to the contention between the demand for convenient web-to-app interactions and the need to properly control the use of these channels. Fundamentally, only the app developer knows whether a cross-WebView navigation request is reasonable and whether the task other apps asking her program to handle stays within the scope of the services she intends to provide. Also the developer is at the best position to balance her need for user retention with the safeguards put in place against the abuse of her app’s capabilities. To mitigate the XAWI attack, an app developer could keep his app’s WebView private, enforce proper domain control on it, or notify user when "suspicious" cross-app navigations (e.g., those without user-interactions) happen. That being said, still there is an important role for the OS to play, which is particularly important given that the developer-end protection inevitably takes a longer time to deploy, with no guarantee to be respected by app vendors (especially when restrictions on cross-app interactions may run against some of their business interests). Therefore in this section, we present a simple, yet effective system-level solution, called *NaviGuard*, for mediating the web-to-app channels.

NaviGuard. The idea of NaviGuard is to identify and control anomalous cross-WebView navigation requests, making them more observable to mobile users. Since it’s infeasible for attackers to programmatically mimic touch event inside a WebView, our approach takes a strategy that allows the requests with evidence of implicit user consents (i.e., triggered by UI interactions) to silently go through, notifies the users of those without such consents and blocks the requests of high risks (e.g., those from background processes), which

reduces the burden on users when such channels are legitimately used. This simple protection is shown to work effectively against all the attacks we discovered.

Specifically, to control the channels, NaviGuard hooks `StartActivity()` to monitor when an activity is launched. When this happens, our approach further determines whether the operation (i.e., `StartActivity()`) comes from WebView and has been issued by a foreground activity. To this end, we hook all JS interfaces APIs (e.g., `addJavascriptInterface`) and WebView callbacks (e.g., `setWebViewClient`), since any Intent initiated from WebView has to go through one of these two channels: Android default schemes are handled by `shouldOverrideUrlLoading` in `WebViewClient`, and deep links can be processed by any of these APIs, depending on its implementation. To link the observed `StartActivity()` to a specific WebView instance, NaviGuard records the thread ID and the WebView ID for each JS interface and callback invocation in a table and removes the IDs once the API call completes. Also stored at that time is the state of the WebView's activity, particularly whether it is on the top (through the API `Activity.isResumed()`). When an Intent and its `StartActivity()` event are observed, NaviGuard looks up the table using the caller's thread ID to find out whether the call indeed comes from a WebView instance. If so, further we check whether the instance (and its activity) runs in the foreground. When this is not the case, NaviGuard immediately stops the launch request from the background WebView, since the user cannot open another activity by operating on a background WebView. Otherwise, NaviGuard tries to link the current operation with a recent user event (e.g., a click), and when the attempt fails, pops up a dialog window to let the user confirm whether she wants the new activity to be activated.

To establish a relation between a URL navigation request and user actions, NaviGuard interposes on user-action related APIs such as `WebView.onTouchEvent()` to obtain the WebView ID should a touch event happen, and keeps the ID in the table. In the meantime, when a `StartActivity()` event occurs, its hook also acquires the caller's WebView ID if the event is issued from a WebView, and looks up the table to find whether a touch event is observed from the same WebView, within a short period of time (1 second set for our implementation). Alternatively, for Android 5.0 and later, we can utilize the API `WebResourceRequest.hasGesture()` to determine the relation between a user's gesture (like a click) and the start of an activity. Note that although these approaches are still subject to clickjacking [36], they make a XAWI attack more visible to the user: even when the remote adversary manages to issue a navigation request using an unrelated user click to infect another app, he cannot command the infected WebView (now in the foreground) to switch to the background through another navigation request without triggering a user dialog. Another way to avoid user interactions is using a whitelist of trusted websites. The developer can include such a list in her app's manifest. Whenever a navigation is directed from any domain on the list, the request is allowed to go through without asking the user.

Evaluation. To evaluate NaviGuard, we chose the 6 vulnerable apps (i.e., Facebook, Twitter, Baidu, etc.) analyzed in Section 3 and Appendix A, together with 44 apps randomly selected from all the vulnerable apps reported by ViewFinder, and installed them on a Nexus 5 device running a customized Android 4.4 with the NaviGuard enhancement. Then we utilized the ADB tool to inject the infectious Intents found by ViewFinder from these apps, which successfully navigated their unprotected WebViews to the sites under our control. In this experiment, however, all these Intents were either blocked (when they were issued from the background) or caused an alert to be raised to get the user's consent. This indicates that no longer can such attacks go unnoticed to the user.

Also important here is the performance of the technique, which should not cause too much delay when there is no infection attempt going on. In our experiment, we ran Monkey, a UI exerciser tool [15], to generate 10,000 random events towards 360 popular apps (top 10 from each of 36 Google Play categories) in the presence of NaviGuard, and then replay the same set of events to the same apps without our protection. During the two tests, we measured the delays introduced, denoted by t_1 and t_2, respectively for these two settings, and further calculated the overhead $((t_1 - t_2)/t_2)$. The study shows that the overhead incurred is very low, around 0.5%.

We further evaluated the compatibility of our techniques with existing apps. For this purpose, we installed 50 popular apps on a Nexus 5. After running Monkey across these apps with 100,000 random events, we found no runtime error caused by NaviGuard reported in the system log, indicating that the security controls put in place will not disrupt these apps' normal operations.

5 LESSON LEARNT

The root cause of XAWI is the capability to cross-WebView communication, particularly navigating another WebView to a given domain from the web. This capability, however, is critical for the integrated service, which is supposed to directly link web content to the most suitable platform (app) to present it. Actually, today's content providers are increasingly utilizing deep linking techniques to indicate to the browser or WebView not only a specific app but also its component for handling the specific content (e.g., video, image, links, etc.) on their web pages. Such cross-WebView content distribution is not a capability that can be curtailed, even given the security implications we discovered.

Indeed, not only Android but also iOS is aggressively using this capability. Actually, the scheme channel was even less protected on iOS until recently, when research shows that URL schemes can be hijacked by a malicious *local app* (installed on the target device) that steals sensitive user information, such as secret tokens from another app [39]. As a result, since iOS 9, any scheme invocation across apps needs the user's approval, which is clearly less convenient than Apple hopes. More recently, Apple is pushing a new deep linking mechanism called *universal links* on iOS 9 and later [11]. This mechanism binds an app to a link, with a certificate-based verification. Through the link, one can directly trigger another app's component (e.g., WebView) and pass parameters (e.g., URL) without asking the user. As a result, this new mechanism, once being widely deployed, could also bring in cross-WebView infections, though more studies are certainly needed to better understand its security risk.

A key lesson learnt from our study is that a smooth cross-WebView channel can also become a path for infection transmission. Safeguards should be in place on the path during the design and implementation of such a communication mechanism. For example, it would not be excessive for the app receiving navigation request to check the security risk of the domain it is about to move into, should the app decide not to confine its WebView within a white-list of domains (for the purposes like user retention). Techniques for protecting web surfing, such as use of blacklists like Google Safe Browsing, could be necessary, if they are made more efficient and more suitable for working on the IPC level. Also, isolation should be applied to protect the WebView with critical capabilities, together with quarantine of the untrusted domain within the WebView that cannot communicate with other activities except those provided by the app that initiates the navigation request. Also, it is important to provide guidance and SDKs to the app developer for putting security checks at the right program locations, as well as develop program testing techniques for systematically detecting the lapses in an app's domain control. Further, incentives should also be given by the content provider to developers for better protecting their apps, through, for example, only linking the web content to the apps of good security quality. On the OS front, at least the URLs passed between the apps could be inspected to identify known malicious domains or anomalies.

6 RELATED WORKS

Attacks on WebView. WebView is a component vulnerable to various attacks. Previous studies show that untrusted web contents can leverage JS interfaces to connect to a smartphone's local resources such as GPS locations [25] and file system [10, 25]. In the meantime, an attack app could also inject malicious JavaScript code into the web contents, sniff and hijack user events [25]. These vulnerabilities are found to be pervasive [28] and are not fixed timely [35]. However, none of these prior studies looked into possibility of cross-WebView, multi-app attacks and security implications of unprivileged WebView (those without any JS interface and call-back capability), which have first been investigated in our research.

Security risks in URL schemes. URL-scheme IPC is known to be vulnerable to hijacking attacks, particularly on iOS and OS X, in which a malicious app claims the scheme used by popular apps to steal the Intents sent to them or impersonate those apps [39]. Scheme-based web-to-app attack is also found to be possible on iOS, with a remote cross-site request forgery reported in a prior study [38]. On Android, as mentioned earlier, Opera and Chrome are found to expose their private functionalities to WebView [37]. Most related to our research is the finding that Samsung's `UniversalMDMClient` can be launched through a URL, asking the user whether she wants to install an update[27]. On the other hand, never before has any systematic effort been made to understand the security implication of cross-WebView navigation, a functionality considered to be legitimate and necessary. Our studies reveal the serious security risks involved in this communication, which enables a remote adversary to attack the mobile users in a way that cannot be imagined before, including remote app infections, persistent app control and multi-app colluding attacks. Our findings point to the fundamental design weakness in URL scheme management on Android and new attack surface it exposes to the remote adversary.

Mobile Phishing. GUI-related Phishing has long been studied [17] and recently mobile Phishing has also been intensively investigated [5–7, 19, 21, 31]. Particularly, prior research investigates the vulnerable links between mobile apps and web sites [19], task hijacking [31] that enables a malicious app to implement UI spoofing, by manipulating system back stacks or a benign app's task stack, side-channel based identification of attack opportunities [7], and other kinds of Phishing activities, such as SMiShing and Vishing [33]. However, none of these studies investigate the risk of a fully remote, multi-app Phishing attack, which our study found is completely feasible. This surprising RDP attack turns out to be extremely powerful, outclassing all existing defense (Section 3.2) and being recognized by the industry to be a realistic threat.

7 CONCLUSION

In this paper, we report our finding of a fundamental design challenge in cross-WebView navigation, a much-needed capability for integrating the services from different apps. Our study reveals a new XAWI weakness overlooked by the prior research, through which a remote adversary can acquire persistent, stealthy control on multiple apps, as soon as his web content is triggered by Chrome. We demonstrate that a series of multi-app, colluding attacks can be launched to perform highly realistic remote Phishing attacks and escalate the remote adversary's privileges. Also such vulnerable apps are found to be pervasive, at least 7.4% among popular apps, including Facebook, Google Drive, Twitter, TripAdvisor, etc. To protect Android users, we developed a new technique to automatically control cross-WebView communication. Most importantly, our study brings to light the contention between the strong demand for convenient web-to-app linking and the security need for controlling the channels for such communication. We show that existing protection on the channels has not been well thought-out and often can be easily bypassed. Further effort is required to better understand the problem and find the solution that closes the attack avenues without undermining the utility of the channels.

8 ACKNOWLEDGMENTS

PKU authors (contact: Xinhui Han) were supported in part by NSFC 61402125 and National Key Research and Development Program of China (Grant No.2016YFB0801302). IU authors (contact: XiaoFeng Wang) were supported in part by NSF CNS-1223477, 1223495, 1527141, 1618493, ARO W911NF1610127 and Samsung Gift fund. IIE authors (contact: Kai Chen) were supported in part by NSFC U1536106 and 61728209, National Key Research and Development Program of China (Grant No.2016QY04W0805), Youth Innovation Promotion Association CAS, and strategic priority research program of CAS (XDA06010701).

A MORE ATTACK CASES

Device state detecting and tampering. We also discovered in our research that from Baidu mobile assistant, an app store app among the most popular Chinese apps (with over 100 million users), a remote XAWI adversary can acquire the capabilities to monitor the user's interactions with her device, identify other apps on the

infected device and even perform an unauthorized app install. Related functionalities are provided by the Baidu app to its WebView through JavaScript interfaces, including the readings of the device's gyroscope, the loudness of the voice perceived by the device's microphone, the existence of a package and installation of an app from the SD card. However, direct navigation from Baidu's appstore page to a malicious website is unlikely, since its WebView does not provide a URL bar and other assistance for browsing unrelated sites. Further, there is protection in place that whenever the WebView leaves a domain under Baidu's control, part of JavaScript interfaces' functionalities are disabled.

In our research, again we use Chrome as the entry point for the attack. The attack content inside Chrome's WebView generates an Intent scheme (with the package name of the Baidu app) to trigger the Baidu activity `UrlHandlerActivity`, which has registered an Intent filter for the scheme http://*/.*/api/calendar (specified in its data field). The activity responds to the attack URL http://attack.com/new/api/calendar, silently navigating the WebView to attack.com. Under the domain, though part of the Javascript interfaces functionalities are stopped, we found that still important capabilities are exposed. Particularly, the JS interfaces `downloadApp` and `getAppInfo` are open to the untrusted domain. So the adversary can find out what app has been installed through querying `getAppInfo` or download app packages through `downloadApp`. Also interestingly, our research shows that Baidu utilizes the WebView callback `shouldOverrideUrlLoading` but fails to protect it. The callback operates on the URLs in the form of appclient:download..., which leads to the download of a file from a specific web location, and appclient:intent intent://..., which creates a deep link for invoking an activity.

We further come up with a new technique to bypass Baidu's domain protection. A problem with Baidu's JavasScript interfaces is that some of the JS interfaces it gives to WebView allow callbacks: e.g., `downloadApp(String url, String callback)`. Here the callback is a piece of JavaScript code to be executed after completion of the function call, in an asynchronous way. This creates a race condition that enables a Time of Check and Time of Use (TOCTOU) attack. Specifically, the attack web content can invoke such an interface, supplying it with JavaScript code as the callback. In the meantime, the content also initiates a navigation to a Baidu domain. The trick here is that once the navigation is complete, even though the adversary loses the control of the WebView, he can regain it when the JS code in the callback is injected back to the current domain, which now is an authorized domain with full JavaScript interfaces capabilities. We successfully executed the attack in our study (also see our demo [1]).

Once the JavaScript interfaces are open, the malicious script can further access user information on the device. We found that through the JavaScript interfaces, the adversary can change the user's calendar, add reminders, collect the readings from its gyroscope and the real-time loudness of the voice when the user is speaking to her phone (which can be a potential side channel), get the user's login state and account information, and even automatically install an app through `installApp` (when the auto-install setting in the app is turned on). All these attacks can happen in a stealthy way, when the infected WebView is running in the background. We reported the vulnerability to Baidu and helped them fixed it.

B FIGURES AND TABLES

`SecureIntentHandlerActivity` is an Activity provided by Facebook Messenger. As illustrated in (Figure 9), this Activity is protected by a permission `FB_APP_COMMUNICATION`, a signature one only given to Facebook's products. The Activity also registers an Intent Filter to receive Intents with scheme fb-messenger-secure://. Once receiving an Intent with such scheme (see example in figure 9), Faceboook Messenger will send out a message without user consent.

```
<!--  Activity  -->
<activity android:name="com.facebook.messenger.intents.
     SecureIntentHandlerActivity" android:permission="com
     .facebook.permission.prod.FB_APP_COMMUNICATION">
  <intent-filter>
    <action android:name="android.intent.action.VIEW"/>
    <category android:name="android.intent.category.
         DEFAULT"/>
    <data android:scheme="fb-messenger-secure"/>
  </intent-filter>
</activity>

<!--  Scheme used to send ''content'' to ''userid''  -->
fb-messenger-secure://autocompose/post?tid=userid&ttype=2&
     s=1&m=content
```

Figure 9: An activity from Facebook Messenger and an exploiting scheme

Activity `UrlInterpreterActivity` in Twitter registers an Intent filter to handle URL as illustrated in Figure 10. Upon receiving a related URL, the Activity can launch another Activity and navigate the latter's WebView to a Phishing page. To trigger the WebView without showing a system dialog, our attack sends an explicit Intent scheme to Twitter.

```
<!--  Activity  -->
<activity android:name="com.twitter.android.
     UrlInterpreterActivity">
  <intent-filter android:autoVerify="true">
    <action android:name="android.intent.action.VIEW"/>
    <category android:name="android.intent.category.
         DEFAULT"/>
    <category android:name="android.intent.category.
         BROWSABLE"/>
    <data android:scheme="http"/>
    <data android:scheme="https"/>
    <data android:host="twitter.com"/>
    <data android:host="www.twitter.com"/>
    <data android:pathPattern="/.*"/>
  </intent-filter>

</activity>
<!--  Handled URL  -->
intent://www.twitter.com/i/redirect?url=http%3A%2F%2
     Fattacker.com%2Fmessages%2Fmedia%2Fattack.html#
     Intent;package=com.twitter.android;scheme=http;end
```

Figure 10: An activity from Twitter and the scheme to trigger it

Table 1 lists several vulnerable apps detected by our tool *ViewFinder*.

Table 1: XAWI-susceptible apps. (✓ *indicates the feature is supported.*)

Package	Installation	JS	HTML5	Custom Scheme	Background
com.google.android.apps.docs	1,000,000,000 ~ 5,000,000,000	✓	○	✓	✓
com.evernote	100,000,000 ~ 500,000,000	○	○	○	✓
vStudio.Android.Camera360	100,000,000 ~ 500,000,000	✓	○	○	✓
com.tripadvisor.tripadvisor	100,000,000 ~ 500,000,000	○	○	○	✓
com.roidapp.photogrid	100,000,000 ~ 500,000,000	○	○	○	✓
com.pinterest	100,000,000 ~ 500,000,000	✓	○	○	✓
com.picsart.studio	100,000,000 ~ 500,000,000	○	○	✓	✓
com.kakao.talk	100,000,000 ~ 500,000,000	○	○	○	✓
com.imo.android.imoim	100,000,000 ~ 500,000,000	○	○	○	✓
com.hola.launcher	100,000,000 ~ 500,000,000	✓	○	○	✓
com.gau.go.launcherex	100,000,000 ~ 500,000,000	✓	✓	○	✓
com.cyworld.camera	100,000,000 ~ 500,000,000	✓	✓	○	✓
com.commsource.beautyplus	100,000,000 ~ 500,000,000	✓	○	○	✓
com.alibaba.aliexpresshd	100,000,000 ~ 500,000,000	○	✓	○	✓
cn.wps.moffice_eng	100,000,000 ~ 500,000,000	✓	○	✓	✓
com.zeroteam.zerolauncher	50,000,000 ~ 100,000,000	✓	✓	○	✓
com.rhmsoft.fm	50,000,000 ~ 100,000,000	✓	○	○	✓
com.nhn.android.search	50,000,000 ~ 100,000,000	✓	○	○	○
com.mobisystems.office	50,000,000 ~ 100,000,000	○	○	○	✓
com.melodis.midomiMusicIdentifier.freemium	50,000,000 ~ 100,000,000	○	○	✓	✓
com.ksmobile.launcher	50,000,000 ~ 100,000,000	✓	○	○	✓
com.intsig.camscanner	50,000,000 ~ 100,000,000	✓	○	✓	✓
com.indeed.android.jobsearch	50,000,000 ~ 100,000,000	✓	○	✓	✓
com.halo.wifikey.wifilocating	50,000,000 ~ 100,000,000	○	✓	○	✓
com.gau.go.launcherex.gowidget.weatherwidget	50,000,000 ~ 100,000,000	○	○	○	✓
com.cootek.smartinputv5	50,000,000 ~ 100,000,000	○	○	✓	○
com.cardinalblue.piccollage.google	50,000,000 ~ 100,000,000	○	○	○	✓
com.audible.application	50,000,000 ~ 100,000,000	✓	○	✓	✓
com.amazon.mShop.android.shopping	50,000,000 ~ 100,000,000	✓	✓	○	○
co.vine.android	50,000,000 ~ 100,000,000	○	○	✓	✓
com.yelp.android	10,000,000 ~ 50,000,000	✓	○	✓	✓
net.daum.android.map	10,000,000 ~ 50,000,000	○	○	✓	✓
net.daum.android.daum	10,000,000 ~ 50,000,000	○	○	✓	○
jp.united.app.cocoppa	10,000,000 ~ 50,000,000	○	✓	✓	✓
jp.co.mcdonalds.android	10,000,000 ~ 50,000,000	○	○	○	✓
de.hafas.android.db	10,000,000 ~ 50,000,000	○	○	✓	✓
com.zynga.wwf2.free	10,000,000 ~ 50,000,000	○	○	○	○
com.xinmei365.font	10,000,000 ~ 50,000,000	○	○	○	✓
com.tokopedia.tkpd	10,000,000 ~ 50,000,000	○	○	○	○
com.toi.reader.activities	10,000,000 ~ 50,000,000	○	✓	○	✓
com.skimble.workouts	10,000,000 ~ 50,000,000	○	○	○	✓
com.ScanLife	10,000,000 ~ 50,000,000	○	✓	✓	✓
com.rhmsoft.fm.hd	10,000,000 ~ 50,000,000	○	○	✓	✓
com.quoord.tapatalkpro.activity	10,000,000 ~ 50,000,000	○	○	○	✓
com.prestigio.ereader	10,000,000 ~ 50,000,000	○	○	○	✓
com.nytimes.android	10,000,000 ~ 50,000,000	○	○	✓	✓
com.naver.linewebtoon	10,000,000 ~ 50,000,000	○	○	✓	✓
com.mt.mtxx.mtxx	10,000,000 ~ 50,000,000	○	○	○	✓
com.makemytrip	10,000,000 ~ 50,000,000	○	○	○	✓
com.mobilesrepublic.appy	10,000,000 ~ 50,000,000	○	○	○	✓
com.lbe.parallel.intl	10,000,000 ~ 50,000,000	○	○	✓	✓

REFERENCES

[1] 2017. Supplement materials. https://sites.google.com/site/xawisite/. (May 2017).

[2] Yousra Aafer, Nan Zhang, Zhongwen Zhang, Xiao Zhang, Kai Chen, XiaoFeng Wang, Xiaoyong Zhou, Wenliang Du, and Michael Grace. 2015. Hare hunting in the wild android: A study on the threat of hanging attribute references. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1248–1259.

[3] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269.

[4] Antonio Bianchi, Jacopo Corbetta, Luca Invernizzi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. 2015. What the app is that? deception and countermeasures in the android user interface. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 931–948.

[5] Kai Chen, Tongxin Li, Bin Ma, Peng Wang, XiaoFeng Wang, and Peiyuan Zong. 2017. Filtering for Malice through the Data Ocean: Large-Scale PHA Install Detection at the Communication Service Provider Level. In *International Symposium on Research in Attacks, Intrusions, and Defenses*.

[6] Kai Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Nan Zhang, Heqing Huang, Wei Zou, and Peng Liu. 2015. Finding Unknown Malice in 10 Seconds: Mass Vetting for New Threats at the Google-Play Scale. In *USENIX Security Symposium*. 659–674.

[7] Qi Alfred Chen, Zhiyun Qian, and Z Morley Mao. 2014. Peeking into your app without actually seeing it: UI state inference and novel android attacks. In *23rd USENIX Security Symposium (USENIX Security 14)*. 1037–1052.

[8] Yangyi Chen, Tongxin Li, XiaoFeng Wang, Kai Chen, and Xinhui Han. 2015. Perplexed messengers from the cloud: Automated security analysis of push-messaging integrations. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1260–1272.

[9] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. 2011. Analyzing inter-application communication in Android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*. ACM, 239–252.

[10] Erika Chin and David Wagner. 2013. Bifocals: Analyzing webview vulnerabilities in android applications. In *International Workshop on Information Security Applications*. Springer, 138–159.

[11] Apple Developer. 2017. Support Universal Links. https://developer.apple.com/library/content/documentation/General/Conceptual/AppSearch/UniversalLinks.html. (May 2017).

[12] Android Developers. 2017. Activity Element. https://developer.android.com/guide/topics/manifest/activity-element.html. (May 2017).

[13] Android Developers. 2017. Android Debug Bridge. https://developer.android.com/studio/command-line/adb.html. (May 2017).

[14] Android Developers. 2017. Tasks and Back Stack. https://developer.android.com/guide/components/tasks-and-back-stack.html. (May 2017).

[15] Android Developers. 2017. UI/Application Exerciser Monkey. http://developer.android.com/tools/help/monkey.html. (May 2017).

[16] Facebook Developers. 2016. App Links. https://developers.facebook.com/docs/applinks. (November 2016).

[17] Rachna Dhamija and J Doug Tygar. 2005. The battle against phishing: Dynamic security skins. In *Proceedings of the 2005 symposium on Usable privacy and security*. ACM, 77–88.

[18] Alon Even. 2016. How to Grow Your Mobile App Retention. http://www.apptamin.com/blog/grow-app-rentention/. (2016).

[19] Adrienne Porter Felt and David Wagner. 2011. *Phishing on mobile devices*. na.

[20] Adrienne Porter Felt, Helen J Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. 2011. Permission Re-Delegation: Attacks and Defenses.. In *USENIX Security Symposium*, Vol. 6. 12–16.

[21] Earlence Fernandes, Qi Alfred Chen, Justin Paupore, Georg Essl, J Alex Halderman, Z Morley Mao, and Atul Prakash. 2016. Android UI Deception Revisited: Attacks and Defenses. In *Proceedings of the 20th International Conference on Financial Cryptography and Data Security*.

[22] iBotPeaches. 2017. Apktool. https://ibotpeaches.github.io/Apktool/. (May 2017).

[23] Yeonjoon Lee, Tongxin Li, Nan Zhang, Soteris Demetriou, Mingming Zha, XiaoFeng Wang, Kai Chen, Xiaoyong Zhou, Xinhui Han, and Michael Grace. 2017. Ghost Installer in the Shadow: Security Analysis of App Installation on Android. In *Dependable Systems and Networks (DSN), 2017 47th Annual IEEE/IFIP International Conference on*. IEEE.

[24] Tongxin Li, Xiaoyong Zhou, Luyi Xing, Yeonjoon Lee, Muhammad Naveed, XiaoFeng Wang, and Xinhui Han. 2014. Mayhem in the push clouds: Understanding and mitigating security hazards in mobile push-messaging services. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 978–989.

[25] Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin. 2011. Attacks on WebView in the Android system. In *Proceedings of the 27th Annual Computer Security Applications Conference*. ACM, 343–352.

[26] Tongbo Luo, Xing Jin, Ajai Ananthanarayanan, and Wenliang Du. 2012. Touchjacking attacks on web in android, ios, and windows phone. In *International Symposium on Foundations and Practice of Security*. Springer, 227–243.

[27] Andre Moulu. 2014. Abusing Samsung KNOX to remotely install a malicious application: story of a half patched vulnerability. https://blog.quarkslab.com/abusing-samsung-knox-to-remotely-install-a-malicious-application-story-of-a-half-patched-vulnerability.html. (November 2014).

[28] Patrick Mutchler, Adam Doupé, John Mitchell, Chris Kruegel, and Giovanni Vigna. 2015. A Large-Scale Study of Mobile Web App Security. In *Proceedings of the Mobile Security Technologies Workshop (MoST)*.

[29] Marcus Niemietz and Jörg Schwenk. 2012. Ui redressing attacks on android devices. *Black Hat Abu Dhabi* (2012).

[30] Chuangang Ren, Peng Liu, and Sencun Zhu. 2017. WindowGuard: Systematic Protection of GUI Security in Android. In *Proc. of the Annual Symposium on Network and Distributed System Security (NDSS)*.

[31] Chuangang Ren, Yulong Zhang, Hui Xue, Tao Wei, and Peng Liu. 2015. Towards discovering and understanding task hijacking in android. In *24th USENIX Security Symposium (USENIX Security 15)*. 945–959.

[32] rovo89. 2017. Xposed Module Repository. http://repo.xposed.info. (May 2017).

[33] Hossain Shahriar, Tulin Klintic, Victor Clincy, et al. 2015. Mobile Phishing Attacks and Mitigation Techniques. *Journal of Information Security* 6, 03 (2015), 206.

[34] Thomas Sommer. 2014. User Retention: Yes, But Which One? http://www.applift.com/blog/user-retention.html. (February 2014).

[35] Tom Sutcliffe and Adrian Taylor. 2015. The Lifetime of Android API Vulnerabilities: Case Study on the JavaScript-to-Java Interface. In *Security Protocols XXIII: 23rd International Workshop, Cambridge, UK, March 31-April 2, 2015, Revised Selected Papers*, Vol. 9379. Springer, 126.

[36] Symantec. 2016. Android ransomware variant uses clickjacking to become device administrator. https://www.symantec.com/connect/blogs/android-ransomware-variant-uses-clickjacking-become-device-administrator. (January 2016).

[37] Mitsui Bussan Takeshi Terada. 2014. Whitepaper – Attacking Android browsers via intent scheme URLs. http://www.mbsd.jp/Whitepaper/IntentScheme.pdf. (March 2014).

[38] Rui Wang, Luyi Xing, XiaoFeng Wang, and Shuo Chen. 2013. Unauthorized origin crossing on mobile platforms: Threats and mitigation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 635–646.

[39] Luyi Xing, Xiaolong Bai, Tongxin Li, XiaoFeng Wang, Kai Chen, Xiaojing Liao, Shi-Min Hu, and Xinhui Han. 2015. Cracking App Isolation on Apple: Unauthorized Cross-App Resource Access on MAC OS. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 31–43.