Identity Confusion in WebView-based Mobile App-in-app Ecosystems

Lei Zhang, Zhibo Zhang, and Ancong Liu, *Fudan University;* Yinzhi Cao, *Johns Hopkins University;* Xiaohan Zhang, Yanjun Chen, Yuan Zhang, Guangliang Yang, and Min Yang, *Fudan University*

https://www.usenix.org/conference/usenixsecurity22/presentation/zhang-lei

This paper is included in the Proceedings of the 31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

Open access to the Proceedings of the 31st USENIX Security Symposium is sponsored by USENIX.

Identity Confusion in WebView-based Mobile App-in-app Ecosystems

Lei Zhang[1,⋆], Zhibo Zhang[1,⋆], Ancong Liu[1], Yinzhi Cao[2], Xiaohan Zhang[1], Yanjun Chen[1]
Yuan Zhang[1], Guangliang Yang[1], Min Yang[1]
1: Fudan University, 2: Johns Hopkins University
1: {zxl, zhibozhang19, acliu19, xh_zhang, yanjunchen20, yuanxzhang, yanggl, m_yang}@fudan.edu.cn
2: yinzhi.cao@jhu.edu
⋆: The first two authors have contributed equally to this work.

Abstract

Mobile applications (apps) often delegate their own functions to other parties, which makes them become a super ecosystem hosting these parties. Therefore, such mobile apps are being called super-apps, and the delegated parties are subsequently called sub-apps, behaving like "app-in-app". Sub-apps not only load (third-party) resources like a normal app, but also have access to the privileged APIs provided by the super-app. This leads to an important research question—determining who can access these privileged APIs.

Real-world super-apps, according to our study, adopt three types of identities—namely web domains, sub-app IDs, and capabilities—to determine privileged API access. However, existing identity checks of these three types are often not well designed, leading to a disobey of the least privilege principle. That is, the granted recipient of a privileged API is broader than intended, thus defined as an "identity confusion" in this paper. To the best of our knowledge, no prior works have studied this type of identity confusion vulnerability.

In this paper, we perform the first systematic study of identity confusion in real-world app-in-app ecosystems. We find that confusions of the aforementioned three types of identities are widespread among all 47 studied super-apps. More importantly, such confusions lead to severe consequences such as manipulating users' financial accounts and installing malware on a smartphone. We responsibly reported all of our findings to developers of affected super-apps, and helped them to fix their vulnerabilities.

1 Introduction

Nowadays, mobile applications (apps) bring significant convenience to people's work and daily lives with rich functionalities. To better serve existing users and keep attracting new users, these mobile apps—or called super- or host-apps—often delegate some of their functions to other parties for content and functionality enrichment. These parties with delegated functions are thus defined as "sub-apps", and the community developing and maintaining sub-apps is called a mobile "app-in-app" ecosystem. Some "app-in-app" ecosystems are extremely popular, e.g., WeChat [1] is hosting >3.8 million sub-apps, which is even more than the number (3.04 million) of Android apps in Google Play [7].

Figure 1 illustrates a typical architecture of an app-in-app ecosystem based on our study of 47 popular super-apps. When a user clicks a Universal Resource Identifier (URI) specifying the super-app protocol and a sub-app ID, the super-app loads the sub-app from its server into a WebView instance. After that, there are two important steps for a sub-app. First, a sub-app may load third-party resources with different identities into web frames of the WebView [11] instance. For example, Pagoda [8], a fruit franchise with 4,000+ stores nationwide in China, loads a cloud provider's domain for remote backup and an advertisement provider domain. Second, a sub-app may access *privileged* APIs provided by the super-app with sensitive and powerful functionalities. Examples of these APIs are access to saved user data (e.g., account, friends, and phone number used in registration) and utilization of OS-level resources reserved for the super-app (e.g., location, camera, and microphone).

One crucial security research question in an app-in-app ecosystem is determining who can call specific privileged APIs provided by the super-app, given the existence of multi-party resources and the access to privileged APIs in one sub-app. This "who" question is an access control issue or, more specifically, an identity check problem. That is, the super-app needs to check the identity of a runtime API invocation and determine whether the invocation is legitimate. While the problem is intuitively simple, the challenge is that many different definitions of identities exist in a super-app. The first is the *domain name*, one crucial element in the web origin triple because WebView is used to render sub-apps. The second type of identity is a *sub-app ID* assigned by the super-app because the super-app loads sub-apps from their

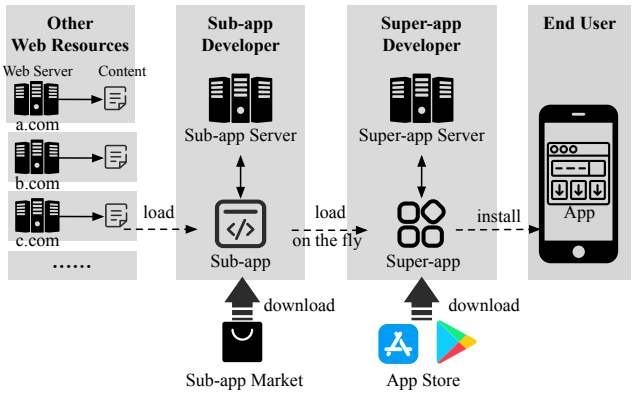

Figure 1: Simplified relationships of participants in the app-in-app ecosystem.

servers hosting the sub-app code. The last is a secret, called a *capability*, shared between a super-app and a sub-app.

Existing super-apps often adopt one of the aforementioned three types of identities to check privileged API invocation. However, none of these identities, at least those adopted by existing super-apps, are atomic, thus disobeying the least privilege principle. For example, a privileged web domain may be embedded in an unprivileged sub-app; a sub-app with a privileged sub-app ID as the identity may contain unprivileged, third-party web domains; similarly, a privileged capability may be obtained by an unprivileged sub-app as well. That is, when a super-app grants a privilege to identity in an app-in-app ecosystem, the intended recipient can be broader than, or different from, the identity that actually represents. Therefore, an adversary can often disguise her own identity to be one with the granted permission, confusing the super-app that performs the identity check. Such a vulnerability, if it exists in an app-in-app ecosystem, is defined as *identity confusion* in the paper.

To the best of our knowledge, no prior works have studied identity confusion vulnerabilities in the app-in-app ecosystem. There are two categories of prior works on WebView vulnerabilities. The first category [21, 31, 33, 36] assumes that WebView as a whole is untrusted, which needs to be isolated from the host app. In such a threat model, identities are clearly defined and separated, i.e., WebView vs. the host app. The second category [35, 45, 48, 52] explores vulnerabilities in WebView itself, e.g., URL display of WebView content. In this threat model, identities are clearly defined as web origins. Fundamentally, our identity confusion vulnerability is due to the introduction of sub-apps, which overlaps with the classic origin identity introduced by WebView from the Web.

In this paper, we perform the *first* systematic study of identity confusion vulnerabilities and their exploits in the real-world app-in-app ecosystems. The adoption of different identities naturally categorizes vulnerabilities into three types:

- Domain name confusion. Such confusion could arise when a malicious sub-app with an unprivileged app ID loads a privileged web domain, and the super-app only checks the domain name for identity. Particularly, we find that there exist race conditions among rendering the web content, obtaining the domain name, and checking the domain name. When the rendered content has a different domain name from what is being checked, domain name confusion arises.
- App ID confusion. Such confusion could arise when an unprivileged web domain resides in a privileged sub-app, and the super-app only checks the app ID. We design a mimicry attack to achieve this purpose in loading malicious URLs into a privileged sub-app. The attack first abuses webpage redirections of some sub-apps and then exploits flawed URI loading checks of super-apps, e.g., string matching that checks suffixes and insecure regular expressions.
- Capability confusion. Such capability confusion may come from either a malicious app ID or a malicious domain name with a privileged capability. We design leak attacks to steal or obtain the capability that can be used to invoke privileged APIs. Specifically, we find that the APIs to obtain capabilities can often be reverse-engineered and called without any protections against adversaries.

Our systematic study involves 47 high-profile super-apps collected from three leading app stores and ranked by their popularity. Our results show that they (both the Android and iOS versions) are *all* vulnerable to at least one type of identity confusion attack despite the diversity in identity checks. We then explore and study the further consequences of identity confusion beyond breaking identity checks. We find that such confusion vulnerabilities lead to consequences such as phishing, privacy leaks, and privilege escalation. Specifically, 31 are further vulnerable to phishing, 35 privacy leaks, and 38 privilege escalations. We report all the vulnerabilities to corresponding super-app developers and help them with the fix. As an example, we have a regular monthly meeting schedule with Alipay for half a year before fixing the vulnerability.

We summarize the contributions of this paper as below:

- We conduct the first systematic study on identity confusion vulnerabilities in super-apps with app-in-app ecosystems by analyzing their design and implementation flaws. We find three types of confusion vulnerabilities: app ID, domain name, and capability.
- We collect and analyze 47 popular real-world super-apps, which exceed 46 billion downloads in total. Our analysis confirms that they are all vulnerable to different types of identity confusion vulnerabilities. Such vulnerabilities can further lead to severe consequences, such as stealing bank accounts and remote installation of malicious apps.
- We thoroughly study why such identity confusion vulnerabilities exist and propose corresponding mitigation strategies based on the causes.

Table 1: Top 15 popular super-apps and their sub-app markets. "-" means there are no public statistics, and it is hard to estimate the number of sub-apps in the corresponding market.

Super-app Name	Category	Downloads	Market Size
TikTok	Social	18.8B+	-
WeChat	Communication	2.1B+	3.8M+
Snapchat	Social	1B+	6+
Kuaishou	Social	780M+	-
Alipay	Finance	690M+	2M+
Line	Communication	500M+	-
UC Browser	Communication	500M+	1K+
Baidu	Tools	410M	420K+
JinRiTouTiao	News & Magazines	220M+	-
Microsoft Teams	Business	100M+	911+
Grab	Maps & Navigation	100M+	-
VK	Social	100M+	219+
Paytm	Finance	100M+	176+
Go-Jek	Travel & Local	50M+	15+
UnionPay	Finance	39.7M+	705+

2 App-in-app Ecosystem: A Survey Study

In this section, we present a brief survey study of existing app-in-app ecosystems. The purpose here is to present how popular such ecosystems are, what structures (including identity checks) super-apps use, and how sub-apps are running atop super-apps.

2.1 Popular Super-app Runtimes

In this subsection, we perform a survey study to crawl and analyze popular app-in-app ecosystems. Our methodology is semi-automatic with three steps. First, we randomly crawl 6,000 popular Android apps from two leading Android app stores (i.e., Google Play and WanDouJia [10]) and automatically analyze these apps for the presence of WebViews. Second, if WebViews are present, we manually analyze these apps, e.g., search them in online engines, to understand whether they support any sub-apps. Lastly, we also study other markets, e.g., iOS's App Store, to find the counterpart super-app and the ecosystem.

Table 1 shows a list of the top 15 popular super-apps ranked by total downloads according to our survey study. These super-apps are diversified, which ranges from communication and social to finance and business and spans across different countries, such as WeChat (China), Line (Korea) [4], and Microsoft Teams (U.S.) [5]. The number of sub-apps in each ecosystem also varies from several million to a few hundred. Note that the number value for some super-apps is "unknown" (marked as "-") because we cannot find a reliable source to estimate the market size, and the super-app disallows broader sub-app crawling.

We further analyze these super-apps and summarize them into a typical structure of an app-in-app ecosystem in Figure 2. A super-app provides a runtime for sub-apps with three major components: (i) an embedded browser instance, (ii) runtime privileged APIs, and (iii) a web-to-mobile bridge.

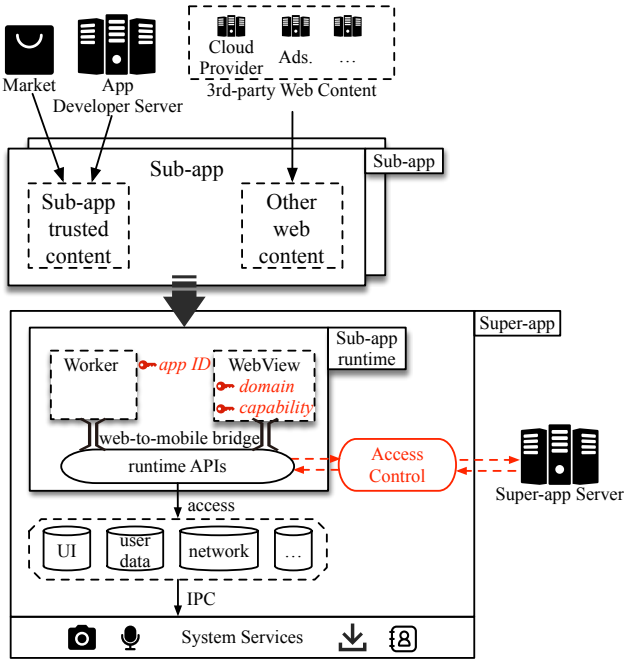

Figure 2: A Typical Structure of App-in-app Ecosystem

First, the embedded browser instance (e.g., WebView[1] for Android and WKWebView [12] for iOS) provides an isolated environment for a sub-app. Such an instance often includes a customized worker to load and execute pre-defined sub-app code. Second, runtime privileged APIs provide access to various resources, such as user data and network access, which are sometimes unique to the super-app. Third, the web-to-mobile bridge connects sub-app code with the native Java code and, most importantly, enforces identity checks for the sub-app. While details of the bridge vary among different super-apps, a typical implementation encapsulates all privileged API invocations from the web side into a message sent to the mobile side via a dispatch method registered through "addJavaScriptInterface". The mobile side parses the received message, finds the corresponding APIs, checks identities, and then invokes them if the check passes.

2.2 Typical Sub-app Programming Model and Lifecycle

This subsection describes the general programming model of sub-apps running atop super-app runtimes as described in Section 2.1. Such sub-apps are usually programmed as mini web applications with JavaScript, HTML, and CSS and have possible access to privileged APIs via the web-to-mobile bridge. Sub-apps not only are hosted on a super-app market, but also fetch content from their own or third-party servers.

Now we describe a typical lifecycle of a sub-app when being loaded by a super-app. First, an end user will either click on or scan a QR code [2] with a deep link [3] pointing to a sub-

[1] Without loss of generality, we use WebView to refer to such embedded web browsers in the paper.

Table 2: The process of identity checks in the top 15 super-apps. D or A means the whitelist of Domain or AppID and their subscripts *sub* or *super* mean who provides them. Symbol → means the check happens in the *Server* or *Native* side.

Super-app	Identifier	Check Policy	Location
TikTok	Domain	Endswith(targetURL, {d|∀d∈D_{sub}})	Loading
	AppID	appID∈A_{super}	API access
WeChat	Domain	targetURL → (*Server*,D_{sub})	Loading
		targetURL → *Server*	API access
	AppID	appID → *Server*	API access
	Capability	Equal(S_{caller},S_{super})	API access
Alipay	Domain	RegMatch(targetURL, {d|∀d∈D_{sub}]})	Loading
		RegMatch(callerURL,{d|∀d∈D_{super}})	API access
	AppID	appID∈A_{super}	API access
		appID → *Server*	API access
UC Browser	Domain	RegMatch(targetURL,{d|∀d∈D_{sub}})	Loading
		RegMatch(callerURL,{d|∀d∈D_{super}})	API access
	AppID	appID∈A_{super}	API access
		appID → *Server*	API access
Baidu	Domain	Endswith(targetURL,{d|∀d∈D_{sub}})	Loading
	AppID	appID → *Server*	API access
JinRiTouTiao	Domain	Endswith(targetURL, {d|∀d∈D_{sub}})	Loading
	AppID	appID∈A_{super}	API access
Teams	Domain	Equal(targetURL,{d|∀d∈D_{sub}})	Loading
		RegMatch(callerURL,{d|∀d∈D_{super}})	API access
VK	AppID	appID → *Server*	API access
Go-Jek	Domain	Equal(targetURL,{d|∀d∈D_{sub}})	Loading
		Equal(callerURL,{d|∀d∈D_{super}})	API access
UnionPay	Domain	targetURL → (*Native*,D_{sub})	Loading
	Capability	Equal(S_{caller},S_{super})	API access

app of a super-app's app-in-app ecosystem. Second, the super-app will find the sub-app based on the app ID embedded in the deep link and then download a bundle of web content from the super-app's market. Such downloaded content is often rendered in a customized worker provided by the WebView. Third, the sub-app, e.g., these running in a worker, will further fetch contents from its own or third-party servers and render them in a WebView instance. Lastly, the sub-app's code, including those downloaded from the super-app's market, its own server, and third-party server, may access privileged APIs via the web-to-mobile bridge.

Note that there are two locations where an identity check can happen. First, when web contents are fetched from sub-app or third-party servers and then rendered in a WebView instance, the super-app will check the identity of fetched contents. Second, when a WebView instance accesses privileged APIs provided by the super-app, the super-app will also check whether the access is allowed based on the WebView instance's identity.

2.3 Existing Identity Checks

In this subsection, we perform a survey study on how identity checks in existing super-apps work. Our methodology is as follows. We manually review all the 15 super-app's source code in Table 1 with the help of static analysis tools and explore the super-app using dynamic analysis. Our purpose is to understand an important question on what identity and corresponding checking policy are used in real-world super-apps.

Let us summarize all identities and their checking policies found in these super-apps below:

- Domain Name. A domain name, as part of web origin, represents a server and contents delivered from the server. We find two main types of domain name based identity checks: (i) strict whitelist and (ii) vague matching. First, some super-apps use a strict method to exactly match a whitelist of web domains. Second, some super-apps adopt a vague matching method, e.g., an `Endswith` to check the suffix of a web domain and a regular expression to match domains with certain patterns.
- App ID. A sub-app ID (or called AppID for short) is an identifier assigned by a super-app to the sub-app. The checking of AppID is usually strict based on a whitelist, and the checking could be performed at either the super-app (native and Java) or the remote server.
- Capability. A capability is a secret issued by either a super-app or a server and checked based on exact match. There are two ways of obtaining a capability in existing super-apps. First, a sub-app obtains a capability on the mobile side via a hidden runtime API. Second, a sub-app obtains a capability from its cloud after a two-way authentication.

Then, let us describe details of our survey study results, i.e., how these identity checks exist in real-world super-apps, in Table 2. First, most super-apps adopt more than one identity in the check to ensure security. Second, most super-apps adopt identity checks at both the content loading and the API access to ensure that the loaded contents are correct and the APIs, especially privileged ones, are accessed with a correct identity. Lastly, identity check policies are very diversified from one super-app to another, making the checks fragmented and local to a specific super-app.

2.4 Super-app Runtime API Analysis

In this subsection, we perform a study on runtime APIs provided by super-apps. The challenges are three-fold. First, these APIs are different from one super-app to another. That is, we need to analyze each super-app. Second, many APIs are hidden, i.e., undocumented, and cannot be discovered by reading documents. Lastly, many APIs are not directly invoked but triggered by a web-to-mobile message via an API pool.

Our detailed steps in discovering these APIs contain both static and dynamic analysis. First, we analyze super-apps statically to find direct hidden API invocations via standard WebView interfaces or event handlers (e.g., methods annotated by "@JavaScriptInterface" or "onConsoleMessage()"). Specifically, we conduct a static control-flow analysis by utilizing both "addJavaScriptInterface" and WebView callbacks as entries. Then, we identify all container objects (e.g., maps, arrays, and sets) during static analysis as potential API pools.

Second, we use dynamic instrumentation to discover indirect hidden API calls. Specifically, we hook statically-identified container objects (e.g., via Xposed [13]) and then

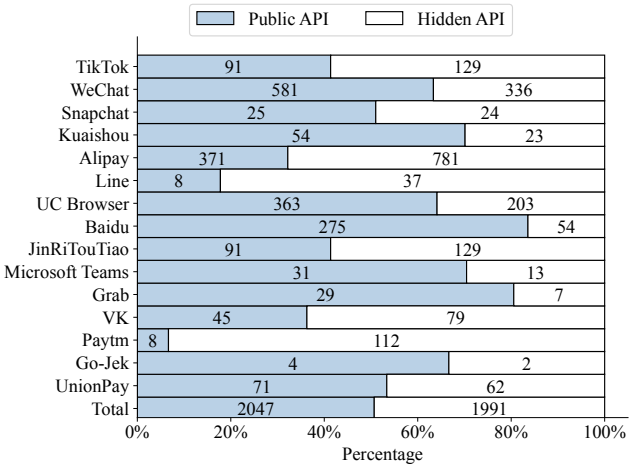

Figure 3: API statistics among top 15 super-apps.

generate test cases to trigger documented public runtime APIs. If these container objects are accessed during a public API call, we will consider them as API pools and read all the stored APIs from the pool. Next, if a hidden API is protected by manually verified identity checks, we will consider it as a privileged hidden API.

Figure 3 illustrates the results of the top 15 super-apps, which show that about 50% of APIs are not well-documented. We also manually sample 200 hidden APIs and check whether they are privileged. Our analysis results show that at least 80% of them, i.e., 160 APIs, are privileged and should not be used by arbitrary sub-apps. For example, a hidden API—named "rpc"—can be used to access the super-app's cloud-side interfaces, like manipulating user accounts. For another example, a hidden API called "getUsageRecord" in TikTok can be abused by sub-apps to monitor users' actions on other sub-apps.

3 Identity Confusion: An Overview

In this section, we give an overview of identity confusion vulnerability by presenting the definition of identity confusion, a motivating example, and our threat model.

3.1 Definition

An identity confusion is a type of vulnerability where a permission (e.g., the access to a privileged API and the loading of web content) is granted to an identity that is broader than (or different from) the intended target, leading to a confusion. An identity confusion is often a disobey of the least privilege principle. In an app-in-app ecosystem, identity confusion arises when multiple definitions of identities co-exist for a given entity, such as a WebView instance. For example, say a permission is granted to an AppID. Then, an identity confusion happens when the sub-app with the AppID is tricked into loading contents from a malicious web domain. For another example, if a permission is granted to a web domain, contents from the web domain may be loaded in a malicious sub-app. That is, one notable reason for such confusion is that web content (loaded in sub-apps) is highly flexible and potentially changes every moment, e.g., web navigation and even sub-app redirection. Thus, it is challenging for the super-app layer to obtain the correct identities, especially when a change happens in the sub-app layer.

Oftentimes, an identity confusion needs to be combined with another vulnerability, e.g., an incorrect domain check or a race condition, for exploitation. Let us take a look at the aforementioned two examples in the previous paragraph again. When the super-app has an incorrect domain check, the adversary can trick a sub-app to load contents from a malicious domain. Similarly, when there exists a race condition in checking domain names, the contents loaded in a malicious sub-app can be recognized as from a permitted web domain. Once identity confusion is exploited, the consequences could be severe because identities are often associated with high privileges, e.g., these APIs accessing user data.

To summarize, a remote adversary (e.g., malicious web content and sub-app provider) with unprivileged identities can disguise own identities, confuse access control enforced in super-apps, and finally call privileged runtime APIs. This can be done by leveraging the privilege assignment and management problem and the asynchronous design between the sub-app and mobile layers. We define such a vulnerability as *identity confusion*.

3.2 A Motivating Example

In this subsection, we describe a motivating example of a super-app WeChat and its sub-app Pingduoduo[2] to illustrate identity confusion vulnerability, which eventually leads to privilege escalation attacks, such as arbitrary APK download and installation on the Android platform.

Let us describe the steps of the end-to-end attack as shown in Figure 4. The attack has 12 detailed steps that can be grouped into three major phases: (i) loading contents in the customized worker, (ii) loading contents in the WebView instance, and (iii) downloading malicious apks.

First, let us look at the first phase in loading contents into the customized worker. In Step (0a), a victim is tricked into clicking on a malicious deep link such as `weixin://encoded(pingduoduo-appID,path,malicious-url)`. WeChat will recognize the deep link for preparing the runtime for Pingduoduo's sub-app in Step (0b). Starting from Step (1a), WeChat downloads and executes Pingduoduo's sub-app code from its own market. Next, in Step (1b), Pingduoduo sends the request for loading the URL embedded in the malicious deep link. Note that the original design of this dynamic URL request is for convenient switches between different online shops maintained by Pingduoduo. This request is hooked by WeChat, which will further send

[2]Pingduoduo is a popular online customer-to-manufacturer market managing over 8.6 million virtual shops.

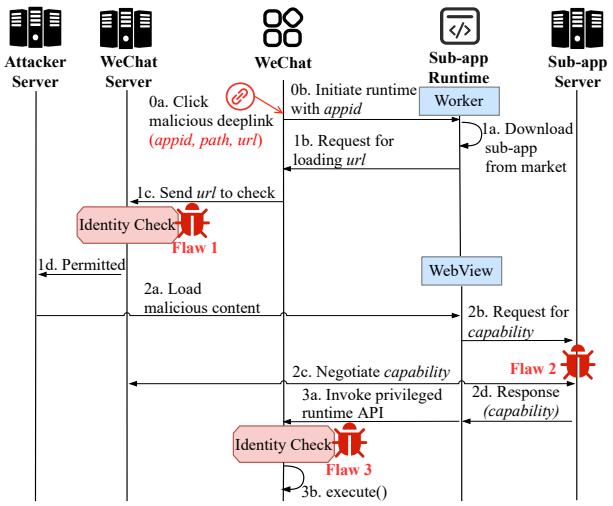

Figure 4: Motivating example for a remote attacker to exploit WeChat app-in-app ecosystem.

the URL to its own server for security check in Step (1c). However, the design and implementation to check the URL are flawed, which leads to our first identity confusion vulnerability.

Here is a description of the flaw. The security check is flawed because the server-side URL parser cannot distinguish the username and the hostname of a common URI. For example, a URL like `https://benign.com:x@malicious.com` will be considered `benign.com`. Therefore, in Step (1d), the WeChat server grants permission for loading this URL.

Second, let us look at the second phase in loading contents into a WebView instance. Because the WeChat obtains a green light for loading the malicious URL, in Step (2a), WebChat loads the contents from `malicious.com` to Pingduoduo's runtime. At this moment, the loaded malicious contents still have no high privileges, because they are isolated in a WebView instance. Although the AppID of Pingduoduo has a high privilege, the access to high privileged APIs still needs a capability, i.e., a secret token.

Here comes our second flaw in the ecosystem. The capability is obtained from a Web service API provided by Pingduoduo, which does not have any access control. That is, any client can call this API to obtain a capability for a higher privilege at WeChat. Specifically, in Step (2b), the malicious contents loaded in the WebView instance can request for a capability from Pingduoduo, which will negotiate with the WeChat server in Step (2c) and then deliver the capability to the WebView in Step (2d).

Third, let us describe the third phase, i.e., downloading and installing a malicious APK. In Step (3a), the malicious contents invoke a privileged API with the obtained capability. Particularly, we use `addDownloadTask()`, a hidden, undocumented API, as an example in Step (3b), which can download and install any APKs on the Android platform.

Note that there exist two types of identity confusion vulnerability for the invocation of `addDownloadTask()`, which are AppID and capability confusions, because the API invocation requires the checking of both the AppID and the capability. The former AppID confusion happens when a malicious domain is loaded in a sub-app with an authenticated AppID. This vulnerability has to be combined with the incorrect policy checking, i.e., Flaw 1 between Steps (1c) and (1d). The latter capability confusion happens when a malicious domain from the correct sub-app can request for a capability from the server, i.e., Flaw 2 between Steps (2a) and (2b), and the requested capability is accepted by WeChat, i.e., Flaw 3 between Steps (3a) and (3b).

3.3 Threat Model

In this subsection, we describe the threat model adopted in the paper. We assume that the super-app and the underlying mobile Operating System (OS) are benign and with no malicious mobile apps installed. Specifically, we consider the following two scenarios:

- Vulnerable Sub-app. A vulnerable sub-app is benign code running atop a super-app with an identity confusion vulnerability (e.g., an AppID or a capability confusion). The adversary in this scenario is a malicious web domain, which has the capability to send a victim user a malicious, phishing deep link pointing to the vulnerable sub-app inside a super-app.
- Malicious Sub-app. A malicious sub-app is code with malicious intent and being crafted by an adversary. The adversary in this scenario is a malicious sub-app developer, which has the capability to upload malicious content to the market of the super-app and trigger an identity confusion vulnerability (e.g., a domain name confusion) of the super-app.

Note that the former scenario—a vulnerable sub-app case—is considered as a stronger threat model compared with the latter. The reasons are two-fold. First, although both threat models need that a victim user clicks on a malicious deep link, the link itself is pointing to a recognized, benign sub-app in the former scenario, but an unrecognized, potentially-malicious sub-app in the latter. Second, the threat model of the malicious sub-app requires that the adversary uploads the malicious code (potentially obfuscated) to the super-app market, which boosts the chance of being detected.

4 Identity Confusion: A Taxonomy Study

In this section, we perform a taxonomy study to break down existing identity confusions into three major types: domain name, AppID and capability-based.

4.1 Domain Name Confusion

A domain name confusion is that the web domain that invokes a privileged API from WebView is different from the domain that a super-app obtains and checks for identity. Specifically,

we classify domain name confusions into two types: timing-based (due to race condition) and frame-based (due to the existence of multiple domains). Table 3 shows the high-level results of whether event handlers of different Android classes are vulnerable to these two types of timing and frame-based race conditions. If super-apps use any of these tested WebView APIs and callbacks to implement domain-based identity checks are all vulnerable. We now describe the details.

4.1.1 Type 1: Timing-based Confusion

The first type—called timing-based—is because of a race condition between different threads of WebView and super-app from a high level. That is, as a simplification of the race condition, when a WebView thread invokes a privileged API and passes the control to a super-app thread, the identity is from say `malicious.com`; but when another super-app thread checks the identity, the identity changes to say `privileged.com` due to redirection, leading to confusion.

We now describe the details. Before that, we need to explain different threads that reside in WebView instances and super-apps.

- WebView Threads. A WebView instance usually has two types of threads, one used for rendering web contents (called a render thread) and the other used for loading web contents (called a browser thread).
- Super-app Threads. A super-app may have three types of threads: (i) a thread that obtains the WebView's domain name as an identity, (ii) a thread that checks the obtained identity and decides whether to allow the execution, and (iii) a thread that dispatches privileged API calls in an asynchronous queue. Note that the existence of these three types of threads depends on the design and implementation of super-apps.

Next, we illustrate two case studies.

Case 1: Race between WebView's rendering and loading threads. This race condition is because WebView's rendering and loading threads may be dealing with content from different web domains. Specifically, on the WebView end, when the loading thread starts to load contents from a new URL after redirection, the rendering thread may still execute contents from the old URL. Then, on the super-app end, there are two threads, one that obtains a new web domain after redirection from the loading thread of WebView as an identity, and the other that checks the new domain name but allows a privileged API call from the old domain.

There are two variations of this race condition depending on the initiation of the URL loading. Figure 5 shows one variation. First, the JavaScript from `malicious.com` running in WebView's rendering thread starts to redirect the webpage to `privileged.com`. Second, the redirection is sent to the loading thread, which starts to load `privileged.com` and triggers the callback (e.g., `onPageStarted()`) registered by the super-app. Third, the corresponding super-app thread triggered by the callback obtains the new domain name as the

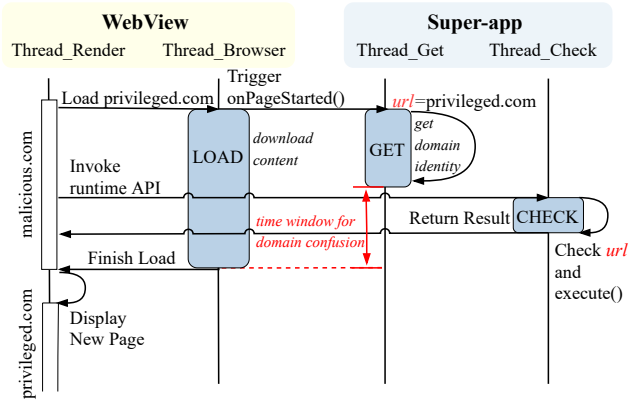

Figure 5: [Domain Name Confusion: Type 1] An illustration of timing-based confusion using `onPageStarted` API due to race conditions between WebView's Rendering (called Render) and Loading (called Browser) Threads.

identity. Lastly, the same JavaScript from `malicious.com` invokes a privileged API call, but the identity has already become `privileged.com`.

Next, Figure 6 shows another variation. First, a thread of the super-app calls `loadUrl()`, which instructs the browser thread to load an URL (`privileged.com`) and returns the new domain name as the identity. Second, the JavaScript code from the old URL (`malicious.com`) invokes a privileged API, which is checked by a thread of the super-app but considered as from `privileged.com`. This leads to domain name confusion.

It is worth noting that many WebView APIs and callbacks are either interacting with or triggered by the browser thread instead of the render thread. That is, the race condition is very common among many WebView APIs. To understand how prevalent such race conditions are, we collect all the WebView APIs from Android's documentation, and then perform a small-scale study on WebView APIs that return URLs and WebView callbacks that have URLs as a parameter. Specifically, we first either register a callback and then redirect the webpage to another URL. Then, we measure whether there exists inconsistency between the callback and the webpage's URL from two perspectives: (1) whether the old webpage can still execute JavaScript code but the URL has been updated to the new one and (2) whether the new webpage can execute JavaScript code but the URL is still the old one.

Here are the detailed steps in measuring the aforementioned two points. We create a webpage that has an endless loop for printing its URL together with the timestamp (i.e., T_{js}), and let the WebView's callbacks print its own URL (i.e., URL_{cb}) with the timestamp (i.e., T_{cb}). Then, for (1), we measure whether $\max(T_{js})$ is larger than T_{cb} and URL_{cb} is the new webpage's; for (2), we measure whether $\min(T_{js})$ is smaller than T_{cb} and URL_{cb} is the old webpage's.

Table 3: The domain name confusion in using WebView's event handlers to obtain identity information. We measure them at time and frame dimensions.

Class Name	Method Signature of Event Handlers	Domain Name Confusion	
		Timing-based	Frame-based
	Getter Method:		
WebView	getOriginalUrl ()	✔	✔
	getUrl ()	✔	✔
	Callback Method:		
WebViewClient	doUpdateVisitedHistory (WebView view, **String url**, boolean isReload)	✔	✔
	onLoadResource (WebView view, **String url**)	✔	
	onPageCommitVisible (WebView view, **String url**)	✔	✔
	onPageFinished (WebView view, **String url**)	✔	✔
	onPageStarted (WebView view, **String url**, Bitmap favicon)	✔	✔
	onReceivedClientCertRequest (WebView view, **ClientCertRequest request**)	✔	✔
	onReceivedError (WebView view, **WebResourceRequest request**, WebResourceError error)	✔	
	onReceivedHttpAuthRequest (WebView view, HttpAuthHandler handler, **String host**, String realm)	✔	
	onReceivedHttpError (WebView view, **WebResourceRequest request**, WebResourceResponse errorResponse)	✔	
	shouldInterceptRequest (WebView view, **WebResourceRequest request**)	✔	
	shouldOverrideUrlLoading (WebView view, **WebResourceRequest request**)	✔	
WebChromeClient	onReceivedTouchIconUrl (WebView view, **String url**, boolean precomposed)	✔	✔

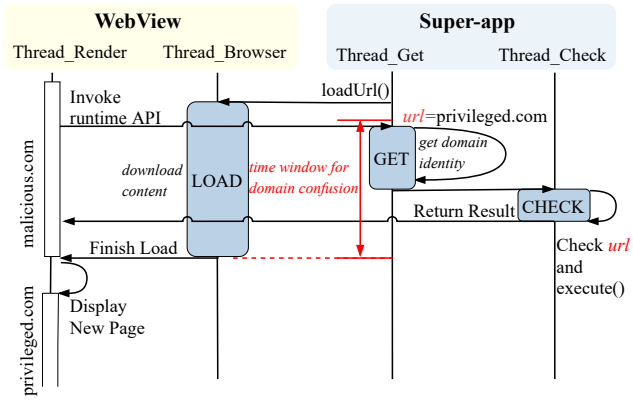

Figure 6: [Domain Name Confusion: Type 1] An illustration of timing-based confusion using loadUrl() API due to race conditions between WebView's Rendering (called Render) and Loading (called Browser) Threads. Although GET and CHECK threads are separate in the figure, they can reside in the same one.

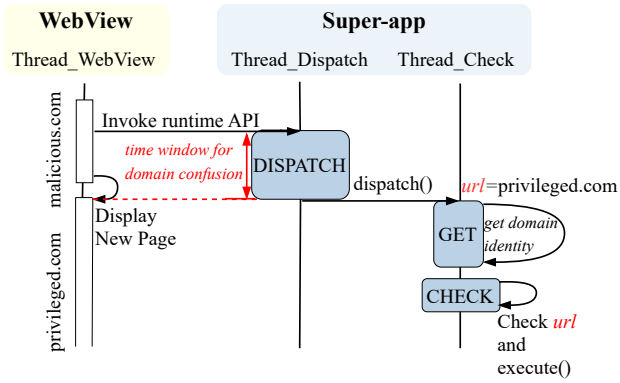

Figure 7: [Domain Name Confusion: Type 1] An illustration of timing-based confusion due to race conditions between super-app's dispatching (called Dispatch) and domain checking (called Check) Threads.

Case 2: Race between super-app's dispatch and checking threads. This race condition is summarized as follows. When super-app's dispatch thread receives a privileged API call, it does not check the identity but instead dispatches to an asynchronous queue. Then, when the checking thread fetches the API call from the queue and checks the identity, the identity obtained from the WebView is out of date.

Figure 7 illustrates such a race condition. First, the JavaScript from `malicious.com` in the WebView render thread calls a privileged runtime API. Next, the WebView thread passes the call to a super-app thread. Then, the super-app thread dispatches the API call to a queue without checking its identity. Next, the JavaScript in the WebView's render thread redirects the webpage to `privileged.com`. Lastly, when another super-app thread fetches the API call from the queue and executes it with identity checks, the obtained identity is `privileged.com` instead of `malicious.com`.

Note that when the super-app finishes the execution of the invoked API, the old webpage (i.e., `malicious.com`) cannot obtain the return value because the webpage is now `privileged.com`. Nevertheless, the attack still succeeds as the privileged API finishes its execution. For example, the `addDownloadTask` API can still download a malicious APK and the `mute` API can silence the mobile phone.

4.1.2 Type 2: Frame-based Confusion

The second type—called frame-based—is that an iframe acts on behalf of the top frame's identity. The reason is that many WebView's APIs and callback functions only return the top frame's URL when multiple sub-frames are embedded as part of a top frame. Then, no matter what identity checks a super-app adopts and how it performs such checks, the super-app can only obtain the top frame's identity if such APIs and callbacks are used. That is, an advertisement from

malicious.com embedded as an iframe of privileged.com can act on behalf of the latter.

We also perform a similar study as we do for timing-based confusions to understand the spread of frame-based confusions among WebView APIs and callbacks. Specifically, we create a webpage with an iframe from a different web domain and run these APIs and callbacks to determine whether they will return multiple domains. Table 3 shows the results: eight out of 14 APIs and callbacks are vulnerable to frame-based confusions. This list includes commonly-used ones like onPageStarted, onPageFinished, and getUrl.

4.2 App ID Confusion

An AppID confusion is that a malicious domain with a privileged AppID invokes a privileged runtime API, thus confusing the super-app's identity checks. We call it AppID confusion because the malicious domain has the correct AppID, but the domain itself is malicious. The key step for AppID confusion is to load a malicious domain within a privileged sub-app. In practice, we find three cases of such AppID confusions in loading malicious URLs into privileged sub-apps.

Type 1: Flawed URL whitelist matching. This flaw is that the URL whitelist used for loading is flawed, thus being able to allow potential malicious URLs to load. The deep reason is the lack of coordination and proper documents between super-app and sub-apps. Specifically, the URL whitelist checking algorithm is provided by the super-app, but the whitelist is provided by the sub-app. Therefore, a misunderstanding of the check algorithm often leads to flaws and we list two scenarios here.

- endswith being misunderstood as strict matching. In this scenario, the super-app provides endswith as the matching algorithm, but the sub-app developer thinks it is a strict matching. Therefore, when the sub-app uses benign.com in the whitelist, an adversary can bypass the check using a domain like maliciousbenign.com.
- Regular expression (regex) being misunderstood as strict matching. In this scenario, the super-app uses regex in the matching, but the sub-app developer still thinks it is a strict matching. Therefore, when the sub-app uses benign.a.com, the dot matches an arbitrary character. That is, an adversary can bypass the check using a domain like benignXa.com.

Type 2: Flawed URL parsing. This flaw is that super-apps have logic errors in parsing URLs and extracting web domains. We listed two types of parsing errors.

- Username and password fields. This parsing error is that the super-app does not recognize username and password fields or a URL. Take https://benign.com:x@malicious.com as an example. A logic error is to extract benign.com as the domain name instead of malicious.com.
- JavaScript protocol. This parsing error is that the super-app does not recognize JavaScript as a protocol. Thus, an attacker can use the URL javascript://payloads to exploit the URL parsing, resulting in code injection attacks and the loading of arbitrary domains.

Type 3: Missing URL checks. This flaw is that super-apps do not check web domains when a sub-app loads a third-party URL into either an iframe or a top frame. Therefore, an adversary can either embed a malicious URL as an advertisement or trick the top frame into visiting a malicious URL and then accesses privileged APIs, such as reading user contacts.

4.3 Capability Confusion

A capability confusion is that the privileged capability used for protecting runtime APIs is leaked to a malicious entity. Specifically, we find two cases of capability confusion: unprotected client-side and server-side APIs.

Type 1: Unprotected client-side API. This flaw is that super-apps use a hidden, unprotected API to transfer capabilities. The super-app assumes that the API is undocumented and will not be used by an adversary, but the API can be reverse-engineered from privileged sub-apps and used by an adversary. It is worth noting that hidden APIs are a widespread problem in super-apps, which takes up to about half of all the runtime APIs (details are in Figure 3 of §5).

Type 2: Unprotected server-side API. This flaw is that a privileged sub-app server exposes unprotected APIs to sign an invocation request that can be accepted by a super-app. Specifically, here is how the attack works. malicious.com first sends a request to the sub-app's back-end servers to sign the invocation request and then forwards the request to super-apps. Because the request is signed by a privileged sub-app server, the super-app will allow the API invocation. Our motivating example in Figure 4 has such a flaw.

5 Measurement: Prevalence & Consequence

In this section, we describe our measurement methodology and results in analyzing the prevalence and consequence of identity confusion vulnerabilities of existing super-apps. We also give a few case studies at the end.

5.1 Methodology

In this subsection, we describe our overall measurement methodology, which has three semi-automatic steps with manual efforts.

5.1.1 Step I: Super-app Discovery

In this step, we use a semi-automatic method to discover more super-apps beyond these that we find in §2. The high-level insight is that super-apps often define many templates (e.g., "miniapp0" and "miniapp1" for process names, and "AbsMakePhoneCallApiHandler" and "AbsChooseAddressApiHandler" for class names) in running different sub-apps, which can be used to find app-in-app ecosystem.

Specifically, there are three sub-steps. First, we use static analysis by utilizing Soot [9], to identify apps with WebViews and JavaScript bridges, e.g., detecting whether they re-implement `addJavaScriptInterface`. Second, we conduct a class name similarity analysis to find super-app runtimes. Specifically, we collect the class name of the activity which contains WebView instances and keep those apps containing at least five similar (i.e., sharing keywords) class or process names as potential super-apps. Then, we use the keyword-based package name matching to filter ads-related WebView instances, which may have bridge implementation but are not sub-apps runtime. Third, we manually verify whether they are truly super-apps with app-in-app ecosystem.

5.1.2 Step II: Vulnerability Analysis

In this step, we analyze each super-app for different identity confusion vulnerabilities. The high-level idea is that we manually write test cases and exploits by the identity confusion taxonomy and check the existence of each vulnerability. Although our analysis is performed on Android, we use the Proof of Vulnerability (PoV) for Android versions of super-apps to verify whether their iOS versions are also vulnerable.

Domain name confusion analysis. The analysis has two major steps: (i) determination of whether a vulnerable API or callback in Table 3 is used, and (ii) manually generating exploits triggering the vulnerability. Let us start with the first step. There are two types of WebViews: Android WebView or iOS counterpart, and customized WebView (e.g., UCWebView). If it is an unchanged WebView, we can directly look up Table 3; otherwise, if it is a customized WebView, we will perform an analysis (as we did in §4.1.1) to determine the vulnerable implementation on API or callback for this WebView.

Second, we create a malicious webpage that invokes privileged runtime APIs with an endless loop, and let the webpage trigger the event handlers, e.g., jumping to a privileged domain. Next, if any of the privileged API executes successfully, it indicates that the super-app is vulnerable to domain name confusion. Let us use `onPageStarted` as an example to explain more clearly. Figure 8 illustrates the webpage we crafted for testing Microsoft Teams—This JavaScript starts a repeated asynchronous loop to invoke the privileged runtime API "getAuthToken" with "window.setInterval()", and then uses "window.location.href" to jump to `https://privileged.com` for triggering WebView's event handler `onPageStarted`. Because there exists a race condition, the return result from the privileged API is accessed by the old page controlled by the adversary.

Note that we need to generate different exploit codes for `onPageStarted` in WebViews with a >72 Chromium kernel. The reason is that the time window for domain name confusion in Figure 5 becomes very small and the race condition is difficult to trigger. Specifically, we ask the webpage to load an error URL (i.e., either a very long URL or an unregistered URL scheme), making WebView execute its error-handling code, thus enlarging the time window of race condition for `onPageStarted`.

```
1  //JavaScript
2  window.setInterval(function(){
3  res = nativeInterface.framelessPostMessage('
   {"id":1,"func":"authentication.getAuthToken","
   args":[["privileged.com"]]}');
4  //res can be leaked to malicious server
5  ... ...
6  },1500);
7  window.location.href = "https://privileged.
   com/";
```

Figure 8: Example for verifying domain name confusion. The getAuthToken is a privileged API of Microsoft Teams. This figure exhibits a race condition: although the webpage is set to navigate to `https://privileged.com`, and so does the domain name, the code is still executed under the old context before the new page is loaded. The return value is accessed by the old page controlled by the adversary during the small interval.

AppID confusion analysis. This analysis checks whether an adversary can ask a super app to load any malicious domain in a sub-app. Specifically, we create a sub-app and set a whitelist for benign domains. Then, we generate a variety of URLs by mutating several initial seeds. Next, we randomly select URLs that cannot match the whitelist to test the sub-app. During the test, we hijack the network traffics and return the same webpage we crafted for invoking privileged runtime APIs, when requesting these URLs. Thus, if any of these URLs is successfully loaded and the JavaScript executed, an AppID confusion is confirmed.

Capability confusion analysis. This analysis checks whether the API of transferring secret is exposed. Specifically, we first collect ten sub-apps for each super-app. Then, we check how they invoke privileged APIs: (1) we first conduct a backward control-flow analysis and extract the adjacent API calls before the invocation of privileged APIs; (2) we cluster the extracted API calls; and (3) if there exists an API which always is invoked before privileged APIs, it may be a secret API. Then, we further check it with manual verification.

5.1.3 Step III: Consequence Analysis

In this step, we analyze the security consequence for each vulnerable super-app. In practice, we find three kinds of such consequences of identity confusion vulnerabilities. We now explain our analysis methodology (mostly manual) below:

- *Privilege Escalation.* We manually inspect whether an adversary can access privileged APIs after successfully confusing the super-app and disguising itself as a privileged identity. We consider the consequence that exists if the adversary can access at least one of such APIs.

Table 4: # of remaining apps in locating super-apps after applying each filter.

Filtering Methods	# Super-apps
Filter 1: Containing WebView	5,436
Filter 2: Redefining `addJavaScriptInterface`	3,463
Filter 3: Class name clustering	291
Filter 4: Manual analysis	47

Table 5: Breakdown of Identity Confusion Vulnerabilities of 47 Super-apps

Identity Confusion		# Super-apps	Examples
Domain	Type 1: Timing-based	15	WeChat, Alipay
	Type 2: Frame-based	15	Microsoft Teams, Go-Jek
	Total	15	
AppID	Type 1: Flawed matching	26	TikTok, Baidu
	Type 2: Flawed parsing	2	WeChat, Go-Jek
	Type 3: Missing checks	10	Microsoft Teams, UnionPay
	Total	38	
Capability	Type 1: Client-side	1	UnionPay
	Type 2: Server-side	1	WeChat
	Total	2	
No identity checks		9	Snapchat, Kuaishou
Total		47	

- *Phishing.* We manually inspect whether a sub-app can load web contents from a malicious domain, with phishing contents.
- *Privacy Leaks.* We manually inspect whether an adversary (e.g., a malicious domain) can access sensitive user data, e.g., via some privileged APIs like `getContacts`.

5.2 Measurement Results

In this part, we describe our measurement results. The first step gives us 47 super-apps: The number of remaining apps after applying each filter is shown in Table 4. Next, we describe the results from Steps II and III.

5.2.1 Vulnerability Prevalence

Table 5 illustrates the overall results of our vulnerability analysis. It shows that they (both the Android and iOS versions) are *all* vulnerable to at least one type of identity confusion attack despite the diversity in identity checks used in these super-apps. Here are the breakdowns. Nine super-apps adopt no identity checks at all, thus being all vulnerable; all 38 super-apps with AppID checks are vulnerable; all 15 super-apps with domain name checks are vulnerable; two super-apps with capability checks, all are vulnerable. Note that the overlaps are because one super-app may adopt more than one identity check.

Additionally, there are two things worth noting for domain confusion vulnerabilities. First, we evaluate the security of customized WebViews used by some super-apps and show the results in Table 6. To summarize, despite the customization, they have the same vulnerabilities as Android's WebView.

Table 6: The customized WebViews affected by [Domain Name Confusion: Type 1], including the iOS's WebView. We collect the iOS version of these 47 super-apps.

Platform	WebView	Domain Name Confusion	Affected Apps
Android	UCWebView	✔	10
	Tencent TBS	✔	4
	Baidu T5	✔	6
	ToutiaoWebview	✔	4
	KsWebView	✔	2
	others	✔	2
iOS	WKWebView	✔	47

Table 7: The system WebViews of stock Android with the latest security patches. $\star$ means attackers should use an error URL to exploit the identity checks implemented on WebView API `onPageStarted`.

Android	Patch Level	WebView's Chromium Version	Domain Name Confusion
< Version 72			
Android 6	2017-10-01	52	✔
Android 7	2019-03-01	51	✔
Android 8	2019-06-05	61	✔
Android 9	2019-08-01	66	✔
> Version 72			
Android 10	2019-09-05	74	✔*
Android 11	2021-07-05	83	✔*

Table 8: Breakdown of Identity Confusion Consequences of 47 Super-apps

Consequences	# Super-apps	Examples
Privilege Escalation	38	Go-Jek, Grab
Phishing	31	TikTok, WeChat
Privacy Leaks	35	Alipay, Microsoft Teams

Second, we evaluate WebViews with Chromium as its kernel in the stock Android from version 6 to the latest 11. The results in Table 7 show that they are all vulnerable.

5.2.2 Vulnerability Consequence

Table 8 illustrates the overall results of our consequence analysis. It shows that such confusion vulnerabilities can lead to phishing, privacy leaks, and privilege escalation. Here are breakdowns: (i) 38 super-apps are vulnerable to privilege escalation; (ii) 31 phishing; and (iii) 35 privacy leaks. This demonstrates the severity of identity confusion.

Interestingly, during our manual inspection, we also find some security consequences that are independent of identity confusion. We list three types below:

- *Permission re-delegation.* When a benign domain applies for permission and the user grants it, the super-app will give this permission to the sub-app, but not the domain. Then, any other domain, e.g., `malicious.com`, in this sub-app can use this permission. We find and confirm that 21 super-apps have this vulnerability.
- *Data leakage.* It is the disclosure of sensitive information to an adversary, such as token and account information. The reason is that a sub-app does not check the destination webpage when sending sensitive data. For example, an

Table 9: The overall result of our flaws detection tool tested on the total 47 super-apps. Symbol "∅" means the host app does not have this type of security enforcement. ✔ means it is vulnerable to our attack.

#ID	Super-app	Domain Name Confusion		AppID Confusion	Capability Confusion	Security Consequences		
		Time-based	Frame-based			Privilege Escalation	Phishing Attack	Privacy Leaks
01	TikTok	✔	✔	✔	∅	✔	✔	✔
02	WeChat	✔	✔	✔	✔	✔	✔	✔
03	Snapchat	∅	∅	∅	∅	✔		✔
04	Kuaishou	∅	∅	∅	∅			
05	Alipay	✔	✔	✔	∅	✔	✔	✔
06	Line	∅	∅	∅	∅	✔		✔
07	UC Browser	✔	✔	✔	∅	✔	✔	✔
08	Baidu	∅	∅	✔	∅	✔	✔	✔
09	JinRiTouTiao	∅	∅	✔	∅		✔	
10	Microsoft Teams	✔	✔	✔	∅	✔		✔
11	Grab	∅	∅	∅	∅	✔		✔
12	VK	∅	∅	✔	∅	✔		✔
13	Paytm	∅	∅	✔	∅	✔		✔
14	Go-Jek	✔	✔	∅	∅	✔	✔	✔
15	UnionPay	✔	✔	✔	✔	✔		✔
16	Kugou	∅	∅	∅	∅			
17	QQ	∅	∅	✔	∅	✔	✔	✔
18	JingDong	∅	∅	✔	∅	✔	✔	✔
19	DingTalk	✔	✔	✔	∅	✔	✔	✔
20	Quark Browser	✔	✔	✔	∅	✔	✔	✔
21	Youku	✔	✔	✔	∅	✔	✔	✔
22	Cainiao	✔	✔	✔	∅	✔	✔	✔
23	Taobao	✔	✔	✔	∅	✔	✔	✔
24	Koubei	✔	✔	✔	∅	✔	✔	✔
25	Gaode	✔	✔	✔	∅	✔	✔	✔
26	iQIYI	∅	∅	✔	∅	✔	✔	✔
27	Tieba	∅	∅	✔	∅	✔	✔	✔
28	Baidu Map	∅	∅	✔	∅	✔	✔	✔
29	XiaoHongShu	∅	∅	✔	∅	✔		✔
30	KanDuoDuo	∅	∅	✔	∅	✔	✔	✔
31	Baidu Netdisk	∅	∅	✔	∅	✔	✔	✔
32	Haokan	∅	∅	✔	∅	✔	✔	✔
33	Meituan	∅	∅	✔	∅	✔		
34	NetEase Cloud Music	∅	∅	∅	∅			
35	Feishu	∅	∅	✔	∅		✔	
36	Yippi	∅	∅	∅	∅	✔		
37	Dianping	∅	∅	✔	∅	✔		
38	Kuaishou-Mini	∅	∅	∅	∅			
39	JinRiTouTiao-Mini	∅	∅	✔	∅		✔	
40	Tiktok-Mini	∅	∅	✔	∅		✔	
41	Suning Finance	∅	∅	∅	∅			
42	QQ-Mini	∅	∅	✔	∅	✔	✔	✔
43	BaiduBaiKe	∅	∅	✔	∅	✔	✔	✔
44	Baidu Browser	∅	∅	✔	∅	✔	✔	✔
45	BaiduHanYu	∅	∅	✔	∅	✔	✔	✔
46	Baidu-Mini	∅	∅	✔	∅	✔	✔	✔
47	YiLu	✔	✔	✔	∅	✔	✔	✔

adversary can craft a deep link with the victim app ID and a malicious URL, and then the sub-app will leak sensitive user data to the malicious URL controlled by the adversary. We randomly collected 200 popular sub-apps and found that 21.5% of them are vulnerable to this attack.

- *Data over-collection.* Data over-collection is when a super-app collects more data than it needs from a sub-app, leading to a privacy concern. Specifically, we find that WeChat hooks all the requests coming from sub-apps, which include sub-app sensitive data, and sends them to WeChat's server.

5.3 Results and Case Studies

In this subsection, we present the overall results and perform case studies of some identity confusion vulnerabilities. Table 9 shows the statistics of collected 47 super-apps, including the vulnerability types and security consequences. Nine super-apps adopt no identity checks, thus all being vulnerable to our attack. Several super-apps have privileged hidden APIs (e.g., "fetchAuthToken" in Snapchat) without any identity checks, thus being vulnerable to privilege escalation or privacy leaks. Kuaishou, Kugou, NetEase Cloud Music, and Suning Finance have little API support and none of them is privileged according to our manual analysis. JinRitouTiao has an AppID confusion, but it redirects the hidden API invocation to another sandbox WebView restricting the actual API calls. Thus, we failed to launch privilege escalation or exploit privacy leaks.

Now, we illustrate two specific interesting examples.

Example 1 [Alipay]: Manipulating Super-apps' Backend Servers. The first example is the domain name and AppID confusions of Alipay, the most popular payment app in China

Table 10: The trigger conditions of WebView's error codes.

WebViewErrorCode	Trigger Condition
ERR_CLEARTEXT_NOT_PERMITTED	Set usesCleartextTraffic as false
ERR_NAME_NOT_RESOLVED	Use a wrong sub-domain name
ERR_CONNECTION_CLOSED	Use long URL, e.g., > 4,000 chars
ERR_UNKNOWN_URL_SCHEME	Use unregistered scheme, e.g., Htttp

with about 690 million downloads. An adversary can further manipulate Alipay's backend servers by exploiting them.

Let us start with identity confusion vulnerabilities. First, Alipay is vulnerable to domain name confusion due to race conditions of a customized WebView called UCWebView. Second, sub-apps of Alipay have AppID confusion due to a flaw in Alipay's URL whitelist matching. Particularly, Alipay uses regular expression on string matching, but many sub-apps think it is a strict matching and add domain names directly to their whitelist.

Next, we describe the security consequences of Alipay's identity confusion. Alipay only checks AppID for any privileged API calls and therefore an adversary can access any privileged API after successful identity confusion exploitation. Specifically, Alipay has about 781 undocumented but accessible APIs as shown in Figure 3. One of them, namely "rpc()", is privileged and can access Alipay's backend cloud sever. Note that this API is designed to be only used by Alipay itself, but in fact, it can be accessed by any sub-app.

Now take a sub-app, 1688, an online wholesale market managing over 920,000 virtual shops, for example, to illustrate the attack and consequence. An attacker can first craft a phishing deep link, e.g., `alipays://platformapi/startapp?appId=[1688]&url=malicious.com...`. Then, when a mobile user clicks the link, the 1688 sub-app will start and execute the malicious JavaScript from `malicious.com`, which invokes the API "rpc()" to access Alipay's cloud servers, e.g., managing user's financial and account data.

Example 2 [TikTok]: Bypassing Security Patches with an Error URL. This second example is the AppID and domain name confusions of TikTok, a popular social app with about 18 billion downloads. The app ID confusion is from the matching of URLs using `endswith` as we discussed in §4.2. Then, the domain name confusion is from the check implemented on customized WebView being vulnerable to the race condition of `onPageStarted`. We reported the vulnerability to TikTok, which then deployed a patch to update its chromium kernel to the latest. However, the patch is still vulnerable because we can utilize an error URL, delay the webpage rendering, and enlarge the time window for the race condition. Note that we further analyzed all WebView's error codes, and found four of them can be easily triggered by attackers as shown in Table 10.

Here are the detailed steps to exploit TikTok's domain name confusion. First, attackers create a malicious webpage, which abuses `benign.com`'s identity by executing the JavaScript "window.location.href = htttp://maliciousbenign.com". Since "htttp" is not a supported scheme, this URL will trigger the race condition of `onPageStarted`.

6 Lessons learned, Mitigation and Discussion

The most important lesson learned from our research is that the identity checks of sub-apps (e.g., for allowing sensitive API invocations) should follow the *least privilege principle*. That is, the definition of identity in the app-in-app ecosystem needs to be atomic, providing clear coordination between developers of super-apps, sub-apps, and WebView.

From our point of view, the atomic definition free of identity confusions is a combination of all three identities used in the wild, i.e., domain name, sub-app ID, and capability. The former two provide a definition of an atomic unit in an Access Control List (ACL), and the latter provides a capability in invoking specific privileged APIs. Specifically, when a sub-app tries to invoke a privileged API of a super-app, the sub-app will provide a secret signed by the private key from the sub-app's server like a digital signature. Next, the super-app obtains the secret using the public key and then verifies the secret, the domain name, and the sub-app ID before allowing the invocation.

Other than the atomic identity definition, the mitigation of identity confusions will also benefit from a domain synchronization between the mobile and web layers of WebView. The mobile code should be empowered to transparently obtain the correct, synchronized, up-to-date domain of any frame in WebView. Draco [42] provides a good example of such a domain synchronization. Specifically, Draco modifies the native code of WebView and supports JavaScript to send the domain information from the render thread. We believe that such a practice should be integrated into the mainstream design of WebView.

Last but not least, sub-app developers should also pay more attention to its security, especially on sensitive but exposed interfaces like the launching webpage. They should also carefully read the documents of super-apps to understand the security checks, e.g., URL whitelisting.

Ethics. We discuss ethical issues of our study, including vulnerability disclosure and experimental setups. First, we have informed all the 47 super-apps of their vulnerabilities. Currently, 29 super-apps have confirmed their vulnerabilities, and 19 have already fixed them. Take Alipay, for an example. We had regular monthly meetings with their developers for half a year. In the end, Alipay not only fixed the vulnerability but also rewarded us $2,500 as part of their bug bounty program. Second, all the attacks are tested on our own devices with our test accounts, which does not harm sub-apps, super-apps, or any of their servers.

7 Related Work

App-in-App Ecosystem. Recent years witnessed several techniques to support app-in-app ecosystems, such as web

apps, hybrid apps, instant apps, and virtual apps. Numerous studies [14, 27, 30, 41, 44, 46, 49–51] have looked into their designs, prevalence, usages, and flaws. For example, DCV-Hunter [46] focuses on differential context vulnerabilities for hybrid apps. Lee et al. [27] investigate privacy issues and side-channel flaws in progressive web apps. MIAFinder [41] studies the link hijacking attacks to instant apps. Zhang et al. [49] reveal the weak isolation between different virtual apps. Lu et al. [30]. focus on analyzing the resource management flaws of app-in-app. Zhang et al. [51] design and implement a novel, scalable crawler, called MiniCrawler, to index over 1.3 million WeChat mini-apps and measure their aggregated statistics, such as resource consumption, API/library usage, obfuscation rate, and app categorization/ratings. As a comparison, our paper focuses on a special type of vulnerability, called identity confusion, with a different threat model from prior works, which has not been studied before.

WebView Security. WebView is becoming a widely-used component for loading web contents in mobile apps and has been studied by many research works [6, 21, 25, 26, 29, 31, 32, 36, 39, 40, 43]. For example, Jin et al. [25], Li et al. [29], Wang et al. [43] show that attackers can inject malicious code into victim apps by exploiting insecure app communication channels (e.g., scheme and intent) in WebView-based hybrid apps. Son et al. [39] analyze WebView-based advertisement apps and find that malicious ads can hijack mobile apps. As a comparison, our work focuses on identity confusion vulnerabilities, e.g., how super-apps protect their APIs in WebView-based sub-app runtime and whether the protection is insecure.

Past works also study the race condition attacks in WebView. Lau et al. [26] present a semi-automated approach to analyze the concurrency flows in the PhoneGap framework and discover event-based race conditions of JavaScript APIs. Another research work [6] also reports several race conditions in WebView's event handlers. As a comparison, our threat model is different from theirs because our sub-app may also include third-party resources. More importantly, our domain name confusion part is much broader research on the WebView's event handlers, which demonstrates the root causes in design flaws and shows more varieties of exploiting such vulnerabilities. Moreover, we also introduce a measurement study to reveal how these event handlers affect the identity checks of real-world mobile apps.

Identity Checks. Many research works investigate identity check flaws in mobile and web apps. We start from the mobile part. Smalley et al. [38] demonstrate the limitation of UID-based Discretionary Access Control (DAC) and bring much more complicated Mandatory Access control (MAC) to the mobile system. Hernandez et al. [24] analyze the issues of enforced security policies. We then describe web apps and their connection with mobile systems. Prior works [15–18, 28, 45, 47] focus on the security issues among multi-origin web pages. NoFrak [23] points out the importance of protecting the web-to-mobile bridge. Then, Draco [42], MobileIFC [37], WIREframe [22], and HybridGuard [34] present frameworks to extend the same origin policy (SOP) to protect web-to-mobile bridges in hybrid applications and enforce fine-grained access control mechanisms. Moreover, prior works [19, 20] discover additional flawed URL parsing and matching examples in different scenarios, such as email senders.

As a comparison, such app-level identification (e.g., UID-based permission validation) and domain-based (e.g., cross-site validation) authorization in mobile apps are different from identity check problems in sub-apps of an app-in-app ecosystem. Specifically, it is much more complicated for WebView based app-in-app ecosystem to integrate both app-level identification and domain verification.

8 Conclusion

In this paper, we perform the first systematic study of so-called identity confusions in real-world app-in-app ecosystems. We categorize and taxonomize existing identity confusions into three types—domain name, app ID, and capability confusions—based on the identity check adopted in the app-in-app ecosystem. Such identity confusion could lead to severe consequences such as manipulating users' financial accounts and malware installation on smartphones. Then, we study 47 most popular super-apps supporting app-in-app ecosystems and find that they are all vulnerable to at least one type of aforementioned identity confusion. We also responsibly report all of the vulnerabilities to corresponding super-app developers.

Acknowledgement

We would like to thank the anonymous reviewers for their insightful comments that helped improve the quality of the paper. This work was supported in part by National Science Foundation (NSF) under grants CNS-20-46361 and CNS-18-54001, National Natural Science Foundation of China (U1736208, U1836210, U1836213, 62172104, 62172105, 61972099, 61902374, 62102093, 62102091), Natural Science Foundation of Shanghai (19ZR1404800), and China Postdoctoral Science Foundation (BX2021079, 2021M690706). Yuan Zhang was supported in part by the Shanghai Rising-Star Program under Grant 21QA1400700. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF. Min Yang is the corresponding author, and a faculty of Shanghai Institute of Intelligent Electronics & Systems, Shanghai Institute for Advanced Communication and Data Science, and Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, China.

References

[1] China bytes vol. 1: Wechat, new trends and chinese wisdom. https://reurl.cc/L769ka Accessed October 6, 2021.

[2] Getting mini program code. https://developers.weixin.qq.com/miniprogram/dev/framework/open-ability/qr-code.html Accessed October 6, 2021.

[3] Getting url link. https://developers.weixin.qq.com/miniprogram/dev/framework/open-ability/url-scheme.html Accessed October 6, 2021.

[4] Line, messenger app. https://line.me/en/ Accessed October 6, 2021.

[5] Microsoft teams on google play. https://play.google.com/store/apps/details?id=com.microsoft.teams Accessed October 6, 2021.

[6] Mind the bridge — new attack model in hybrid mobile application. https://conference.hitb.org/hitbsecconf2021ams/materials/D2T1%20-%20A%20New%20Attack%20Model%20for%20Hybrid%20Mobile%20Applications%20-%20Ce%20Qin.pdf Accessed October 6, 2021.

[7] Number of available applications in the google play store from december 2009 to december 2020. https://reurl.cc/ox5Oj3 Accessed October 6, 2021.

[8] Pagoda company profile. https://www.pagoda.com.cn/en Accessed October 6, 2021.

[9] Soot. https://github.com/soot-oss/soot Accessed October 6, 2021.

[10] Wandoujia. https://www.wandoujia.com/ Accessed October 6, 2021.

[11] Webview. https://developer.android.com/reference/android/webkit/WebView Accessed October 6, 2021.

[12] Wkwebview, apple development documentations. https://developer.apple.com/documentation/webkit/wkwebview Accessed October 6, 2021.

[13] Xposed module repository. https://repo.xposed.info/ Accessed October 6, 2021.

[14] Yasemin Acar, Michael Backes, Sven Bugiel, Sascha Fahl, Patrick McDaniel, and Matthew Smith. Sok: Lessons learned from android security research for appified software platforms. In 2016 IEEE Symposium on Security and Privacy (SP), pages 433–451. IEEE, 2016.

[15] Yinzhi Cao, Zhichun Li, Vaibhav Rastogi, Yan Chen, and Xitao Wen. Virtual browser: a virtualized browser to sandbox third-party javascripts with enhanced security. In Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security, pages 8–9, 2012.

[16] Yinzhi Cao, Vaibhav Rastogi, Zhichun Li, Yan Chen, and Alexander Moshchuk. Redefining web browser principals with a configurable origin policy. In 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pages 1–12. IEEE, 2013.

[17] Yinzhi Cao, Yan Shoshitaishvili, Kevin Borgolte, Christopher Kruegel, Giovanni Vigna, and Yan Chen. Protecting web-based single sign-on protocols against relying party impersonation attacks through a dedicated bi-directional authenticated secure channel. In International Workshop on Recent Advances in Intrusion Detection, pages 276–298. Springer, 2014.

[18] Yinzhi Cao, Vinod Yegneswaran, Phillip A Porras, and Yan Chen. Pathcutter: Severing the self-propagation path of xss javascript worms in social web networks. In NDSS, 2012.

[19] Jianjun Chen, Jian Jiang, Haixin Duan, Tao Wan, Shuo Chen, Vern Paxson, and Min Yang. We still don't have secure cross-domain requests: an empirical study of CORS. In 27th USENIX Security Symposium (USENIX Security 18), pages 1079–1093, 2018.

[20] Jianjun Chen, Vern Paxson, and Jian Jiang. Composition kills: A case study of email sender authentication. In 29th USENIX Security Symposium (USENIX Security 20), pages 2183–2199, 2020.

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

[22] Drew Davidson, Yaohui Chen, Franklin George, Long Lu, and Somesh Jha. Secure integration of web content and applications on commodity mobile operating systems. In Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, pages 652–665, 2017.

[23] Martin Georgiev, Suman Jana, and Vitaly Shmatikov. Breaking and fixing origin-based access control in hybrid web/mobile application frameworks. In Proc. of the Network and Distributed System Security Symposium (NDSS'14), 2014.

[24] Grant Hernandez, Dave Jing Tian, Anurag Swarnim Yadav, Byron J Williams, and Kevin RB Butler. Bigmac: Fine-grained policy analysis of android firmware. In 29th USENIX Security Symposium (USENIX Security 20), pages 271–287, 2020.

[25] Xing Jin, Xuchao Hu, Kailiang Ying, Wenliang Du, Heng Yin, and Gautam Nagesh Peri. 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, pages 66–77, 2014.

[26] Phi Tuong Lau. Static detection of event-driven races in html5-based mobile apps. In International Conference on Verification and Evaluation of Computer and Communication Systems, pages 32–46. Springer.

[27] Jiyeon Lee, Hayeon Kim, Junghwan Park, Insik Shin, and Sooel Son. 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, pages 1731–1746, 2018.

[28] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. Detecting Node.Js Prototype Pollution Vulnerabilities via Object Lookup Analysis, page 268–279. Association for Computing Machinery, New York, NY, USA, 2021.

[29] Tongxin Li, Xueqiang Wang, Mingming Zha, Kai Chen, XiaoFeng Wang, Luyi Xing, Xiaolong Bai, Nan Zhang, and Xinhui Han. 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, pages 829–844, 2017.

[30] Haoran Lu, Luyi Xing, Yue Xiao, Yifan Zhang, Xiaojing Liao, XiaoFeng Wang, and Xueqiang Wang. Demystifying resource management risks in emerging mobile app-in-app ecosystems. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, pages 569–585, 2020.

[31] Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin. Attacks on webview in the android system. In Proceedings of the 27th Annual Computer Security Applications Conference, pages 343–352, 2011.

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

[33] Patrick Mutchler, Adam Doupé, John Mitchell, Chris Kruegel, and Giovanni Vigna. A large-scale study of mobile web app security. In Proceedings of the Mobile Security Technologies Workshop (MoST), page 50, 2015.

[34] Phu H Phung, Abhinav Mohanty, Rahul Rachapalli, and Meera Sridhar. Hybridguard: A principal-based permission and fine-grained policy enforcement framework for web-based mobile applications. In 2017 IEEE Security and Privacy Workshops (SPW), pages 147–156. IEEE, 2017.

[35] Vaibhav Rastogi, Rui Shao, Yan Chen, Xiang Pan, Shihong Zou, and Ryan Riley. Detecting hidden attacks through the mobile app-web interfaces. In Proc. of the Network and Distributed System Security Symposium (NDSS'16), 2016.

[36] Claudio Rizzo, Lorenzo Cavallaro, and Johannes Kinder. Babelview: Evaluating the impact of code injection attacks in mobile webviews. In International Symposium on Research in Attacks, Intrusions, and Defenses, pages 25–46. Springer, 2018.

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

[38] Stephen Smalley and Robert Craig. Security enhanced (se) android: Bringing flexible mac to android. In Proc. of the Network and Distributed System Security Symposium (NDSS'13), 2013.

[39] Sooel Son, Daehyeok Kim, and Vitaly Shmatikov. What mobile ads know about mobile users. In Proc. of the Network and Distributed System Security Symposium (NDSS'16), 2016.

[40] Wei Song, Qingqing Huang, and Jeff Huang. Understanding javascript vulnerabilities in large real-world android applications. IEEE Transactions on Dependable and Secure Computing, 17(5):1063–1078, 2018.

[41] Yutian Tang, Yulei Sui, Haoyu Wang, Xiapu Luo, Hao Zhou, and Zhou Xu. All your app links are belong to us: understanding the threats of instant apps based attacks. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pages 914–926, 2020.

[42] Guliz Seray Tuncay, Soteris Demetriou, and Carl A Gunter. 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, pages 104–115, 2016.

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

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

[45] GuangLiang Yang, Jeff Huang, and Guofei Gu. Iframes/popups are dangerous in mobile webview: studying and mitigating differential context vulnerabilities. In 28th USENIX Security Symposium (USENIX Security 19), pages 977–994, 2019.

[46] GuangLiang Yang, Jeff Huang, and Guofei Gu. Iframes/popups are dangerous in mobile webview: studying and mitigating differential context vulnerabilities. In 28th USENIX Security Symposium (USENIX Security 19), pages 977–994, 2019.

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

[48] Guangliang Yang, Abner Mendoza, Jialong Zhang, and Guofei Gu. Precisely and scalably vetting javascript bridge in android hybrid apps. In International Symposium on Research in Attacks, Intrusions, and Defenses, pages 143–166. Springer, 2017.

[49] Lei Zhang, Zhemin Yang, Yuyu He, Mingqi Li, Sen Yang, Min Yang, Yuan Zhang, and Zhiyun Qian. App in the middle: Demystify application virtualization in android and its security threats. Proceedings of the ACM on Measurement and Analysis of Computing Systems, 3(1):1–24, 2019.

[50] Xiaohan Zhang, Yuan Zhang, Qianqian Mo, Hao Xia, Zhemin Yang, Min Yang, Xiaofeng Wang, Long Lu, and Haixin Duan. An empirical study of web resource manipulation in real-world mobile applications. In 27th USENIX Security Symposium (USENIX Security 18), pages 1183–1198, 2018.

[51] Yue Zhang, Bayan Turkistani, Allen Yuqing Yang, Chaoshun Zuo, and Zhiqiang Lin. A measurement study of wechat mini-apps. In Proceedings of the 2021 ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems, 2021.

[52] Zicheng Zhang, Daoyuan Wu, Lixiang Li, and Debin Gao. On the usability (in)security of in-app browsing interfaces in mobile apps. In International Symposium on Research in Attacks, Intrusions, and Defenses, 2021.