POSTER: PLUGINPERMCHECK: Preventing Permission Escalation in App Virtualization

Shou-Ching Hsiao* and Hsu-Chun Hsiao*
*National Taiwan University, Taipei, Taiwan
d10922007@ntu.edu.tw, hchsiao@csie.ntu.edu.tw

I. INTRODUCTION

App virtualization provides the host app with the ability to load ordinary apps inside its process. These loaded apps, called plugin apps, execute with the identity of the host app, which is transparent to Android OS. Popular host apps, like Dual Space and Parallel Space, have more than 100 million downloads on Google Play. The most prevalent application is supporting mobile users to run multiple instances of the same app simultaneously for logging in to different accounts. However, while such a new paradigm brings convenience to users, it also leads to additional security issues. To support permission requirements of plugin apps, the host app over-claims permissions in advance. Since plugin apps and the host app share the same user ID (UID), plugin apps can utilize permissions of the host app even without any permission declaration, causing the opportunity of *permission escalation* attack. A malicious plugin app can evade app vetting or malware detection that uses declared permissions as an important indicator. Worse still, an attacker can be more stealthy by deferring the malicious behaviors until run-time permission has been granted to the host app. As the current UID-based permission control is insufficient to check permissions in the context of app virtualization, we aim to provide a solution that can prevent permission escalation attacks while allowing normal execution of benign plugin apps.

Previous work proposes defense for benign plugin apps to prevent execution inside the host app [3, 1]. However, merely aborting the execution does not fulfill our purpose since users can no longer execute these plugin apps. Our research is the first to provide an actionable solution for users to prevent such attacks while still preserving normal functionalities. We present PLUGINPERMCHECK that checks the plugin's permissions based on the process ID (PID) and provide the following desired properties: (1) effectiveness of blocking undeclared permissions; (2) acceptable overhead; (3) no requirement for pre-configuring host apps and plugin apps; and (4) supporting a variety of apps and OS versions.

To achieve these properties, our system needs to overcome the following challenges. First, unlike UID, which is fixed unless an app is reinstalled, PID is susceptible to change. Our system updates the PID for each plugin app to accomplish effective permission checking. Second, the overhead of additional checking is inevitable. We need to minimize the latency by only applying our system to plugin apps executing in the host app. Third, to be freed from pre-configuration, our system should identify these targets at run-time and extract the information before any permission request. This also enhances the support of a wide range of plugin apps. Fourth, because API-permission mappings vary with different OS versions, we need to dynamically obtain the API-corresponding permissions from the currently executing system instead of relying on a hard-coded API-permission table statically.

II. BACKGROUND AND THREAT MODEL

A. App Virtualization

The host app acts as a proxy layer to intercept and wrap system calls from plugin apps and communicates with the Android OS on their behalf. By doing so, the host app and plugin apps share the same UID and thereby equal permission sets and resources. Since Android OS does not know the existence of plugin apps, the host app helps plugin apps complete launching, loading, and execution. While launching plugin apps, intent wrapping [4] is a common trick to pass the verification of Activity Manager Service (AMS), where the host app extracts the plugin component from one intent before saving the same component inside another intent's field.

Originally, app virtualization was designed for hot-patching or modular development purposes, where the same party develops the host app and plugin apps. However, an increasing trend has been observed to utilize the host app as a platform, encapsulating various plugin apps from different developers. This unintended usage of app virtualization has brought new security issues, especially when the host app lacks proper permission control for plugin apps.

B. Threat Model

We consider an attacker developing a malicious app that aims to conduct permission-required malicious behaviors inside a typical host app.

Permission Escalation. The attacker's goal is to abuse high-risk permissions without being detected. The attack will be easily detectable if the attacker declares such permissions in the app's manifest file or asks for user granting in run-time. Thus, to evade app vetting, the attacker can manipulate the feature of "shared UID," abusing the host app's permissions with null permission declaration. Also, to avoid raising user suspicion, the attacker can probe if the host app has already been granted permission through `checkSelfPermission`. Once the permission has been granted, the attacker can successfully escalate permission, over which users have neither information nor control.

We test 20 host apps collected from Google Play and find that all of them are vulnerable to such attacks, including dominant ones with more than 100 million downloads. Even worse, some of the host apps proactively request permissions exactly during the first launch so that plugin apps can stealthily escalate run-time permissions more easily.

III. System Design

The core idea of PluginPermCheck is to identify the source of permission requests through PID and determine if the permission should be blocked through PID's corresponding permission sets. Our system consists of three main modules. When launching plugin apps, the first and second modules detect app virtualization and obtain information on user-loaded plugin apps. During permission-required system calls, the last module enforces the permission policy.

A. Detecting App Virtualization

This module detects the host app and user-loaded plugin apps through intent wrapping and then records their package names and signing certificates. Since some host apps may have multiple packages, we use certificates to link them to the same host app. We monitor the two steps of intent wrapping by first logging the plugin component returned by `Intent.getComponent` followed by comparing it to the input of component saving APIs, like `setType`, `setData`, `putExtra`, and `putExtras`. Upon matching, we identify the current package as the host app and the package of the saved component as the plugin app. These detecting results are used in the next module.

B. Parsing Plugin Apps

This module identifies the plugin app instantiated in the host app process and then extracts the plugin's PID and the declared permission sets. Utilizing the package name and certificates from the first module, we inspect if the current instance executing `Instrumentation.newApplication` belongs to the host app and the return object belongs to the plugin app. To avoid unnecessary overhead, checking the host app package can filter out the same dual-instance app launching in the Android OS; and checking the plugin app package can filter out the non-plugin process in the host app. As the Application object of every process is instantiated at process creation, we can update the corresponding PID of each plugin app accurately every time this parsing module is triggered.

C. Checking Permission

To check permission for plugin apps, we monitor the permission requests at the system service, exactly the invoke of `AMS.checkComponentPermission` so that permission checking from apps and system service are both applied. Then, our system obtains PID and the requesting permission from method parameters and enforces the following permission policy—if the permission is not in the PID's corresponding declared permission set, the permission should be denied.

TABLE I
Results of Effectiveness

Plugin App	Removed Permissions	Dual App		Dual App†		Dual Space		Dual Space†	
		N	**R**	**N**	**R**	**N**	**R**	**N**	**R**
com.waze	LOCATION	✓	✓	✓	✗	✓	✓	✓	✗
com.twitter.android	CAMERA	✓	✓	✓	✗	✓	✓	✓	✗
	STORAGE	✓	✓	✓	✗	✓	✓	✓	✓
com.google.android.apps.nbu.files	STORAGE	✓	✓	✓	✗	✓	✓	✓	✗
com.simpler.dialer	PHONE	✓	✓	✓	✗	△	△	✓	✗
	CONTACTS	✓	✓	✓	✗	△	△	✓	✗
net.sourceforge.opencamera	CAMERA	✓	✓	✓	✗	✓	✓	✓	✗
com.hitrolab.audioeditor	MICROPHONE	✓	✓	✓	✗	✓	✓	✓	✗

†With PluginPermCheck applied. **N**: Normal. **R**: Repackaged.
✓: Normal functionality. ✗: Unable to use the functionality.
△: Other exceptions. Dual App: com.ninetyplus.dualapp.
Dual Space: com.ludashi.dualspace.

IV. Implementation and Evaluation

Implementation. For demonstration purpose, a prototype of PluginPermCheck tool is implemented via hooking using the *LSPosed* Framework [2]. This framework dynamically instruments Android OS so that users need not re-flash the mobile system. While our prototype tool can act as a quick fix for users, Android OS can also adopt our design to address permission escalation attacks in app virtualization.

Evaluation. We evaluate our prototype on an ASUS Zenfone M2 (X01AD) running Android 9.0.

1) Effectiveness: To simulate permission escalation attacks, we use repackaging techniques to generate the adversarial samples by removing permission declarations in the plugin app's manifest file. The results in Table I show that no false positive is generated by misidentifying a normal plugin app as a permission escalated case. On the other hand, we also show that undeclared permissions of repackaged apps can be intercepted by our system, except for one false-negative case of the storage permission. An initial guess is that the external storage permission is enforced by the kernel in Android 9.0. Other exceptions, such as abnormal Google Play services in host apps, are not caused by our system.

2) Overhead: We use the `top` command to gauge the computational overhead of installing, launching, and executing a plugin app. We add up the CPU utilization of these operations within a 60-second window. On average, our tool consumes 1.68% more CPU per second.

V. Conclusion and Future Work

In this work, we propose a PID-based plugin permission checking system to prevent permission escalation attacks in app virtualization. Future work includes (1) kernel-level permission enforcement and (2) a larger-scale evaluation.

References

[1] Deshun Dai et al. "Parallel Space Traveling: A Security Analysis of App-Level Virtualization in Android". In: *ACM SACMAT*. 2020.

[2] LSPosed. URL: https://github.com/LSPosed/LSPosed.

[3] Tongbo Luo et al. "Anti-plugin: Don't let your app play as an Android plugin". In: *Blackhat Asia* (2017).

[4] Luman Shi et al. "VAHunt: Warding Off New Repackaged Android Malware in App-Virtualization's Clothing". In: *ACM SIGSAC CCS*. 2020.