The Dark Forest: Understanding Security Risks of Cross-Party Delegated Resources in Mobile App-in-App Ecosystems

Zhibo Zhang, Lei Zhang, Guangliang Yang, Yanjun Chen, Jiahao Xu, and Min Yang

***Abstract*— In app-in-app ecosystems, mobile applications (i.e., host apps) often delegate their rich resources to hosted parties (i.e., sub-apps), which can be utilized to provide millions of effective services including shopping, banking, and government. These resources vary from system abilities (e.g., web socket and GPS location) to app and user data (e.g., storage and phone number). This leads to an important research question—carefully design and enforce security regulations on these cross-party delegated resources (CPDR). Real-world host apps, according to our study, adopt 11 common security regulations in protecting the integrity, confidentiality, and availability of CPDR. However, existing practice and compliance between host apps and sub-apps are vague and inconsistent, leading to violations of these security regulations. To the best of our knowledge, no prior works have studied these security regulations. In this paper, we perform the first systematic study of the security regulations and their security weaknesses in real-world app-in-app ecosystems. We propose three novel attack vectors including masquerade attack, data-driven attack, and channel hijacking. We find that violations of the common security regulations are widespread among all 9 studied app-in-app ecosystems. More importantly, such security weakness can lead to severe consequences such as manipulating sub-apps' back-end servers and stealing sensitive user data. We responsibly report all of our findings to host app developers of affected app-in-app ecosystems and help them fix their vulnerabilities. The code of this work is available at https://github.com/TitaniumB/MiniAppSecurity.git.**

***Index Terms*— App-in-app ecosystem, security regulation, vulnerability analysis.**

Manuscript received 7 November 2023; revised 13 February 2024; accepted 3 April 2024. Date of publication 19 April 2024; date of current version 22 May 2024. This work was supported in part by the National Key Research and Development Program under Grant 2021YFB3101200; and in part by the National Natural Science Foundation of China under Grant 62172104, Grant 62172105, Grant 61972099, Grant 62102093, and Grant 62102091. The associate editor coordinating the review of this manuscript and approving it for publication was Dr. Angelo Spognardi. *(Corresponding author: Min Yang.)*

This work involved human subjects or animals in its research. Approval of all ethical and experimental procedures and protocols was granted by the Ethics Committee, School of Computer Science, Fudan University.

Zhibo Zhang, Lei Zhang, Guangliang Yang, Yanjun Chen, and Jiahao Xu are with the School of Computer Science, Fudan University, Shanghai 200433, China (e-mail: zhibozhang19@m.fudan.edu.cn; zxl@m.fudan.edu.cn; yanggl@m.fudan.edu.cn; yanjunchen20@fudan.edu.cn; jiahaoxu21@m.fudan.edu.cn).

Min Yang is with the School of Computer Science, Fudan University, Shanghai 200433, China, also with Shanghai Institute of Intelligent Electronics & Systems, Shanghai 200433, China, and also with the Engineering Research Center of Cyber Security Auditing and Monitoring, Shanghai 200433, China (e-mail: m_yang@fudan.edu.cn).

Digital Object Identifier 10.1109/TIFS.2024.3390553

I. Introduction

NOWADAYS, mobile applications (apps) bring significant convenience to people's work and daily lives. To provide better services and attract more users, high-profile mobile apps are involving numerous essential services; thus, allowing their users to do anything (e.g., online shopping and restaurant reservation) but without leaving these apps. To this end, some mobile apps–or called host apps–apply a new design paradigm, namely "app-in-app", that offers an in-app runtime, and provides rich functionalities to plentiful "sub-apps". Thus, complicated services, such as file management, one-click login, and in-app payment, can be broken down into smaller tasks that are dealt with by different sub-app participants. As of today, over 47 popular super-apps[1] (e.g., TikTok and Snapchat), as well as ten device vendors (e.g., Xiaomi and Huawei), offer app-in-app features. As a prominent example, WeChat, one of the most used apps in the world, has more than 3.8 million sub-apps with over 450 million daily active users [1], even much more than the apps in Google Play [2]. With the emerging popularity of app-in-app ecosystem,[2] its security becomes critical.

Figure 1 illustrates a typical architecture of the app-in-app ecosystem. Commonly, a host app[3] provides sub-apps with rich resources, which help sub-apps to implement a large number of effective online services, such as banking, education, government, shopping, and pandemic. We find these resources are often sensitive and diverse, including system functionalities and resources (e.g., web socket, GPS location, and microphone), and also user and app data (e.g., phone number, user account, and contact). One important research problem is ensuring the security of these cross-party delegated resources (CPDR) in the app-in-app ecosystem. However, there still lacks a thorough understanding of CPDR, and its security is rarely investigated. To this end, millions of users and crucial institutions are at risks, including one-click user account hijacking in TikTok [3], and potential data leakage in numerous banking services [4]. Motivated by the security concerns, we conduct the first systematic study on the app-in-app ecosystem and the security protection of CPDR. Nevertheless, this is not an easy task, as there exists no standard security practices for

[1] A super-app is often granted with a lot of system permissions and may have partnerships with the mobile device vendor.

[2] We define a community that develops and maintains sub-apps as the mobile app-in-app ecosystem.

[3] Without loss of generality, we use host app to refer to both super-app and device vendor that support app-in-app feature in the paper.

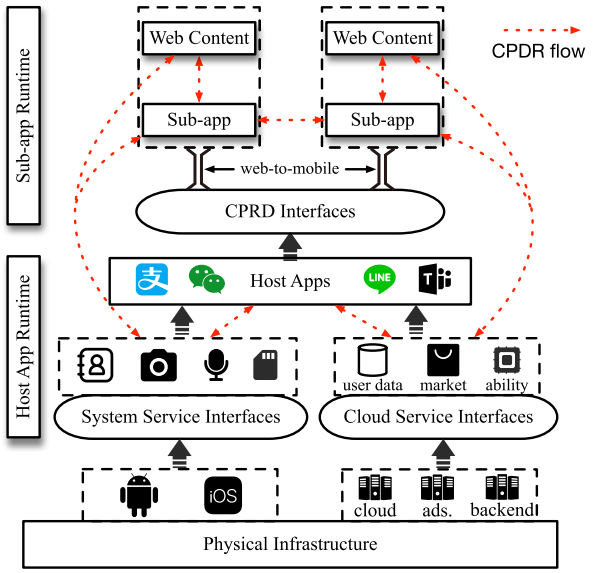

Fig. 1. Typical structure of app-in-app ecosystems with cross-party resource delegation.

TABLE I

TOP 9 POPULAR HOST APPS AND THEIR SUB-APP MARKETS. MAU INDICATES THE AVERAGE MONTHLY ACTIVE USERS

Ecosystem	Description	MAU	Market Size
TikTok	Video sharing.	1B+	-
WeChat	Instant messaging client.	1.3B+	3.8M+
KuaiShou	Video sharing.	600M+	-
Alipay	Third-party mobile and online payment platform.	1.2B+	2M+
Line	Instant messaging. Social networking service.	178M+	-
Baidu	News headlines. Hot video.	600M+	420K+
Microsoft Teams	Chat, meet, call, and collaborate in one app.	270M+	911+
Huawei Quick App	Preinstalled app of EMUI.	730M+	8.7K+
Xiaomi Quick App	Preinstalled app of MIUI.	500M+	1K+
Total	-	4.3B+	6.2M+

the protection of CPDR. Different host apps often enable and enforce totally different security implications. Worse still, the security regulations applied in host apps are often quite vague, as many regulations are essentially provided and deployed in the server side. In addition, a part of security responsibilities are left to sub-app developers, who often have inadequate security expertise or improper–in some cases–compliance with security regulations. Such differentiated and vague security practices make it difficult in protecting the CPDR in the app-in-app ecosystem.

To address the challenges, our first step is investigating and understanding the security regulations that should be followed and enforced by real-world host and sub-apps. One key observation is that although such security enforcement is often unknown and deployed in server side or require involvement of the sub-app, the code of host apps, as well as their development manuals (for sub-app development), can actually reflect the security implications of their app-in-app features. Therefore, we perform code reverse-engineering and documentation analysis, and conduct a comparative study on nine most popular app-in-app ecosystems (Table I). Consequently, we summarize and unify 11 common security regulations (CSRs) on CPDR from the perspectives of integrity, confidentiality, and availability (see §II-C for more details.).

Next, we further study whether there are security violations in practice. The fact is, beneath the similarity of resource delivery, security enforcement in different app-in-app ecosystems is inadequate and inconsistent, potentially causing serious security hazards. There exists security flaws in both sides of host apps and sub-apps:

- On the aspect of host apps, they fail to provide a sound and robust isolation and access control mechanisms in protecting the resources in multi-party environment. For instance, host apps rely on the in-app component (e.g., WebView) as the infrastructure of in-app runtime, which is intended to load trusted content but does not facilitate precise identity recognition [5]. Host apps cannot correctly determine whether the CPDR interface caller is authorized to enforce a right access control.
- On the aspect of sub-apps, due to the lack of security standardization and testing, their development and usage of CPDR interfaces do not also comply with security regulations, exposing sensitive resources to attackers. For instance, in the case of Baidu, it delegates an encrypted phone number to the sub-app using the CPDR interface called `getPhoneNumber` and requires the sub-app server to obtain the decryption key for decryption. However, sub-apps frequently decrypt in their front end for convenience, violating the back end decryption regulation, which can result in any account hijacking.

In summary, our analysis reveals seven types of security weakness, grouped into three attack vectors: (i) an unprivileged web content or sub-app can launch *masquerade attack* to gain privileged abilities (detailed in § IV); (ii) malicious web content or sub-app can launch *data-driven attack* for manipulating other sub-apps and steal their sensitive data (detailed in § V); and (iii) a man-in-the-middle attacker launches *channel hijacking* for accessing sensitive data and even tampering the sub-app's back-end servers (detailed in §VI).

To get close to the above attacks' security impact in real-world app-in-app ecosystems, we first measure the prevalence of security weaknesses with the help of differential analysis and penetration samples (detailed in § VII). We select nine representative[4] host apps and find that all of them have failed to enforce CSRs. On average, 27.5% of their sub-apps (out of a total of 202,273) become attack surface for loading malicious web content; besides, a half of sub-apps failed in compliance with CSRs may leak their sensitive data. More results can be found in § VIII.

We also perform case studies to illustrate the security consequences of these identified weaknesses and found that, the security consequences are profound and widespread.

- A weakness of host apps can compromise the security of millions of sub-apps. For example, with encryption failure of phone number in host app, sub-apps are vulnerable to account hijacking even if they enforce encryption in transmission to their server side.
- One small leak of sub-app can sink the great ship of host app. For example, when malicious web content is loaded into one vulnerable sub-app in Alipay, this web content can exploit privileged abilities of Alipay and access sensitive data stored in its back-end server.

[4]We choose host apps from manufacturers based on popularity, degree of development, and uniqueness. As detailed in subsection II-A.

More interestingly, our study shows that the insecure resource delegation brings significant bucket effects to app-in-app ecosystems. That is, a vulnerable delegation process could break the security assumption in another app-in-app ecosystem running the sub-app or mobile app developed by the same sub-app provider, leading to a more serious data leakage. For example, with a leaked login token of HuoLaLa sub-app, its mobile app version with the same authentication process is also vulnerable, thus attacker can exploit the app's service, which contains more functionalities including access to contact book, e-wallet, and even bank card.

A. Contributions

We summarize the contributions of this paper as below:

- *New understanding of the app-in-app ecosystems.* We perform the first systematic measurement study to demystify the complexity of CPDR in app-in-app ecosystem. We conclude 11 common security regulations with the goal of integrity, confidentiality, availability of sensitive data and privileged abilities.
- *Insecure resource delegation identification and verification.* We identify three novel attack vectors, containing seven security weaknesses in real-world app-in-app ecosystems. Such weaknesses can further lead to severe consequences such as ability re-delegation, data leakage, and privilege escalation. We further design and implement a penetration and differential analysis framework to verify the security weakness in CPDR.
- *Shed light on securing app-in-app ecosystems.* We thoroughly study why these insecure resource deliveries exist and help to secure the CPDR in app-in-app ecosystem.

B. This Submission is an Extension of Our Conference Paper [6]

In comparison, we present new contributions especially on the analysis of diversified security enforcement including: (a) new understanding of cross-party interaction in app-in-app ecosystems; (b) new understanding in security enforcement of summarized 11 common security regulations; (c) new attack surfaces and weaknesses found in sub-apps and host apps. (d) new detection method of certain security weaknesses for both host apps and sub-apps (old method in our previous work only work for identity confusion); (e) new security consequence analysis results for vulnerable mobile vendors, sub-apps, and remote servers;

II. Security Regulation Investigation on App-in-App

In this section, we present our security regulation investigation against existing app-in-app ecosystems. Our investigation purpose includes (i) understanding the resource delegation model of existing app-in-app ecosystems; and (ii) recovering the security regulations for the protection of CPDR.

A. Understanding App-in-App

In this section, we analyze the top-nine most popular app-in-app ecosystems (listed in Table I). We select these ecosystems based on the following criteria. First, we chose super-apps and device vendors with user bases exceeding 100 million, as they generally demonstrate a higher level of refinement in design and implementation. Second, the ecosystems have accessible and detailed sub-app developer documentation, which can help attract more sub-app developers to join their app-in-app community. Third, we select one representative host app for each app vendor, as multiple host apps from the same vendor typically share the similar architectural design.

Our study unifies the architecture of app-in-app ecosystems as well as CPDR across different parties (Figure 1), including host app, sub-app, and third-party web content. In general, a host app provides sub-app runtime with three major components: (i) an embedded web browser, (ii) CPDR interfaces provided by the host app, and (iii) web-to-mobile bridge.

First, the embedded browser instance (e.g., WebView[5] for Android and WKWebView [7] for iOS) provides an isolated environment for a sub-app, which is written in HTML, CSS and JavaScript. Such an embeded-browser instance often includes a customized worker to load and execute sub-app code downloaded from market. The sub-app code can further load web content from third-party servers such as advertisements and news articles, in an isolated render frame. Second, host apps provide rich CPDR interfaces to access to various system- and app-level resources (e.g., microphone, bank card info). For example, WeChat provides 917 customized interfaces, and sub-apps only need to import the JavaScript SDK provided by WeChat to call and access rich resources of WeChat. Third, the web-to-mobile bridge connects sub-app code with the native Java code. While details of the bridge vary among different host apps, a typical design encapsulates native Java API invocations from the web side into a message sent to the mobile side. For example, in Android, app can expose a dispatch method registered through `addJavaScriptInterface`. The mobile side parses the received message, finds the corresponding APIs, performs security checks, and allows the API invocations if the check passes.

B. Modeling CPDR

In this section, we dive into the model of CPDR. We perform reverse-engineering against the implementation and usage of CPDR interfaces in the host apps and sub-apps. In particular, we review the code of the nine host apps using decompilation tool (i.e., JADX [8]), and explore the CPDR interfaces using dynamic analysis. Our purpose is to understand the CPDR sensitivity and lifecycle. The results are depicted in Figure 1 with red dashed lines, and details are described below.

1) Resource Sensitivity: First, host apps often delegate system- and app-level resources to sub-app runtime (categorized in Table II). These resources can be divided into two classes: (i) ability and (ii) data. In particular, abilities allow sub-apps to access system services (e.g. camera), which are granted to host app with corresponding permissions (e.g. *Manifest.permission.CAMERA*). Furthermore, sub-apps can transmit or access user sensitive data, including phone numbers used in registration, health info, and even bank card info. Second, some CPDR are delegated in channels that connect sub-apps, web content, and remote servers. For example,

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

TABLE II
STATISTICS OF CPRDIS IN TOP 9 POPULAR APP-IN-APP ECOSYSTEMS

Ecosystem	Resources		Hidden APIs	Java APIs
	System	App		
TikTok	26	51	129	220
WeChat	55	27	336	917
KuaiShou	20	58	23	77
Alipay	31	42	781	1152
Line	2	15	37	45
Baidu	30	30	54	329
Teams	4	10	13	44
Huawei	54	33	309	619
Xiaomi	54	33	309	619

the CPDR interfaces "request" or "connect" allows sub-apps to establish the HTTP request or connect a WebSocket with a remote server; likewise, the "postMessage" interface provides the channel for sub-apps to receive message from web content.

More importantly, we find there exists many CPDR interfaces that can be accessed but undocumented. We manually sample 200 of these 'hidden' APIs and check the usage of them. Our analysis shows that at least 80% of them, i.e., 160 APIs, are privileged and should not be used by any sub-apps. For example, a hidden API—named "rpc"—can be used to access the host app's server-side interfaces, like manipulating user accounts. According to our study, we believe these abilities are supposed to be only accessed by the first-party or partner developers.

In comparison, all nine app-in-app ecosystems support delegation of access to sensitive abilities and data, including loading third-party web content, accessing hidden APIs, system storage, network resources, and user phone numbers. Note that different ecosystems may provide different name of CPDR interface. On the other hand, some hidden APIs are unique to individual ecosystems, such as the "rpc" API provided by Alipay.

2) Delegation Chain: Real-world CPDR involves multiple layers and can be cross-ecosystem. The delegation chain among multiple parties is often quite long and complex, involving several layers. These characteristics may pose new security hazards.

First, the CPDR are frequently accessed from one layer and subsequently passed across multiple layers, including (i) system layer, (ii) host app layer, (iii) sub-app layer, and (iv) cloud layer. Let us consider an example of the user login process in Pagoda,[6] which involves resource delegation among the host app (WeChat), the sub-app (Pagoda), and its web content (advertisements). Specifically, Pagoda has the ability to use user's phone number delegated from WeChat in the login process. And then, it can re-delegate this resource to dynamically-loaded web content for personalized advertising. Note that, the phone number is encrypted by WeChat, thus Pagoda needs delegate the encrypted data to its back-end server for decryption, using secret keys assigned from WeChat.

Second, the delegation chain may span across different ecosystems. That is, the sub-app server act as a link to facilitate the flow of resources in different app-in-app ecosystems. For instance, the purchase and payment info in Pagoda can be synchronized among different ecosystems of WeChat, Alipay, and its own Pagoda mobile app. In this scenario, the payment information, including the user address delegated from WeChat, is also passed to the sub-app running in Alipay. Unfortunately, the data isolation in Alipay is vulnerable and can be abused, which could potentially result in a data leak to a malicious attacker. Thus, the protections in different ecosystems should be consistent and robust against the increasing complexity of the delegation chain. However, our further security analysis verifies these is quite difficult in practice, which inevitably introduces significant security risks.

[6]A fruit retailer with more than 4,000 stores in China, and it has developed its own online shopping sub-app in WeChat and Alipay.

C. Unifying Security Regulations

After modeling CPDR, we perform a study on what common security regulations (CSRs) should be satisfied by sub-apps and host apps. As discussed above, the existing design and implementation of CPDR is multi-layer and cross-ecosystem. Furthermore, there lacks standard security practices, and many real-world protections are often ad-hoc. Developer documentations in different app-in-app ecosystems delineate distinct security guidelines for the secure utilization of resources.

Therefore, we generalize and unify the security regulations that should be conformed to in the app-in-app ecosystem. Below, we present our methodology of extracting security regulations. First, we analyze the security related descriptions from developer documentations provided by host apps, which indicate the secure usage of CPDR interfaces. These descriptions are frequently emphasized as tips beneath each API and contain directive phrases (such as "developer must" or "one should not"). Second, we manually review the code and explore the implementation of CPDR interfaces, which helps uncover undocumented security enforcement in host apps.

As a result, we successfully extracted and unified 11 CSRs from three aspects, including ability delegation control, data isolation, and channel security (Table III). At the micro level, the security regulations follow the *least privilege principle*. It considers CPDR as protected objects in a zero-trust environment, requiring permission and identity check for ability delegation, storage isolation between different parties, and also input validation. This fundamental principle ensures the CPDR could not be read, modified, or accessed by unauthorized parties. At the macro level, security regulations should be consistent across the delegation chain. First, it indicates the tighter or at least similar security regulations from traditional OSes and browsers. This includes the permission scope, protocol security, and the same origin policies. Second, different ecosystems with the similar delegation scenarios should be unified and consistent. To achieve this, host apps often provide cryptography protection on phone number, health data, and transmission channels.

Our study concludes several important findings: (i) **Collaborative efforts** of both the host app, sub-app, and remote servers are necessary. That is, a robust implementation of security enforcement are required in host apps, and secure usage of CPDR should be verified in sub-apps. (ii) **Customization** of security enforcement varies in different ecosystems. For example, host apps often implement different identity checks to secure CPDR interfaces. There exist two types of identity. One is a domain name as part of web origin, and the other is a sub-app ID (or called AppID for short) assigned by a host app. (iii) **Incomplete** security regulations are adopted in existing

TABLE III
THE SURVEY RESULTS OF COMMON SECURITY REGULATIONS

MajorCategory	MinorCategory	Summarized Description	# of Mentioned Ecosystem
Ability Delegation Control	Identity Check	R1. Sub-app should maintain the domain allowlist of web content and determine whether the loaded content is legal.	Alipay, WeChat, Baidu, TikTok
		R2. Host app should check the caller's identity of privileged CPDR interfaces and determine whether the invocation is legitimate.	Alipay, WeChat, Baidu, TikTok
	Permission Check	R3. Host app should enforce permission check for CPDR interfaces that can delegate sensitive OS resources.	Alipay, WeChat, Baidu, TikTok, KuaiShou
Data Isolation	Storage Isolation	R4. Host app should isolate the storage by userID and sub-app ID.	Alipay, TikTok
		R5. Host app should isolate the storage of sub-app and web content.	Alipay, Huawei, Xiaomi
		R6. Host app should isolate the storage of different web content following the same origin policies.	-
	Input Validation	R7. Sub-app should not trust the data delegated from other sub-apps/web content.	WeChat
Channel Security	Protocol Security	R8. Sub-app must use secure protocol to communicate with remote server.	Alipay, WeChat, TikTok
	Encryption	R9. Host app should encrypt the storage.	Alipay
		R10. Host app should encrypt and sign the user's private data.	Alipay, WeChat, Baidu, TikTok, KuaiShou,
		R11. Sub-app should decrypt and verify the integrity of delievered user's private data on server-side.	Alipay, WeChat, Baidu, TikTok, KuaiShou

app-in-app ecosystems. As a comparison, Alipay involves the most security regulations; in contrast, Huawei and Xiaomi lacks corresponding protection on identical CPDR. These characteristics indicate existing protection of CPDR are vague and fragile, and sensitive CPDR can be exposed to malicious parties if any of the CSRs are violated by host apps or sub-apps. As declared in TikTok [9], the violation of security regulations can jeopardize the confidentiality, integrity, and availability of app-in-app ecosystems.

III. SECURITY ANALYSIS: AN OVERVIEW

In this section, we present an overview of the security threats in CPDR, including three novel attack vectors, adversary scenarios, and attack roadmap.

A. Attack Vector Definition

The attack vectors appear when CSRs are violated during the implementation and usage of CPDR interfaces in real-world host apps and sub-apps. This is because the delegatee or delegator of CPDR may be malicious or compromised in the insecure delegation chain. Thus, according to the categories of CSRs, we propose three attack vectors: masquerade attack (Section IV), data-driven attack (Section V), and channel hijacking (Section VI). The definitions are described as follows:

- Masquerade Attack. This attack lets unprivileged party pretend to be privileged one, thus abusing privileged abilities. In an app-in-app ecosystem, masquerade attack happens because of a disobey of the least privilege principle. For example, a privileged sub-app is tricked into loading unprivileged web content, and host apps only check the permission granted to this sub-app. Consequently, the loaded unprivileged content finally obtains the abilities of the privileged sub-app.
- Data-driven Attack. This attack manipulates data received or stored by other parties, tampering with the code logic. This type of attack typically involves exploiting weaknesses in the data processing of sub-apps and data storing components of host apps. Consequently, malicious sub-apps or web content can exploit critical functionalities of victim sub-apps and steal their sensitive data.
- Channel Hijacking. This kind of attack lets man-in-the-middle attacker access data in cross-layer transmission channels, thus compromising their integrity and confidentiality. As a result, attackers can access rich sensitive data and even manipulate sub-apps back-end servers.

B. Adversary Scenarios

In this section, we describe the adversary scenarios adopted in the paper. We assume that the host apps and underlying mobile operating systems are benign. The attacker is sophisticated with app reverse engineering, communication engineering (i.e., traffic analysis with mitmproxy [10]), and even social engineering skills (i.e., sending malicious deep-link to victim users). Specifically, we consider the ecosystems are facing the following threat scenarios:

- Inside Attacker. In this scenario, the attacker can launch attack as a malicious party (i.e., sub-app, web content from malicious server) in the app-in-app ecosystem. The attacker can either upload a malicious sub-app to the market or launch masquerade attack to load malicious web content into a vulnerable sub-app. Note that if either the host app or the sub-app violates *R1*, malicious web content can be loaded into the sub-app runtime, despite the runtime being designed to exclusively load web content trusted by sub-apps.
- Outside Attacker. In this scenario, the attacker has the ability to compromise resource delegation channels. Specifically, the adversary in this scenario could be a man-in-the-middle (MITM) server or malicious mobile app installed on the victim's phone. He can exploit OS vulnerabilities [11] to access local files of host apps.

Note that the former scenario is considered more powerful than the latter one. This is because the inside attack (i) is more deceptive due to the simplified launch condition through phishing deep links and reuse of the runtime of benign sub-apps; (ii) can access to much powerful functionalities, such as payment and back-end service of host apps, leading to significant security breaches and damage.

C. Attack Road-Map

This section presents the attack roadmap detailing how three attack vectors operate in two adversary scenarios.

As an inside attacker, who can (a) start with injecting malicious web code into the sub-app layer. Specifically, a malicious deep link such as scheme://encoded(sub-appID,path,

malicious-url) is crafted by attacker, and clicked by victim user. Host app recognizes the deep link for loading the sub-app with sub-appID and page path, which receives a URL as input to load web content in its runtime. With the inadequate allowlist check, malicious web content can be easily loaded for successor attacks. (b) The injected code seeks to expand the scope of attacks via API manipulation. That is, in masquerade attack, attacker breaks the host apps' permission and identity check to access privileged abilities. Besides, in data-driven attack, the attacker accesses sensitive data of other sub-apps in the insecure data isolation. Moreover, to achieve API manipulation, the attacker can manipulate other privileged sub-apps that lack input validation by simply crafting malicious input and sending it to them.

Alternatively, the attacker can initiate channel hijacking from outside the app-in-app ecosystem. If the delegation of sensitive data lacks encryption protection, the delegation channel can be tampered with, allowing the attacker to pilfer sensitive data such as secret keys, tokens, and phone numbers.

IV. Masquerade Attack: Ability Exposure

This section details our first attack vector, referred to as *masquerade attack*, wherein remote attackers upload malicious sub-apps or web content posing as trusted entities, consequently escalating their privileges to critical APIs. The root cause of such an attack lies in the complexity of the `CPDR` delegation model, which complicates the identification of trusted entities when the host app delegates critical functionalities to sub-apps or web content. In brief, the vulnerabilities may happen when host apps check the identity of `CPDR` interface caller (AppID and domain name), or re-delegate OS permission to sub-apps and web content.

A. App ID Confusion

An AppID confusion occurs when the malicious web content is loaded into a sub-app (i.e., violate *R1*). Thus, AppID confusion opens the attack surface for inside attackers. Most importantly, the loaded malicious code can unexpectedly access `CPDR` interfaces for further exploits, i.e., using a privileged AppID to invoke a privileged `CPDR` interface, which can confuse the host app's identity checks. Note that the reason that we call it AppID confusion is that the malicious domain has the correct AppID, but the domain itself is malicious. In practice, we find three cases of such AppID confusion.

1) Type 1: Flawed URL allowlist matching: This flaw is that the URL allowlist used for loading is flawed, thus being able to allow potential malicious URLs to load. The main reason is the lack of coordination and proper documents between the host app and sub-apps. Specifically, the URL allowlist checking algorithm is provided by the host app, but the allowlist 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 host 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 allowlist, an adversary can bypass the check using a domain like maliciousbenign.com.
- Regular expression (regex) being misunderstood as strict matching. In this scenario, the host 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.

2) Type 2: Flawed URL parsing: This flaw is that host apps have logic errors in parsing URLs and extracting web domains. Specifically, we find two types of parsing errors that exist in studied host apps. The first is caused in many cases that the host app does not correctly recognize the username and password fields of the URL. Take https://benign.com:x@malicious.com as an example. A logic error is to extract benign.com as the domain name, rather than malicious.com. The second is that host app does not forbid JavaScript as a protocol. Thus, an attacker can use the URL javascript://payloads to exploit the URL parsing, resulting in code injection attacks.

3) Type 3: Missing URL checks: This flaw means the host 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 access privileged APIs, such as reading user contacts.

Take-aways of §IV-A

Party Responsibility: The host app implements flawed methods for allowlist matching and URL parsing. The sub-apps mistakenly configure allowed domain.

Consequence: Injection of malicious web content.

B. Domain Name Confusion

A domain name confusion is that the web domain that invokes a privileged API from web content is different from the domain that a host app obtains and checks for identity, disobeying the *R2*. 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. Host apps depend on WebView callbacks to acquire the identity of the caller. Nevertheless, these callbacks do not provide sufficient accuracy to obtain the correct identities, especially when a change happens in the sub-app layer. Specifically, we classify domain name confusions into two types: timing-based (due to race conditions) and frame-based (due to the existence of multiple domains). We now describe the details.

1) Type 1: Timing-Based Confusion: The first type—called timing-based—is because of a race condition between different threads of WebView and host 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 host app thread, the identity is from say malicious.com; but when the host app thread checks the identity, the identity changes to say privileged.com due to redirection, leading to confusion. Figure 2 shows an exploitation of such an attack on Microsoft

```
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/";
```

Fig. 2. Example for exploiting domain name confusion. The getAuthToken is a privileged API of Microsoft Teams.

Teams. This figure exhibits a race condition: although the webpage is set to navigate to https://privileged.com, and so is 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.

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 are conducted by the host app and how such checks are performed, the host 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.

Table IV shows the confusing WebView event handlers in existing host apps. This list includes commonly used ones like `onPageStarted`, `onPageFinished`, and `getUrl`.

Take-aways of IV-B

Party Responsibility: The host app uses inadequate WebView event handlers to obtain identity.

Consequence: Privilege escalation of critical abilities.

C. Permission Inconsistency

Permission inconsistency happens when host apps fail to restrict unauthorized sub-apps or web content to access re-delegate sensitive system resources protected by OS permission (i.e., violate *R3*). We list two scenarios here.

- *Permission Scope.* In this scenario, the permissions designed by OS developers for protecting sensitive system resources are always updated ahead of host apps. To this end, host app developers cannot obtain a precise permission mapping for used system resources. Note that, the host apps are often granted with critical system permissions due to their close partnership with the operating system providers. This results in both temporal gaps and missing scope in defining permissions. For example, after the Android Q version, it was announced that access to the clipboard requires signature-level permission [12]. However, all of the host apps including Alipay, WeChat, Baidu, Xiaomi, and Huawei do not provide a consistent permission scope for the `CPDR` interfaces with access to clipboard. For another example, host apps like Alipay and WeChat often provide different `CPDR` interfaces (e.g., `previewImage` and `saveImage`) with access to the same system API (i.e., `getExternalStorageDirectory`), but only `saveImage` has the corresponding permission protection. In our observation, the previewImage can also access files in the external storage directory that stores sensitive photos of users, exposing them to malicious attackers.
- *Enforcement Scope.* In this scenario, host app developers only enforce the permission check for the sub-appID. That is, if the sub-app has already been granted with the permission for accessing system microphone, any loaded web content in the sub-app can obtain such abilities without permission request to users. Thus, malicious content can easily abuse the delegated system's abilities.

As a result, both the malicious sub-app and malicious web content can easily obtain unauthorized abilities and abuse system resources.

Take-aways of IV-C

Party Responsibility: The host app implements inconsistent permission check in delegating system resources to sub-apps and web content.

Consequence: System permission re-delegation.

V. Data-Driven Attack: Data Manipulation

This section details our second attack vector, known as data-driven attack, in which sub-applications fail to validate requests received from other entities, or data storage is inadequately isolated.

A. Weak Storage Isolation

As detailed in Table III, the widely used storage should be protected following *R4-6*. However, none of these 9 studied app-in-app ecosystems implement these regulations comprehensively.

1) Sub-App Storage: By design, the storage of sub-apps is naturally isolated from that of web content. However, this isolation may be compromised due to the existence of covert channels. Specifically, Baidu provides an `CPDR` interface called `setData`, which exposes the storage of current sub-app runtime to its web content, violating *R5*. It is important to note that sub-apps often load diverse web content from third-party servers (e.g., advertisement), and these web content can potentially access and share sensitive data stored by the sub-app, incurring data theft.

2) Web Content Storage: Although the concept of SOP has already been introduced and enforced by web browsers, we find existing host apps may fail to protect their local storage when delegated through `CPDR` interfaces, thereby violating *R6*. Specifically, Alipay provides `setStorage` and `getStorage` for web content to easily save and read its own data, including user tokens, nicknames, and cookies. However, we find that Alipay only enforces a coarse-grained isolation among different domains. That is, the web content of malicious.com can access the stored sensitive data of benign.com, leading to data leakage.

TABLE IV
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	getUrl ()	✔	✔
Callback Method:			
WebViewClient	onPageFinished (WebView view, **String url**)	✔	✔
	onPageStarted (WebView view, **String url**, Bitmap favicon)	✔	✔
	shouldOverrideUrlLoading (WebView view, **WebResourceRequest request**)	✔	

Take-aways of V-A

Party Responsibility: The host app provides coarse-grained storage isolation.

Consequence: Data leakage.

B. Missing Input Validation

With strict storage isolation protection, sub-apps become difficult to share data among sub-apps and web content. Thus, host apps provide `CPDR` interfaces `postMessage` and `navigateToMiniprogram` to satisfy such data delegation needs. One often relies on the input data from other parties to complete their service tasks. For example, the shopping sub-app A can send the payment order to finance sub-app B to complete the payment process. However, this requires sub-app to carefully check the communication target when delivering sensitive resources. In this scenario, we assume web content and other sub-apps can be malicious.

1) PostMessage: This flaw is that a sub-app does not validate messages sent from its web content via the `CPDR` interface `postMessage`, violating *R7*. Sub-apps often handle the messages that are sent from loaded web content, and delegate the resources they are obsessed to web content. For example, the web content can send message to sub-app to achieve user login and retrieve the login token to present further VIP services. Without validations on the sender, malicious web content can easily craft the request with malicious input to launch data-driven attack, abusing the data and abilities of the vulnerable sub-app.

2) Sub-App Navigator: This flaw is that a sub-app does not validate input data from other sub-apps via `CPDR` interface called `navigateToMiniprogram`, violating *R7* too. Using this API, one can easily delegate tasks (e.g., payment and face recognition) to other sub-apps by sending query data to and receiving task results from other sub-apps. However, it is important for the receiver sub-app to check the sender's appID; otherwise, a malicious sub-app can launch data-driven attack to manipulate a vulnerable sub-app to abuse its ability or steal sensitive data.

Take-aways of V-B

Party Responsibility: The sub app fails to check the identity of message sender.

Consequence: Data leakage, ability re-delegation.

VI. CHANNEL HIJACKING: CRYPTOGRAPHY FAILURES

This section details our third attack vector, wherein an outside attacker can tamper with the resource delegation channels including network and storage. The root cause lies in the cryptography failure of sub-apps and host apps on the above two channels. This attack vector can bring extensive threats to the service of sub-apps, such as manipulating sub-app web servers.

A. Storage Hijacking

Note that many security researchers have highlighted security issues of storage without encryption [13], [14]. They advocate for app developers to refrain from storing sensitive data, such as passwords, tokens, and user information, in bare storage. However, we still find that seven host apps do not provide encryption to local storage files used by sub-apps or web content, violating *R9*. And sub-apps are more likely to trust the storage channel, assuming it possesses the same level of security as their host app, suffering storage hijacking attack. A sophisticated malicious app can exploit Android privilege escalation vulnerabilities to gain access to these storage files. Such attackers can steal or modify sensitive data, resulting in data leakage or tampering with the sub-app's services.

Take-aways of VI-A

Party Responsibility: The host app provides no storage file encryption.

Consequence: Data leakage.

B. Cloud Traffic Hijacking

1) Insecure Protocol: Although all host apps allow sub-apps to use network channels (i.e., `Request`), we find that seven of them do not provide strict secure protocol enforcement (i.e., SSL), violating *R8*. That is, sub-app developers that lack security awareness still use insecure communication protocols, such as http:// and ws:// (for WebSocket), and can suffer from MITM atttack. Note that, Line and Microsoft Teams do not provide such customized interfaces, but their sub-apps can use JavaScript request methods (e.g., `XMLHttpRequest`) to communicate with a remote server. However, their developer documentation does not state the corresponding security regulation; thus, their sub-apps may also suffer from the attack. To exploit this vulnerability, one MITM attacker can hijack the network traffic, and modify or steal sensitive data including user phone numbers for registration, user tokens, and even payment orders.

2) Phone Number in PlainText: Sub-apps often assume the integrity of delegated user phone numbers is guaranteed, they often rely on it to complete the user login and authentication process. However, we find Huawei and Xiaomi delegate the phone number without encryption, violating *R10*. To exploit this, the MITM attacker can easily modify the phone number used for user verification through a network monitor or Xposed hook, logging in to any victim's account.

3) Encryption Key Leakage: The delegated sensitive user data are often critical for backend server logic, such as user login and payment processes. To prevent integrity breaches, the delegated data must remain encrypted in the front end and be processed in sub-app's server. However, the ambiguous and ad-hoc security descriptions provided by host app developer may mislead sub-apps into attempting to decrypt them on the front end. As a consequence, this could lead to the leakage of secret keys to malicious attackers, thereby violating *R11*. Note that there are usually two ways to leak the secret key. First, the sub-app may embed its private keys in the sub-app code, which can be easily extracted. Second, the sub-app may dynamically send the key from its server-side and execute the decryption in the front end. With leaked secret keys, the attacker can gain unauthorized access to back-end servers of sub-app developers or generate fake signatures to steal sensitive user data.

Take-aways of VI-B

Party Responsibility: The host app failes to encrypt the delegated user data. The sub app uses insecure communication protocol and does not follow secure cryptography procedure.

Consequence: Data leakage.

VII. Measurement: Prevalence and Consequence

In this section, we conduct a measurement study on the nine most popular app-in-app ecosystems in Table I to identify violations of `CSRs` in real-world host apps and sub-apps.

A. Overall Methodology

Thoroughly confirming the aforementioned security concerns is a challenging task due to the semantic gap between the implementation of different host apps. This discrepancy makes it costly and time-consuming to identify vulnerabilities across various app-in-app ecosystems. Our proposed methodology operates at the program behavior level and includes a set of formalized testing templates or security rules. Based on the summarized `CSRs` (detailed in Section II-C), we design a set of violation samples to test the host app implementation of `CPDR` interfaces. Note that, all of the tests are constructed in JavaScript language which can be deployed across ecosystems with the help of uni-app [15]. The uni-app can automatically convert our penetration samples to suit the different API supported by respective ecosystems. In cases where the host app lacks support for the tested API, we consider it is non-vulnerable. Simultaneously, we carry out differential analysis to identify evidence of a violation of security rules when sub-apps and web content are utilizing `CPDR` interfaces.

B. Penetration Test Configuration

1) Identity Check: Our experiment evaluates whether an unprivileged identity can access privileged abilities. We consider the penetration test in two scenarios: (i) a sub-app without privileged AppID; and (ii) web content without privileged domain. Our penetration test samples check whether an adversary can ask a host app to load any malicious domain in a sub-app. And whether the domain can further exploit privileged `CPDR` interfaces.

- Sample1. Specifically, we create a sub-app and set an allowlist for benign domains. Then, we generate a variety of URLs by mutating several initial seeds. Next, we randomly select URLs that cannot match the allowlist to test the sub-app. During the test, we hijack the network traffic and return the same webpage we crafted for invoking privileged `CPDR` interfaces when requesting these URLs. Thus, if any of these URLs are successfully loaded and the JavaScript executed, an AppID confusion is confirmed.
- Sample2. To validate a domain name confusion, 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 APIs execute successfully, the host app is vulnerable to domain name confusion.

2) Permission: Our experiment evaluates whether a sub-app without any permission can access a permission-protected system resources. Although previous work [16] uses the dynamic testing technique to find permission inconsistency for common Android permissions, they ignore the clipboard permission granted by user dynamically, and network permission may be required under different network connection situations. Thus, our penetration test considers all of these situations to better understand the permission inconsistency in different ecosystems. Specifically, our penetration test sample includes invoking `getClipBoard`, `setClipBoard`, and other system APIs such as files management, camera usage, and network request. According to the consistency II-C, host apps should create a pop-up window to alert the user that one sub-app is requiring permission when calling these `CPDR` interfaces. Based on this, we can identify permission inconsistencies by monitoring the presence of consistent permission pop-ups after running test samples.

3) Storage Isolation: Our experiments evaluate whether the implementation of storage protection meets all of the three security regulations (i.e., *R4-R6*). Host apps must carefully isolate the storage from different entities (i.e., sub-apps, users, and web content), and prevent malicious parties from abusing them. Specifically, we configure a group of penetration samples within our own sub-apps.

- Sample1. The sub-app A invokes `setStorage` to add a key-value pair data to the storage, i.e., < "testkey", "sub-appA"> . The web content invokes `getStorage` with key = "testkey". If the return result is "sub-app", it indicates a violation of *R5* on read operation.
- Sample2. The web content a.com invokes `setStorage` with key = "testkey", value = "a.com", and sub-app invokes `getStorage` with key="testkey". If the return result is "a.com", it indicates a violation of *R5* on write operation.

- Sample3. The web content b.com invokes `getStorage` with key "key". If the return result is "a.com", it indicates a violation of *R6*.
- Sample4. Let another sub-app B invokes `getStorage` with key = "testkey". If the return result is "sub-appA", it indicates a violation of *R4*.

4) Encryption: Our experiments evaluate whether the encryption of storage files and delegated user data meet security regulations following *R9* and *R10* individually. For storage files, we reuse the test samples from storage isolation and then analyze whether the storage files are encrypted, which indicates the violation of *R9*. For user data delegation, our penetration test samples invoke the `CPDR` interfaces including `getPhoneNumber`, `getUserProfile`, and `getRunData`. Then we use MITM proxy server to check whether the delegated data is encrypted. If the data is not encrypted, it indicates a violation of *R10*.

C. Differential Analysis Configuration

1) Input Validation: To find a sub-app that violates *R7*, we perform static taint analysis from cross-party communication event handler (i.e., `onMessage` for `postMessage` and `onLoad` for `navigateToMiniprogram`). Our static analysis is implemented based on JS-WALA [17], which provides an intermediate language presentation of the JavaScript code. To this end, we can build the control-flow graph and data-flow graph using its functionality. Specifically, we model these event handlers and identify their parameters that contain the cross-party delegated data as tainted. Then we build the data-flow graph for these tainted data and analyze whether there are data processed by delegatee sub-app. We identify such data delegation as vulnerable when the delegatee sub-app lacks check of the sender's identity (e.g., AppID and domain name).

2) Protocol Usage: We analyze the network channel security from the usage of communication interfaces in sub-apps. In our analysis, we first identify the *Request* and *Connect_WebSocket* APIs as sinks, and launch backward data-flow analysis on the target server address variable (modeled manually) to find the source string value that propagate to the variable. Specifically, we analyze the object assignment statement to check whether a constant String value is assigned to the tainted variable. If we find a constant String value, we further try to parse the String as URI and extract its protocol. If a sub-app uses "http" or "ws" as communication protocol, it is vulnerable to channel hijacking.

3) Key Leakage: Our differential analysis tries to find the evidence on the sub-app's code that has insecure front-end decryption, which violates *R11*. Previous work [18] measures the master key leakage in sub-apps of WeChat, however, their work is limited to only detect the first type of key leakage. And they only focus on the secret keys that are mainly used in the communication between the sub-app and host apps, thus ignores other cross-party communications such as sub-app to its backend server. Our differential analysis considers a generic key leakage problem. To identify the embedded key, we utilize the entropy-based method like previous work [19] to find secrets that should not be present in any sub-app code. We calculate the information entropy of the secret keys provided by host apps, and choose 1.5 as the threshold. If one string has entropy exceeding 1.5, we consider it as a development key leakage. To identify the dynamically leaked keys, we can identify whether there are common decryption algorithms usage for private user data such as phone number, user profile, and user motion data. Specifically, we use the function names of common encryption and decryption algorithms (i.e., RSA and AES algorithms) provided in the sub-app developer documentation as matching features.

TABLE V
OVERALL RESULTS OF HOST APPS. ✔ MEANS IT HAS VULNERABLE SECURITY ENFORCEMENT, AND ✘ HAS NOT. - MEANS THE HOST APP DOES NOT HAVE THIS TYPE OF SECURITY ENFORCEMENT

Ecosystem	Masquerade Attack			Data-Driven	Channel Hijack	
	§IV-A	§IV-B	§IV-C	§V-A	§VI-A	§VI-B2
TikTok	✔	✔	✔	✘	✔	✘
WeChat	✔	✔	✔	✘	✔	✘
KuaiShou	✘	-	✔	✘	✔	✘
Alipay	✔	✔	✔	✔	✔	✘
Line	✔	-	✔	-	-	-
Baidu	✔	-	✔	✔	✔	✘
M. Teams	✔	✔	-	-	-	-
Huawei	✔	-	✔	✔	✔	✔
Xiaomi	✔	-	✔	✔	✔	✔

D. Sub-App Collection

Now we describe how we collect the source code of sub-apps. Each app-in-app ecosystem has its customized sub-app market, thus we implement our sub-app crawler based on MiniCrawler [20] for individual ecosystems. While sub-apps from WeChat are encrypted, we can use decryption tool [21] to recover their source code. To this end, we collect 202,273 sub-apps, including 57,514 from WeChat, 93,128 from Alipay, 49,630 from Baidu, 29 from Kuaishou, 969 from Huawei, and 1,032 from Xiaomi. Note that, TikTok, Line, and Teams have anti-crawler protection; thus, we fail to analyze their sub-apps. But we still can perform security tests on these host apps.

VIII. MEASUREMENT RESULTS

In this section, we describe our empirical study results in terms of analyzing the prevalence and consequence of the aforementioned three attack vectors in real-world app-in-app ecosystems. We also give a few real-world case studies and proof-of-concept exploitation to demonstrate the attack scenarios at the end.

A. Vulnerability Prevalence

1) Host Apps' Pitfalls: Table V shows that all of the studied ecosystems are vulnerable to at least one type of attack despite the diversity in security enforcement implemented in app-in-app ecosystems. Here are the breakdowns. Nine app-in-app ecosystems suffer from masquerade attacks. Four app-in-app ecosystems suffer from data-driven attacks. And seven app-in-app ecosystems suffer from channel hijacking.

2) Sub-Apps' Pitfalls: Table VI shows the prevalence of attack surface. The ranking is top-down: For masquerade attack, around 27.5% of privileged sub-apps have AppID confusion. Thus they are all controllable under the malicious deep links. For data-driven attack, nine host apps do not provide a fine-grained identity for input validation, thus affecting all sub-apps. Specifically, 27.3% sub-apps are exploitable from other

TABLE VI

OVERALL RESULTS OF SUB-APPS. ✔ SUB-APPS ARE ALL VULNERABLE DUE TO HOST APPS LACK OF SECURITY ENFORCEMENT

Ecosystem	Data-Driven	Channel Hijack	
	§V-B	§VI-B1	§VI-B3
TikTok	✔	-	-
WeChat	✔	-	**10.46%**
KuaiShou	✘	**86.21%**	**3.45%**
Alipay	✔	**77.5%**	**8.68%**
Line	-	-	-
Baidu	✔	**22%**	**3.26%**
M. Teams	-	-	-
Huawei	✔	**29.2%**	**2.48%**
Xiaomi	✔	**52.4%**	**3%**

sub-apps and 18.1% from web content. Thus they open a vast attack surface for app-in-app ecosystems. We randomly select 100 sub-apps from each app-in-app ecosystem and further validate around 4.5% of sub-apps concatenate sensitive data to URLs that can be manipulated by malicious attackers. These sensitive data often include user login tokens and address info. For channel hijacking, 22%-86.21% sub-apps still use insecure protocols in their communication with remote servers. And at most 10.46% sub-apps leak the sensitive keys to the front-end.

B. Vulnerability Consequence

We find the vulnerability consequences can be vast and far-reaching, by combining the PoCs in the aforementioned three attack vectors, leading to ability re-delegation, data leakage, and even privilege escalation. More details are presented below:

- *Ability Re-delegation.* The permission check inconsistency happens between OS and host app, or between sub-app and web content. Moreover, when benign web content applies for permission and the user grants it, the host 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.
- *Data Leakage.* It is the disclosure of sensitive information to an adversary. Utilizing clustering analysis, we categorize the leaked data. The result shows a disaster, which includes leakage of PII data (around 64%) such as user tokens, addresses, and device information. Other highly ranked sensitive data include session information and secret keys. We manually analyze the leaked secret keys, and find there are diversified key abilities, including encrypt and decrypt phone numbers and access third-party cloud service APIs (e.g., manipulate remote storage).
- *Privilege Escalation.* We manually inspect whether malicious web content can access privileged APIs after successfully confusing the host app and disguising itself as a privileged identity. The consequence includes manipulating host app's backend server via a privileged API called "rpc", exposing huge security risks.

C. Case Studies

1) Example [WeChat] One-click Privilege Escalation: Now we describe a motivating example of a host app WeChat and its sub-app Pingduoduo[7] to illustrate identity confusion vulnerability, which eventually leads to privilege escalation attacks, such as arbitrary APK download and installation on the Android platform.

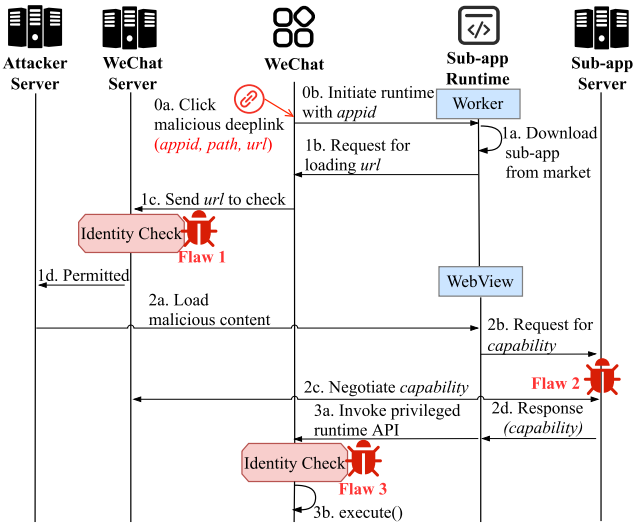

Fig. 3. One-click attack example for a remote attacker to exploit WeChat app-in-app ecosystem.

As shown in Figure 3, the end-to-end attack has 12 detailed steps that can be grouped into three major phases: (i) injecting malicious code to Pingduoduo's runtime, (ii) obtaining privilege capabilities, and (iii) downloading malicious apps.

First, a web attacker can send malicious deep link such as weixin://encoded(pingduoduo-appID, path, malicious-url) to victim. When the link is clicked by the user, Pingduoduo will be loaded into the customized worker, and send the request for loading the URL embedded in the malicious deep link. This request is hooked by WeChat, which will further check the domain allowlist on its server-side in Step(1c). However, Pingduoduo is affected by flawed URL parsing IV-A; therefore, in Step (1d), the WeChat server grants permission to load this URL. To this end, malicious.com is loaded into Pingduoduo's WeView instance. Second, up to now, the malicious web content can only invoke unprivileged APIs because WeChat performs capability-based access control. In Pingduoduo, such capability is delegated from its server side to the sub-app through network channel, and protected by an authorization key. However, the Pingduoduo leaks the secret key to the sub-app, which can be obtained by malicious attacker, and finally abused for obtaining the privileged capability (Step 2c). This finally leads to privilege escalation for sensitive APIs. For example, `addDownloadTask()`, a hidden, undocumented API, which can download and install any APKs on the Android platform.

2) Example [TikTok] Bypassing Security Patches with an Error URL: This example is the AppID and domain name confusion of TikTok, a popular social app with about 18 billion downloads. The AppID confusion is from the matching of URLs using `endswith`. 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

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

TABLE VII
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

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 VII.

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`.

3) Example [HuoLaLa] Bucket Effect: This example illustrates the amplified channel hijacking in HuoLaLa, a cargo transport service provider with more than 16 million monthly active users. The channel hijacking is from the plaintext phone number delegated from Huawei to HuoLaLa. Specifically, the phone number is supposed to be authenticated as belonging to the real phone user and used by HuoLaLa for password-free user login. However, the MITM attacker can modify the delegated phone number, breaking the authenticity assumption. If the exploit is considered to occur in Huawei's app-in-app ecosystem, our attacker can achieve any account login in HuoLaLa's sub-app, and get the user token from the response. Note that, the above token can be used to log in to the same user account in HuoLaLa's mobile app, even though its app version has a more secure phone number encryption. To this end, the malicious attacker can replace the token in app's login procedure, and achieve the mobile account hijacking. Attackers can leverage the bucket effect to further access much private user information that does not appear in the sub-app version, including the user's car refueling bill, home address, and even the lending business. This example illustrates that the security of service providers is highly constrained by the bucket effect in app-in-app ecosystems. Indeed, the bucket effect will bring great damage to the data and abilities that are delegated in app-in-app ecosystems.

Root Cause: With a deeper analysis of the root cause of the *bucket effect*, we find that the sub-app server, acting as a link, connects user privacy data from different app-in-app ecosystems. Consequently, a privacy data breach on one ecosystem can further impact the security measures of the same sub-app on another ecosystem.

Prevalence: To understand the prevalence of bucket effect, we manually sample 100 popular sub-apps and examine their user login implementations across different ecosystems, including Alipay, WeChat, Baidu, Huawei, and their mobile apps. The results indicate that 84 sub-apps have versions deployed on multiple ecosystems, with an average of one sub-app deployed on three ecosystems. Among these, we identify 17 cross-platform sub-apps where the login tokens are shared, suggesting the potential presence of the *bucket effect*.

IX. DISCUSSION

1) Lessons Learned: The most important lessons learned from our research is that an appropriate security regulation should adhere to the two principles, namely the *least privilege principle* and *delegation consistency*. First, the identity definition of delegatee and delegator in the app-in-app ecosystem needs to be atomic, providing clear coordination between parties of host apps, sub-apps, and web content. Second, both the host app, sub-apps and web content should enforce the standard security checks for each delegated resource. It is important to emphasize that neither party should place trust in the other. Adopting the client-side trust model is advisable, whereby each party involved in cross-party delegation is regarded as untrusted.

Other than the involved parties should enforce proper security regulations, the mitigation of security flaws will also benefit from the cooperation with underlying infrastructure, i.e., obtain a precise domain information provided by the WebView component. Draco [22] provides a good example of such domain identity 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.

2) Security Regulation Design: In this paper, we conduct a detailed analysis of the implementation and enforcement of security regulations on different app-in-app ecosystems. However, it is also worth discussing the adequacy of the design of security regulations. Take security regulation *R1* (i.e., web content domain allowlist) as an example, its security assumption based on domain-based allowlist is coarse-grained, thereby this mechanism may be unreliable in restricting untrusted web content. We believe that this work can inspire further research into the proper design of security regulations in app-in-app ecosystem.

3) Automatic Security Analysis of Cross-Party Resource Delegation: Similarly, different scenarios of cross-party delegation security analysis have also been proposed, as exemplified in IoT scenario [23]. Specifically, Yuan et al. proposes a semi-automatic way in identifying inconsistent authorization in IoT device access using formal verification technique. However, due to the closed-source feature of studied ecosystem, there is a lack of standards in security implementation, leading to different implementations across similar ecosystems. To this end, significant manual expertise is required to model the security regulations for different ecosystems. Guided by the two fundamental security principles of *least privilege* and *delegation consistency*, it is possible to utilize various analysis techniques to achieve the goal of security detection, such as static analysis, penetration testing, and even using LLM in program behavior abstraction and analysis. We anticipate that substantial effort will continue to be devoted to this subject.

4) Ethics: We discuss the ethical issues of our study, including vulnerability disclosure and experimental setups. First, we have informed all nine host apps of their vulnerabilities. Currently, eight host apps have confirmed their vulnerabilities. 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, host-apps, or any of their servers. Note that, we reported the vulnerable sub-apps to the corresponding host app developer because they are responsible for informing these vulnerable sub-apps of rectification.

X. Related Work

A. App-in-App Ecosystem

Recent studies reveal the model of app-in-app ecosystems, and their various advantages in different aspects of social life, including health, education, government, and marketing [24], [25], [26], [27], [28], [29], [30], [31], [32]. Additionally, some studies provide program analysis techniques by leveraging as dynamic analysis [33], [34], and static analysis [35], [36]. In the domain of research on attacks and defense in app-in-app ecosystems, a few pioneering studies [6], [16], [18], [37], [38] have delved into the defense mechanisms and vulnerabilities of these ecosystems. Specifically, Lu et al. [16] and Zhang et al. [38] investigate the permission inconsistency problem, and Zhang et al. [6] proposed the novel identity confusion flaws in protecting runtime APIs. Wang et al. [39] developed a tool to identify hidden APIs in app-in-app ecosystems, and demonstrated the potential security risks being exposed to third-party sub-apps. Zhang et al. [18] and Yang et al. [37] measure the security prevalence of information leakage. As a comparison, our paper has a different goal focusing on the `CPDR`, implementation of security regulations, and their security threats, which have not been thoroughly studied before.

B. WebView Security

WebView is becoming a widely-used component for loading web content in mobile apps and has been studied by many research works [40], [41], [42], [43], [44], [45], [46], [47], [48], [49], [50]. For example, Jin et al. [40], Li et al. [41], Wang et al. [42] 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. [43] analyze WebView-based advertisement apps and find that malicious ads can hijack mobile apps. As a comparison, our work focuses on how host apps and sub-apps should protect the delegated resources.

Past works also study the race condition attacks in WebView. Lau et al. [44] 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 [45] also reports several race conditions in WebView's event handlers. As a comparison, our measurement study reveals the security implications in real-world host apps and sub-apps.

C. Mobile App Security Mechanism

Smalley and Craig [51] 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. [52] analyze the issues of enforced security policies. We then describe web apps and their connection with mobile systems. Prior works [53], [54], [55], [56], [57], [58], [59] focus on the security issues among multi-origin web pages. NoFrak [60] points out the importance of protecting the web-to-mobile bridge. Then, Draco [22], MobileIFC [61], WIREframe [62], and HybridGuard [63] 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 [64], [65] discover additional flawed URL parsing and matching examples in different scenarios, such as email senders. As a comparison, app-in-app ecosystem often requires more sophisticated security enforcement because it involves more parties with sensitive resources, which is much more complicated than mobile apps.

XI. Conclusion

In this paper, we unveil the critical security question surrounding cross-party delegated resources in app-in-app ecosystems. We identify 11 core security regulations within host apps, yet their inconsistent implementation with sub-apps leads to violations and vulnerabilities. By investigating various ecosystems, we propose three novel attack vectors, demonstrating the widespread prevalence of security regulation breaches. Our findings emphasize the urgency of consistent security practices. Through responsible disclosure, we engage with developers to remediate vulnerabilities, underscoring the imperative of collaborative security efforts in the evolving app-in-app landscape.

References

[1] *WeChat Miniprograms Have More Than 450 Million DAU*. Accessed: Oct. 6, 2021. [Online]. Available: https://finance.sina.com.cn/tech/2022-01-06/doc-ikyakumx8630297.shtml

[2] *Number of Available Applications in the Google Play Store From December 2009 to December 2020*. Accessed: Oct. 6, 2021. [Online]. Available: https://reurl.cc/ox5Oj3

[3] (2023). *Tiktok One-Click Attack*. [Online]. Available: https://bit.ly/42Hmo6b

[4] (2023). *CNCERT/CC Security Report*. [Online]. Available: https://www.cert.org.cn/publish/main/upload/File/2020

[5] *Android Developers*. Accessed: Feb. 15, 2023. [Online]. Available: https://developer.android.com/develop/ui/views/layout/webapps

[6] L. Zhang et al., "Identity confusion in WebView-based mobile app-in-app ecosystems," in *Proc. 31st USENIX Secur. Symp. (USENIX Security)*, Aug. 2022, pp. 1597–1613. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/zhang-lei

[7] WKWebView. (2021). *Apple Development Documentations*. Accessed: Oct. 6, 2021. [Online]. Available: https://developer.apple.com/documentation/webkit/wkwebview

[8] *Jadx—Dex to Java Decompiler*. Accessed: Feb. 15, 2023. [Online]. Available: https://github.com/skylot/jadx

[9] (2021). *Tiktok Mini-App Secure Development Guide*. [Online]. Available: https://microapp.bytedance.com/docs/zh-CN/mini-app/develop/guide/anquan/anquankaifa/

[10] *Mitmproxy—An Interactive HTTPS Proxy*. Accessed: Feb. 15, 2023. [Online]. Available: https://mitmproxy.org/

[11] (2022). *A Very Powerful Clipboard: Analysis of a Samsung In-the-Wild Exploit Chain*. Accessed: Feb. 15, 2023. [Online]. Available: https://googleprojectzero.blogspot.com/2022/11/a-very-powerful-clipboard-samsung-in-the-wild-exploit-chain.html

[12] (2023). *Limited Access to Android Clipboard Data*. [Online]. Available: https://android.googlesource.com/platform/frameworks/base/

[13] (2018). *Man-in-the-Disk: Android Apps Exposed via External Storage*. Accessed: Feb. 15, 2023. [Online]. Available: https://research.checkpoint.com/2018/androids-man-in-the-disk/

[14] J. Reardon, Á. Feal, P. Wijesekera, A. E. B. On, N. Vallina-Rodriguez, and S. Egelman, "50 ways to leak your data: An exploration of apps' circumvention of the Android permissions system," in *Proc. 28th USENIX Secur. Symp. (USENIX Security)*, 2019, pp. 603–620.

[15] *dcloudio/uni-app: A Cross-Platform Framework Using vue.js*. Accessed: Feb. 15, 2023. [Online]. Available: https://github.com/dcloudio/uni-app

[16] H. Lu et al., "Demystifying resource management risks in emerging mobile app-in-app ecosystems," in *Proc. 27th ACM SIGSAC Conf. Comput. Commun. Security (CCS)*, 2020, pp. 569–585.

[17] (2023). *JS-WALA*. [Online]. Available: https://github.com/wala/JS_WALA

[18] Y. Zhang, Y. Yang, and Z. Lin, "Don't leak your keys: Understanding, measuring, and exploiting the AppSecret leaks in mini-programs," 2023, *arXiv:2306.08151*.

[19] M. Meli, M. R. McNiece, and B. Reaves, "How bad can it git? Characterizing secret leakage in public GitHub repositories," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2019, pp. 1–15.

[20] Y. Zhang, B. Turkistani, A. Y. Yang, C. Zuo, and Z. Lin, "A measurement study of wechat mini-apps," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 49, no. 1, pp. 19–20, Jun. 2021.

[21] MP-Unpack. *Unpacking WeChat wxapkgs*. Accessed: Feb. 15, 2023. [Online]. Available: https://xuedingmiaojun.github.io/mp-unpack

[22] G. S. Tuncay, S. Demetriou, and C. A. Gunter, "Draco: A system for uniform and fine-grained access control for web code on android," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2016, pp. 104–115.

[23] B. Yuan, Y. Jia, L. Xing, D. Zhao, X. Wang, and Y. Zhang, "Shattered chain of trust: Understanding security risks in cross-cloud IoT access delegation," in *Proc. 29th USENIX Secur. Symp. (USENIX Security)*, 2020, pp. 1183–1200.

[24] L. Hao, F. Wan, N. Ma, and Y. Wang, "Analysis of the development of WeChat mini program," *J. Phys., Conf. Ser.*, vol. 1087, Sep. 2018, Art. no. 062040.

[25] Z. Tang, Z. Zhou, F. Xu, and M. Warkentin, "Apps within apps: Predicting government WeChat mini-program adoption from trust–risk perspective and innovation diffusion theory," *Inf. Technol. People*, vol. 35, no. 3, pp. 1170–1190, Apr. 2022.

[26] F. Wang, L. D. Xiao, K. Wang, M. Li, and Y. Yang, "Evaluation of a WeChat-based dementia-specific training program for nurses in primary care settings: A randomized controlled trial," *Appl. Nursing Res.*, vol. 38, pp. 51–59, Dec. 2017.

[27] K. Zhou et al., "Benefits of a WeChat-based multimodal nursing program on early rehabilitation in postoperative women with breast cancer: A clinical randomized controlled trial," *Int. J. Nursing Stud.*, vol. 106, Jun. 2020, Art. no. 103565.

[28] A. Cheng, G. Ren, T. Hong, K. Nam, and C. Koo, "An exploratory analysis of travel-related wechat mini program usage: Affordance theory perspective," in *Proc. Int. Conf. Inf. Commun. Technol. Tourism*. Nicosia, Cyprus: Springer, Jan. 2019, pp. 333–343.

[29] Q. Rao and E. Ko, "Impulsive purchasing and luxury brand loyalty in WeChat mini program," *Asia Pacific J. Marketing Logistics*, vol. 33, no. 10, pp. 2054–2071, Oct. 2021.

[30] X. Chen, X. Zhou, H. Li, J. Li, and H. Jiang, "The value of WeChat application in chronic diseases management in China," *Comput. Methods Programs Biomed.*, vol. 196, 2020, Art. no. 105710.

[31] Q. Liang and C. Chang, "Construction of teaching model based on WeChat mini-program," *Int. J. Sci.*, vol. 16, no. 1, pp. 54–59, 2019.

[32] Y. Sui, T. Wang, and X. Wang, "The impact of WeChat app-based education and rehabilitation program on anxiety, depression, quality of life, loss of follow-up and survival in non-small cell lung cancer patients who underwent surgical resection," *Eur. J. Oncol. Nursing*, vol. 45, Apr. 2020, Art. no. 101707.

[33] Y. Liu et al., "Industry practice of Javascript dynamic analysis on WeChat mini-programs," in *Proc. 35th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Sep. 2020, pp. 1189–1193.

[34] J. Hu, L. Wei, Y. Liu, and S.-C. Cheung, "*w*test: WebView-oriented testing for Android applications," 2023, *arXiv:2306.03845*.

[35] C. Wang, R. Ko, Y. Zhang, Y. Yang, and Z. Lin, "Taintmini: Detecting flow of sensitive data in mini-programs with static taint analysis," in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng. (ICSE)*, May 2023, pp. 932–944.

[36] W. Li et al., "MiniTracker: Large-scale sensitive information tracking in mini apps," *IEEE Trans. Dependable Secure Comput.*, vol. 14, no. 8, pp. 1–17, Aug. 2015.

[37] Y. Yang, Y. Zhang, and Z. Lin, "Cross miniapp request forgery: Root causes, attacks, and vulnerability detection," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2022, pp. 3079–3092.

[38] J. Zhang, L. Yang, Y. Han, Z. Xiang, and X. Hei, "A small leak will sink many ships: Vulnerabilities related to mini-programs permissions," in *Proc. IEEE 47th Annu. Comput., Softw., Appl. Conf. (COMPSAC)*, Jun. 2023, pp. 595–606.

[39] C. Wang, Y. Zhang, and Z. Lin, "Uncovering and exploiting hidden APIs in mobile super apps," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.* NY, USA: Association for Computing Machinery, Nov. 2023, pp. 2471–2485, doi: 10.1145/3576915.3616676.

[40] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri, "Code injection attacks on HTML5-based mobile apps: Characterization, detection and mitigation," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2014, pp. 66–77.

[41] T. Li et al., "Unleashing the walking dead: Understanding cross-app remote infections on mobile WebViews," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 829–844.

[42] R. Wang, L. Xing, X. Wang, and S. Chen, "Unauthorized origin crossing on mobile platforms: Threats and mitigation," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, 2013, pp. 635–646.

[43] S. Son, D. Kim, and V. Shmatikov, "What mobile ads know about mobile users," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2016, pp. 1–14.

[44] P. T. Lau, "Static detection of event-driven races in HTML5-based mobile apps," in *Proc. Int. Conf. Verification Eval. Comput. Commun. Syst.* Porto, Portugal: Springer, 2019, pp. 32–46.

[45] (2021). *Mind the Bridge—New Attack Model in Hybrid Mobile Application*. Accessed: Oct. 6, 2021. [Online]. Available: https://conference.hitb.org/hitbsecconf2021ams/materials/D2T1%20-%20A%20New%20Attack%20Model%20for%20Hybrid%20Mobile%20Applications%20-%20Ce%20Qin.pdf

[46] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin, "Attacks on WebView in the Android system," in *Proc. 27th Annu. Comput. Secur. Appl. Conf.*, Dec. 2011, pp. 343–352.

[47] T. Luo, X. Jin, A. Ananthanarayanan, and W. Du, "Touchjacking attacks on web in Android, iOS, and windows phone," in *Proc. Int. Symp. Found. Pract. Secur.* Montreal, QC, Canada: Springer, 2012, pp. 227–243.

[48] E. Chin and D. Wagner, "Bifocals: Analyzing WebView vulnerabilities in Android applications," in *Proc. Int. Workshop Inf. Secur. Appl.* Jeju Island, South Korea: Springer, 2013, pp. 138–159.

[49] C. Rizzo, L. Cavallaro, and J. Kinder, "BabelView: Evaluating the impact of code injection attacks in mobile webviews," in *Proc. Int. Symp. Res. Attacks, Intrusions, Defenses*. Heraklion, Greece: Springer, 2018, pp. 25–46.

[50] W. Song, Q. Huang, and J. Huang, "Understanding Javascript vulnerabilities in large real-world Android applications," *IEEE Trans. Dependable Secure Comput.*, vol. 17, no. 5, pp. 1063–1078, Sep. 2020.

[51] S. Smalley and R. Craig, "Security enhanced (SE) Android: Bringing flexible MAC to Android," in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, 2013, pp. 1–18.

[52] G. Hernandez, D. J. Tian, A. S. Yadav, B. J. Williams, and K. R. Butler, "BigMAC: Fine-grained policy analysis of Android firmware," in *Proc. 29th USENIX Security Symp. (USENIX Security)*, 2020, pp. 271–287.

[53] G. Yang, J. Huang, and G. Gu, "Iframes/popups are dangerous in mobile WebView: Studying and mitigating differential context vulnerabilities," in *Proc. 28th USENIX Security Symp. (USENIX Security)*, 2019, pp. 977–994.

[54] G. Yang, J. Huang, G. Gu, and A. Mendoza, "Study and mitigation of origin stripping vulnerabilities in hybrid-postMessage enabled mobile applications," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2018, pp. 742–755.

[55] Y. Cao, V. Rastogi, Z. Li, Y. Chen, and A. Moshchuk, "Redefining web browser principals with a configurable origin policy," in *Proc. 43rd Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2013, pp. 1–12.

[56] Y. Cao, V. Yegneswaran, P. A. Porras, and Y. Chen, "PathCutter: Severing the self-propagation path of XSS JavaScript worms in social web networks," in *Proc. NDSS*, 2012, pp. 1–14.

[57] Y. Cao, Z. Li, V. Rastogi, Y. Chen, and X. Wen, "Virtual browser: A virtualized browser to sandbox third-party JavaScripts with enhanced security," in *Proc. 7th ACM Symp. Inf., Comput. Commun. Secur.*, May 2012, pp. 8–9.

[58] Y. Cao, Y. Shoshitaishvili, K. Borgolte, C. Kruegel, G. Vigna, and Y. Chen, "Protecting web-based single sign-on protocols against relying party impersonation attacks through a dedicated bi-directional authenticated secure channel," in *Proc. Int. Workshop Recent Adv. Intrusion Detection*. Gothenburg, Sweden: Springer, 2014, pp. 276–298.

[59] S. Li, M. Kang, J. Hou, and Y. Cao, "Detecting Node.js prototype pollution vulnerabilities via object lookup analysis," in *Proc. 29th ACM Joint Meeting Eur. Softw. Eng. Conf.* New York, NY, USA: Association for Computing Machinery, 2021, pp. 268–279.

[60] M. Georgiev, S. Jana, and V. Shmatikov, "Breaking and fixing origin-based access control in hybrid web/mobile application frameworks," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2014, pp. 1–15.

[61] K. Singh, "Practical context-aware permission control for hybrid mobile applications," in *Proc. Int. Workshop Recent Adv. Intrusion Detection*. Rodney Bay, St. Lucia: Springer, 2013, pp. 307–327.

[62] D. Davidson, Y. Chen, F. George, L. Lu, and S. Jha, "Secure integration of web content and applications on commodity mobile operating systems," in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, Apr. 2017, pp. 652–665.

[63] P. H. Phung, A. Mohanty, R. Rachapalli, and M. Sridhar, "HybridGuard: A principal-based permission and fine-grained policy enforcement framework for web-based mobile applications," in *Proc. IEEE Secur. Privacy Workshops (SPW)*, May 2017, pp. 147–156.

[64] J. Chen, V. Paxson, and J. Jiang, "Composition kills: A case study of email sender authentication," in *Proc. 29th USENIX Security Symposium (USENIX Security)*, 2020, pp. 2183–2199.

[65] J. Chen et al., "We still don't have secure cross-domain requests: An empirical study of CORS," in *Proc. 27th USENIX Security Symposium (USENIX Security)*, 2018, pp. 1079–1093.

Zhibo Zhang received the B.S. degree in information security from Fudan University, Shanghai, China, in 2019, where he is currently pursuing the Ph.D. degree in cyberspace security. He focuses primarily on the areas of mobile security and program analysis, specifically exploring novel attacks and security threats in mobile platforms.

Lei Zhang received the Ph.D. degree from the School of Computer Science, Fudan University, in 2020. He focuses on the system and software security, including vulnerability detection, exploitation, and repairing by utilizing program analysis and AI technique.

Guangliang Yang received the Ph.D. degree from Texas A&M University in 2019. He received a Post-Doctoral Fellowship from Georgia Tech from 2019 to 2021. He is currently an Assistant Professor with the School of Computer Science, Fudan University. His research interests primarily focus on computer system security and AI security.

Yanjun Chen received the bachelor's degree from Xiamen University, China, in 2020, and the master's degree in cyberspace security from Fudan University in 2022. Her research interests include mobile and web security.

Jiahao Xu received the bachelor's degree from China University of Geosciences, Wuhan, China, in 2016. He is currently pursuing the master's degree in cyberspace security with Fudan University. His research interests include mobile and web security.

Min Yang received the B.Sc. and Ph.D. degrees in computer science from Fudan University, Shanghai, China, in 2001 and 2006, respectively. He is currently a Professor with the School of Computer Science, Fudan University. His research interests include system security and AI security.