Demystifying Resource Management Risks in Emerging Mobile App-in-App Ecosystems

Haoran Lu, Luyi Xing*, Yue Xiao, Yifan Zhang, Xiaojing Liao, XiaoFeng Wang, Xueqiang Wang
Indiana University Bloomington
{haorlu, luyixing, xiaoyue, yz113, xliao, xw7, xw48}@iu.edu

ABSTRACT

App-in-app is a new and trending mobile computing paradigm in which native app-like software modules, called *sub-apps*, are hosted by popular mobile apps such as Wechat, Alipay, Baidu, TikTok and Chrome, to enrich the host app's functionalities and to form an "all-in-one app" ecosystem. Sub-apps access system resources through the host, and their functionalities come close to regular mobile apps (taking photos, recording voices, banking, shopping, etc.). Less clear, however, is whether the host app, typically a third-party app, is capable of securely managing sub-apps and their access to system resources. In this paper, we report the first systematic study on the resource management in app-in-app systems. Our study reveals high-impact security flaws, which allow the adversary to stealthily escalate privilege (e.g., accessing the camera, photo gallery, microphone, etc.) or acquire sensitive data (e.g., location, passwords of Amazon, Google, etc.). To understand the impacts of those flaws, we developed an analysis tool that automatically assesses 11 popular app-in-app platforms on both Android and iOS. Our results brought to light the prevalence of the security flaws. We further discuss the lessons learned and propose mitigation strategies.

1 INTRODUCTION

A new mobile-computing paradigm, dubbed *app-in-app*, is gaining popularity in the past years. Under this paradigm, a mobile app, called *host app* or *host*, operates a set of *sub-apps* as its in-app components. These sub-apps give users native app like experience and enriched functionalities (e-commerce, banking, health, travel management, food ordering, etc.), thereby increasing their "stickness" to the host: one does not need to leave the app if it does everything [93]. They are installed by the users from the host's sub-app store, just like Google Play and Apple app store, which fosters an ecosystem around the host app. As a prominent example, it is reported that Wechat, the 5th most-used app in the world [27], has one million sub-apps in its sub-app store and 200 million daily sub-app users [5, 6]. In the meantime, sub-app vendors also benefit from the host's large customer base. For example, YuXiaoge, an e-commerce retailer is known to make more than 1.5 million USD of monthly sales through its sub-app [28] in Wechat, which allows it to access Wechat's one billion users worldwide [25]. Another example is Pinduoduo, a group-buying platform that has acquired 300 million users in three years from its wildly successful Wechat sub-app [25, 29]. Nowadays, this app-in-app paradigm has been supported by many popular apps such as Wechat, Alipay, TikTok, Baidu, Chrome, Firefox, etc., with other major app vendors also actively participating in these hosts' ecosystems.

*Corresponding author.

New security risks in the app-in-app paradigm. An app-in-app system is under the control of the host app, which allocates resources to sub-apps, including system resources it acquires from the OS. For this purpose, the host app acts like the OS, e.g., providing its own APIs (a.k.a., *sub-app API*) to its sub-apps for resources access. Also like the OS, the host app mediates sub-apps' access to security-critical resources (e.g., GPS location, microphone, camera, Bluetooth, photos, etc.) with its own permission-based access control mechanism. Further, a sub-app's interactions with the user go through the host's user interface (UI): usually the host creates a dedicated (fullscreen) window for each sub-app. Even the lifecycle of the sub-app is controlled by the host, which decides when it should be closed if resources are in high demand.

What comes with the app-in-app systems are new security risks inherent to the paradigm. Prior research on mobile permission re-delegation attacks [56] (a.k.a., confused deputy) shows that a privileged app (deputy) may leak OS resources to malicious apps with the exposure of its internal functions (through unprotected public interface) either unintentionally or deliberately (when the deputy opts to not implement protection since it will not bear the consequences of the attacks). For an app-in-app system, however, the host is designed to mediate sub-apps' access to sensitive system resources (e.g., location, microphone, camera, photos, etc.) and avoid the re-delegation risks. However, it is never clear whether a third-party app – the host – is capable of properly managing the OS resources. Indeed, this can be very difficult, since the host does not have full information about resource management at the OS level: for example, the full list of permissions required for resource access has never been made public by mobile OS vendors, and is very hard to obtain [35, 46].

Besides resources accessed through system APIs, user interface (UI) also needs protection, whose improper management opens avenues to phishing attacks [38, 49]. State-of-the-art defense against mobile phishing [49, 58, 96] relies on identifying the foreground app: if the user knows exactly which app she is interacting with, she would not expose secrets to the unintended one. Such protection, however, does not work under the app-in-app UI model: since a sub-app is rendered in the host's window, a malicious sub-app can be hard to differentiate from the host and other sub-apps providing security-critical services (e.g., e-commerce, banking, health, etc.). With the importance of those potential security risks, little has been done so far to understand whether the app-in-app paradigm has been adequately protected.

Our study. In this paper, we report the *first* systematic security analysis on app-in-app systems across Android and iOS. Our research brought to light the fundamental conflicts between the rich, native app-like functionalities expected from the sub-app, and the fundamental lack of capabilities for a third-party app – the host –

to securely mediate access to system resources it acquired. More seriously, we found that the functionality- and user-oriented app-in-app ecosystems actually undermine the security protection of modern mobile OSes, by introducing new risks to their otherwise secure permission-based resource control.

In particular, the hosts generally fail to provide sound security policies and effective control to protect sensitive system resources from unauthorized sub-apps. For example, `Wi-Fi scan` is guarded by a *dangerous* permission (i.e., `location`) on Android and an *entitlement* on iOS, because this capability can leak user location [37]. However, Wechat discloses it to sub-apps (on both OSes) without proper authorization – a serious violation of permission protection on user `location`. Similar problems are also discovered on other resources managed by popular host apps, which allow the adversary, through an unauthorized sub-app, to stealthily gain access to microphone, location, camera, etc., in the absence of a user consent. Fundamentally, the problem comes from *a knowledge gap*: the host developer fundamentally lacks the OS level knowledge, and therefore is not in position to design sound sub-app level permission policies that comply with the OS level resource protection. Bridging this knowledge gap presents new challenges to the design of app-in-app systems, such as misleading developer documentations and conflicting OS-level security policies between Android and iOS, which have not been systematically studied before (Section 3.1).

Also our research shows that an app-in-app system undermines the protection enforced by a mobile OS by rendering the state-of-the-art UI deception defense ineffective (Section 3.2). We present an attack vector that is inherent in the app-in-app UI model, which introduces realistic security hazards during app-user interactions. Exploiting the weakness, a malicious sub-app, leveraging its UI embedded in its host's window, can strategically impersonate the host to get sensitive information from the user, such as account passwords. Our research shows that leading host apps such as Safari (on iOS), Chrome (on Android), Wechat and Alipay (on both OSes) are all vulnerable, allowing a malicious sub-app to steal the user's passwords for Amazon, Google, various mobile wallets, etc.

Further, as mainstream app vendors participate in app-in-app paradigm by releasing sub-apps that are functionality-equivalent to their regular mobile apps [29] (e.g., e-commerce, banking, travel, health, etc.), their (sub-)apps can no longer benefit from the protection offered by modern mobile OSes – which have been hardened after years of open security research. Safeguarding sub-apps today mostly relies on individual host vendors, who however fail to demonstrate that they are up to this challenge: our study shows that even leading host apps with more than 100 million downloads on Google Play and Apple App Store (e.g., Wechat, Alipay, Tiktok) expose new attack surfaces and exploitable weaknesses through their app-in-app supports (Section 3).

Impacts. To understand the scope and magnitude of the newly discovered security risks (called app-in-app flaws or *APINA flaws* for short), we analyzed 11 most popular commodity app-in-app platforms on both iOS and Android, including Wechat, Alipay, TikTok, JinRiTouTiao, Chrome, Safari, etc. To enable systematic measurement and analysis of these flaws on a large scale, we developed a scanner called *Apinat* (short for App-in-app Threat Scanner), to report whether a host app contains APINA flaws, leveraging a combination of test case generation, dynamic analysis and lightweight computer vision techniques.

Running *Apinat* on the popular platforms, we found significant and broad impacts of APINA flaws (Section 4.3): every single app-in-app system we studied is vulnerable and exploitable with serious consequences (Table 4). More specifically, APINA flaws allow the adversary (i.e., either a malicious sub-app or native app) to stealthily acquire critical system capabilities (e.g., accessing camera, microphone, connecting to arbitrary Wi-Fi access point, connecting the phone to arbitrary Bluetooth device, etc.) without the user consent, and steal users' private data (e.g., location, account credentials of Amazon and top banks, credit cards, mailing address, travel logs, health statistics). We reported our findings to all affected vendors, who acknowledged that what we found are real and significant. Google, Wechat and Alipay all awarded us through their bug bounty programs. The demos of our attacks are available [12].

Contributions. The contributions are outlined as follows:

- We conducted the first systematic security analysis on the app-in-app paradigm and discovered a series of unexpected, security-critical flaws. Our findings bring in new insights into the fundamental security limitations and challenges in designing this new computing platform and ecosystem, and are invaluable to the enhancement of its security protection. Also, we believe that our findings are just a tip of the iceberg, and will inspire the follow-up research on this direction.
- We developed new techniques to detect APINA flaws and measure their pervasiveness and impacts in 11 real-world app-in-app systems. We demonstrate that the APINA flaws are indeed prevalent, across both Android and iOS, with severe security and privacy implications. We release the source code of our tool [12].
- We discuss mitigation strategies and the lessons learned for building a more secure app-in-app system.

2 MOBILE APP-IN-APP SYSTEMS

A unique feature of app-in-app paradigm is its sophisticated sub-apps, which are designed to provide *native-app experience* [5]. As an instance, Wechat has provided "mini-programs" since 2017, which are essentially sub-apps hosted inside the Wechat app. Another example of the app-in-app paradigm in our study is the mobile browser supporting *standalone* Web App, since Web App is designed to act like regular, native app: a Web App is launched from the home screen, and takes a dedicated entire window (separated from the browser window) just like a standalone, native app – as advocated by Google and Apple [44, 78]. Using Web App, such as Pinterest, is not like interacting with a webpage [4], but more like launching the regular Pinterest mobile app, except that its whole window is created by the browser, which is transparent to the users.

Table 1 shows 11 popular app-in-app ecosystems we studied: besides Wechat, other host apps are also remarkably popular, e.g., Alipay has one billion monthly active users and 120,000 sub-apps [9]. Also, high-profile retailers, app and service providers (e.g., Amazon, Microsoft, Airbnb, Samsung, HSBC, JD.com, Starbucks, McDonald's) actively contribute to the trending app-in-app paradigm by releasing sub-apps to these hosts.

Table 1: Popular host apps and the amount of sub-apps available in the host apps.

Host App	Functionality (of the host itself, except sub-apps)	Number of Sub-App	Number of Downloads
WeChat	Instant Messaging, Social Networking, Social Media, VoIP, Mobile Wallet	1M+	200M+
Alipay	Mobile Wallet, Finance/Investment Management, Shopping, News, Social Networking, Utility Bill Management, Travel	120,000+	1B+
Chrome	Web Browser	N/A†	1B+
Safari	Web Browser	N/A†	N/A†
TikTok	Video Sharing, Instant Messaging, Personal Blog, Game Center	N/A†	500M+
JinRiTouTiao	News Feeding, live streaming	N/A†	1M+
QQ	Instant Messaging, Video/Audio Chatting, File Transfer, Personal Blog, Game Center, Mobile payment	N/A†	10M+
Firefox	Web Browser	N/A†	100M+
Opera	Web Browser	N/A†	100M +
Baidu	Search engine, News, Short Videos, Voice Recognition, Readings	N/A†	230M+
DingTalk	Address Book, Mobile Office Tool Box, Video Conference	20,000	500K+
Total	-	1M+	2.6B+

†: Lack public information.

2.1 Architecture

To enable native-app experience for sub-apps, each host app provides a set of sub-app APIs for sub-apps to access diverse system resources (such as the camera, microphone, Bluetooth, NFC, Contacts, photos). Also, the sub-app is cross-platform: it is typically developed in script languages (particularly JavaScript [20]) and runs on both the Android and iOS version of its host app. These features are fulfilled by the host app through a cross-platform runtime, called *sub-app runtime* in this paper. Below, we summarize its typical architecture, which we learned through reverse engineering popular host apps (Table 1).

Figure 1 outlines a typical app-in-app architecture from the perspective of resource access. Specifically, the host app consists of a sub-app layer, where all sub-apps live, and a sub-app runtime, where a generic abstraction was provided for system and host app resource access. Sub-app runtime bridges sub-app API calls (in JavaScript) into the native layer (in Java on Android, and Objective-C [39] or Swift [40] on iOS) of host apps. The native layer then enforces home-grown permissions of the host app (a.k.a., *sub-app permissions*), and either accesses host resources or calls corresponding system APIs to access system resources.

Specifically, in the script layer, invoking a sub-app API (e.g., `connectWiFi`) will trigger an *Encapsulation lib*, which then calls a native function (e.g., `dispatcher.dispatch('connectWiFi')`) to bridge the control flow into native layer. Particularly, we found two common bridge techniques adopted by host apps, WebView [79] and React Native [55]. To use Webview, a host app leverages a WebView API `addJavascriptInterface(obj, 'dispatcher')` to expose a native object `obj` aliased 'dispatcher' to the script layer; its *dispatch* function is then accessible in script layer, as mentioned above. Given each sub-app API, the *dispatch* function invokes the corresponding native-layer libraries to check sub-app permission and access resources. To use the React Native bridge, the major difference here is how to expose the native object to script layer: `dispatcher` object's class in the native layer, namely `Dispatcher`, must inherit an Interface `ReactPackage` of React Native framework;

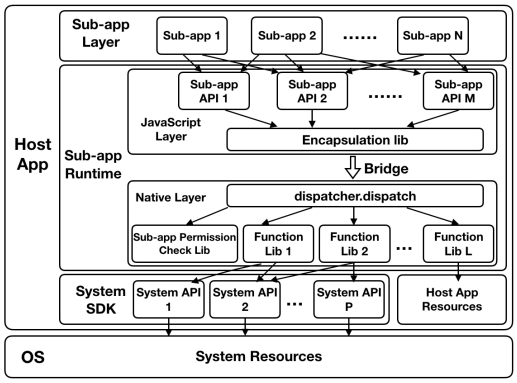


Figure 1: A typical app-in-app architecture

`dispatcher` object is then specified in a configuration file, and exposed to the script layer by React Native framework [55].

2.2 Security Model in Resource Management

• *Sub-app permission.* The app-in-app system uses the traditional permission label assignment model to protect sub-app APIs. In particular, sub-app requests a list of sub-app permissions, which govern the access to the sub-app APIs. If the sub-app APIs access sensitive system resources (i.e., protected by *dangerous* [76] permissions on Android or *entitlements* [17] on iOS[1]) or sensitive host app resources (non-system resources under host app's control, such as user profile, ID, gender managed by Wechat), a host app enforces sub-app permissions for calling the sub-app APIs. For example, Wechat defines a sub-app permission named `scope.record` to protect its sub-app API `wx.startRecord`, which is used to record audio using the phone's microphone. Note that host apps typically do not intend to protect less sensitive system resources, i.e., those protected only by Android *normal* permissions or not protected by OSes.

• *Isolation.* Each sub-app has its unique ID assigned by the host app. Sub-app cannot access resources of the OS, host app, and other native apps and sub-apps without going through an authorized channel provided by the host. Each sub-app has a unique data storage, managed by the host app.

• *Sub-app vetting.* Like traditional app stores, all sub-app stores in our study also require a sub-app to be reviewed before being published. The sub-app vetting process consists of risk assessment, along with additional criteria, to determine whether a sub-app violates user privacy or data security requirements. Examples of the security requirements include any sub-app shall not request or induce users to enter the host app's user ID and passwords [30] and user data collected via a sub-app may not be sold, transferred, traded, or disclosed [51].

2.3 Comparison with Existing App Encapsulation Systems

Similar to the app-in-app system, other approaches can also "encapsulate" an application to run in the execution environment provided

[1]iOS *entitlements* which confer security permissions and capabilities [41], are strictly protected by Apple and must be explicitly declared and vetted [108].

Table 2: Comparison with existing app encapsulation systems (○: Not Supported; ◐: Partially Supported; ●: Fully Supported).

System Features	Android Plugin Framework	App Sanboxing	Browser Extension	PhoneGap App	App-in-app
Sub-app API ecosystem for system resource access	○	○	◐†	◐†	●
The host has rich functionalities apart from hosting	○	○	●	○	●
Mandatory sub-app destruction	○	○	○	○	●

†: supported by HTML5 API, which is standardized by W3C and thus much easier to manage than sub-app API.

by another app. A prominent example is Android *Plugin framework* (e.g., Parallel Space [23], DroidPlugin [15], and VirtualApp [26]), an app virtualization technology through which a "host" app can load other Android apps (called "plugin" apps) from their APK files to run in the host's own process [111, 112]. This is done through wrapping the system APIs for invoking plug-in apps, which share the host's UID and permissions during their executions [99, 111]. Also mobile *app sandboxing* (e.g., Boxify [47], and NJAS [50]) is another way to virtualize apps. The idea is to encapsulate untrusted apps in a restricted execution environment within the context of a trusted sandbox app, which provides an emulation layer to abstract the underlying OS and interposes all interactions (e.g., system calls, binder IPC) between the untrusted app and the OS [47]. Other app-encapsulation solutions include *Browser extension*[2] (e.g., AdBlock [8], EditThisCookie [16]), where a small program customizes a web browser with the support of HTML5 (for controlled system-resource access) and extension APIs (for browser-resource access) [13], and *PhoneGap apps (or HTML5 based mobile apps)*, where a middleware framework (e.g., PhoneGap [24], Cordova [11], Ionic [19]) is used to allow the developer to create mobile apps using web technologies [31, 86].

Table 2 compares the above solutions with the app-in-app system, in terms of their security implications. As we can see here, unlike the app virtualization based Plugin framework and app sandboxing, the app-in-app platform builds up its own API ecosystem (i.e., sub-app APIs) for sub-apps to access system resources, which entails non-trivial efforts to define and maintain its security model, such as the sub-app API permission policies. Such policies are standardized for HTML5 APIs used by browser extensions [103], but are ad-hoc, opaque and often inadequate for the sub-app APIs that are much more powerful and access more sophisticated system resources than HTML5, thereby opening new attack avenues (see Section 3.1 and Table 7 in Appendix). Furthermore, compared with other solutions dedicated to application encapsulation, the app-in-app host has its own rich functionalities (e.g., mobile wallets, messaging, see Table 1), which are often shared with the sub-apps. For example, e-commerce sub-apps (e.g., Amazon, Pinduoduo) leverage the host's built-in wallet feature for convenient payment process (see Section 3.2). In our study, we found that such interactions (between the sub-app and the host app) actually exposes a new attack surface (Section 3.2). Also, the upper limit on the number of sub-apps that can run concurrently on an app-in-app platform is public and fixed (8 in Chrome, 5 in Wechat, 4 in Alipay): whenever the limit is reached, the host has to terminate a sub-app in order to launch a new one. This feature, which does not exist in other systems, can be abused for deception attack, as discovered in our research (Section 3.3).

[2]Browser extension is different from the deprecated *Browser Plugin* architecture [14]. *Browser Plugin* is generally not considered as an encapsulation system since the plugins are independent executables and their access to system resources are not encapsulated by the browser.

2.4 Adversary Model

We assume the host app is benign, which aims to provide a secure environment to run sub-apps. On the victim's Android or iOS devices where at least one app-in-app system is used, we consider an adversary with his malicious app[3] or sub-app installed. Note that, none of our attacks requires the victim to install both. In Section 3, we show that such malware can successfully pass the vetting of popular sub-app stores, Google Play, third-party app stores, etc. The malware sample does not have system privilege and in some of our attacks may need to ask for a set of common permissions from the users (e.g., Android's `READ_EXTERNAL_STORAGE` and `WRITE_EXTERNAL_STORAGE`) depending on specific attacks. Such permissions are extensively requested by popular apps such as Facebook, Gmail, and Pinterest. Therefore, we believe that claiming them by the malicious app will not arouse obvious suspicion.

3 DESIGN CHALLENGES AND PITFALLS

In this section, we report our security analysis on resource management in app-in-app systems. Our research shows that all app-in-app systems we studied, including Wechat, Alipay, TikTok, Chrome, Safari, etc., are subject to various APINA flaws, leading to serious exposure of system resources and new deception attacks. More specifically, the security analysis has been systematically performed by assessing two general categories of resources, system resources accessible through system APIs and user interfaces. For each resource, we looked at the security policies that should be in place, the mechanism that enforces the policies and the user interactions with security implications.

3.1 System Resource Exposure

Sub-apps access system resources through sub-app APIs (Section 2.1). However, for the host, as a third-party app, constructing sound sub-app API permission policies that govern system resources entails challenges that are unexpected before. As a result, sensitive system resources are often unwittingly exposed to unauthorized sub-apps.

Escaped sub-app API. Sub-app APIs are protected by sub-app permissions, which are supposed to be consistent with the permissions required by the OS when using the relevant system APIs they wrap. For example, since the OS requires permissions to use its

[3]In this paper, we use the term *app* and *native app* interchangeably. Also, although the host app is a native app itself, the term *app* or *native app* in this paper always refers to an app that is not a host app for ease of presentation.

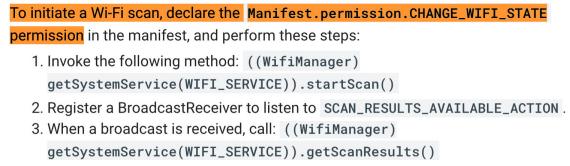


Figure 2: Misleading Android documentation

audio recording APIs (`RECORD_AUDIO` [72] permission on Android and `Microphone Usage` [42] entitlement on iOS), the sub-app API to record audio (e.g., `wx.startRecord` in Wechat) should also be protected by a sub-app permission (e.g., by sub-app permission `scope.record` in Wechat). However, in our study, we found the inconsistency in permission requirement between sub-app API and system API, indicating that the host fails to create the correct permission policies that govern sub-app API for system resource access. As a result, certain sub-app APIs, that wrap permission-protected system APIs, are completely open to any sub-apps without requiring a sub-app permission, exposing sensitive system resources. We name such unprotected sub-app API *escaped sub-app API*, and the problem *System Resource Exposure*.

An example discovered in our research is the unprotected Wechat sub-app API `wx.getWifiList` that performs `WiFi scan`, a capability protected by the underlying OSes against user location leak, e.g., through an *entitlement* `Hotspot Helper` on iOS and *dangerous* level `location` permissions (`ACCESS_COARSE_LOCATION` and `ACCESS_FINE_LOCATION`) on Android. This exposes the `WiFi scan` capability to sub-apps, leaking out user location to them. Note that a high-profile host app (e.g., Wechat) is typically granted the `location` permission and many other permissions due to its rich functionalities (see Table 9 in Appendix). As a result, through the delegation, the host makes to the sub-app, escaped sub-app API effectively opens a door for the adversary to gain unauthorized access to system resources. In addition to `wx.getWifiList`, we found many other escaped APIs in popular app-in-app systems, affecting diverse sensitive resources across iOS and Android (Section 4.3).

Looking into the possible root causes, we found that it is very challenging for an app-in-app system to soundly define sub-app API permission policies that comply with the OSes in governing system resources, due to the incomplete knowledge about permission checks that happen at related system APIs. Specifically, official API-permission maps for both Android and iOS have never been made public. Although several prior attempts [35, 45, 46, 56] have been made to automatically map Android APIs to their required permissions, we found that the prior results are incomplete (Section 4.3) and highly likely to introduce *escaped sub-app APIs* if they are used to design a sub-app permission system. Consider the above example of `WiFi scan`: recent results [35, 46] (on Android API level 25) report that a related Android API `WifiManager.getScanResults` requires just a *normal* permission `ACCESS_WIFI_STATE`, but fail to mention that *dangerous* permission, `location`, is also required (required since Android API level 23 [80]). Further, Section 4.3 reports more *escaped sub-app APIs* possibly caused by the incomplete API-permission maps.

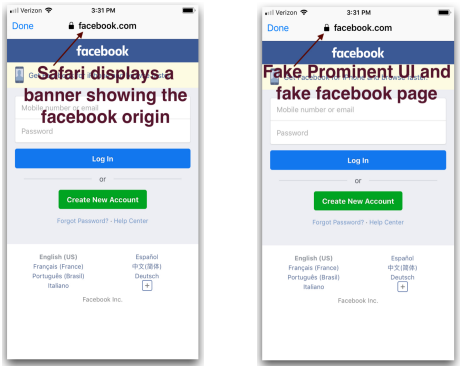
Such API-permission mapping information has also not been well recorded by the developer documentation. We found that such documentation can be misleading or confusing. Again let us use `Wi-Fi scan` as an example: as shown in Figure 2, the Android documentation [75] may introduce the misconception that declaring `Manifest.permission.CHANGE_WIFI_STATE` is sufficient for `Wi-Fi scan`. Similar misguidance also appears on the Android documentation for `Bluetooth scan`, etc., and has likely caused other *escaped sub-app APIs* we found (Section 4.3).

Another observation of our study is that, since sub-app API is designed to be cross-platform (Section 2.1), the effort to build sub-app API permission policies that comply with all OSes has been complicated by the inconsistency between Android and iOS in resource protection. Consider `Bluetooth scan` as an example: the Android API `BluetoothLeScanner.startScan` requires the *dangerous* `location` permission, while iOS does not. Such a discrepancy likely causes confusion in app-in-app design which is typically built for both OSes with uniform security policies. When this happens, we found that the app-in-app designer tends to leave the related sub-app API unprotected. For example, the sub-app APIs for `Bluetooth scan` are not protected in multiple host apps (e.g., `wx.getBluetoothDevices` in Wechat). Actually, the difference in policy-level protection could result from an additional security guard put in place by the OS, which is unaware to the app-in-app designer. For example, combining together `Signal Strength` and `Device ID` of nearby Bluetooth devices reveals geo-locations [37]. On Android the pair is returned together by the above `Bluetooth scan` API `BluetoothLeScanner.startScan`, which thus needs a permission check; on iOS, however, the pair has been partially obfuscated (`Device ID` is randomized), and is thus unprotected. We suspect that the inconsistency, as observed on capabilities/resources `Wi-Fi scan`, `Wi-Fi connect`, `iBeacon`, etc. is likely responsible for other *escaped sub-app APIs* we discovered (see Section 4.3).

Discussion. Android 6.0 (API level 23) started to use a runtime permission model: a user grants permissions at the app's runtime, when permissions are requested (e.g., when the app launches or when the user accesses a specific feature) [73]. In this scenario, when an *escaped sub-app API* tries to silently access a resource, the host may have not been granted a permission to access that resource, thus a permission will be explicitly required. However, the OS' permission granting window will show to the user that it is the host app, who requests the permission, not the sub-app. This misleading request may lead to the wrongly inherited trust associated with the hosts, and make users grant the permission inappropriately. This is also the case on iOS when a permission window pops up to confirm the host's access to a resource.

Furthermore, Table 9 in Appendix lists all Android *dangerous* permissions and iOS *entitlements* requested by the vulnerable hosts right after launching. It shows that the hosts proactively request necessary permissions/*entitlements* even before accessing any specific features, due to usability consideration [92]. Those capabilities of the hosts will always be stealthily delegated to the sub-app via escaped sub-app APIs, which present serious risks.

Attacks and vendor acknowledgements. We implemented an attack sub-app which ran in Wechat v6.7.3 to steal user's location on *both Android and iOS*. The attack sub-app utilized Wechat *escaped sub-app API* `wx.getWifiList` to perform `Wi-Fi scan` and sniffer the surrounding Wi-Fi access points. The scan results included sensitive information, i.e., `BSSID`, `Signal Strength` and `Device ID` of nearby Wi-Fi access points, which enabled to infer the geolocation



(a) Real Prominent UI (b) Fake Prominent UI

Figure 3: Safari's Prominent UI confusion on iOS

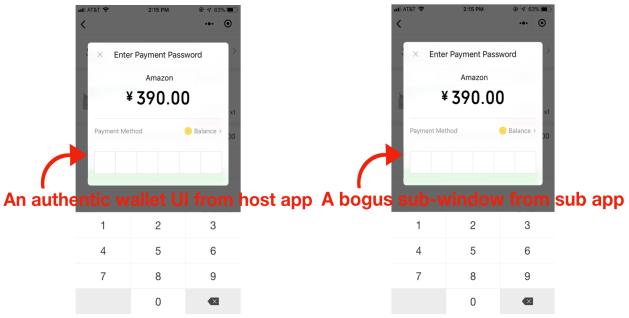
of a user [37]. This attack sub-app successfully passed the vetting of Wechat sub-app store and ran on both iOS and Android versions of Wechat. Note that, our attack sub-app has less than 200 lines of code, which indicates that it is cost-effective for the adversary to develop APINA flaws malware in the wild. Besides, we successfully built attack sub-apps in other host apps, e.g., Alipay, DingTalk, etc., and reported the problems to all affected host-app vendors. They all acknowledged the problem; in particular, Wechat and Alipay awarded us through bug bounty programs for this flaw.

3.2 Sub-window Deception

Another unique app-in-app feature is the functionality-level interactions between the host and sub-app. For this need, the host does not fully isolate itself from sub-apps, which unwittingly introduces new attack surfaces and high spoofing risks against host-app UI.

Browsers' Prominent UI confusion. To bring a native app-like experience, Safari on iOS and Chrome on Android open each sub-app (i.e., Web App, Progressive Web App, or PWA) in an entire dedicated window separate from the browser window just like a native app [44, 78] – without any browser UI such as the address bar. Web App is programmed with JavaScript and HTML, but must meet certain standards [68]. With user confirmation, the browser can install a Web App to the phone's home screen (through an `APK` [69] package created by Chrome, or a `Web Clip` [44] created by Safari). Such Web App takes the full screen if "standalone" or "fullscreen" is specified in its `manifest` file. Each Web App is clearly associated with specific Web domain when installed. When the user navigates to out-of-scope URL (i.e., of different origin) in a Web App, the browser displays a Prominent UI – a banner at the top of the screen (Figure 3a) showing the origin and secure connection status – due to the lack of address bar. According to Web App documentation [67], users rely on such Prominent UI to be aware when they navigate out of scope, an important security notice.

Based on a study [57], mobile phishing attacks tend to happen when users become accustomed to familiar, repeated contexts. More specifically, if users frequently encounter legitimate context, they will become conditioned to reflexively respond to it. Here, the current design of Web App conditions users to see the Prominent UI in out-of-scope navigation, which we found poses a spoofable context.



(a) Authentic wallet UI (b) Bogus wallet UI

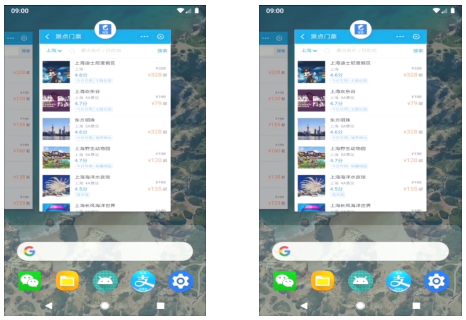
Figure 4: Mobile wallet UI confusion

Specifically, Web App is designed to embrace out-of-scope browsing [104] and let users freely navigate to other Web domains for services, such as login through Facebook/Google (a.k.a., SSO [107]), payment through PayPal, etc. The problem is that, when a victim user navigates out of scope, e.g., to Facebook login page, a malicious Web App can actually navigate to an in-scope phishing page imitating Facebook, showing a bogus Prominent UI (as if shown by the browser) to misleadingly inform the victim that she is on a real Facebook domain (Figure 3b). In this way, the malicious Web App can collect victim users' secrets such as Facebook passwords.

We call this problem *Sub-window Deception* – *sub-window* means part of a window. Prior phishing attacks in browsers explored how a webpage can go full screen and spoof the address bar [1, 2]; as another instance of mobile browser UI attacks, our attack complements the prior understanding, by exploiting Prominent UI – the counterpart of address bar in the context of native app-like Web App. Note that, address bar differs from Prominent UI in design: based on the Web specification [106], browsers *"should provide a means of exiting fullscreen that always works and advertise this to the user"* [106], so users can see the *real* address bar when they want (although browsers may not implement this protection properly [89]); Web App users, however, lack practical mechanisms to verify the authenticity of Prominent UI today. This attack applies to general Web App on both iOS and Android: we reported it to W3C[4], Apple, Google, Firefox, and Opera, which all acknowledged the importance of the problem.

Mobile Wallet UI confusion. Wechat and Alipay both feature mobile wallets, which can be leveraged by e-commerce sub-apps to provide users a convenient payment process. Amazon sub-app, for example, after a user clicks its "checkout with Wechat wallet" button, will invoke a sub-app API (e.g., `wx.requestPayment` in Wechat) to trigger the host's wallet; the host then reclaims a central portion of the screen from the sub-app, to show its wallet UI (highlighed in Figure 4a), for the user to enter her wallet password and finish the payment. Such a context can condition users to reflexively provide wallet passwords at checkout, and therefore presents a practical spoofing target. Specifically, when a victim user clicks the "check out" button in a malicious e-commerce sub-app, it can show a bogus wallet UI (Figure 4b) to collect the user's wallet password, instead of invoking the real wallet.

[4]We do not provide the link to our report for anonymization purpose



(a) Real sub-app task (b) Bogus sub-app task

Figure 5: Dedicated task for every sub-app in *Recents screen*

Up to our knowledge, state-of-the-art mobile anti-phishing techniques *cannot* identify our attacks: for example, [49, 58, 96] mainly rely on identifying the app of the whole foreground window, so the user would not expose secrets to the unintended app; this cannot identify a sub-window at the sub-app level – whether it belongs to a sub-app or the host app.

Our study shows that a bogus sub-window presents a practical attack vector in app-in-app, since a window portion (sub-window) can be used by both the sub- or host- app in accustomed contexts without differentiation through clear identification. From a broader sense, we believe the spoofing risk is also high if the OS shows its pop-up atop an app (so malicious app can show a bogus one when the OS does not); however, in sharp contrast, such pop-up is not *sub-window* (but *whole* window), and known anti-phishing techniques (e.g., [49]) to identify (the owner of) the whole window can defeat it.

Attacks and vendor acknowledgements. We implemented PoC attacks in Safari on iOS, Chrome on Android, and Wechat on both OSes: in Safari and Chrome, our malicious Web App impersonates Prominent UI to spoof Facebook login page; in Wechat, our malicious sub-app mimics the wallet UI to steal wallet password (see video demo online [12]). Our malicious sub-app passed the vetting of Wechat's sub-app store. Note that Web App does not have app stores and can be installed after visiting its URL. In addition, we reported the problems to all affected host vendors who all acknowledged that the threats were practical and severe. Wechat and Alipay awarded us through their bug bounty programs.

3.3 Sub-app Lifecycle Hijacking

As app vendors follow the app-in-app trend by releasing sub-apps, it is unclear whether their users remain equally protected as before. Unfortunately, our study shows that even highly credible host apps cannot fully eliminate new attack surfaces incurred by app-in-app.

***Recents screen* takeover.** The *Recents screen* (a.k.a., recent task list, or recent apps) [64] is a system-owned UI of Android to list recently accessed *task* (Figure 5a). In particular, each launched native app will have a task in *Recents screen*; similarly, a host app creates a separate task for every launched sub-app, which appears in *Recents screen*. Limited by the OS, each host app can only create a fixed number of tasks in *Recents screen*, pre-specified as `activities` in its manifest file. In our research, we found that host apps all specify a small magic number (e.g., 8 in Chrome, 5 in Wechat, 4 in Alipay, 3 in JinRiTouTiao) for the task limit.

When an additional sub-app will be opened after the task limit is reached, the host app silently executes a mandatory recycling process: closing the first opened sub-app to recycle its task for launching the new one. The recycling process is transparent to users for the possible purpose of not interrupting the user experience. Such a mandatory sub-app recycling mechanism introduces a new risk: once the recycling is traceable by a malicious app, i.e., the adversary knows which sub-app has been recycled and at what time, it can stealthily insert a phishing task into the *Recents screen* to imitate the silently recycled sub-app. For this purpose, the adversary needs to acquire the information about a host's task recycling, which turns out to be feasible, due to the presence of various information-leak avenues that can be hard to eliminate. In our study we found a side channel that enables recycling tracking in multiple host apps. In particular, possibly due to the rich functionalities of sub-apps, which require resource files (e.g., user data, logs, icon and image texture), the host app creates a folder in Android *external storage* [18] to cache sub-app resources. Such external storage can be monitored by an Android native app to track which sub-app is launched and at what time (by monitoring sub-app icon which will be downloaded at launch, see PoC attack below). Thereby, once the number of launched sub-apps reaches the host's task limit, the sub-app to recycle can be inferred. Hence, a malicious native app can insert a bogus task in *Recents screen* to mimic the recycled sub-app; at this point, the bogus task is the only one in *Recents screen* that represents the target sub-app that's spoofed (Figure 5b).

This attack is a kind of mobile *task hijacking* attacks [52, 97]. Regarding its root cause, the mandatory sub-app recycling opens a new attack surface for app-in-app systems, since it makes it possible for malicious apps to track a sub-app's lifecycle termination. Note that, the attack surface becomes exploitable in the presence of side channels, e.g., external storage, which itself is a relatively trivial bug. Other side channels could also be used for the attack once they arise (see discussion below). Eliminating this attack surface may not be trivial in practice, since the maximum task number of a host in `Recents screen` is not dynamically scalable (statically specified in manifest) limited by the OS. Although the very recent Android supports dynamic task, no host app we studied (Table 3) adopted it, possibly for the purpose of supporting older Android versions, given the notorious Android version fragmentation [74].

Attacks and vendor acknowledgements. We implemented PoC attacks on Alipay and Wechat, which cache sub-app resources in the external storage and enable recycling tracking. Specifically, Alipay stores the caches at `/sdcard/alipay/multimedia/`. When a sub-app, such as Amazon, is launched, its icon will be downloaded to it if not already there. Our malicious Android app with a common permission `WRITE_EXTERNAL_STORAGE` can remove cached icons periodically and monitor when they reappear, to know what specific sub-app is launched and at what time. In our attack of Alipay, once the malicious app infers that five sub-apps have been launched (submit-app task limit of Alipay is 4), it will insert a bogus task in *Recents screen* to mimic the first sub-app which is recycled. Once the victim user enters the spoofing task through *Recents screen*, it can lure the user to leak secrets such as passwords. Note that

our malware successfully passed the vetting of popular app stores, including Google Play. We reported the problem (demo online [12]) to Wechat and Alipay which both acknowledged the problem and awarded us through bug bounties.

Discussion on other side channels. As mentioned earlier, other side channels, once found, can also be used for impersonating the recycled sub-app. Prior research [113] shows that the operations of Android apps can be easily tracked through the Android process file system (*procfs*). Although Android has been continuously beefing up its privacy protection, until Android 9, still some global statistics under procfs are exposed. Particularly, we found that the TCP connection status, under `/proc/net/tcp`, can be accessed by any app without permission on Android 9 and below. From the file, we can see the IP addresses of all TCP connections on an Android phone, which are labeled by randomly assigned User IDs. Interestingly, we found that an invoked sub-app makes a sequence of TCP connections and the endpoints of the connections (domains) can uniquely fingerprint its initiation: e.g., Amazon sub-app connects to 4 unique endpoints with 8 consecutive HTTPS requests in the first few seconds after it is launched. In our research, we sampled 8 high-profile sub-apps in Wechat and identified the sequences of their communication endpoints as observed when they are triggered (see Table 8 in Appendix). Through this side channel, a malicious native app is able to infer the launch of a specific sub-app, so it can wait until it is terminated for task recycling (e.g., after 5 other sub-apps are found to be started through the procfs, when the magic number is 5) to create a task to impersonate the sub-app. Note that, the malicious app can resolve the endpoint domains from IP addresses using Android API `Inetaddress.getCanonicalHostName` [81] without requiring any permission, which was implemented and confirmed in our study.

Until April 2020, 84% of the Android devices are still running Android 9 or earlier versions [33], and therefore are vulnerable to our attack. On Android 10, which was released in September 2019, the *procfs* used in our attack has been closed by SEAndroid [70, 83] and thus the attack is ineffective. Seeking a more powerful attack avenue is left to the future research.

4 APINA FLAWS IN THE WILD

In this section, we report a measurement study that reveals the scope and magnitude of APINA flaws in the wild. The study is made possible by an automatic analysis tool we built, called *Apinat* (short for App-in-app Threat Scanner), which helped us identify app-in-app hosts that are susceptible to APINA flaws.

4.1 Identifying System Resource Exposure

Design. *Apinat* aims to find System Resource Exposure flaws by identifying *escaped sub-app API*. The idea is to find sub-app API that is unprotected by sub-app permission, while its corresponding system API is permission-protected. A naive idea is to call each sub-app API and find out whether the OS checks the permissions of the host app, e.g., by looking at permission pop-ups or system logs. This does not work because the permission asked by the OS can be related to the host app's own rich functionalities, not the sub-app API. To systematically find *escaped sub-app API*, our methodology tracks the system API invocations triggered by a sub-app API via a dynamic analysis (**Step S1**, **S2**), and then utilizes system API-permission maps to derive required permissions of the sub-app API (**Step S3**), which will be compared to the corresponding sub-app permission. If a sub-app API is not protected by any sub-app permission while its corresponding system APIs are guarded by the Android *dangerous* permissions[5] or iOS entitlements, an escaped sub-app API is found (**Step S4**). Following we describe how this design works and how it was implemented in our research on Android. Since sub-app APIs are cross-platform, we were able to extend all the findings made by our technique from Android to iOS, by simply validating these findings on the Apple platform.

Step S1: generating test cases. To conduct a dynamic analysis, we first need to generate test cases with valid arguments to invoke each sub-app API without triggering exceptions. For each API, we target that at least one invocation is successful. The argument types of JavaScript APIs are either *primitive* or *Object* [21]. To produce test cases for *primitive* arguments, we leveraged Mozilla's funfuzz [22], a JavaScript toolkit. For *Object* arguments, we recursively generated test cases leveraging funfuzz because an *Object* includes key/value pairs: its key is *primitive*, and the value is either *primitive* or *Object*. All test cases were then used by a sub-app we built to invoke sub-app APIs. If all test cases for an API failed, we manually inspected the API, to address the issues such as formatted inputs (e.g., UUID), stateful calls (an API needs to be invoked before another), etc.

Step S2: tracking system API. As mentioned in Section 2.1, the execution of a sub-app API enters the native layer of the host through a `dispatcher.dispatch` function, and then its control flow reaches system APIs (Figure 1). Therefore, Step S2 first identifies the `dispatcher.dispatch` function in the host app's native layer, then tracks its call graph to find the system APIs triggered by the sub-app API call, as elaborated below.

(1) Find the dispatcher.dispatch function in the native layer. As mentioned earlier, Webview and React Native are bridge techniques to expose `dispatcher.dispatch` to JavaScript layer. Note that though the object alias (*dispatcher*) and function name (*dispatch*) vary across host apps, they can be obtained by inspecting function invocations in *Encapsulation lib* (Figure 1), which is a preprocessing step in our study. Note that, the *Encapsulation lib* usually can be found in JavaScript files packaged in the host app (e.g., under `assets/` folder). So all we need is to identify the *dispatcher* object in the native layer based on the obtained alias.

To this end, **S2** implements an Xposed [7] (a framework to instrument Android app execution) module to hook Webview API `addJavascriptInterface(obj, alias)` (that exposes Java object to JavaScript through an alias) and inspect its arguments at runtime: if the "alias" argument's value matches the *dispatcher* alias discovered, the "obj" is the *dispatcher* object that we are looking for. The analysis of React Native bridge is slightly different: we hook its API `createNativeModules` to get its Java objects exposed to JavaScript; our Xposed module then calls each exposed object's `getName` method to get its alias – if this alias matches the *dispatcher* alias discovered, the object is what we are looking for.

[5]Host apps are typically third-party apps that cannot call privileged system APIs protected by "signature" or "signatureOrSystem" level permissions [65], or they typically do not delegate the access to such APIs through sub-app APIs.

(2) Track execution of the dispatcher.dispatch function. Our tool then tracks the call graph of the *dispatch* function to find the called system APIs – which are called to fulfill the sub-app API call. Specifically, our Xposed module hooks *dispatch* function and Android API, and inspects their respective call stack traces. However, we found that *dispatch*'s thread sometimes does not call Android APIs directly: it triggers additional thread to call Android APIs, likely for not to block the *dispatch* thread. This introduces a challenge: Xposed outputs the above two stack traces by threads – one stack trace with Android API call and the other with *dispatch* function call – but cannot correlate two threads if one triggers the other. Therefore, we still cannot confirm that the Android APIs are invoked due to *dispatch*. Hence, we developed techniques to deal with the multi-threading challenge.

In our study, we found that `Handler`, a typical multi-threading mechanism of Android, is commonly used by *dispatch*'s thread to trigger a new thread. Thus, we implemented our own thread correlation technique based on the `Handler` mechanism in our Xposed module. More specifically, in the `Handler` mechanism, a triggering thread, that holds a `handler` instance, will submit a `runnable` to a message queue via calling the `handler.post(runnable)` method. Here, the `runnable` is an instance of Java Runnable [3], with code in its `run()` method. Then the `runnable` is taken out of the queue by another thread, which we name as the `handler thread`, and gets executed in the `handler thread`. Hence, as checked in our Xposed module, if the executed `runnable` in a `handler thread` is the same instance with one submitted by a triggering thread, then we correlate the two threads. Through such techniques, we found Android APIs are indeed called in such `handler thread` following the *dispatch* thread's triggering – this confirms the Android API call caused by the dispatcher.dispatch function.

Step S3: synthesizing API-permission mapping. As mentioned in Section 3.1, system API-permission mappings are incomplete, which can lead to *escaped sub-app APIs*. How to derive a complete mapping remains an open question in system security research and program analysis. Here, our idea is to supplement recent mappings [35, 46] leveraging documentation analysis: we automatically extract *explicitly* declared permissions for system APIs in the developer manuals, assuming that *explicit* specification at least reflects intended protection of the OS with rare faults. Note that certain portion of the documentation (Figure 2) can be misleading to human readers due to incomplete presentation, but we observed that an exhaustive, machine-based search on the whole corpus of mobile OSes (e.g., the entire documentation for all 2,497 SDK classes of Android API level 27) can help us derive a significantly more complete mapping than state-of-the-art mappings [35, 46] (see evaluation below). For instance, after a full scan of Android WifiManager manual [75], we were able to flag `ACCESS_FINE_LOCATION` as a required permission for multiple `Wi-Fi scan` related APIs (see Figure 8 in Appendix), which has never been included in state-of-the-art mappings [35, 46].

In our implementation, we developed a Web crawler to fetch all manuals from Android developer portal [62] and extracted the description for each public API. From the description, we extracted the declared permissions based on a few key observations: if the description of an API does not contain a *"permission"* keyword, it is very likely that the API does not *explicitly* require permissions; otherwise, its description usually includes a *"permission"* keyword and the exact permission string constants defined in Android class `android.Manifest.permission` [77], such as `ACCESS_FINE_LOCATION`. Based on such observation, we performed a light-weight string matching to find *explicit* permission requirement of Android APIs (we discuss the sophisticated, NLP-based direction in Section 5).

Step S3 also derives a sub-app API-permission map: given its relatively small size (i.e., 50 to 172 sub-app APIs in each host, compared to 35,847 public Android APIs found by our Web crawler), we extracted the map by searching "permission" related keywords in sub-app developer manual and manually confirmed the result.

Step S4: reporting flaws. S4 reports an *escaped sub-app API* if it is not protected by sub-app permission while its corresponding Android API(s) is protected by *dangerous* permission(s). That is, a *System Resource Exposure* flaw is found.

For iOS, however, a comprehensive detection is much more difficult than Android, especially because obtaining a system API-entitlement mapping is much more challenging. First, we are not aware of any work that has produced a mapping on iOS for us to start with. Second, we observed that iOS documentation is not as precise as Android in this regard, which describes required *entitlements* for iOS classes but not for individual system API. To still help us understand the impact of *escaped sub-app API* on iOS , we extended our findings of *escaped sub-app API* from Android to iOS: if its corresponding iOS API requires an entitlement, it is considered to affect iOS as well. This was done by manually searching corresponding iOS API for a given *escaped sub-app API*, and checking whether any entitlement is asked when the iOS API is invoked in an iOS app we built.

Evaluation. We evaluated the effectiveness and performance of our analysis tool (source code released [12]) on a Google Pixel 2 phone and iPhone 8 against 11 host apps. The evaluation was performed on three Android versions (8.1, 9, 10, corresponding to API level 27, 28, 29) and two iOS versions (12, 13.4).

• *Overall detection results. Apinat* found 39 *System Resource Exposure* flaws: all of them affect Android 8.1, 9 and 10; 13 of them affect iOS 12 and 13.4. We manually validated that all discovered flaws are true in two steps (1) confirming that the sub-app API requires no sub-app permission through a testing sub-app; (2) confirming that the corresponding system API(s) has permission protection through a testing native app on Android and iOS.

• *Evaluating individual steps.* Step S1 generated test cases for all 927 sub-app APIs under test in less than one hour. All sub-app APIs were successfully invoked at least once without exception on each Android version. In this experiment, we did not measure the host's code coverage because the host includes implementation of its own rich functionalities that are unrelated to sub-app API. Thus, it is not meaningful to compute the percentage of covered host app code.

Step S2 took less than 30 seconds for each sub-app API, to report its corresponding system APIs on each Android version. The number of system APIs (protected by dangerous permissions) hooked by S2 is 111, 129, 145 on the three Android versions respectively. To evaluate the correctness of our thread correlation, we created

a test app which triggered new threads leveraging the `Handler` mechanism. Our test showed that our tool correctly correlated the triggering and triggered threads on each Android version.

Step S3 produced the API-permission mapping, including 94 public APIs protected by dangerous permissions at API level 27, 103 APIs at API level 28, and 119 at API level 29. This step took our approach three hours to process the descriptions of 35,847, 42,302 and 44,323 public APIs on the three API versions respectively. To validate the correctness of the mapping, we constructed a test app to invoke all the APIs and verify the required permissions at runtime. For all three Android versions, our permission mapping of all except 6 APIs are correct. Specifically, the mapping of 92 APIs out of 94 is correct (97.9% precision) at the API level 27. The two false cases are unprotected APIs `hasPermission` and `requestPermission` in class `UsbManager` [66]. At API 28, the mapping shows a precision of 97.1%, with one new false case than what was found on API 27 (3 cases in total); API 29 shows a precision of 95%, with three new cases than API 28 (6 cases in total). Figure 6 and 7 in Appendix list all six false cases. Their descriptions are more complex and we envision that a natural language processing (NLP) based approach can further improve our precision (see discussion in Section 5).

To evaluate the coverage of our documentation analysis tool, we randomly selected the documentation of 153 classes, 641 APIs (out of 2,164 classes), and then manually reviewed and identified 52 API-permission mappings as the ground truth. Running on the aforementioned documentation, our analysis tool shows a precision of 100% and a recall of 100% to identify API-permission mapping from the documentation.

We also compared the completeness of our mapping with recent mappings [35, 46]. They collectively reported 50 public Android APIs that require *dangerous* permissions on API level 25 (we were not able to get results of newer Android from the authors); as a significant improvement, our tool was able to find 91 APIs (82% more) that require *dangerous* permissions on the same API version.

Discussion. *Apinat* may not find all *escaped sub-app API*s. In particular, Step S3 cannot derive a complete mapping if the documentation is incomplete [56] or too sophisticated to process; *Apinat* only looks for system resource exposure by public system APIs, and undocumented system APIs used by host apps may also lead to the flaw but remain undetected. Also, *Apinat* falls short in analyzing obfuscated host apps, e.g., that uses dynamic code loading, and implementations in C/C++. Hence, *Apinat* did not scan Web App APIs due to the extensive C/C++ implementation in browsers. In addition, incorrect documentation (which we assume is rare, and, indeed, our tool did not report any false *escaped sub-app API*) might lead to false alarms if not properly validated.

4.2 Finding UI Deception Flaws

Design. To find out the opportunities for *Sub-window Deception* (Section 3.2) in a host app, *Apinat* looks for the app's sensitive sub-windows, which can be the targets of the attack. As discovered in the wallet UI confusion (Figure 4b), the host's sub-windows that are likely to show up in the presence of a sub-app, tends to be those that can be triggered through sub-app APIs. If such a sub-window also contains sensitive information, our approach then reports it as a potential target for *Sub-window Deception*.

Implementation. To detect such sub-windows, *Apinat* monitors the change of the host UI in response to each sub-app API call to identify the sub-window triggered by the API call, and then determines whether it carries sensitive content. Specifically, our approach first constructs a testing sub-app with an empty UI. Then, it runs the sub-app API test cases in Section 4.1 to call each sub-app API through the testing sub-app and further inspects the UI screenshots before and after the call for changes. This is done through a pixel-to-pixel comparison of two screenshots. To capture the screenshots, we developed a screen projection service on Android using the `Media Projection` [63] API. Also, our screenshot comparison was implemented using the OpenCV `cv2.absdiff` API [85]. For each sub-window identified, our approach continues to find those associated with sensitive information. For this purpose, it extracts strings from sub-window screenshots using the online OCR service [34]. From the strings, *Apinat* further utilizes a list of 121 sensitive data keywords and the keyword pair list from recent research [91] to recognize those carrying sensitive content. Note that, like a mobile OS, an app-in-app system also provides a set of sub-app APIs for sub-apps to dynamically generate UI components, such as `wx.createCanvasContext.draw()` in Wechat. To control false positives, *Apinat* omits such sub-app APIs, based on the API category in the sub-app API documentation. As a result, *Apinat* successfully detected 6 *Sub-window Deception* flaws (Section 4.3). For iOS, we got the same result since sub-app APIs are across-platform: the one triggers a sub-window on one OS also does that on the other (see Section 2).

Evaluation. *Apinat* detected 5 sub-app APIs associated with the UI deception flaw; we manually confirmed that all of them trigger sub-windows with sensitive information. Also, all of them affect both Android (version 8.1, 9, 10) and iOS (version 12, 13.4). Performance-wise, the screenshot comparison and OCR analysis together took less than 5 seconds on a desktop with Intel i5 2.7 GHz CPU and 8 GB memory.

Discussion. Soundly detecting whether a host app is susceptible to *Sub-app Lifecycle Hijacking* (Section 3.3) is difficult, especially because it entails detection of side-channel information leakage, which can happen in many ways and is hard to capture in general. To measure a lower bound of the pervasiveness of the problem, we checked whether the hosts in our study (Table 1) had the same information leakage and lifecycle design as reported in Section 3.3. As a result, 3 host apps on Android (Wechat, Alipay, DingTalk) are susceptible, which were all manually confirmed to be exploitable.

4.3 Measurement of Impact

Flaw landscape. With the help of *Apinat*, we analyzed 11 highly popular host apps (Table 1) on Android and iOS, and found they are all vulnerable to APINA flaws (Table 3). Altogether we found 52 APINA flaws, including 39 *System Resource Exposure*, 10 *Sub-window Deception*, and 3 hosts vulnerable to *Sub-app Lifecycle Hijacking*.

For *System Resource Exposure*, in total, we uncover 39 escaped sub-app APIs associated with 5 categories of exposed system resources, 4 Android dangerous permissions and 6 iOS entitlements. Table 5 shows the flaw number for each category of resources (the second column), and illustrates 8 *escaped sub-app API*s, with corresponding Android permission(s) and iOS entitlement(s) ignored to enforce

Table 3: Eleven vulnerable host apps on Android and iOS (A: Alipay; B: Baidu; C: Chrome; D: DingTalk; F: Firefox; J:JinRiTouTiao O: Opera; Q: QQ; S: Safari; T: TikTok W: Wechat). "N/A" denotes the flaw does not apply to the OS.

Type of APINA Flaw	iOS	Android
System Resource Exposure	A, D, J, Q, T, W	A, D, J, Q, T, W
Sub-window Deception	A, B, D, W, S, Q	A, B, C, D, F, W, O, Q
Sub-app Lifecycle Hijacking	N/A	A, W, Q

on them . All 39 *escaped sub-app API*s are listed in our released dataset [12]. Also, Table 6 illustrates 6 *Sub-window Deception* in non-browser hosts; additionally, all four browsers we studied (Chrome, Safari, Firefox, Opera) are susceptible to the same Prominent UI attack.

Resource comparison between Android and iOS. Although finding the full API-entitlement mapping for iOS is hard, we observed a few differences between Android and iOS in system resource protection.

• *iOS is less affected than Android.* Out of 39 *escaped sub-app API*s, 13 are related to iOS, which appears to be less affected and harder to be exploited. There are two reasons behind the observation, as discovered in our study. First, certain iOS APIs return less information than its Android counterparts (i.e., those related to Bluetooth, see Section 3.1), and thus is less susceptible to privacy risk (location leakage). Hence, all 27 Bluetooth related *escaped sub-app API*s do not affect iOS (see Table 5). Second, WiFi related *escaped sub-app API*s (see Table 5) are much harder to exploit on iOS, since Apple has extra protection besides entitlement. Specifically, right before the programmatic WiFi operation, an iOS app must obtain the user's explicit consent: the app must instruct the user to manually open the WiFi setting page in the Settings app and then return to the iOS app [43] – an indicator of user intention for the app to operate WiFi.

• *iOS entitlement fails to communicate risks to the user.* On Android, when the permission `ACCESS_FINE_LOCATION` is used for protecting the operation `Wi-Fi scan`, it indicates a risk that the WiFi operation may leak location, a highly private data. On iOS, however, the entitlement for `Wi-Fi scan`, i.e., `Hotspot Helper`, fails to communicate such a risk. Actually, to get this entitlement for an iOS app, app developers must file a special written request, by justifying that the app does not have malicious intention. Apple reviews the request internally, and possibly assesses the risks of location leakage behind the scene. However, the name of this entitlement and its documentation we could find does not show that `Wi-Fi scan` needs `location` protection on iOS. This may have been one of the factors leading to *escaped sub-app API* `wx.getWifiList` (see Table 5), which performs `Wi-Fi scan` without protection.

Attack consequences. The system resources exposed by *escaped sub-app API* and the risks can be identified from the API names and the permissions they violate (see Table 5). For example, unauthorized sub-apps can use *escaped sub-app API* `wx.getRecorderManager` to record audio (violating Android's `RECORD_AUDIO` permission). We summarize the attack consequences in Table 4. Further, *Sub-window Deception* leads to phishing on popular Web services (e.g., Facebook), mobile wallets, and sensitive host UIs. For example, Wechat has a UI triggered by sub-app API (`wx.addPhoneContact`) for users to enter contact information (Table 6); attacking this UI may leak contacts, which are highly private. Last, *Sub-app Lifecycle Hijacking* affects three hosts (Table 3), and the risks come from imitating sub-apps supported by these hosts, such as sub-apps for Amazon, health, banking, Airbnb, etc. (see Table 4).

Table 4: Examples of APINA attack consequences

Flaw Type	Consequences (what the adversary is able to do)
System Resource Exposure	record audio, operate camera, infer victims' geo-location (thus track users), connect victim's device to arbitrary Wi-Fi (thus manipulate network traffic, perform man-in-the-middle (MITM), or phishing), and steal sensitive QR code, etc.
Sub-window Deception	steal account credentials on Facebook, Google, Amazon, etc., steal mobile wallet passwords, contacts, etc.
Sub-app Lifecycle Hijacking	steal money (payments in shopping sub-app), steal personal secrets (e.g., answers to security questions, health records, etc), steal business documents (from a document processing and management sub-app published by Microsoft), steal account credentials of Amazon, ICBC, CCB, HSBC (1st, 2nd, 7th largest banks in the world), Airbnb, Weibo, Starbucks, McDonald's, etc.

Cases Studies. We discuss a few other problems we observed in app-in-app ecosystem.

• *Keeping up with the OS evolution.* Sub-app API permission policy should keep up as the OS always evolves: newer OS version often introduces more strict protection against sensitive resources. For example, since Android 10, an app that accesses location in the background needs an additional *dangerous* permission `ACCESS_BACKGROUND_LOCATION`, in addition to the regular `location` permission [82]. This protects user location against illicit tracking. However, sub-app API for accessing location in hosts TikTok and Alipay did not keep up with such a new protection, thus leaving their users at risk.

As another example, for the WiFi and Bluetooth scan that can leak location, Android 9 and 10 require `ACCESS_FINE_LOCATION` permission, while Android 8.1 is less restrictive and `ACCESS_COARSE_LOCATION` is sufficient (see Table 5). Note that Android uses the first permission to control access to precise location (e.g., GPS data), and uses the second one for coarse-grained positioning (with an accuracy approximately at the level of a city block [10]). Interestingly, all hosts we studied only define a general `location` sub-app permission to govern location API (e.g., `scope.userLocation` in Wechat), but do not differentiate the two levels of accuracy.

• *Too many APIs for one permission.* APIs of apparently different functionalities should never be governed by the same permission. However, in the host TikTok (an extremely popular short-video sharing app, with more than 500 million downloads on Google Play), three sub-app APIs `tt.scanCode`, `tt.chooseImage`, and `tt.chooseVideo` (for scanning QR code, accessing photos and videos) are governed by the same permission `scope.camera`, and thus inadequately protected. For example, a malicious sub-app that is granted permission by the user to scan QR code, also, behind the scene, silently gets access to API `tt.chooseImage` that accesses user photos, posing a serious privacy risk.

5 LESSONS AND MITIGATION STRATEGY

Lessons learned. The most important lesson learnt from our research is the caution one should take when building a third-party

Table 5: Summary of *System Resources Exposure* flaws (N/A means the flaw does not affect the OS)

System Resource Exposed	# of Flaws	Example of Escaped Sub-app API	Dangerous Permission	Android API	Entitlement	iOS Class
Microphone	1	Wechat: wx.getRecorderManager	RECORD_AUDIO	media.MediaRecorder CaptureRequest	Microphone Usage	AVAudioRecorder
Camera	3	Alipay: my.scan	CAMERA	CameraManager.open-CameraCaptureRequest	Camera Usage	AVCaptureSession
Wi-Fi	4	WeChat:wx.getWifiList	ACCESS_COARSE_LOCATION or ACCESS_FINE_LOCATION ‡	WifiManager. getScanResults	Hotspot Helper †	NEHotspotHelper
		JinRiTouTiao/TikTok: tt.getConnectedWifi	ACCESS_COARSE_LOCATION or ACCESS_FINE_LOCATION ‡	WifiManager. getConnectionInfo	Access WiFi Information †	CNCopyCurrent-NetworkInfo
Bluetooth	27	Alipay: my.startBluetooth DevicesDiscovery	ACCESS_COARSE_LOCATION or ACCESS_FINE_LOCATION ‡	BluetoothLe-Scanner.startScan	**N/A**	**N/A**
		DingTalk: dd.getBluetoothDevices				
		Alipay: my.getBLEDeviceServices				
iBeacon	4	Alipay: my.getBeacons	ACCESS_COARSE_LOCATION or ACCESS_FINE_LOCATION ‡	BluetoothLe-Scanner.startScan	Location Usage	CLBeaconRegion

†: Much harder to exploit on iOS due to the requirement of explicit user approval.
‡: Android 9 and 10 require `ACCESS_FINE_LOCATION` permission, while Android 8.1 requires `ACCESS_FINE_LOCATION` or `ACCESS_COARSE_LOCATION`.

Table 6: Examples of *Sub-window Deception* flaws

Host App	Triggering Sub-App API	Legitimate (and spoofable) Functionality of the Sub-window
Alipay	my.tradePay	Request password of Alipay mobile wallet
Baidu	swan.requestPolymerPayment	Request passwords of multiple mobile wallets
DingTalk	dd.pay	Request password of Alipay mobile wallet
Wechat	wx.addPhoneContact	Enter new contacts
	wx.requestPayment	Request password of Wechat mobile wallet
QQ	new in-app message	Display instant messages and enter a reply

sub-ecosystem on top of the operating system. In such a case, the third-party OS-like app strives to provide a secure, self-contained environment. However, whether it undermines OS security and brings in new risks to modern mobile applications, need to be evaluated to identify the security gap between the new sub-ecosystem and the OSes. Our measurement study shows that popular host apps are generally susceptible to APINA flaws, indicating that the app-in-app system vendors today either fail to identify or are unprepared for the new risks.

Fundamentally, APINA risks come from the limited app-level capabilities the host app has in managing OS resources, lack of OS-level support, and missing of sub-app API standard. Hence, down the road, it will require non-trivial, joint efforts to fundamentally eliminate APINA risks. More specifically, *escaped sub-app API* shows that a third-party host fundamentally lacks the full OS-level knowledge on system resource protection. While such knowledge can come from different sources, e.g., program analysis on relevant OS modules [35, 46], developer documentations, technical reports, etc., each source only contributes a partial view. Hence, joint efforts need to be made in the future to synthesize such knowledge. Our study also brings to light that such efforts are by no means trivial – even the two leading mobile OSes have apparently confusing, conflicting security policies. Synthesizing such a knowledge base will not only benefit the app-in-app paradigm, but also make the OS' security mechanism more transparent for thorough auditing.

Our findings also highlight that prior OS lessons cannot fully eliminate new risks incurred by the new paradigm, which necessitates thorough assessment regarding its new attack surfaces. For example, a malicious sub-app, a sub-window, and a task in *Recents screen* all come as new attack vectors that pose credible new threats to users' most sensitive credentials and privacy.

Mitigation strategy. Following we propose the short-term mitigation and discuss the long-term strategy.

• *Escaped Sub-app API.* Eliminating the risk requires the host to have full OS-level knowledge about resource control (e.g., API-permission mapping), which is difficult. In the meantime, our tool *Apinat* together with our API-permission mapping (released online [12]) can help host app vendors identify *escaped sub-app API*s. Note that, since *Apinat* can identify *escaped sub-app API* with its ignored permissions with respect to specific OS version (see Section 4.3), host vendors may leverage our tool to assess whether their sub-app API permission policy is always compliant with newly released OS version, and thus keep up the pace.

Further, we envision two possible future directions that could lead to more viable solutions. First, as demonstrated by our analysis tool *Apinat*, although it is hard for today's program analysis techniques to derive a complete system API-permission mapping [35, 46], we could significantly improve its accuracy and comprehensiveness through developer documentation analysis. Our current documentation analysis is rather preliminary, but still effective, as demonstrated in our study (Section 4). Down the road, we could develop natural language processing (NLP) based techniques for deriving more complete knowledge about API permission policies. With the techniques, an analysis could cover a wider spectrum of literature, including iOS documents, research papers, technical reports, etc., which will help derive up-to-date knowledge about resource-management policies.

Second, the app-in-app community should work together to standardize sub-app APIs and their permission policies, as done in the HTML5 community on HTML5 APIs. For the time being,

different app-in-app vendors bring in their own APIs for accessing a wide spectrum of systems resources (see our comparison between sub-app API and HTML5 API in Table 7 in Appendix), often without proper safeguard in place, not to mention any effort to standardize the protection.

• *Sub-window deception.* The risk comes from the UI confusion, whether a window portion in the sub-app window belongs to the sub-app or the host (Figure 3b). To mitigate the risk, we envision a higher degree of UI isolation between the sub-app and the host: the sub-app's window should always be separated from that of the host and the host should never reclaim any window portion from the sub-app's window. Regarding the Wallet UI confusion (Section 3.2), for example, Wechat should display its wallet UI only in its own window, and advertise to users that any UI component inside the sub-app window belongs to the sub-app.

A question arises whether a malicious sub-app can fake a crash and then show spoofing content in the sub-app window to resemble the host. This can be mitigated through additional protection supported by the host. Specifically, the host app could keep monitoring the UI of its sub-apps (e.g., through `PixelCopy` API on Android [71]) to detect whether a UI object in the sub-app is similar to a sub-window or a security-sensitive UI object associated with host app. For performance concern, the host app could utilize the state-of-the-art object detection model (e.g., Fast Region-based Convolutional Network based model [61], and YOLO [32, 95]) for the real-time security-sensitive UI object recognition: as reported in prior research [95], Fast YOLO yields the performance of 155 frames per second. Note that, the effectiveness, efficiency and deployment feasibility of such an approach need to be carefully explored and evaluated, which we leave for our future research.

• *Sub-app lifecycle hijacking.* To mitigate this risk, a short-term defense is to place all sub-app resources in the phone's internal storage, so our current exploit (leveraging monitoring the phone's external storage) to track sub-app lifecyle termination (Section 3.3) can be mitigated. However, it is always a concern that sub-apps may deplete the limited space on a phone's internal storage. A long term solution needs to consider how to make the sub-app tasks dynamically scalable, to ensure that the sub-app recycling is no longer traceable and thus the attack surface is fundamentally eliminated. This, nonetheless, calls for support from modern OSes for the emerging app-in-app paradigm.

6 RELATED WORK

As mentioned in Section 2, recent years witnessed several techniques to support the functionality of app encapsulation (e.g., Plugin frameworks, app sandboxing, browser extension, and PhoneGap app, see Table 2). Accordingly, numerous studies have looked into the flaws in these app encapsulation systems: considering the security analysis of Plugin frameworks, a set of flaws were revealed [99, 111, 112], including no permission management and no isolation of the virtualized apps. Compared with Plugin frameworks, the app-in-app paradigm has a much stronger security model, and our study investigates the design challenges and pitfalls in the security mechanism. Meanwhile, the security and privacy implications of browser extensions have been extensively studied [48, 54, 84, 98]. Furthermore, the current security studies on PhoneGap apps focus on the flaws of PhoneGap framework's access control policy [60, 90, 100, 101]. For example, Song et al. [101] revealed that the bridges added by the framework to the browser are not correctly protected by the same origin policy. Jin et al. [86] introduced multiple channels (e.g., barcode, SMS, file system, Contact) for code injection attacks in PhoneGap apps. Different from these works, we focus on the unique features of the app-in-app paradigm (Table 2), and reveal multiple new attack surfaces, which have never been studied before.

The closer to our study are the works of Web App and WebView vulnerability assessment. Up to our knowledge, only one paper [87] analyzed security issue of Web App, in which the security and privacy risks on HTML5 features in PWA were investigated. In our attack on Web App, we did not focus on HTML5 features but a general UI isolation and management model in app-in-app paradigm. Also, although WebView technique is used in some of the app-in-app systems, in sharp contrast to the works on Webview weaknesses [36, 60, 88, 102, 105, 109, 110], our study explores the security implications and challenges for a third-party app to manage operating system resources. Another set of works related to our study is the exploitation of mobile UI deception. Prior phishing attacks [38, 49, 53, 57, 97] mislead victims about the foreground app, aiming at information theft in spoofing apps. Corresponding countermeasures [49, 58, 96] emphasize identifying the app of the whole foreground window, and thus cannot thwart *Sub-window Deception* due to the inability to identify sub-windows. Further, another angle of UI deception is clickjacking [59, 94], which are different from our mobile-phishing attacks.

7 CONCLUSION

In this paper, we present the first systematic security analysis on resource management in high-profile app-in-app systems. Our study shows that such systems are susceptible to a set of serious security flaws. Our measurement study further sheds light on their pervasiveness in the real-world. Further, our study brings in new insights into the fundamental security limitations of this new paradigm, and further contributes to a better understanding of its mitigation strategy, an important step towards building a more secure app-in-app ecosystem.

8 ACKNOWLEDGMENT

We would like to thank our shepherd Yanick Fratantonio and the anonymous reviewers for their insightful comments. This work is supported in part by the NSF CNS-1618493, 1801432, 1838083.

REFERENCES

[1] 2011. HTML5 Fullscreen API Attack. https://feross.org/html5-fullscreen-api-attack/.
[2] 2011. The Inception Bar. https://jameshfisher.com/2019/04/27/the-inception-bar-a-new-phishing-method/.
[3] 2018. Java Runnable. https://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html.
[4] 2018. A one year PWA retrospective. https://medium.com/pinterest-engineering/a-one-year-pwa-retrospective-f4a2f4129e05.
[5] 2018. Wechat 1 million sub-apps. https://www.scmp.com/tech/article/2153705/tencents-wechat-now-host-1-million-mini-programs.
[6] 2018. WeChat reaches 1M sub-apps, half the size of Apple's App Store. https://techcrunch.com/2018/11/07/wechat-mini-apps-200-million-users/.
[7] 2018. Xposed. https://api.xposed.info/reference/packages.html.
[8] 2019. AdBlock. https://getadblock.com.

[9] 2019. Alibaba's alternative to the app store reaches 230M daily users. https://techcrunch.com/2019/01/29/alibaba-alipay-mini-programs-230m-users/.
[10] 2019. Android Location permissions. https://developers.google.com/maps/documentation/android-sdk/location.
[11] 2019. Apache Cordova. https://cordova.apache.org/.
[12] 2019. Apina Supporting Website. https://sites.google.com/view/appinapp/.
[13] 2019. Browser Extension JavaScript APIs. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Browser_support_for_JavaScript_APIs.
[14] 2019. Browser Plug-in. https://developer.mozilla.org/en-US/docs/Plugins/Guide/Plug-in_Basics#Understanding_the_Runtime_Model.
[15] 2019. Droid Plugin. http://droidpluginteam.github.io/DroidPlugin/.
[16] 2019. EditThisCookie. http://www.editthiscookie.com/.
[17] 2019. Entitlements. https://developer.apple.com/documentation/bundleresources/entitlements.
[18] 2019. External-Storage. https://developer.android.com/guide/topics/data/data-storage.
[19] 2019. Ionic. https://ionicframework.com.
[20] 2019. JavaScript. https://en.wikipedia.org/wiki/JavaScript.
[21] 2019. JavaScript data types and data structures. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures.
[22] 2019. MorzillaSecurity Funfuzz. https://github.com/MozillaSecurity/funfuzz/blob/master/src/funfuzz/js/shared/random.js.
[23] 2019. Parallel Space App. http://parallelspace-app.com.
[24] 2019. PhoneGap. https://phonegap.com.
[25] 2019. Pinduoduo. https://www.digitalcommerce360.com/2019/04/16/why-china-ecommerce-is-going-crazy-for-wechat-mini%E2%80%91programs/.
[26] 2019. Virtual App. https://github.com/asLody/VirtualApp.
[27] 2019. WeChat became the 5th most-used app in the world. https://www.dragonsocial.net/blog/wechat-mini-programs/.
[28] 2019. WeChat Mini-programs and social e-commerce. https://walkthechat.com/wechat-mini-programs-simple-introduction/.
[29] 2019. WeChat Mini Programs: The Complete Guide for Business. https://topdigital.agency/wechat-mini-programs-the-complete-guide-for-business/.
[30] 2019. Wechat sub-app review process. https://developers.weixin.qq.com/miniprogram/en/product/reject.html.
[31] 2019. What is a Hybrid Mobile App? https://www.telerik.com/blogs/what-is-a-hybrid-mobile-app-.
[32] 2019. YOLOv3: An Incremental Improvement. https://pjreddie.com/media/files/papers/YOLOv3.pdf.
[33] 2020. Android version distribution. https://gs.statcounter.com/os-version-market-share/android.
[34] 2020. OCR SPACE. https://ocr.space.
[35] Yousra Aafer, Guanhong Tao, Jianjun Huang, Xiangyu Zhang, and Ninghui Li. 2018. Precise Android API Protection Mapping Derivation and Reasoning. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. ACM, 1151–1164.
[36] Yasemin Acar, Michael Backes, Sven Bugiel, Sascha Fahl, Patrick McDaniel, and Matthew Smith. 2016. Sok: Lessons learned from android security research for appified software platforms. In 2016 IEEE Symposium on Security and Privacy (SP). IEEE, 433–451.
[37] Jagdish Prasad Achara, Mathieu Cunche, Vincent Roca, and Aurélien Francillon. 2014. Short paper: WifiLeaks: underestimated privacy implications of the access_wifi_state android permission. In Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks. ACM, 231–236.
[38] Simone Aonzo, Alessio Merlo, Giulio Tavella, and Yanick Fratantonio. 2018. Phishing Attacks on Modern Android. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18). ACM, New York, NY, USA, 1788–1801. https://doi.org/10.1145/3243734.3243778
[39] Apple. 2014. Objective-C. https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html.
[40] Apple. 2018. Swift. https://developer.apple.com/swift/.
[41] Apple. 2019. Apple Entitlements. https://developer.apple.com/library/archive/documentation/Miscellaneous/Reference/EntitlementKeyReference/Chapters/AboutEntitlements.html.
[42] Apple. 2019. Microphone Usage Entitlement. https://developer.apple.com/documentation/avfoundation/avaudiosession/1616601-requestrecordpermission.
[43] Apple. 2019. NEHotspot Requirement. https://developer.apple.com/library/archive/qa/qa1942/_index.html.
[44] Apple. 2019. Web Clip. https://developer.apple.com/library/archive/documentation/AppleApplications/Reference/SafariWebContent/ConfiguringWebApplications/ConfiguringWebApplications.html.
[45] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. Pscout: analyzing the android permission specification. In Proceedings of the 2012 ACM conference on Computer and communications security. ACM, 217–228.
[46] Michael Backes, Sven Bugiel, Erik Derr, Patrick D McDaniel, Damien Octeau, and Sebastian Weisgerber. 2016. On Demystifying the Android Application Framework: Re-Visiting Android Permission Specification Analysis.. In 25th USENIX Security Symposium. 1101–1118.
[47] Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp von Styp-Rekowsky. 2015. Boxify: Full-fledged app sandboxing for stock android. In 24th USENIX Security Symposium. 691–706.
[48] Sruthi Bandhakavi, Nandit Tiku, Wyatt Pittman, Samuel T. King, P. Madhusudan, and Marianne Winslett. 2011. Vetting Browser Extensions for Security Vulnerabilities with VEX. Commun. ACM 54, 9 (Sept. 2011), 91–99. https://doi.org/10.1145/1995376.1995398
[49] 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.
[50] Antonio Bianchi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. 2015. Njas: Sandboxing unmodified applications in non-rooted devices running stock android. In Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices. ACM, 27–38.
[51] ByteDance. 2019. Tiktok sub-app review process. https://developer.toutiao.com/dev/cn/mini-app/operation/agreement/agreement.
[52] 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). USENIX Association, San Diego, CA, 1037–1052. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/chen
[53] Qi Alfred Chen, Zhiyun Qian, and Zhuoqing Morley Mao. 2014. Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks.. In USENIX Security Symposium. 1037–1052.
[54] Vladan Djeric and Ashvin Goel. 2009. Securing script-based extensibility in web browsers. University of Toronto.
[55] Facebook. 2019. React Native. https://facebook.github.io/react-native/.
[56] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android permissions demystified. In Proceedings of the 18th ACM conference on Computer and communications security. ACM, 627–638.
[57] Adrienne Porter Felt and David Wagner. 2011. Phishing on mobile devices. na.
[58] 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 International Conference on Financial Cryptography and Data Security. Springer, 41–59.
[59] Yanick Fratantonio, Chenxiong Qian, Simon P Chung, and Wenke Lee. 2017. Cloak and dagger: from two permissions to complete control of the UI feedback loop. In 2017 38th IEEE Symposium on Security and Privacy (SP). IEEE, 1041–1057.
[60] Martin Georgiev, Suman Jana, and Vitaly Shmatikov. 2014. Breaking and fixing origin-based access control in hybrid web/mobile application frameworks. In NDSS symposium, Vol. 2014. NIH Public Access, 1.
[61] Ross Girshick. 2015. Fast r-cnn. In Proceedings of the IEEE international conference on computer vision. 1440–1448.
[62] Google. 2018. Android Developers. https://developer.android.com/.
[63] Google. 2018. Android MediaProjection. https://developer.android.com/reference/android/media/projection/MediaProjection.
[64] Google. 2018. Android Recents Screen. https://developer.android.com/guide/components/activities/recents.
[65] Google. 2018. Android Signature Permissions. https://developer.android.com/guide/topics/permissions/overview#signature_permissions.
[66] Google. 2018. Android UsbManager. https://developer.android.com/reference/android/hardware/usb/UsbManager.
[67] Google. 2018. Progressive Web App. https://developers.google.com/web/progressive-web-apps/.
[68] Google. 2018. PWA Checklist. https://developers.google.com/web/progressive-web-apps/checklist.
[69] Google. 2018. PWA WebAPK. https://developers.google.com/web/fundamentals/integration/webapks.
[70] Google. 2018. Security Enhancements in Android 6.0. https://source.android.com/security/enhancements/enhancements60.
[71] Google. 2019. Android PixelCopy API. https://developer.android.com/reference/android/view/PixelCopy.
[72] Google. 2019. Android Record Audio Permission. https://developer.android.com/reference/android/Manifest.permission.html#RECORD_AUDIO.
[73] Google. 2019. Android Requiement of Accquiring Dangerous Level Permission. https://developer.android.com/guide/topics/permissions/overview.
[74] Google. 2019. Android Version Fragmentation. https://developer.android.com/about/dashboards.
[75] Google. 2019. Android WifiManager Documentation. https://developer.android.com/reference/android/net/wifi/WifiManager.
[76] Google. 2019. Dangerous Permission Level. https://developer.android.com/guide/topics/permissions/overview#dangerous_permissions.
[77] Google. 2019. Manifest.permission. https://developer.android.com/reference/android/Manifest.permission.

[78] Google. 2019. Progressive Web Apps. https://developers.google.com/web/fundamentals/app-install-banners/promoting-install-mobile.

[79] Google. 2019. WebView. https://developer.android.com/reference/android/webkit/WebView.

[80] Google. 2020. Android 6.0 Change. https://developer.android.com/about/versions/marshmallow/android-6.0-changes#behavior-hardware-id.

[81] Google. 2020. Android InetAddress API. https://developer.android.com/reference/java/net/InetAddress#getCanonicalHostName().

[82] Google. 2020. Privacy changes in Android 10. https://developer.android.com/about/versions/10/privacy/changes.

[83] Google. 2020. Security and Privacy Enhancements in Android 10. https://source.android.com/security/enhancements/enhancements10#filesystem-restrictions.

[84] Arjun Guha, Matthew Fredrikson, Benjamin Livshits, and Nikhil Swamy. 2011. Verified security for browser extensions. In 2011 IEEE symposium on security and privacy. IEEE, 115–130.

[85] Itseez. 2015. Open Source Computer Vision Library. https://github.com/itseez/opencv.

[86] Xing Jin, Xuchao Hu, Kailiang Ying, Wenliang Du, Heng Yin, and Gautam Nagesh Peri. 2014. Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM, 66–77.

[87] Jiyeon Lee, Hayeon Kim, Junghwan Park, Insik Shin, and Sooel Son. 2018. Pride and Prejudice in Progressive Web Apps: Abusing Native App-like Features in Web Applications. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. ACM, 1731–1746.

[88] Tongxin Li, Xueqiang Wang, Mingming Zha, Kai Chen, XiaoFeng Wang, Luyi Xing, Xiaolong Bai, Nan Zhang, and Xinhui Han. 2017. Unleashing the walking dead: Understanding cross-app remote infections on mobile webviews. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. ACM, 829–844.

[89] Meng Luo, Oleksii Starov, Nima Honarmand, and Nick Nikiforakis. 2017. Hindsight: Understanding the evolution of ui vulnerabilities in mobile browsers. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. 149–162.

[90] 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).

[91] Yuhong Nan, Zhemin Yang, Xiaofeng Wang, Yuan Zhang, Donglai Zhu, and Min Yang. 2018. Finding Clues for Your Secrets: Semantics-Driven, Learning-Based Privacy Discovery in Mobile Apps. In NDSS.

[92] Katarzyna Olejnik, Italo Dacosta, Joana Soares Machado, Kévin Huguenin, Mohammad Emtiyaz Khan, and Jean-Pierre Hubaux. 2017. SmarPer: Context-aware and automatic runtime-permissions for mobile devices. In 2017 IEEE Symposium on Security and Privacy (SP). IEEE, 1058–1076.

[93] Olivia Plotnick. 2018. How Brands Are Using WeChat Mini Programs. https://mavsocial.com/wechat-mini-programs-for-brands/.

[94] Andrea Possemato, Andrea Lanzi, Simon Pak Ho Chung, Wenke Lee, and Yanick Fratantonio. 2018. ClickShield: Are You Hiding Something? Towards Eradicating Clickjacking on Android. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. ACM, 1120–1136.

[95] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You only look once: Unified, real-time object detection. In Proceedings of the IEEE conference on computer vision and pattern recognition. 779–788.

[96] 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).

[97] 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). USENIX Association, Washington, D.C., 945–959. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/ren-chuangang

[98] Iskander Sanchez-Rola, Igor Santos, and Davide Balzarotti. 2017. Extension Breakdown: Security Analysis of Browsers Extension Resources Control Policies. In 26th USENIX Security Symposium. USENIX Association, Vancouver, BC, 679–694. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/sanchez-rola

[99] Luman Shi, Jianming Fu, Zhengwei Guo, and Jiang Ming. 2019. " Jekyll and Hyde" is Risky: Shared-Everything Threat Mitigation in Dual-Instance Apps. In Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services. ACM, 222–235.

[100] Kapil Singh. 2013. Practical context-aware permission control for hybrid mobile applications. In International Workshop on Recent Advances in Intrusion Detection. Springer, 307–327.

[101] Wei Song, Qingqing Huang, and Jeff Huang. 2018. Understanding JavaScript Vulnerabilities in Large Real-World Android Applications. IEEE Transactions on Dependable and Secure Computing (2018).

[102] Guliz Seray Tuncay, Soteris Demetriou, and Carl A Gunter. 2016. Draco: A system for uniform and fine-grained access control for web code on android. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM, 104–115.

[103] W3C. 2017. Web App Javascript API Permissions Working Draft. https://www.w3.org/TR/permissions/.

[104] W3C. 2019. Web App navigation scope. https://www.w3.org/TR/appmanifest/#navigation-scope.

[105] 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.

[106] WHATWG. 2020. WHATWG Javascript Fullscreen API Living Standard. https://fullscreen.spec.whatwg.org/#security-and-privacy-considerations.

[107] Wiki. 2019. Single-Sign-On. https://en.wikipedia.org/wiki/Single_sign-on.

[108] 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 X and iOS. In Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15). ACM, New York, NY, USA, 31–43. https://doi.org/10.1145/2810103.2813609

[109] Guangliang Yang and Jeff Huang. 2018. Automated generation of event-oriented exploits in android hybrid apps. In Proc. of the Network and Distributed System Security Symposium (NDSS'18).

[110] Guangliang Yang, Jeff Huang, Guofei Gu, and Abner Mendoza. 2018. Study and mitigation of origin stripping vulnerabilities in hybrid-postmessage enabled mobile applications. In 2018 IEEE Symposium on Security and Privacy (SP). IEEE, 742–755.

[111] Lei Zhang, Zhemin Yang, Yuyu He, Mingqi Li, Sen Yang, Min Yang, Yuan Zhang, and Zhiyun Qian. 2019. App in the Middle: Demystify Application Virtualization in Android and Its Security Threats. Proc. ACM Meas. Anal. Comput. Syst. 3, 1, Article 17 (March 2019), 24 pages. https://doi.org/10.1145/3322205.3311088

[112] Cong Zheng, Tongbo Luo, Zhi Xu, Wenjun Hu, and Xin Ouyang. 2018. Android Plugin Becomes a Catastrophe to Android Ecosystem. In Proceedings of the First Workshop on Radical and Experiential Security (RESEC '18). ACM, New York, NY, USA, 61–64. https://doi.org/10.1145/3203422.3203425

[113] Xiaoyong Zhou, Soteris Demetriou, Dongjing He, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, Carl A Gunter, and Klara Nahrstedt. 2013. Identity, location, disease and more: Inferring your secrets from android public resources. In Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. ACM, 1017–1028.

setRequireOriginal LINK

Added in API level 29

```
public static Uri setRequireOriginal (Uri uri)
```

Update the given `Uri` to indicate that the caller requires the original file contents when calling `ContentResolver#openFileDescriptor(Uri, String)`.

This can be useful when the caller wants to ensure they're backing up the exact bytes of the underlying media, without any Exif redaction being performed.

If the original file contents cannot be provided, a `UnsupportedOperationException` will be thrown when the returned `Uri` is used, such as when the caller doesn't hold `Manifest.permission.ACCESS_MEDIA_LOCATION`.

Parameters	
`uri`	`Uri`: This value must never be `null`.

Returns	
`Uri`	This value will never be `null`.

See also:

`getRequireOriginal(Uri)`

(a) MediaStore.setRequireOriginal

setGeolocationEnabled

Added in API level 5

```
public abstract void setGeolocationEnabled (boolean flag)
```

Sets whether Geolocation is enabled. The default is `true`.

Please note that in order for the Geolocation API to be usable by a page in the WebView, the following requirements must be met:

- an application must have permission to access the device location, see `Manifest.permission.ACCESS_COARSE_LOCATION`, `Manifest.permission.ACCESS_FINE_LOCATION`;
- an application must provide an implementation of the `WebChromeClient#onGeolocationPermissionsShowPrompt` callback to receive notifications that a page is requesting access to location via the JavaScript Geolocation API.

Parameters	
`flag`	`boolean`: whether Geolocation should be enabled

(b) WebSettings.setGeolocationEnabled

setDestinationUri

Added in API level 9

```
public DownloadManager.Request setDestinationUri (Uri uri)
```

Set the local destination for the downloaded file. Must be a file URI to a path on external storage, and the calling application must have the WRITE_EXTERNAL_STORAGE permission.

The downloaded file is not scanned by MediaScanner. But it can be made scannable by calling `allowScanningByMediaScanner()`.

By default, downloads are saved to a generated filename in the shared download cache and may be deleted by the system at any time to reclaim space.

For applications targeting `Build.VERSION_CODES.Q` or above, `WRITE EXTERNAL STORAGE` permission is not needed and the `uri` must refer to a path within the directories owned by the application (e.g. `Context#getExternalFilesDir(String)`) or a path within the top-level Downloads directory (as returned by `Environment#getExternalStoragePublicDirectory(String)` with `Environment#DIRECTORY_DOWNLOADS`).

Parameters	
`uri`	`Uri`: a file `Uri` indicating the destination for the downloaded file.

Returns	
`DownloadManager.Request`	this object

(c) Request.setDestinationUri

Figure 6: Descriptions of Android APIs that Apinat failed to generate permission mapping (Part1).

requestPermission

Added in API level 12

```
public void requestPermission (UsbDevice device,
            PendingIntent pi)
```

Requests temporary permission for the given package to access the device. This may result in a system dialog being displayed to the user if permission had not already been granted. Success or failure is returned via the `PendingIntent` pi. If successful, this grants the caller permission to access the device only until the device is disconnected. The following extras will be added to pi:

- `EXTRA_DEVICE` containing the device passed into this call
- `EXTRA_PERMISSION_GRANTED` containing boolean indicating whether permission was granted by the user

Permission for USB devices of class `UsbConstants#USB_CLASS_VIDEO` for clients that target SDK `Build.VERSION_CODES.P` and above can be granted only if they have additionally the `Manifest.permission.CAMERA` permission.
Requires the `PackageManager#FEATURE_USB_HOST` feature which can be detected using `PackageManager.hasSystemFeature(String)`.

Parameters	
`device`	`UsbDevice`: to request permissions for
`pi`	`PendingIntent`: PendingIntent for returning result

(a) UsbManager.requestPermission

hasPermission

Added in API level 12

```
public boolean hasPermission (UsbDevice device)
```

Returns true if the caller has permission to access the device. Permission might have been granted temporarily via `requestPermission(android.hardware.usb.UsbDevice, android.app.PendingIntent)` or by the user choosing the caller as the default application for the device. Permission for USB devices of class `UsbConstants#USB_CLASS_VIDEO` for clients that target SDK `Build.VERSION_CODES.P` and above can be granted only if they have additionally the `Manifest.permission.CAMERA` permission.
Requires the `PackageManager#FEATURE_USB_HOST` feature which can be detected using `PackageManager.hasSystemFeature(String)`.

Parameters	
`device`	`UsbDevice`: to check permissions for

Returns	
`boolean`	true if caller has permission

(b) UsbManager.hasPermission

addNetworkSuggestions

Added in API level 29

```
public int addNetworkSuggestions (List<WifiNetworkSuggestion> networkSuggestions)
```

Provide a list of network suggestions to the device. See `WifiNetworkSuggestion` for a detailed explanation of the parameters. When the device decides to connect to one of the provided network suggestions, platform sends a directed broadcast `ACTION_WIFI_NETWORK_SUGGESTION_POST_CONNECTION` to the app if the network was created with `WifiNetworkSuggestion.Builder` flag set and the app holds `ACCESS_FINE_LOCATION` permission.

NOTE:

- These networks are just a suggestion to the platform. The platform will ultimately decide on which network the device connects to.
- When an app is uninstalled or disabled, all its suggested networks are discarded. If the device is currently connected to a suggested network which is being removed then the device will disconnect from that network.
- If user reset network settings, all added suggestions will be discarded. Apps can use `getNetworkSuggestions()` to check if their suggestions are in the device.
- In-place modification of existing suggestions are allowed. If the provided suggestions `WifiNetworkSuggestion#equals(Object)` any previously provided suggestions by the app. Previous suggestions will be updated

Requires `Manifest.permission.CHANGE_WIFI_STATE`

Parameters	
`networkSuggestions`	`List`: List of network suggestions provided by the app. This value must never be `null`.

(c) WifiManager.addNetworkSuggestions

Figure 7: Descriptions of Android APIs that Apinat failed to generate permission mapping (Part2).

getScanResults

Added in API level 1

```
public List<ScanResult> getScanResults ()
```

Return the results of the latest access point scan.

Returns	
`List<ScanResult>`	the list of access points found in the most recent scan. An app must hold `ACCESS_FINE_LOCATION` permission in order to get valid results.

(a) WifiManager.getScanResults

getConfiguredNetworks

Added in API level 1

```
public List<WifiConfiguration> getConfiguredNetworks ()
```

Return a list of all the networks configured for the current foreground user. Not all fields of WifiConfiguration are returned. Only the following fields are filled in:

- networkId
- SSID
- BSSID
- priority
- allowedProtocols
- allowedKeyManagement
- allowedAuthAlgorithms
- allowedPairwiseCiphers
- allowedGroupCiphers

Requires `Manifest.permission.ACCESS_FINE_LOCATION` and `Manifest.permission.ACCESS_WIFI_STATE`

Returns	
`List<WifiConfiguration>`	a list of network configurations in the form of a list of `WifiConfiguration` objects.

(b) WifiManager.getConfiguredNetworks

Figure 8: The identified documentation associated with ACCESS_FINE_LOCATION permission.

Table 7: Examples of system resources accessed by sub-app APIs in real-world app-in-app systems, which are missing in standardized HTML5.

Capabilities/Resources	# Related Sub-app API
Authentication/User Identification	4
Background Audio Playback	24
Bluetooth Scan	3
Contacts	3
Clipboard	8
iBeacon	5
Make Phone Call	4
mDns	10
Phone Number	2
NFC Read/Write	10
Wifi Scan	9
Total	82

Table 8: Examples of popular sub-apps in Wechat and corresponding endpoint-request sequences. *n means n continuous requests.

Sub-app	Catergory	Endpoint-request sequence
Google	Technology	nq2kp0q0.api.lncld.net drawtogether.googleminiapps.cn nq2kp0q0.api.lncld.net*2
SEPHORA	Beauty	beacon-mp.tingyun.com fp-it.fengkongcloud.com api.sephora.cn*5 mp.sephora.cn sensor.sephora.cn experiment.appadhoc.com ssl1.sephorastatic.cn zhls.qq.com img.sephorastatic.cn
Amazon	Shopping	api-cslp-emt.amazon.cn www.amazon.cn images-cn.ssl-images-amazon.com*2 api-cslp-emt.amazon.cn*2 images-cn.ssl-images-amazon.com*2 miniapp.amazon.cn
Walmart	Retailer	mapi.ghsmpwalmart.com cdn.ghsmpwalmart.com*2 zhls.qq.com*2 btrace.qq.com statistic.ghsmpwalmart.com*2
McDonald's	Catering	jice.fw4.me*4 api.miniapp.mcdonalds.com.cn*2 cdn.jaxcx.com tracking.mcdonalds.com.cn*4 cdn.miniapp.mcdonalds.com.cn
KFC	Catering	trackingprd.hwwt8.com orders.kfc.com.cn imgorder.kfc.com.cn*6 fp.hwwt8.com
Airbnb	Travel	z1.muscache.cn api.airbnb.cn*2 z1.muscache.cn*2 www.airbnb.cn z1.muscache.cn api.airbnb.cn*3
BMW	Automotive	cdn.bmwwechat.cn dcs.bmw.com.cn cdn.bmwwechat.cn applet.bmwwechat.cn cdn.bmwwechat.cn*2 applet.bmwwechat.cn*3 cdn.bmwwechat.cn*3
Nike	Shopping	analytics.nike.com*3 makecellapi.nike.com*24 insights.nike.com*22 unite.nike.com* 29 www.nike.com*4
Lofter	Blog	hubble.netease.com*4 mini.lofter.com sentry.yuedu.163.com yaolu.yuedu.163.com
Mongo TV	Streaming	0img.hitv.com*4 1img.hitv.com*4 2img.hitv.com*4 3img.hitv.com*4 4img.hitv.com*4 d-weixin-v0.log.mgtv.com*2 i5.hitv.com http://mobileso.bz.mgtv.com st.bz.mgtv.com thirdpart.api.mgtv.com*2

Table 10: Examples of popular sub-apps and corresponding host apps

Sub-app	Category	Host App
Google	Technology	Wechat
Microsoft	Technology	Wechat
SEPHORA	Beauty	Wechat
Amazon	Shopping	Wechat
Dell	Manufacturer	Wechat
Walmart	Retailer	Chrome, Wechat
McDonald's	Catering	Alipay, Wechat
KFC	Catering	Wechat
Airbnb	Travel	Wechat
GIORGIO ARMANI	Luxury	Wechat
Bookings	Travel	Alipay, Baidu, Wechat
Starbucks	Catering	Alipay, Wechat
HSBC	Banking	Alipay, Wechat, Chrome
Huawei	Electronics	Alipay, Wechat
Samsung	Electronics	Wechat
BMW	Automotive	Chrome, Wechat
ICBC	Banking	Wechat

Table 9: Android *dangerous* permissions and iOS *entitlements* requested by vulnerable host apps when first launched, and when the host apps' own functionalities are triggered.

Host App	Android Permissions When Launched	Other Android Permissions	iOS Entitlements When Launched	Other iOS Entitlements
Alipay	READ_PHONE_STATE WRITE_EXTERNAL_STORAGE READ_EXTERNAL_STORAGE CAMERA ACCESS_COARSE_LOCATION ACCESS_FINE_LOCATION	RECORD_AUDIO GET_ACCOUNTS READ_CONTACTS WRITE_CONTACTS	Push Notification Camera Location	Photo Library Contacts Microphone Usage
Baidu	READ_PHONE_STATE WRITE_EXTERNAL_STORAGE READ_EXTERNAL_STORAGE ACCESS_COARSE_LOCATION ACCESS_FINE_LOCATION	CAMERA GET_ACCOUNTS READ_CONTACTS WRITE_CONTACTS RECORD_AUDIO	Push Notification Location	Camera Microphone Usage Photo Library
DingTalk	READ_PHONE_STATE WRITE_EXTERNAL_STORAGE READ_EXTERNAL_STORAGE	READ_CALENDAR WRITE_CALENDAR CAMERA GET_ACCOUNTS READ_CONTACTS WRITE_CONTACTS ACCESS_COARSE_LOCATION ACCESS_FINE_LOCATION RECORD_AUDIO	Push Notification Contacts	Calendars Camera Microphone Usage Photo Library Location
QQ	READ_PHONE_STATE WRITE_EXTERNAL_STORAGE READ_EXTERNAL_STORAGE	CAMERA GET_ACCOUNTS READ_CONTACTS WRITE_CONTACTS ACCESS_COARSE_LOCATION ACCESS_FINE_LOCATION RECORD_AUDIO READ_CALENDAR WRITE_CALENDAR	Push Notification Contacts	Camera Microphone Usage Photo Library Location
Wechat	READ_PHONE_STATE WRITE_EXTERNAL_STORAGE READ_EXTERNAL_STORAGE	CAMERA GET_ACCOUNTS READ_CONTACTS WRITE_CONTACTS ACCESS_COARSE_LOCATION ACCESS_FINE_LOCATION RECORD_AUDIO BODY_SENSORS	Push Notification Camera Contacts Montion & Fitness	Photo Library Microphone Usage Location