RootFree Attacks: Exploiting Mobile Platform's Super Apps From Desktop

Chao Wang
The Ohio State University

Yue Zhang
Drexel University

Zhiqiang Lin
The Ohio State University

ABSTRACT

In recent years, there has been a surge in the popularity of mobile super apps, which consolidate a variety of services, including messaging, ride-hailing, and e-commerce, into a single application, eliminating the need to switch between different apps. Originally tailored for mobile usage, super apps like WeChat and WeCom have expanded their reach to desktop platforms, including Windows. However, different operating systems have different threat models (e.g., Windows can directly grant users with root privilege but Android and iOS do not). Therefore, the single super app (including both its host app and miniapps) can face completely different threats in different platforms. In this paper, we systematically study the attacks caused by the discrepancies from different platforms. Specifically, we show that there are at least two classes of attacks, dubbed RootFree attacks, against mobile super apps: layer below that attacks the super apps from privileged software, and layer up that attacks the super apps from the internal malicious miniapps. We have disclosed our attacks and the corresponding vulnerabilities to the host app vendor, and received bug bounties. These vulnerabilities all are ranked as high severity vulnerabilities, and some of them have already been patched.

CCS CONCEPTS

• **Security and privacy** → **Web application security**; **Mobile and wireless security**.

KEYWORDS

Hidden APIs, Superapp Security, Miniapp Security, Web Security, App-in-App Security

ACM Reference Format:
Chao Wang, Yue Zhang, and Zhiqiang Lin. 2024. RootFree Attacks: Exploiting Mobile Platform's Super Apps From Desktop. In *Proceedings of the 2024 ACM ASIA Conference on Computer and Communications Security (AsiaCCS '24), July 1–5, 2024, Singapore.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

A super app is a comprehensive application designed to offer users access to a wide array of diverse services, encompassing functions like online shopping, ride-hailing, and instant messaging, all conveniently accessible through a single platform. This innovative concept has gained significant traction, with numerous prominent super apps emerging on the global stage. One of the pioneering examples of a super app is China's WeChat, which has garnered immense popularity. The concept of super apps is rapidly expanding its reach. For instance, even platforms that initially served a narrower purpose are evolving into super apps. Whatsapp, initially known for its messaging capabilities, is gradually transforming into a super app. Many other super apps have also followed, and these include Paytm from India, Grab from Singapore, GoTo from Indonesia, Zalo from Vietnam, and Kakao from South Korea [6]. Among these multifaceted applications, WeChat stands out, boasting an extensive user base of over one billion monthly users [22]. Its significance is underscored by its provision of an incredibly diverse range of everyday services. These encompass not only routine tasks like online shopping and instant messaging but also more specialized functions like booking medical appointments and even facilitating legal processes such as divorce filings [1].

However, the task of providing an exhaustive array of daily services by a single super app company is an unfeasible endeavor. To address this challenge, super app giants like WeChat have introduced a solution — they have opened up their platforms by providing Application Programming Interfaces (APIs) to third-party developers. This strategic move has given rise to what is commonly referred to as the miniapp or app-in-the-app paradigm, a revolutionary concept in the super app landscape [25]. Essentially, this paradigm enables external developers to craft applications that run seamlessly within the ecosystem of the super app. WeCom, as an enterprise-oriented extension of WeChat, has also followed this suit. The implications of this approach are far-reaching and transformative. By allowing third-party developers to contribute to the super app's ecosystem, the platform becomes a hub for a multitude of miniapps, each catering to specific needs and services. This concept is exemplified by WeChat, which boasts an astounding offering of over 4.3 million miniapps [20]. In comparison, the runner-up in this realm, Alipay, presents around 120 thousand miniapps [3].

Interestingly, we notice that WeChat, WeCom, and Alipay have taken steps towards integration with desktop platforms, particularly Windows. An interesting distinction emerges, with WeChat and WeCom exhibiting a more assertive approach compared to Alipay, allowing all their miniapps to function seamlessly on both desktop and mobile environments. This expansion, however, introduces a complex challenge: the discrepancies in threat models between desktop and mobile platforms. Windows, for instance, can grant users root privileges, a power not directly available on Android and iOS. This raises valid concerns about the capacity of WeChat and WeCom to ensure robust security for not only their overarching applications but also the miniapps. This is especially pertinent when considering that the host app lacks kernel privileges, yet it functions in a manner akin to an operating system from the perspective of a miniapp. This presents new opportunities for potential attacks on WeChat from the desktop version without compromising the mobile counterpart. The focal point of this paper is to address these uncertainties. We conduct a systematic exploration of the various attack surfaces presented by distinct platforms, intending to identify potential vulnerabilities within the super app framework. Through this analysis, we seek to shed light on the intricate interplay between security and adaptability in the context of super apps, ultimately contributing to a deeper understanding of their operational landscape.

Particularly, we demonstrate that there could exist at least two classes of attacks, dubbed RootFree attacks, depending on where they are originated from: layer below that start from the from privileged software, and layer up that start from the internal malicious miniapps. While layer below attack is conceptually similar to hypervisor attacks (e.g., Bluepill [26]) against guest OS or malicious OS against user applications (e.g., Iago [15]), there are still substantial differences in the super app context, particularly because there is a large body of sensitive data accessible through super apps and super app developers are also aware of such threat. For instance, we notice WeChat in Windows encrypts its miniapp packages using AES [12]. In contrast, the miniapp packages in Android and iOS do not use encryption at all. However, it is extremely challenging to protect user level software without the help from privileged software, but WeChat aims to do so through encryption. Unfortunately, we show that its current encryption implementation can be broken through dynamic hooking of cryptographic APIs as well as data dependence and differential analysis.

Meanwhile, since the running environments and the APIs provided by the super apps should be the same given that the miniapps running atop the super apps use the same suite of code, there are discrepancies in both the sandbox implementation as well as the API implementation across platforms. Particularly, mobile platforms (e.g., Android, iOS) use stricter isolation policies when compared with Windows, where the stronger sandbox is enforced to isolate apps (including the miniapps running atop a host app) from each other. Being a super app that can run on Windows, it should prevent the "inside" miniapps from accessing resources that are located "outside" of super apps. Although on Windows, WeChat introduces a software level isolation to prevent malicious miniapps from accessing other apps and OS's files, we show that its implementation is flawed, allowing a malicious miniapp to launch various attacks including remote code execution against OSs and host apps. Since this attack is from a malicious miniapp at the upper layer, we call it layer up attack.

Contributions. We make the following contributions:

- We show that super apps running atop different platforms may have a variety of discrepancies from threat models and implementations, some of which can lead to security vulnerabilities.
- We develop two classes of RootFree attacks (layer below and layer up) based on vulnerabilities identified from our binary analysis. We show these attacks can cause grave security consequences to the massive super apps users as well as miniapp developers. We articulate the attack details, as well as to provide a snapshot of the ecosystem.
- In addition to the responsible disclosure of these vulnerabilities to the super app provider, we also shed light on the possible countermeasures, so that other super app providers can properly design their apps and avoid the same security mistakes.

2 BACKGROUND

In this section, we begin by delving into the concept of miniapps (§2.1). We then explore the security measures implemented within super apps (§2.2). Lastly, we will analyze the distinctions among miniapps when they operate on various platforms (§2.3).

2.1 Super Apps: the Hybrid Mobile and Web Apps

In the desktop era, web apps dominated by running inside browsers, accessible via URLs. However, they lack native app capabilities like directly accessing device features (e.g., Bluetooth, sensors, and NFC). Moreover, a lack of a centralized vetting mechanism to assess the security of web applications exacerbates the situation. This gap enables individuals to deploy their own web apps with minimal restrictions as long as they possess a web server. Consequently, this scenario paves the way for the proliferation of numerous malicious web applications.

In the mobile era, users shifted to app stores for native mobile apps due to centralized vetting and native OS support. But, storage limits, download requirements, and platform-specific development pose challenges. Super apps like WeChat expanded from social networks to offer varied services. They run miniapps, benefiting from web and native capabilities, enhancing user experience. Super apps host miniapps in a sandbox, offer in-app APIs for system resources, and use script and markup languages for cross-platform support.

2.2 Security Mechanisms in Super Apps

Miniapps enforce stricter distribution policies compared to native or desktop apps, primarily as a defense against malware. Preinstallation vetting measures serve as the initial line of defense, where miniapps undergo thorough reviews before distribution. These miniapps are exclusively available through official markets, distinct from desktop and mobile apps that can be sourced from alternative app stores. To ensure security compliance, any miniapp discovered to contain malicious payloads must rectify them or face removal from the market. Additionally, dynamic code updates, a vector for malware, are mitigated in miniapps through limitations on dynamic code execution and WebAssembly usage.

Post-installation protection measures come into play during direct user interaction with miniapps, aiming to safeguard against potential malware attacks. Local resource access protection restricts miniapps from accessing system resources without user authorization, similar to Android's permission mechanism. Additionally, miniapps are constrained in their capabilities, including limitations on accessing user files, SMS messages, keystrokes, or background execution. If a miniapp remains in the background for more than five minutes, the super app terminates it.

2.3 The Differences Across Platforms

While super apps are initially designed to be executed only on mobile platforms such as Android and iOS, recently we also notice that some of them, particularly WeChat and WeCom, can also be executed on top of desktop platforms such as Windows. However, since different OSs can have different threat models and implementations, the execution of the super apps (including its miniapps as well) can be impacted due to the discrepancies. As summarized in Table 1, we conducted a comprehensive systematic study of superapps' documentations (e.g., WeChat [29], Alipay [28]) and identified the presence of at least five distinct types of discrepancies.:

(I) **User Privileges.** In mobile platforms, end users typically do not have any root privileges, so do the apps executed. Therefore, super apps can assume the trust of the operating systems (e.g., no malware from the kernels). In fact, it is extremely costly and also challenging to obtain the root privilege for both mobile apps and mobile users. For instance, a zero-click jailbreak vulnerability can cost millions of dollars [16]. There are also tools (e.g., Appdome Prevention [9]) to defend against app rooting. Unfortunately, however, desktop users can easily become administrators without exploiting any vulnerabilities since the system owner by default can have such a privilege (desktop PCs tend to give their owners more flexible control), and they can install privileged kernel modules or any other software (e.g., a debugger) to monitor the execution of super apps.

(II) **Process Isolations.** Different OSs could have different process isolation policies and mechanisms. In general, mobile OSs tend to have stricter policies and stronger isolations than desktop OS (due to the rich privacy data stored in mobile platforms), as shown in Table 1, more specifically:

- **Android.** The Android platform leverages the Linux user-based protection to identify and isolate app resources from each other including malicious apps. To do so, Android assigns a unique user ID (UID) to each Android app and makes it run in its own process, and meanwhile a strong mandatory access control from SE-Linux is enforced over all processes even for the ones running with root privilege [5]. Therefore, an Android app at runtime is protected in a sandbox and it cannot be easily escaped to access any other app's code and data. Additionally, trusted execution environment such as TrustZone is also widely available in ARM processors (e.g., MediaTek [21] and Qualcomm [8]). We notice super apps (e.g., Alipay) have started to take advantage of this feature to protect the app secrets.
- **iOS.** Similar to the app running atop Android, the apps' data, code and runtime on iOS are also strongly isolated. To be more specific, although iOS can directly leverage its UNIX kernel/user isolation (iOS uses Darwin UNIX [24] as its kernel) to isolate its apps, it chooses to implement a customized sandbox on its own, which is a set of fine-grained rules limiting the app's access to files, hardware resources, and so forth. In particular, iOS will create a data directory for each app under "`/private/var/mobile/Containers/Data/Application`" directory, and the directory name is a random GUID. Moreover, the app cannot access any data directories other than its own sandboxed one. Also, Apple provides secure enclaves [4], a hardware enforced trusted execution environment (TEE), for iOS apps to protect sensitive data from being compromised.
- **Windows.** Unlike mobile platforms, there is almost no strong app data isolation enforced by Windows (unless users intentionally places their apps into Windows SandBox [7, 32]). As such, the app running atop Windows can access other apps' data. Moreover, although CPU vendors have made considerable efforts in trusted computing (e.g., Intel launched the SGX Trusted Execution Extension in 2016 [11]), Windows currently does not provide a unified APIs to allow 3rd party developers fully leverage SGX for app secret protection. Meanwhile, recently Intel even dropped the support of SGX in desktop CPUs [2]. Therefore, it is very unlikely that super apps will soon use TEEs in desktops.

(III) **In-App APIs.** The miniapps can access resources under the supervision of the host app through the in-app APIs. Typically, those APIs define the types of requests that can be made by these miniapps, allowing the miniapps developers to customize the functionalities independently. While most of the APIs are the same regardless of the platforms, we notice that there are indeed some APIs implemented differently across the platforms (and this observation has also been explored by most recent work APIDiff [34]). To name a few, mobile platforms such as Android and iOS have native support of multiple sensors, while Windows does not have. As such, there are a set of APIs that can only be invoked at the Android or iOS version of the miniapps. For example, the NFC APIs are only supported in Android platform, and similarly APIs including Gyroscope, Compass and Accelerometer are usually enabled at the mobile platforms.

(IV) **Rendering Engines.** A rendering engine is needed to render a variety of view components (e.g., input boxes, buttons) of the miniapps, and this engine is an important component of the super app. Currently, both the Android and Windows version of WeChat and WeCom use XWeb [30], whereas the iOS version uses WKWebView, part of the WebKit [10] framework.

	Mobile OS		**Desktop OS**
	Android	**iOS**	**Windows**
(I) User Privileges	Non-root	Non-root	Root
(II) Process Isolations			
├App Code	✓	✓	✗
├App Data	✓	✓	✗
├App Runtime	✓	✓	✗
└Trusted Execution Environment	✓	✓	✗
(III) In-App APIs	All APIs	All APIs	Subset of APIs
(IV) Rendering Engines	XWeb	WKWebView	XWeb
(V) Logic Engines	JSCore	JavaScriptCore	JSCore

Table 1: The discrenpancies of super app execution across platforms

(V) **Logic Engines.** A logic engine (also known as JavaScript engine) is the super app's component that executes JavaScript code. Currently, WeChat and WeCom use JavaScriptCore in iOS, and JSCore (which is built upon V8 engine) in Android and Windows. Note that the JSCore engine has also been used by other super apps including Tiktok [17].

3 OVERVIEW

In this section, we present an overview of the attacks against super apps. We first describe the key observation we have and the goal of this work in §3.1; then describe our assumptions and the scope in §3.2; and finally provide an overview of our proposed attacks in §3.3.

3.1 Key Observation and Research Questions

Key Observation. Having explained the various discrepancies in §2.3, we can notice that super apps when running in different OSs will obviously have different isolation polices and also different privileges. Particularly, unlike the desktop Windows which inherits from the traditional multi-user model and grants computer owners root privileges, mobile OSs are designed to be exclusively used by a single user without the need of granting any root privileges since they just need to make sure the owner can download and run multiple apps (some of which may be untrusted), and a malicious app cannot access other apps' data.

Once allowing a super app to be executed on top of both platforms, super app developers must take into considerations of these platform discrepancies and their corresponding threats, and protect their assets (e.g., the secrecy and integrity of the miniapps's code and data, as well as the entire ecosystem including the secret data of other super app users). Fortunately, we do notice that super apps such as WeChat and WeCom have considered these threats (or at least partially). For instance, we notice WeChat and WeCom have provided a software level sandbox preventing a miniapp from accessing data outside the sandbox in both desktop and mobile platforms. Meanwhile, they also encrypt the miniapp code and data distributed to the desktop platforms, whereas the miniapps in mobile platforms do not have such protection.

Research Questions. However, there are still many research questions to be answered. We now list the research questions:

RQ1 *Have* WeChat *and* WeCom *systematically guarded the threats from desktop platforms, such as the privileged software attack which has been proven to be extremely challenging to defend against?*

RQ2 *While they have used encryption to protect the secrets, has the corresponding key properly managed and distributed?*

RQ3 *Although they have sandboxed the execution of miniapps, will the discrepancies across platforms weak the protection?*

RQ4 *Since desktop and mobile OSs have different policies when managing resources, can this be exploited by miniapps?*

Therefore, this work aims to answer these questions, by systematically examining the attack surfaces, inspecting the implementations, and designing proof of concept attacks to raise the awareness, so that future super app vendors can learn these lessons.

3.2 Scope and Assumptions

Scope. Our study concentrates on WeChat among the various super apps available due to several key factors. With a massive one billion monthly active users, WeChat's vulnerabilities could have significant impacts. Additionally, its pioneering of the miniapp concept has led to over 4.3 million miniapps, outstripping competitors like Alipay and Snapchat. Furthermore, our study necessitates that the targeted superapps have a desktop version capable of running miniapps. We have identified that, among the superapps we examined, only WeChat supports the execution of miniapps on its desktop platform. In contrast, other superapps like Alipay, while they may have desktop versions, do not currently support miniapps on their desktop platforms. Also, our investigation confirms that WeCom shares the same miniapp framework, making WeChat the primary focus for our proof of concept.

Additionally, although we have outlined five categories of discrepancies as discussed in Table 1, we only focus on exploiting the first two, namely user privileges, and process isolation. We do not focus on API differences since this has been studied by APIDiff [34]. While it is true that logic engines and render engines of the super apps are implemented differently in different platforms, they all serve the same purposes without much change (e.g., rendering the view components or executing the JavaScript code). As such, exploiting the logic engine and render engine of the super apps are out of our focus.

Assumptions. We assume the trustworthiness of the code from the OSs and the super apps, since we do not believe they have any malicious intentions (otherwise, they can trivially make any attacks, as in Bluepill [26] to attack guest OS from hypervisor, or Iago [15] to attack native apps from malicious OS). That is, we exclude malicious kernel module attacks (e.g., Windows rootkits) by assuming the trustworthiness of the OS kernel code.

We also assume the trustworthiness of the super apps' backends or miniapps' backends, since they are out of direct reach from attackers. To make our attacks more focused, we assume that there could be a malicious native app such as a Windows spyware, or a malicious miniapp installed onto a victim's device. This assumption is reasonable since previous efforts [25] made a similar assumption, and also practical since there are numerous Windows malwares in the wild.

Lastly, we do not assume that Windows users have administrator privileges by default. Instead, our paper assumes that the attacker will not obtain administrator privileges unless explicitly stated otherwise.

3.3 Attack Overview

In this paper, we focus on systematically uncovering the discrepancies of super apps across platforms, and understanding their security consequences by developing the corresponding RootFree attacks. As summarized in Table 2, there could be two classes of attacks depending on where they are originated:

- **RootFree Attacks from Layer Below (§4).** The super apps can be subject to attacks from layer below such as the malicious OS attacks (e.g., a compromised kernel with rootkits without user's awareness). The attackers in this category can exploit the user

	Vulnerable Platform			Victim	Trusted Party	Attacker
	Android	iOS	Windows			
Attacks From Layer Below						
├(A1) Code Confidentiality and Integrity Attacks against Benign Miniapps	✗	✗	✓	📄,👤	⊞, 💬,👤	🕵,Ⓐ
└(A2) Data Confidentiality and Integrity Attacks for Malicious Miniapps	✗	✗	✓	📄,💬,👤	⊞, 💬,👤	🕵,Ⓐ
Attacks From Layer Up						
└(A3) Sandbox Escaping due to Flawed and also Weak File System Isolation	✗	✗	✓	👤,⊞, 💬,	⊞, 💬,👤	📄

Table 2: Summary of Attacks. 📄 Innocent Miniapps; 📄 Malicious Miniapps; Ⓐ Malicious Host Apps; ⊞ OSs; 💬 Host Apps; 👤 Innocent Desktop Users; 🕵 Malicious Desktop Users;

privilege discrepancies. For instance, the attacker (who can be a malicious user of the OS) can install kernel malware to monitor super app's execution, such as stealing the sensitive information from or inject malicious data to a victim miniapp. Also, attackers can install user level spyware (e.g., a debugger) to monitor super app's execution without using any kernel level privilege. We still call all these attacks layer below to ease the understanding and also differentiate from layer up attacks. Note that mobile platforms such as Android and iOS have enforced centralized vetting to prevent malicious apps from being installed, but Windows allows administrators to have complete control such as installing arbitrary software (both user space and kernel space), thereby significantly threatening the secrecy and integrity of super apps.

- **RootFree Attacks from Layer Up (§5).** Since the super apps provide the running environment for miniapps, they can be the targets of the layer-up attackers from malicious miniapps. Specifically, the attackers from layer up can exploit the process isolation discrepancies. For instance, a malicious miniapp running atop Windows can jail-break the isolation enforced by their super apps, and then they can read, write data on the OSs, and even execute code remotely.

3.4 Our Methodology

Workflow. In this paper, we focus on examining the security risks associated with super apps and miniapps on desktop platforms. Our goal is to identify potential attack surfaces, develop concrete attacks, and understand the security consequences. To accomplish this, we employ a reverse engineering methodology by developing code based on the target. We inspect the resources (code and data) that can be accessed on different platforms and compare the enabled protections such as encryption and file isolation. Having identified the attack surfaces, we then developed an attack confirmation code and concrete case studies to demonstrate the attacks. Although theoretically, attacks could be launched against all files that have weak protections, we only select some of them for proof of concepts. The attack confirmation code is mainly used to verify if the identified weaknesses are exploitable, such as breaking the encryption of miniapp code. Finally, with the verified weaknesses, we develop concrete code to launch the attacks on a case-by-case basis.

We acknowledge that there are certain manual efforts involved. Reverse engineering is a significant manual effort where we analyze and deconstruct the target applications and their components. It involves manually dissecting the code, understanding its structure, and identifying potential vulnerabilities. Second, identifying potential attack surfaces requires a manual assessment of the application's architecture and dependencies. We must pinpoint areas where vulnerabilities may exist, which can be a meticulous process. Finally, creating concrete case studies to demonstrate the attacks involves planning and executing specific scenarios that showcase the security risks.

Testing Environment. To study the security issues and impact of our targeted super app WeChat, we have registered several user accounts, downloaded the corresponding miniapp development tools and SDKs, followed their official documents to build testing miniapps. Some experiments require us to reverse engineer WeChat, and therefore, we used JEB and IDA Pro [18] to inspect the de-complied code statically and programmed Frida [19] scripts to dynamically verify our findings or exploits when needed. In total, we have developed around 2,000 lines of code (LoC) using five different languages (i.e., JavaScript, Bash, Golang, C/C++, and TypeScript).

4 ROOTFREE FROM LAYER BELOW

4.1 Attack Surface Analysis

Insight. To launch an attack against WeChat, we have to first understand its attack surface, particularly the critical conditional branches such as the permission checks and resources accesses. However, WeChat is a giant software, whose Windows, Android, and iOS version has 172, 223, and 230 MB, respectively. Statically analyzing all of these binary code, to uncover these conditional branches will be extremely challenging. Therefore, we have to resort to dynamic analysis — executing WeChat in the three platforms and observing the execution differences, from which to uncover the critical code and data for the attacks with a differential analysis.

However, there are still multiple challenges when using the differential analysis. First, we still have to identify the alignment in the WeChat's execution; but WeChat has extremely complicated control flows. Second, the implementation at the binary code level is also quite different, since WeChat in different platforms is developed using different languages (e.g., Windows in C/C++, Android in Java, and iOS in Objective-C). Therefore, we have to identify the appropriate code that is commonly executed in all three platforms. Fortunately, one such code is the miniapp code, since WeChat's goal is to allow 3rd-party developers to develop one program and execute it in all platforms. Meanwhile, at the WeChat host code, we can hook at the system call level (e.g., file opening) and align them, because the host binary code is quite different across platforms but system call level behavior is understandable.

Differential Analysis. As such, we have developed a testing miniapp, and hooked the system APIs of the three different platforms to observe the operating system level behaviors, such as the files that are involved in the execution of a miniapp. During the analysis, we mainly focused on two categories of files: (i) the files that are platform independent; (ii) the OS unique files, particularly the "Windows-unique" files, given that those files may be important to some Windows unique features. Below, we summarize our findings in these two categories:

- **Files contained in a miniapp package.** A miniapp is packed in a compressed file in the format of `wxapkg`. As shown in Table 3, each packed miniapp package contains the following file types: (i) the code of the miniapps (usually in the format of JavaScript files), which is not platform specific and must be accessed every time miniapps are launched; (ii) the resource files, which can be images, audios, or videos; (iii) the package configuration files, which specify the general features of the miniapps such as permission the miniapp requires; (iv) the UI pages, which describe how each UI window looks like (e.g., the how many buttons and input boxes). All these files are packed as a whole, and different OSs use different measures to protect the confidentiality and integrity of the package. Since Android and iOS have strict isolation, the packages of the downloaded miniapps are out of reach of attackers. Windows, however, does not have such protection, and the super app encrypts the package to protect them.
- **Files produced during miniapp execution.** There are also files that are produced by the miniapps. As shown in Table 3, we also summarize those files as follows. (i) Log files. Regardless of the platforms, miniapps usually maintain log files for debugging purposes; (ii) Database files. Interestingly, we notice that for WeChat in Windows, some databases files are created. One such file is `WeChatApps.data`, which is a database that manages the mini-apps that the user can invoke. In particular, on Windows, miniapps can be cached, and users can also create a shortcut for future references. To do so, WeChat will add the appID of the corresponding miniapp into the database, and the user can then invoke it directly by clicking the shortcut. On Android (as well as iOS), there is a feature named "add to home screen", which has a similar workflow to allow a user to cache a miniapp with a shortcut to the user's home screen. However, on Android and iOS, the feature is implemented at the system level, and the host app is not involved in this process. (iii) The miniapp specific configuration files. We noticed that the miniapps might create and access some configuration files used to record the miniapp specific data (e.g., their user's credentials and game archives).

Attack Target. Having analyzed the execution differences of miniapps, we inspect further the two particular files: the miniapp code, and the `WeChatApps.data`. Interestingly, we find these two files are actually both encrypted in Windows, but not in mobile platforms. More importantly, we find they are two very critical files that need to be protected in Windows platform. Specifically, the miniapp is developed using JavaScript, and its intellectual property (e.g., the specific algorithm and business model) can be easily leaked from the code (since analyzing scripting code is generally easier than analyzing native binaries). Second, with the leaked miniapp code, the attacker may preform white box testing, and even reuse some of the components to attack the miniapps. Finally, the attacker can also easily modify the script code for malicious purposes, e.g., inserting malicious code in payment miniapp to steal user's money. Therefore, WeChat encrypts the miniapp code in the Windows platform, since arbitrary users can open the corresponding folder and inspect the miniapp code, but users in mobile platform do not have this capability.

File Type	Mobile OS		Desktop OS
	Android	iOS	Windows
Miniapps package			
├Code	📦	📦	🔒
├Resources	📦	📦	🔒
├Package Configurations	📦	📦	🔒
└UI pages	📦	📦	🔒
Files produced by miniapps			
├Database	📦	📦	🔒
├Configuration	📦	📦	🔒
└Log	📦	📦	✗

Table 3: Summary of File Protections on Different OSs. 📦 represents "isolation". 🔒 represents "encryption".

With respect to `WeChatApps.data`, it is also a crucial file encrypted in Windows. In particular, it concerns whether a miniapp can be executed or not. In Windows, we find that WeChat will only execute the miniapp indexed in `WeChatApps.data`. Only WeChat vetted miniapps will be added into this file. Encrypting `WeChatApps.data` will prevent malicious miniapps from being distributed. For example, if there is no database that records which miniapp is added by the user, the attacker can freely distribute their malicious miniapps by releasing the miniapp packages in the corresponding folders and creating shortcuts to invoking them, thereby completely bypassing the centralized vetting process enforced by WeChat.

4.2 Concrete Attacks

Our attack surface analysis has revealed that WeChat has used encryption to protect the secrecy and integrity of important files, and such protection is performed at software layer without using any hardware enforced TEE techniques. Since the only secret of modern cryptography is the cryptographic keys [27], we just have to break the keys (such as the key generation or distribution) in order to break the encryption and decryption. While WeChat has used heavy obfuscation to thwart binary analysis, we can dynamically hook the cryptography APIs (e.g., AES and 3DES) using Frida [19], and use backward slicing [37] from their arguments to understand how the keys are generated. In the following, we describe in greater detail of how to perform these analysis, and launch our code confidentiality and integrity attacks against benign miniapps (i.e., break the secrecy and integrity of miniapp code), and data confidentiality and integrity attacks for malicious miniapps (i.e., integrity protection of `WeChatApps.data`).

(A1) Code Confidentiality and Integrity Attacks against Benign Miniapps. During the miniapp execution on Windows platform, we noticed the miniapp code will be decrypted first, since what downloaded from the WeChat server is an encrypted package

appID	App Categories	Attack Methods	Affected Feature	Attack Consequences
wx70c04******99f70	Hotel	Inspecting hard-code passcode	Code confidentiality	Unlocking Hotel rooms
wx09dac******fa7bc	Promotion App	Inspecting the code	Code confidentiality	Coupon code leakage
wxb647c******f14ad	Promotion App	Inspecting the hard-coded coupon code	Code confidentiality	Coupon code leakage
wx432c7******877f9	Promotion App	Inspecting the hard-coded coupon code	Code confidentiality	Coupon code leakage
wx79a83******a7978	Short Video	Launching phishing attacks and tricking users into inputting passwords	Code Integrity	User's credentials leakage
wx7ddec******7276a	Express Delivery	Recording user's address and shopping orders	Code Integrity	User's information leakage
wxca1fe******52094	Online Shopping	Launching phishing attacks and tricking users into inputting bank information	Code Integrity	User's bank account leakage
wx71d58******e3321	Public Transportation	Recording the user's locations	Code Integrity	Tracking the user's locations
wxdbee9******d6263	Instant Messages	Recording user's messages	Code Integrity	User's information leakage
wx94679******b069d	Online Shopping	Launching phishing attacks and tricking users into inputting bank information	Code Integrity	User's bank account leakage

Table 4: Attack case-studies

whereas in Android and iOS the miniapp code is in plaintext. By dynamically hooking all cryptographic APIs, and running different miniapps multiple times, we noticed one special function located in `WeChatWin.dll`, which always takes three parameters: a 32-bytes buffer, the cphiertext (which is a part of the encrypted miniapp packages), and a buffer that is used to store the decrypted data. We then confirmed that the 32 bytes buffer is the key, and traced the buffer to understand how the key is generated. Interestingly, we confirmed that the key is the output of PBKDF2 (Password-Based Key Derivation Function 2 [23], which is a standard key derivation function with a sliding computational cost), whose input is the appID. Therefore, we conclude that very likely WeChat uses the miniappID as the key. We then performed a forward slicing by tracing the appID, and identified that WeChat actually also uses the second last byte in the appID to generate an `XOR` key, which works together with the AES key (the key derived from PBKDF2) to encrypt and also decrypt aminiapp package.

Therefore, WeChat essentially uses appID, a 18-bytes string assigned by the WeChat as the key. Using miniappID as an encryption or decryption key makes a lot of sense. First, the key cannot be user specific; otherwise WeChat has to generate different keys for the same miniapp. Second, the key cannot be device specific too (though sometimes MAC address has been used to derive cryptographic keys) for the similar reason. Therefore, the key is likely miniapp specific.

Having identified the decryption function and the involved keys, we then trace further to understand exactly how WeChat decrypts a miniapp and this finding is presented in Figure 1 (Note that it turns out to be a non-standard crypto algorithm customized by WeChat). In particular, we notice that WeChat will first feed the appID, and a constant string "`saltiest`" into PBKDF2 to derive a 32-bytes decryption key `Key1` (Step ❶). Next, WeChat will take the first 1024 bytes of the miniapp (`C1`) and decrypt it using the produced decryption key, through which WeChat will obtain a data block that is 1,024 bytes (Step ❷). Next, WeChat will take the second last byte of the appID as the `XOR` key, and use it to exclusive or (`XOR`) the rest of the encrypted package of the miniapp (Step ❸). Finally, WeChat will put the derypted data block `P2` produced in the Step ❷, and the data blocks `P3` in the Step ❸, then add a string "`V1MMWX`" (i.e., `P1`) as an identification in the file header, which forms the final output of the decrypted package (Step ❹). In this process, despite WeChat's integrity check, which prevents modified mini-apps from executing, we can still make it run because the attacker is using a rooted model.

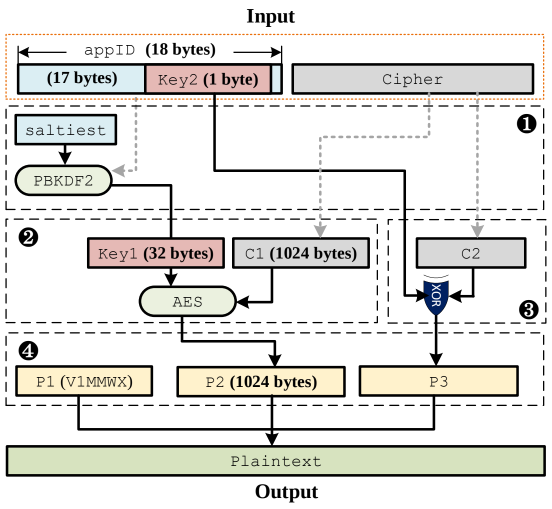

Figure 1: Workflow of the miniapp code decryption algorithm

Attack Case Studies. By using the decryption algorithm revealed above, an attacker (or a malicious app co-located with the super app) can decrypt (and encrypt) any miniapps of interest. Specifically, an attacker can

- **Break Code Confidentiality.** We developed a malware to scan the miniapps downloaded at "`C:/User/username/WeChat Files /Applet/` ", automatically decrypt them, and then unpack them to search strings of interest (e.g., "password", "coupon", "keys"). If so, it then uploads these to a remote server. As shown in Table 4, we have launched attacks to break the code confidentiality against four miniapps. To be more specific, the collected strings including hard-coded keys (which is used to encrypt or decrypt sensitive information), password, and coupon code. For example, we have found a hotel check-in miniapp (whose appID is "`wx70c 04***99f70`") has leaked the passcodes (which can be used to unlock the hotel rooms). As shown in Figure 2, `roomNo` is the room number of a hotel room, and `deviceCode` is the password that can be used to unlock the door of that room. There are five hotel rooms that are subject to this type of attack (line 7 – line 13). At line 16 – line 30, the check-in miniapp sends the passcode to a Bluetooth locker miniapp, which opens the door, and returns

```
  checkInOrder: function checkInOrder() {
     var self = this;
     console.log("Check-in", self.dataOrderDetails.roomNo);
     var deviceCode = "";
     deviceCode = self.dataOrderDetails.roomBlueDeviceCode;
     //      var deviceCode = "";
     //      if("0818" == self.dataOrderDetails.roomNo|| "8101" ==
     ↪  self.dataOrderDetails.roomNo || "8102" ==
     ↪  self.dataOrderDetails.roomNo|| "8103" ==
     ↪  self.dataOrderDetails.roomNo|| "8104" ==
     ↪  self.dataOrderDetails.roomNo|| "8105" ==
     ↪  self.dataOrderDetails.roomNo){
     //        deviceCode = "8E20FE921A07C4F3125AF13E"
     //      }else if("0816" == self.dataOrderDetails.roomNo){
     //        deviceCode = "9BF084673F860CB2B715B1F7";
     //      }else if("0817" == self.dataOrderDetails.roomNo){
     //        deviceCode = "48481B9ED0A894E36D993D3A";
     //      }else if("0901" == self.dataOrderDetails.roomNo){
     //        deviceCode = "6799BE3922FC0CB2B71BDAB4";
     //      }
     //Open a Bluetooth locker miniapp to open the door
       _wepy2.default.navigateToMiniProgram({
         appId: "wxe23c5b15a4150858",
         path: "",
         extraData: {
             DeviceCode: deviceCode
         },
         envVersion: "trial",
         success: function success(res) {
             // Door Opened
             console.log("Check-In", "Door opened");
         },
         fail: function fail(res) {
             console.log("Check-In", "Cannot open the door");
         }
     });
 }
```

Figure 2: A hotel check-in miniapp that leaked the passcode

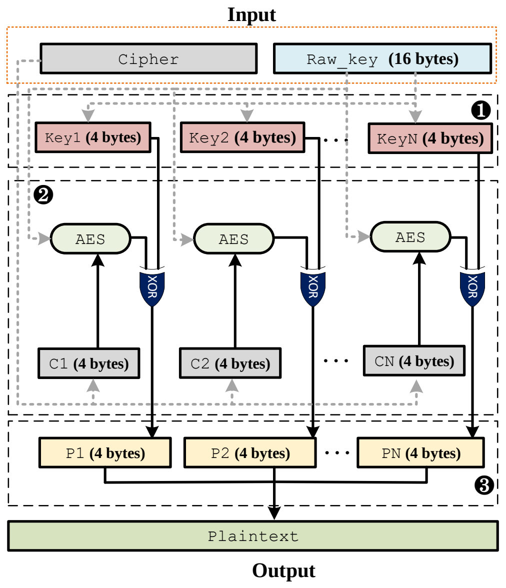

Figure 3: Workflow of the data decryption algorithm

the status (e.g., whether the door is opened or not). As such, theoretically, as long as the attacker can obtain the passcode from the code, he or she can create a malicious Bluetooth miniapp, send the passcode to the hotel lock to unlock the door.

- **Break Code Integrity.** We also add a few payloads in our malware and enable it to modify the victim miniapp for malicious purposes, such as tracking the victim's location). For example, a public transportation miniapp whose appID is "`wx71d58***e3321`", allows a user to track the next bus in real-time (e.g., whether the bus is close to the user or not). However, by adding the malicious code into this victim miniapp, attackers can now collect the user's real-time locations, and send the locations to a remote server. In some other examples (e.g., miniapp whose appID is "`wx94679***b069d`"), the attacker can modify the miniapps' control flow to steal the user's password or bank account. In those examples, in addition to adding a few pieces of code in the existing code, we also added a few UI pages and packed the pages into the miniapp packages to make the victim miniapps become phishing miniapps (e.g., the newly added pages may trick the user into believing they are parts of the victim miniapps, so that the user may input the information as they think they are interacting with the victim miniapps).

(A2) Data Confidentiality and Integrity Attacks for Malicious Miniapps. We followed the same approach as in A1 to trace the crypto function execution, and identify the corresponding keys. In particular, we hook all of the crypto functions, and inspect the ones that consume the data from `WeChatApps.data`. During this process, we identified one function located at `0x1131E260` that takes 6 parameters: `source_buffer`, `destination_buffer`, `length`, `aes_key_st`, `raw_key` and `isEncrypt`. Among them, the buffer `source_buffer` points to the content read from `WeChat Apps.data.`, and the other buffer `destination_buffer` points to the decrypted plaintext. Therefore, we conclude `raw_key` is the key.

However, unlike miniapp code encryption which directly uses the appID as the key, this time, the key generation is more complicated, and we have to trace back multiple functions to eventually resolve the key generation. Specifically, we performed a backward slicing with the trace, and identified that WeChat actually uses devices related information to generate the key, which is a string generated from the CPU information obtained by invoking `GetCurrentHwProfileW` (using bytes shifting). The produced key `raw_key` is 16 bytes.

Once we have resolved the key, we next trace it further to understand how the decryption is performed on `WeChatApps.data`, and this finding is presented in Figure 3. Interestingly, we also noticed that the decryption algorithm is a non-standard cryptographic algorithm customized by WeChat, which involves both AES decryption and `XOR` operations. At a high level, the algorithm first takes the produced key `raw_key` and the cipher as its inputs, then splits the produced key `raw_key` into multiple blocks, each of which is 4 bytes (Step ❶). Next, the algorithm splits the ciphers into multiple blocks, and each block is also 4 bytes. The algorithm decrypts each block using the original `raw_key`, and then uses the keys produced in Step ❶ to `XOR` each byte sequentially (e.g., the first key block is used to `XOR` the first cipher block). If the number of cipher blocks is larger than the number of the key blocks, the algorithm will reuse key blocks sequentially until there is no other cipher block left (Step ❷). Each produced output is a part of the plaintext, and WeChat finally puts all the outputs together to form the original plaintext (Step ❸).

Attack Case Studies. Similar to code confidentiality and integrity attacks, an attacker can launch the following data confidentiality and integrity attacks:

- **Breaking Data Confidentiality.** We developed a malware to scan the records of `WeChatApp.data`, through which to infer the users' privacy, since this file stores the frequently used miniapps. For example, if the attacker identifies that the user creates a shortcut for a hospital's miniapp, the attacker then knows that the user is likely a patient of that hospital. Then, if the hospital is a specialized hospital (e.g., andrology hospital), the attacker may further infer what kind of disease the user may have. As another example, the attacker can also know what kind of products fall into users' interests by inspecting the records of `WeChatApp.data` (e.g., the user may add a shortcut of clothing store), and then launch more targeted attacks (e.g., pushing cheaper clothing advertisement).
- **Breaking Data Integrity.** The attacker may distribute the malicious miniapps without WeChat's vetting by breaking data integrity. To be more specific, WeChat has enforced security vetting to guard its market, and every miniapp must go through the vetting process before delivering to the users. The vetting process can protect the users from malware attacks and violent content (e.g., WeChat will not allow miniapps to contain terrorism content). However, now that the attacker can freely add arbitrary appID into the database, he or she can directly place a malicious miniapp shortcut on victim user's `WeChatApp.data`, completely bypassing the vetting, which threatens not only victim's user privacy (e.g., collecting user's phone number, and bank account) but also breaks the entire ecosystem (e.g., spreading terrorism content) guarded by the super apps. In our case study, we have created two malware (in total 531 LoC): a miniapp-based phising malware, and a native malware that modifies `WeChatApp.data` to allow the execution of the phishing miniapp.

5 ROOTFREE FROM LAYER UP

5.1 Attack Surface Analysis

Insights. Similar to how we attack the super apps from layer below, we have to also first understand the attack surfaces from a miniapp's perspective before we could launch any attacks from the layer up. Since the only interface for layer up attack is the APIs, we have to inspect them to understand how an attacker would use them to develop malicious miniapps, to escape the sandbox and attack the super apps or the entire OS for example.

However, there are still non-trivial challenges when analyzing these APIs, particularly because of the large number of APIs we have to analyze. More specifically, there are 985 APIs [29] in total, meaning we have to at least test 985 times independently for each API. Given that there are three different platforms offered by WeChat, theoretically we have to test 2,955 times in order to observe the discrependices among them. Therefore, we have to narrow down the scope of the APIs for our analysis. From security and privacy perspective, we feel APIs related to system resources (e.g., microphones, cameras) and the user's sensitive data (e.g., locations, phone number, contact information) should be of our focus.

Differential Analysis. As such, we have developed a testing miniapp using the resource access APIs to observe the discrepancies among the three different platforms. During our analysis, we mainly focused on two categories of resources that can be accessed by the miniapps: (i) the resources (e.g., audio, video, user's contact) that the miniapps have to request for authorizations from the user; Intuitively, those resources are not freely accessible by the miniapps, and therefore, they may be highly sensitive and cause security and privacy impacts if a miniapp can exploit them; (ii) the resources (e.g., the file system) that miniapps can access without the users' awareness but have potential risks if being freely accessed. While such resources do not need the user's authorization, the super apps have enforced permission protections to prevent them from being freely accessed. Also, note that there are indeed some other resources such as battery information and screen brightness that can be accessed without the user's authorization and the super app's protection, but they often lead to side channel attacks, and we therefore leave their exploration to future works. In particular, a miniapp can access the file system through API `wx.getFileSystemManager`, by which the miniapps can read, write and delete the files provided by the super apps. If no strong file system isolation is provided, a malicious miniapp may steal the user' sensitive information, and even overwrite them for malicious purposes (e.g., the miniapps can be a ransomware, which can encrypt the entire disk for profit). Given that Windows has weaker protection (where there is no strong file system isolation), the super apps have to implement the file system isolation on their own. We noticed that on Windows machine, they create a sandboxed directory for each of the miniapps to isolate the miniapps from accessing files outside.

Attack Targets. Through our differential analysis, it becomes apparent that some of the resource access APIs can be exploited, particularly in Windows platform. Meanwhile, the file system sandbox implementation in Windows may be exploitable, due to the lack of strong isolation from the operating systems (as well as the privilege discrepancies) compared to Android and iOS. For instance, we do notice that the file system isolation in Windows is achieved through a software abstraction `wxfile`, where the root directory of a miniapp is the sandbox directory, and all the operations (e.g., write and read) are supposed to occur only inside this folder. However, it is not clearly whether there is any flawed implementation of this abstraction.

5.2 Concrete Attacks

(A3) Sandbox Escaping due to Flawed and also Weak File System Isolation. To protect files from being accessed by malicious miniapps, WeChat uses a sandbox mechanism with security checks in `FileSystemManager` to ensure the file access can only occur inside the the sandbox. We therefore traced the functions of class `FileSystemManager`, and inspected the implementations of file operations. During this analysis, we identified function `getNativePathByJsPath` (whose address is `0xABE042`) located in `applet::AppletUtils`, to be of our particular interests.

Specifically, as shown in Figure 4, there exist some interesting input validation checks. At a high level, the super app needs to ensure there is no escaping of the sandbox to access other directories, and one way for attacker to do so is to use `../` or `..\`, and WeChat

```
1   /* code omitted */
2   sub_427C36(a2);
3   if (sub_BD3510(v36, v42, "wxfile://", v50, 1))
4   {
5     std::string::string(a2);
6     v64 = std::char_traits<char>::length(byte_DB44A0);
7     v7 = std::char_traits<char>::length("wxfile://");
8     _cfltcvt(v73, 0, "wxfile://", v7, byte_DB44A0, v64);
9     std::string::string((void *)byte_DB44A0);
10    std::string::string("../");
11    // replace "../" to ""
12    sub_4765D5((int)v73, v22, v24, v26, v28, v30, v32,
13        v34, v37, v43, v48, v53);
14    std::string::string((void *)byte_DB44A0);
15    std::string::string("..\\");
16    // replace "..\\" to ""
17    sub_4765D5((int)v73, v23, v25, v27, v29, v31, v33,
18        v35, v38, v44, v49, v54);
19    sub_45FB4C();
20    /* code omitted */
21  }
22  /* code omitted */
```

Figure 4: The decompiled code for converting 'wxfile' scheme to native path in getNativePathByJsPath implementation

indeed performs this check and will replace it to empty string (as shown in line 10 – 13, and 15 – 18). Unfortunately, we found this check is flawed, given that the security check only checks the file path before the replacement of "../", but does not check the file path after the replacement. As such, there are multiple ways to go around such checks, as long as the attacker can make sure there are "../" even after the replacement. For example, an attacker can craft a file path "../../../../../../../password.txt". After replacing "../" with an empty string, attacker can still have a valid path to "../../../../password.txt", thereby completely escaping the sandbox to access other files. Meanwhile, Windows does not impose any additional access control if the users are executing the super app with administrator privilege, which is different compared to Android whose SE-Linux can stop further any sandbox escaping with this flawed implementation.

Attack Case Studies. Given the flawed security check, a malicious miniapp can access the files outside of its sandbox. Again, if such an attack is launched on Android or iOS, the impact is relatively small due to its stronger isolation. However, when this miniapp is executed on Windows, the attacker can modify arbitrary files or even achieve remote code execution to completely take over the machine. In the following, we demonstrate concretely how to launch such attacks.

For proof of concept, we have developed an attack as shown in Figure 5. First, the attacker sends the victim user a URL or an instant message pointing to a malicious miniapp. Once the victim clicks it, the malicious miniapp will be loaded by WeChat (Step ❶). Next, the malicious miniapp obtains wx.getFileSystemManager (Step ❷), and then directly initiates file write request to write malicious code onto the disk (Step ❸). Since the security check is flawed (Step ❹), the malicious miniapp can write the malicious code at anywhere on the victim machine (Step ❺). For example, the malicious miniapp can place the malicious payload to C:\ProgramData\Microsoft\Windows\Start Menu\Programs\StartUp without any warning if the super app is executed with administrator privilege, and the malicious code will be executed when the user reboots the computer. However, there is also alternative to execute the malicious code instantly. For instance, we noticed that there is an API named

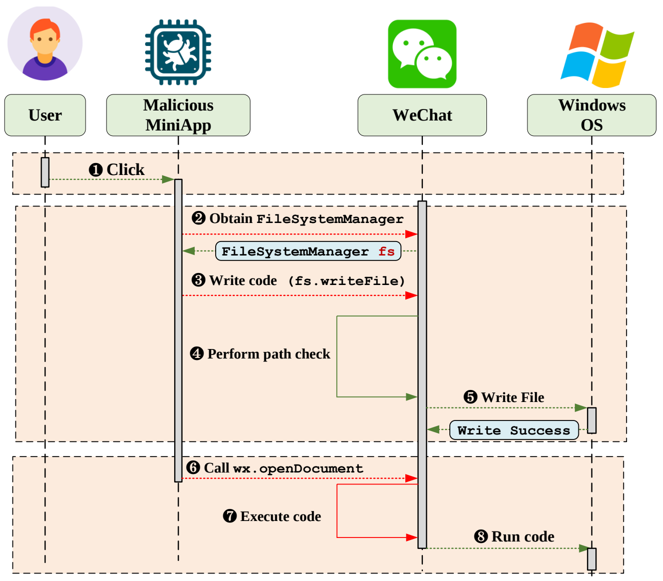

Figure 5: Remote code execution attacks in Windows

wx.openDocument, which will open non-executable files (e.g., PDF, DOCX). When a miniapp calls wx.openDocument (Step ❻), an executable file WeChatXFile located at the WeChat's installation folder, will be loaded to execute the document opening logic (Step ❼). As such, the malicious miniapp can overwrite WeChatXFile, and then calls wx.openDocument if WeChat XFile is not loaded to the memory yet. As a result, the malicious code can be executed instantly (Step ❽), and there is no warning message even when the super app is executed without administrator privileges.

With the capabilities of breaking the file system isolation, the malicious miniapp running atop Windows can also launch arbitrary data manipulation attack. For instances, the attacker can traverse the folders to steal user's sensitive files or overwrite important system configuration files (if the super app is executed under administrator privilege) such as the system configuration file hosts, which is used to map domain names to IP addresses (e.g., www.google.com to 142.250.190.142). The attacker can change hosts by overwriting a correct IP address with phising websites under attacker's control. Attackers can also use these attacks to distribute malware in a large scale, given that the user base of WeChat has reached 1.2 billion [22]. Please note that in all of our attacks, only the RCE attack requires such permissions. Even if these permissions are not granted, the miniapp can still inspect sensitive user information and delete important files. However, it will not be allowed to install malicious software in such a case.

6 DISCUSSION

6.1 Generality of Our Methodology

While the paper primarily focuses on WeChat, it's important to note that the workflow it introduces is not limited to this specific application. The methodology and steps outlined can be applied to other super apps and miniapps, making it a versatile and adaptable framework for evaluating security risks in similar platforms. Despite the current prevalence of desktop miniapps being primarily associated with WeChat, we believe our methodology can be extended to analyze other platforms should they incorporate support for miniapps in the near future. For example, the comprehensive examination of resources, including code and data, is a step that transcends specific applications and can be seamlessly applied to assess the security posture of any super app or miniapp. Additionally, the evaluation of security measures like encryption and file isolation, which are common in software development, remains relevant to any comparable platform. Furthermore, the process of identifying attack surfaces, while subject to some variation in specifics, upholds the universal practice of seeking potential vulnerabilities within an application, regardless of its nature or origin.

6.2 Root Causes

The issue of our attacks stems from the underlying discrepancies that exist across different platforms for super apps. It becomes imperative to delve into the reasons behind these discrepancies in order to gain a comprehensive understanding of the matter at hand.

It becomes evident that the variance in threat models plays a pivotal role in this context. Mobile operating systems are designed to accommodate the execution of multiple applications, with a crucial emphasis on isolating these applications from one another. This isolation serves as a safeguard against potential breaches, ensuring that untrusted apps are barred from accessing the data of other apps. In stark contrast, desktop OSs adhere to a different threat model, embracing the multi-user paradigm while affording the owner of the computer the root privileges. Consequently, the migration of mobile apps to desktop environments precipitates a shift in the threat model, creating a scenario where these apps are exposed to disparate security challenges. It is worth noting that web browsers, which evolved during the desktop era, assume the role of super apps, yet remain unaffected by the issues faced by contemporary mobile super apps. Consider attack (A2) where decryption breach manipulates databases and deploys malicious miniapps. In the web realm, web apps operate with fewer restrictions, increasing the risk. Attack (A1), compromising miniapp code confidentiality and integrity, holds less incentive in the web environment due to limited secrets stored in cached files.

Moving forward, it is crucial to underscore the disparities in built-in security mechanisms across these platforms. While some attacks can be attributed to implementation flaws, such as in the case of A3, where developers of super apps failed in executing proper security checks, the repercussions of these flaws manifest in varying degrees across different platforms. Notably, on Android and iOS, the impact of attack (A3) is curtailed due to the robust system-level isolation. This isolation obstructs the ability of miniapps to access files belonging to other applications, thereby mitigating the consequences of breaches.

6.3 Lessons Learned

We can learn valuable lessons from the research, and use this knowledge to further educate the community and contribute to the field of super apps and mobile/desktop security. First, the research highlights the importance of security in the context of super apps and their expansion (e.g., miniapps) across different platforms. We should emphasize the significance of security awareness and the need for robust security measures, especially when integrating third-party applications. Second, the discrepancies in security models and implementations across mobile and desktop platforms underscore the need for standardized security practices. We should explore these differences further, conduct comparative studies, and propose solutions for bridging the gaps in security between platforms. We can explore additional attack vectors, develop tools or methodologies for detecting and mitigating these attacks, and provide guidelines for super app developers to enhance their security against such threats. Third, one proactive approach that WeChat could consider involves the addition of an extra layer within its operating environment. An alternative course of action could entail a thorough reevaluation of specific features within their desktop super app, such as WeChat itself. This may necessitate the temporary suspension of support for particular functionalities, such as miniapp execution, until robust security mechanisms can be established. Although such decisions may present challenges, they underscore a resolute commitment to prioritizing user security in the face of evolving threats.

6.4 Ethics and Responsible Disclosure

We have followed the community's best practice in our study. In particular, we only launched the attacks in the controlled environment using our own accounts and machines (e.g., Windows, Android, and iOS). While we have programmed a set of malware to demonstrate the attacks (e.g., A1, A3), and those malware can in theory be widely distributed using the methods introduced in A2, we keep all of them private (without distributing them to the public at all). In addition, although our A1 can break the code confidentiality and integrity of miniapps, we did not release our tool, and will never to do so, in order not to cause any harms to any users, miniapp developers, and platform providers (i.e., Tencent in this case).

More importantly, we have reported all our findings to Tencent using 3 independent vulnerability disclosure reports, each of which described one specific type of attacks (i.e., A1, A2, A3). Without any surprise, all of them have been ranked as the high severity vulnerabilities, and Tencent awarded us with bug bounties. Meanwhile, Tencent's WeChat security engineers have been very actively working with us in the past year (e.g., we have met online multiple times to discuss the possible fixes).

6.5 Countermeasures

In our comprehensive analysis of vulnerabilities affecting super apps operating within the Windows platform, we have unveiled three major concrete attack vectors that could potentially compromise their security. These attacks not only raise concerns but also emphasize the importance of implementing robust countermeasures to safeguard these super apps and the sensitive data they handle. In the subsequent sections, we delve into the potential strategies and countermeasures that can be employed to mitigate the risks posed by these attacks.

Patching the Implementation Vulnerabilities. A critical step towards enhancing the security of super apps involves addressing the implementation vulnerabilities that have been exploited in our attacks, specifically the sandbox escaping checks targeted in attack A3. We recommend that the development teams responsible for these super apps, like WeChat, promptly patch these flaws to prevent potential exploits. Notably, it is encouraging to note that Tencent has already taken action in this regard by releasing patches to address the vulnerabilities exploited in attack A3. We have confirmed that A3 no longer works. This responsive approach showcases their commitment to securing their platform and user data.

Thwarting the Binary Code Analysis. The vulnerabilities leveraged in attacks A1 and A2 underscore the significance of reevaluating the techniques employed for code analysis and reverse engineering. These attacks capitalized on methods such as API hooking and dynamic data dependence analysis to uncover critical cryptographic operations and associated keys. To deter such analysis-driven attacks in the short term, incorporating obfuscation techniques can prove to be effective. Currently, WeChat has taken proactive measures by patching attack A1 through the utilization of VMProtect, a tool known for safeguarding code through execution within a specialized virtual machine. Furthermore, their ongoing efforts to address attack A2 using similar obfuscation methods exemplify a commitment to fortifying their defense mechanisms. While it's important to acknowledge that obfuscation may not offer absolute protection against determined adversaries, it remains an industry-standard practice for deterring binary analysis attempts.

Using Stronger Security Mechanisms. A distinctive challenge faced by super apps on the Windows platform, as opposed to their counterparts on Android and iOS, is the absence of robust security mechanisms provided by the operating system and hardware. Notably, super apps running on Windows lack the comprehensive security features like secure enclaves available on other platforms. This leaves them susceptible to potent attacks, including those orchestrated by malicious operating systems. One prospective avenue for enhancing the security of super apps is the incorporation of Trusted Execution Environments within the desktop environment. Technologies like Intel Software Guard Extensions (SGX) have demonstrated their ability to bolster security by creating isolated enclaves for executing sensitive operations. However, the recent discontinuation of SGX support for desktop systems by Intel, as reported in [31], poses a challenge to this approach's feasibility at present.

In light of the absence of TEE support, Tencent and similar developers must strategize on alternative measures. A potential course of action could involve reconsidering the inclusion of certain features in their desktop super app, like WeChat. This might entail temporarily discontinuing support for specific functionalities, such as miniapp execution, until viable security mechanisms can be established.

7 RELATED WORK

The first study of miniapp security can be dated back to the work by Lu et al. [25] for the investigation of the security management of resources within app-in-app systems. Then in 2021, we [40] introduced MiniCrawler, a tool designed to download WeChat miniapps. We conducted an extensive assessment encompassing various aspects (e.g., resource usage, API utilization). In 2022, Wang et al. [35] put forward WeDetector, a solution aimed at pinpointing three prevalent bug patterns in WeChat mini-programs: inappropriate reliance on platform-specific APIs, inadequate adaptation of layouts, and vague handling of arguments in callbacks. Zhang et al. [39] examined the susceptibility of app-in-app ecosystems to identity confusion vulnerabilities. Their scrutiny extended to 47 super apps, revealing that each one was susceptible to identity confusion attacks. We introduced a novel type of attack, CMRF [38], and developed CMRFScanner to identify vulnerabilities of this nature. Our findings indicated that an extensive number of WeChat and Baidu miniapps were at risk of CMRF attacks.

In 2023, Zhang et al. [42] revealed the Trusted Domain Compromise Attack for phishing. Cai et al. [13] discussed user activity and advocated user-centric account security in super apps. Wang et al. [36] compared traditional browser and super-app threat models. Zhao et al. [43] delve into signature verification in enhancing transaction and data security in the miniapp ecosystem. Lately, we turned our focus towards the protocols governing sensitive resource access in miniapps. Our analysis led to the discovery of vulnerabilities related to master key leakage [41]. We also introduced TaintMini [33] a solution designed to track the flow of sensitive data in mini-programs using a comprehensive data flow graph-based approach. Meanwhile, we also discovered hidden APIs provided by super apps [14], showcasing their untapped potential for exploitation.

Most recently, our APIDiff [34] revealed variations in API execution across diverse platforms within the WeChat framework. We accomplished this by automatically generating API test cases, which in turn facilitated the recognition of three distinct categories of discrepancies: API presence, API permissions, and API outcomes. This work also explores the discrepancies between platforms, but with a key focus on user privileges and process isolation by challenging different threat models and implementations. More specifically, we introduce two categories of RootFree attacks, namely, those operating in the layer below and the layer above, which are based on vulnerabilities we have identified through binary analysis. We also demonstrate the potential for these attacks to have severe security implications for both the vast user base of super apps and the developers of miniapps.

8 CONCLUSION

In this paper, we have shown that there are cross platform discrepancies for super apps and these include different threat model, user privilege, and isolation mechanism. We have demonstrated that these discrepancies can be exploited to launch RootFree attacks from layer below and layer up against the super apps including their miniapps. While our study mainly focuses on WeChat, we hope our findings can help many other super app vendors to avoid the same mistakes as in WeChat, when integrating the miniapp model and executing it on different platforms such as desktops.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their invaluable feedback. This research was supported in part by NSF award 2330264. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of NSF.

REFERENCES

[1] "How facebook, apple, google copied china's wechat messaging app," https://exbulletin.com/tech/274959/.

[2] "Intel's dropping of sgx prevents ultra hd blu-ray playback on pcs - ghacks tech news," https://www.ghacks.net/2022/01/14/intels-dropping-of-sgx-prevents-ultra-hd-blu-ray-playback-on-pcs/.

[3] "The race to create the world's next super-app - bbc news," https://www.bbc.com/news/business-55929418.

[4] "Secure enclave - apple support," https://support.apple.com/guide/security/secure-enclave-sec59b0b31ff/web.

[5] "Security-enhanced linux in android," https://source.android.com/security/selinux.

[6] "What is a super app and why haven't they gone global?" https://www.cnbc.com/video/2021/07/16/what-is-a-super-app-and-why-havent-they-gone-global.html.

[7] "Windows sandbox - windows security | microsoft docs," https://docs.microsoft.com/en-us/windows/security/threat-protection/windows-sandbox/windows-sandbox-overview.

[8] A. G. Adami, L. Burget, S. Dupont, H. Garudadri, F. Grezl, H. Hermansky, P. Jain, S. S. Kajarekar, N. Morgan, and S. Sivadas, "Qualcomm-icsi-ogi features for asr." in *INTERSPEECH*, 2002.

[9] Alan Bavosa, "No-code Jailbreak & Root Prevention in iOS & Android apps ," 2020, Available athttps://www.appdome.com/blog/jailbreak-detection-root-detection/.

[10] J. Andrus, N. AlDuaij, and J. Nieh, "Binary compatible graphics support in android for running ios apps," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, 2017, pp. 55–67.

[11] G. Ayoade, V. Karande, L. Khan, and K. Hamlen, "Decentralized iot data management using blockchain and trusted execution environment," in *2018 IEEE International Conference on Information Reuse and Integration (IRI)*. IEEE, 2018, pp. 15–22.

[12] P. Bulens, F.-X. Standaert, J.-J. Quisquater, P. Pellegrin, and G. Rouvroy, "Implementation of the aes-128 on virtex-5 fpgas," in *International Conference on Cryptology in Africa*. Springer, 2008, pp. 16–26.

[13] Y. Cai, Z. Zhang, D. Li, Y. Guo, and X. Chen, "Shared account problem in super apps," in *Proceedings of the 2023 ACM Workshop on Secure and Trustworthy Superapps*, 2023, pp. 47–50.

[14] W. Chao, Y. Zhang, and Z. Lin, "Uncovering and exploiting hidden apis in mobile super apps," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023.

[15] S. Checkoway and H. Shacham, "Iago attacks: Why the system call api is a bad untrusted rpc interface," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 253–264, 2013.

[16] DAN GOODIN, "Zeroday exploit prices are higher than ever, especially for iOS and messaging apps ," 2019, Available athttps://www.reddit.com/r/apple/comments/adoka8/zeroday_exploit_prices_are_higher_than_ever/.

[17] B. Dean, "Tiktok user statistics (2022)," https://backlinko.com/tiktok-users.

[18] C. Eagle, *The IDA pro book*. no starch press, 2011.

[19] Frida, "Firda–dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers." https://frida.re/docs/android/, 2012.

[20] T. GRAZIANI, "What are wechat mini-programs? a simple introduction - walkthechat," https://walkthechat.com/wechat-mini-programs-simple-introduction/.

[21] M. Inc., "Mediatek inc." https://www.mediatek.com/.

[22] T. Inc, "55+ wechat statistics - 2022 update," https://99firms.com/blog/wechat-statistics/#gref.

[23] S. Josefsson, "Pkcs# 5: Password-based key derivation function 2 (pbkdf2) test vectors," *Internet Engineering Task Force (IETF), RFC Editor, RFC*, vol. 6070, 2011.

[24] G. Lee and C. Gray, "L4/darwin: Evolving unix," in *Conference for Unix, Linux and Open Source Professionals, Melbourne, Vic, Australia*, 2006.

[25] H. Lu, L. Xing, Y. Xiao, Y. Zhang, X. Liao, X. Wang, and X. 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*, 2020, pp. 569–585.

[26] J. Rutkowska, "Subverting vistatm kernel for fun and profit," *Black Hat Briefings*, 2006.

[27] B. Schneier, *Applied cryptography: protocols, algorithms, and source code in C*. john wiley & sons, 2007.

[28] statista, "Number of mobile monthly active users across alibaba's online shopping properties from 3rd quarter 2017 to 3rd quarter 2020," https://www.statista.com/statistics/663464/alibaba-cumulative-active-mobile-users-taobao-tmall/, 2020.

[29] Tencent, "WeChat Chinese Documentation," https://developers.weixin.qq.com/miniprogram/en/dev/api/, 06 2020.

[30] ——, "WeChat English Documentation," https://developers.weixin.qq.com/miniprogram/en/dev/api/, 06 2020.

[31] B. Toulas, "New intel chips won't play blu-ray disks due to sgx deprecation," https://www.bleepingcomputer.com/news/security/new-intel-chips-wont-play-blu-ray-disks-due-to-sgx-deprecation/, 2022.

[32] A. Đuranec, S. Gruičić, and M. Žagar, "Forensic analysis of windows 10 sandbox," in *2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO)*. IEEE, 2020, pp. 1224–1229.

[33] 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 *ICSE*.

[34] C. Wang, Y. Zhang, and Z. Lin, "One size does not fit all: Uncovering and exploiting cross platform discrepant apis in wechat," in *31st USENIX Security Symposium (USENIX Security 23)*, 2023.

[35] T. Wang, Q. Xu, X. Chang, W. Dou, J. Zhu, J. Xie, Y. Deng, J. Yang, J. Yang, J. Wei *et al.*, "Characterizing and detecting bugs in wechat mini-programs," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 363–375.

[36] Y. Wang, Y. Yao, S. Shi, W. Chen, and L. Huang, "Towards a better super-app architecture from a browser security perspective," in *Proceedings of the 2023 ACM Workshop on Secure and Trustworthy Superapps*, 2023, pp. 23–28.

[37] M. Weiser, "Program slicing," *IEEE Transactions on software engineering*, no. 4, pp. 352–357, 1984.

[38] Y. Yang, Y. Zhang, and Z. Lin, "Cross miniapp request forgery: Root causes, attacks, and vulnerability detection," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 3079–3092.

[39] L. Zhang, Z. Zhang, A. Liu, Y. Cao, X. Zhang, Y. Chen, Y. Zhang, G. Yang, and M. Yang, "Identity confusion in {WebView-based} mobile app-in-app ecosystems," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 1597–1613.

[40] Y. Zhang, B. Turkistani, A. Y. Yang, C. Zuo, and Z. Lin, "A measurement study of wechat mini-apps," in *SIGMETRICS '21: ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems, Virtual Event, China, June 14-18, 2021*, L. Huang, A. Gandhi, N. Kiyavash, and J. Wang, Eds. ACM, 2021, pp. 19–20. [Online]. Available: https://doi.org/10.1145/3410220.3460106

[41] Y. Zhang, Y. Yang, and Z. Lin, "Don't leak your keys: Understanding, measuring, and exploiting the appsecret leaks in mini-programs." in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023.

[42] Z. Zhang, Z. Zhang, K. Lian, G. Yang, L. Zhang, Y. Zhang, and M. Yang, "Trusted-domain compromise attack in app-in-app ecosystems," in *Proceedings of the 2023 ACM Workshop on Secure and Trustworthy Superapps*, 2023, pp. 51–57.

[43] Y. Zhao, Y. Zhang, and H. Wang, "Potential risks arising from the absence of signature verification in miniapp plugins," in *Proceedings of the 2023 ACM Workshop on Secure and Trustworthy Superapps*, 2023, pp. 59–64.