ANTI-PLUGIN: DON'T LET YOUR APP PLAY AS AN ANDROID PLUGIN

Tongbo Luo, Cong Zheng, Zhi Xu, Xin Ouyang
Email: {tluo,cozheng,zxu,xouyang}@paloaltonetworks.com
Palo Alto Networks Inc.

ABSTRACT

The Android plugin technology is an innovative application-level virtualization framework that allows a mobile application to dynamically load and launch another app without installing the app. This technology was originally developed for purposes of hot patching and reducing the released APK size. The primary application of this technology is to satisfy the growing demand for launching multiple instances of a same app on the same device, such as log in two Twitter accounts for the personal and business simultaneously. The most popular app powered by this technology, Parallel Space, has been installed 50 million times in Google Play.

However, as we know, it never takes malware authors long to catch on to new mobile trends. In the wild, by applying the plugin technology, a newly discovered Android malware "Dual-instance" dynamically loads and launches the original Twitter app's APK file within itself and also hijacks user's inputs (e.g. password) to launch the phishing attack. Besides, after we have comprehensively analyzed security risks of the Android plugin technology, we find that the data stored by the plugin app can be stolen by the malicious host app or other plugin apps. In our Wildfire product, we have captured 119, 898 samples using the Android plugin technology, among which 114, 630 samples are malicious or grey. Thus, the Android plugin technology is becoming a new security threat to normal Android apps.

Our proposal demystifies the Android plugin technology in depth, explains the underlying attack vector and investigates fundamental security problems. We propose a lightweight defense mechanism and release a library, named `Plugin-Killer`, which prevents an Android app from being launched by the host app using the Android plugin technology. Once a benign Android app embeds the library, the app can detect the potential threats from virtual environment and terminates itself when it is launched.

1. INTRODUCTION

The Android plugin technology is an innovative application-level virtualization/proxy, which was originally developed for purposes of hot patching, reducing the released APK size, and solving the 65535 methods limitation. Technically, the Android plugin technology is very different with the widely known dynamic code loading (e.g. loading a dex or jar file), since it can load/launch a whole app (e.g. an APK file) and the host app does not need to declare any specific interfaces or components for loaded apps. The primary application of the Android plugin technology is to launch multiple instances of any apps on the same device without installing apps. Based on our observations, two incentives of applying the Android plugin technology to the app development are: using multi-accounts in social apps and instantly launching apps in the app store app. Both application scenarios are derived by from user requirements. For example, one user has two Twitter accounts for the personal and business respectively, and doesn't want to switch accounts by login and logout repeatedly or simply use two smartphones. The most popular app powered by this technology, "Parallel Space" [2], has been installed 50 million times in Google Play. Figure 1 shows all of the popular mobile apps and libraries powered by plugin technology.

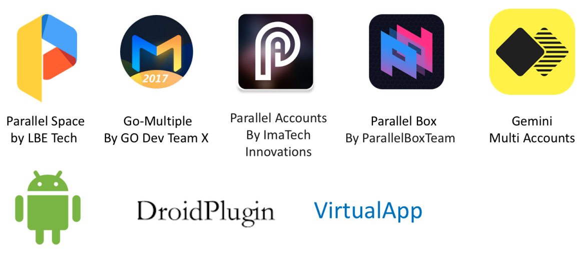


Figure 1: List of All Popular Virtual Environment.

However, as we know, it never takes malware authors long to catch on to new trends, so cybercriminals have recently taken it upon themselves to create malicious apps based on the Android plugin technology. A newly discovered Android malware "Dual-instance malware", reported by AVAST Software [14], adopted the plugin technology. In this malware, the malware writer developed a fake Twitter app (host app), which can dynamically load and launch the real Twitter app from the Twitter's APK file (plugin app) without installing Twitter App in devices. Thus, by nature, it is a good way to launch the phishing attack as users interact with the real Twitter App indeed. The plugin technology allows the malicious host app to fully control and hook the process of plugin apps, so the host app can steal the Twitter account credentials through hijacking related functions of the EditText in the Twitter login window.

The Android plugin technology is a double-edged sword. Even though it brings users and developers more convenience and less development and maintenance cost, it also have many security issues and concerns. The discovered dual-instance malware is just a drop in the ocean compared to the millions of suspicious apps utilized this technology. It is confident to predict that plugin technology would be used by attackers as the new attack vector in the future. This new attack vector completely bypass all of the existing malware detection systems since the attacker did not compromise the integrity of the victim app's APK file.

The most serious security concern is that the trust computing environment in the Android system has changed because of the plugin technology. Previously, the trusted computing base relies on the assumption of secure Android system, which means that the Android system is free of vulnerabilities. But, even if this assumption still holds, the trusted computing environment cannot be guaranteed as the Android app may run in the Android plugin environment instead of the real Android system. Once an app is loaded and launched in the plugin environment, it is completely controlled by the host app. The potential security risks of legitimate apps running in the the plugin environment include: 1) All data stored in the file system by the app can be stolen by the host app or other apps running as plugin instances. 2) User inputs, such as login credentials, can be stolen by the host app. Therefore, legitimate apps are facing a new security threat from the Android plugin technology.

In this paper, we demystify the Android plugin technology in depth, explain the underlying attack vector and investigate fundamental security problems. We propose a lightweight defense mechanism and release a library, named "PluginKiller", which prevents an Android app from being launched by the host app using the Android plugin technology. Once a normal Android app embeds the library, the app can detect the Android plugin environment and terminates itself when it is launched. [1]

2. PLUGIN TECHNOLOGY DEMYSTIFY

There are many ways to implement plugin technology (e.g. DroidPlugin, VirtualApp and DynamicAPK), but all of them share the similar design. In this paper, we will use DroidPlugin as the example to explain. Figure 2 depicts the high-level picture of how DroidPlugin works. It involves three major parts: the Android framework, the host app with DroidPlugin SDK embedded, and the plugin as each individual APK file. The essential component in the DroidPlugin library is called `Proxy Hook`. It locates between plugin and Android framework, and intercepts invocations of the Android APIs from plugin app. The intercepted invocation will be modified by the DroidPlugin, such as changing the passed parameters, before sending to the Android framework, and this is magic of DroidPlugin to launch an APK file without installation.

2.1 Virtual Environment

The essential mechanism of plugin technology to launch multiple instances of an app is to create a virtual environment on the top of the Android framework. This virtual environment is transparent to the Android framework so that plugin can bypass system's restriction. The magic of DroidPlugin is to leverage the *Proxy Hook* component to intercept certain invocations of the Android APIs from the plugin app, and modify the parameters of them.

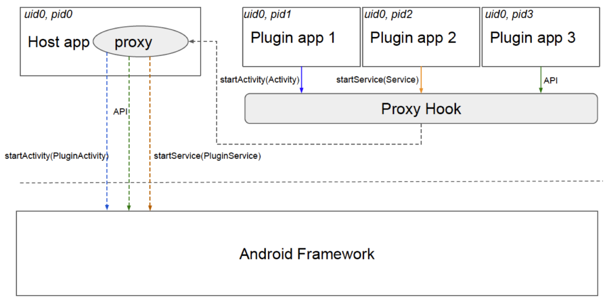


Figure 2: DroidPlugin Overview

Usually, hooking system [6, 19, 23] is a standard way to play this man-in-the-middle game. When we talk about the design of a hooking system, we actually needs to answer the following two questions: 'How to Hook API'? and 'What API to Hook'? The first question is easy to answer because hooking an API in Java is pretty standard. Java provides the "*Dynamic Proxy API*" for creating dynamic proxy of a class or instance using proxy design pattern. Some of the APIs is defined inside Android framework, we also need to use reflection to hook them. The second question is more complicated and involving the knowledge of how Android framework works. Basically, DroidPlugin hooked the APIs to perform the following tasks:

- Load and launch plugin (APK) without installation.
- Manage the lifecycle of app components.
- Inter-plugin communication.
- Plugin management (download, update).

This technology sounds similar to the DCL [7, 9, 12, 13], `Dynamic Code Loading`, a technique allows an app to load and execute code that is not part of its initial static code base at runtime. Both of them are used to run additional piece of code that may not necessarily be presented in the app package at installation time. DCL can only load a small piece of code that tightly depended on the base app's context, but Plugin technology is more advanced since it can launch a whole APK file, containing the code to perform more complicated functionalities and more interactions with the system.

Hook ClassLoader - Launch plugin without installation. In Android system, APK or dex files are loaded by ClassLoader. The loaded files and classes are stored in a list defined in the ClassLoader object called `DexElement`. Whenever the ClassLoader needs to load a class, it will scan through this list to match the given classname. If failed to locate in the current ClassLoader, it may trace back to the parent ClassLoader iteratively. There are several types of ClassLoader in Android: BootClassLoader is used to load the system class; PathClassLoader is used to load app class. But all of them are derived from the base class `BaseDexClassLoader`.

[1]https://github.com/irobert-tluo/AntiPluginLib

Originally, system goes to a specific path, usually is '*data/app*' folder, to find the installed app's APK file and related resources. At the launching step, only the APK file of the host app is parsed and saved in DexElement list. Since the plugin APK file is not under that specific path, it cannot be automatically loaded by the system. Android system relies on this ClassLoader object in **`Activity Thread`** to load and launch new activity as the following code shows [1].

```
java.lang.ClassLoader cl =
    r.packageInfo.getClassLoader();
activity = mInstrumentation.newActivity(cl,
    component.getClassName(), r.intent);
StrictMode.incrementExpectedActivityCount(activity.getClass());
r.intent.setExtrasClassLoader(cl);
```

Therefore, if system attempts to use current ClassLoader to load a class defined in plugin file, nothing can be located and the plugin cannot be launched. DroidPlugin will hook this ClassLoader and insert a parsed plugin APK file to the **`DexElemement`** list of it. It can be done to invoke the function **`loadDex`** with the path to the plugin file as the parameter (Figure 3). Once this hooking step is done, the classes defined in the plugin are available to search and launch in the normal way. This trick is similar to the one used int hot patching in Android.

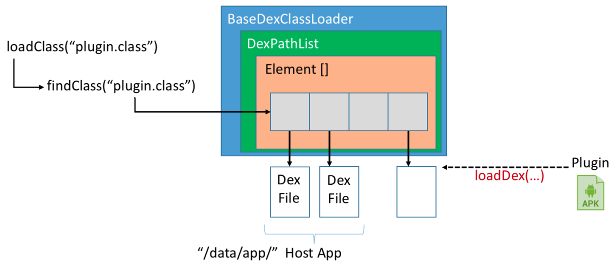


Figure 3: Hook ClassLoader to Launch Plugin without Installation

Share the UID. Android's package manager creates a unique user id (UID) and group (GID) when it installs an application and these are retained until the application is un-installed. For the plugin app, although it is been dynamically loaded and launched by the hooked classloader, it is not treated as a new app from system's perspective. Therefore, all plugin apps share the same UID with the host app, but different PIDs. Since all apps use the same UID, the Android permission model and the data isolation model in the Android system cannot be enforced to ensure the security of plugin apps.

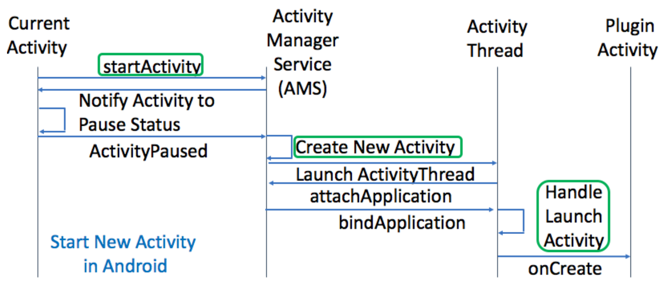


Figure 4: Flow to start new Activity in Android

Pre-defined Stub Components. Launching plugin APK file without installation is just the first step. DroidPlugin also needs to maintain the lifecycle of the app components used in plugin app. App components, such as Activity, Service, Broadcast Receiver, Content Receiver, are the basic blocks to build the essential functionalities of a mobile app. Unlike the UI components (e.g. button, view), app components is special because only the system can manage the lifecycle of them. It means the using these components involves lots of interaction with the Android framework. Due to the way the plugin is launched, DroidPlugin has to apply more tricks.

I will use starting a new activity in the plugin as an example to explain the tricks of DroidPlugin. Activities are served as the entry point for a user's interaction with an app, and are also central to how a user navigates within an app or between apps. It is commonly used in apps. Figure 4 illustrates the whole flow to start a new activity in Android platform. Android apps cannot create a new activity by itself, they need to use the service called **`Activity Manager Service`** or **`AMS`** provided by the system. AMS handles the management of the lifecycle for each activity, such as creating new activity or destroying closed activity.

The standard way to create an activity is to invoke the API called **`startActivity`** either explicitly or implicitly. Then, the AMS will perform some tasks, such as pausing the current activity, creating new activity, and maintaining the activities in a stack. After AMS finished those tasks, it will return the control back to the new activity and notify the **`Activity Thread`** to load and execute the new activity code, like the callback in the **`onCreate`** function of the activity class.

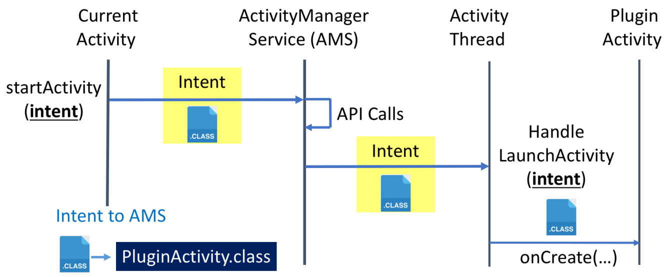


Figure 5: Intents to AMS when creating new Activity

For our paper, we only focused on the three steps: *startActivity*, *AMS*, and *handleLaunchActivity*. These functions are the places where the activity communicates with the AMS. As the simplified Figure 5 shows, function *startActivity* will send an intent to the AMS with the content as the class of new activity to be created. This is how the currently activity tells AMS which is the new activity. Once AMS set up the context of new activity, it will forward this intent to the *Activity Thread* in the app and invoke the callback function *handleLaunchActivity* to handle the intent. This function will extract the class of the new activity from the intent, load the class and begin to execute the code. That is how the system starts a new activity.

However, if we want to start an activity defined in the plugin app, it leads to a failure in the AMS. This is because the plugin activity is not defined in the host app's manifest file. Users may launch any plugin app, and the DroidPlugin

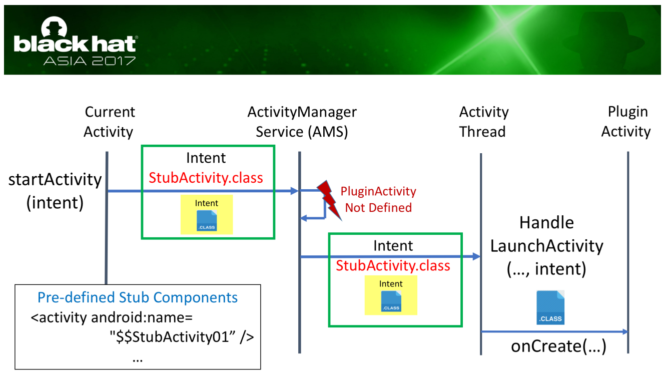
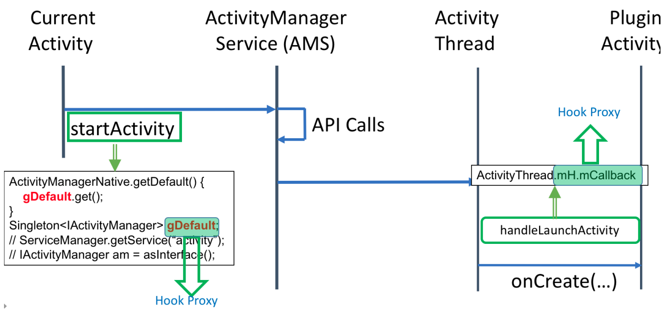


Figure 6: Replace the Intent content with Stub Activity Class

library cannot predict the name of them. Therefore, it is not possible to copy the definition of each plugin activity to the host app before installation. DroidPlugin solve the problem by pre-defining several stub components in host app's manifest file, such as stub activity with name like *stubActivity01*, which is same for all of the host apps. The host app working as a proxy has already pre-defined all components (e.g. service, activity, receiver and content provider) and permissions in its manifest. Generally, the host app pre-defines 10 stub components for each type of component separately, and all permissions in its AndroidManifest.xml. Thus, both the host app and plugin apps use these pre-defined stub components.

Hook AMS - App Component Without Definition. During the runtime, DroidPlugin will intercept the intent sending to the AMS from current activity, and wrap it into a new intent with the class of the stub activity (Figure 6). With the modified intent, DroidPlugin can fool the AMS to create the activity for the stub one, and this time won't have any failure. But it is not done since the AMS will also forward the intent to the new activity, activity thread will load the class of the stub activity based on the content in the wrapped new intent. Therefore, DroidPlugin also need to unwrap the replaced intent and feed the original one to the activity thread.



```
ActivityManagerNative.getDefault() {
    gDefault.get();
}
Singleton<IActivityManager> gDefault;
// ServiceManager.getService("activity");
// IActivityManager am = asInterface();
```

Figure 7: Intents to AMS when creating new Activity

To wrap the intent to the AMS, DroidPlugin hooked the function *startActivity* as we just mentioned. The core part of the code in this function is to get the *binder* of AMS in order to send intent to it. As the code in Figure 7 shows, activity does not need to get the binder of AMS from the system for each time. Since the communication to AMS is quite frequently, system saved a cope to the local object called *gDefault* as a cache. All of the framework will get the binder of AMS from this object. Once DroidPlugin hooked this cache object, it can serve the hooked binder instance to every invocation of *startActivity* API and intercept the intent before sending to the AMS. The same trick can be used to hook the function *handleLaunchActivity* to unwrap the forwarded intent.

If we look into the procedure for the system to create other type of app components, such as service, content provide, and broadcast receiver, we can find a similar flow since they also need to use the same mechanism to contact AMS. The only difference is the API invoked by current activity. For example, Figure 8 shows the flow to start a new service by invoking the API *startService*. DroidPlugin applied the same hooking point to intercept the intent, and replaced the class of the target service in the intent to the class of stub service.

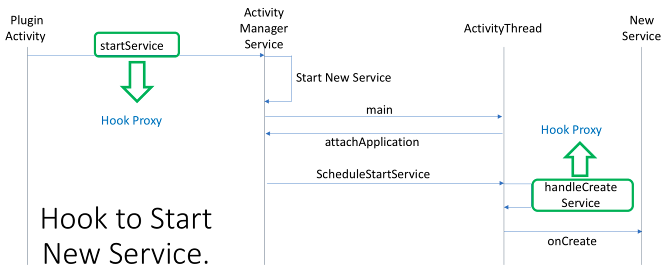


Figure 8: Hook to Start the Plugin Service

3. ABUSING OF PLUGIN TECHNOLOGY BY MALWARE

Plugin technology is powerful and useful, which makes the user experience of Android apps more convenience. However, we want to pose a fact that this new technology had been abused by malware. In this section, we will show some statistics and the real malware samples to prove it.

Statistics We searched the number of apps in our internal APK database, and we have found around 1 hundred and 20 thousands apps contain the DroidPlugin library/SDK or customized version of it. 5268 of them are benign but 114630 of them are malicious. (In our database, we marked them as malicious based on the suspicious behaviors discovered by our dynamic analysis module)

We measure the trend of new malware samples powered by DroidPlugin from July of 2015 to last month. As you can see from this figure, until Jan of 2016, the number of malicious samples is quite small; But there is a big jump at the first quarter of 2016 and followed by another hike in the second quarter. For the last several months of 2016, the number of new samples in increate by around 1000 to 2000.

Why Plugin Technology is abused? Based on the samples we have analyzed, we believe there are 3 reasons:

- Updating Malware without rooting the phone. With plugin technology, once the attackers can trick users to install one of its malware app to the device, they

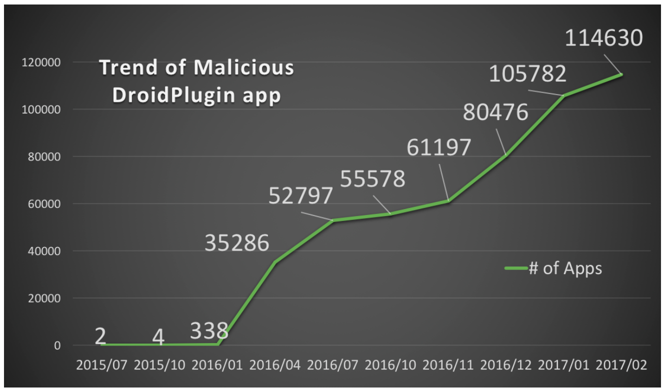


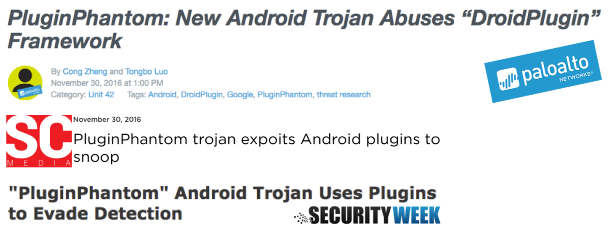
Figure 9: Trend of Abusing DroidPlugin SDK

can install new malware or update existing malware without rooting the phone. Because malware will put the core malicious code into the plugin as an individual APK file, to update it, malware only needs to download the new APK file from the remote server and replace the old one. By doing it, attackers significantly reduced the chance to be detected by users, since no user interaction is involving in the whole process.

- Evading Static Detection. Since the malware code is in the plugin which is dynamically fetched at runtime from remote server, and the host app doesn't contains the code to invoke any suspicious APIs, it can help attackers to evade some static detections. Without dynamically inspect the malware, it is hard to find evidence of malicious behavior only from the host app.
- Phishing without repackaging. Along with the development of detection technique [8], the chance of repackaging a popular app becomes smaller and smaller, which is used to be the most pervasive way for phishing. However, plugin technology can be used as an alternative way to launch phishing attack without repackaging the app. Attackers can load the authenticated APK as a plugin without modifying it, and put the malicious code in another plugin or in the host app to steal user's input or other credential information.

Malware Case Study: PluginPlathom. The first discovered malware family that utilized DroidPlugin library is named as `PluginPhathom` by us [5]. The blog we published at research center of Palo Alto Networks on November of 2016 shows our analysis on the samples of PluginPhathom, and it has been reprinted in mediums like SC Media [3], SecurityWeek [11] and BleepingComputer [4]. The behaviors of PluginPhathom is quite common, including taking pictures, capturing screenshots, recording audios, and intercepting and sending SMS messages. The uniqueness of it is that PluginPhathom modularized each malicious functionalities as a plugin, and it leverages the host app to dynamically launch each plugins at runtime to perform attacks.

PluginPhathom malware family was upgraded from its earlier version called `Trojan.Ihide` (first discovered at July of 2016 [14]), and attackers integrated the DroidPlugin library by completely refactoring the code to modularized each malicious functionalities to an individual APK file. It


PluginPhantom: New Android Trojan Abuses "DroidPlugin" Framework

By Cong Zheng and Tongbo Luo
November 30, 2016 at 1:00 PM
Category: Unit 42 Tags: Android, DroidPlugin, Google, PluginPhantom, threat research

November 30, 2016
PluginPhantom trojan expoits Android plugins to snoop

"PluginPhantom" Android Trojan Uses Plugins to Evade Detection

Figure 10: Medium Coverage of PluginPhathom Malware.

contains 9 plugins. For example, the contact plugin contains the malicious code to steal user's contact information; and the file plugin contains the code to steal the local files. It also has a plugin called `update` which will update each module by downloading the new APK file form the remote server.

Malware Case Study: Dual Instance. Another scenario to abuse plugin technology is to launch multiple instance of an official and authentic victim app (Figure 11). `Dual Instance` is a malware family targeting the authenticated *Twitter* App which first discovered by Avast Threat Intelligence Team [15]. One of the samples, *DualTwitter*, is downloaded by users who want to login multiple twitter accounts in the same device simultaneously, but definitely at price. Once the user log into twitter using this app, the malware will log the keyboard input and steal user's credential information. DualTwitter is powered by the library `VirtualApp`, a different implementation of plugin technology, to load multiple instances of twitter app at the same time without repackaging the original twitter app. We have also find the similar malware targeting on authenticated *Instagram* app.

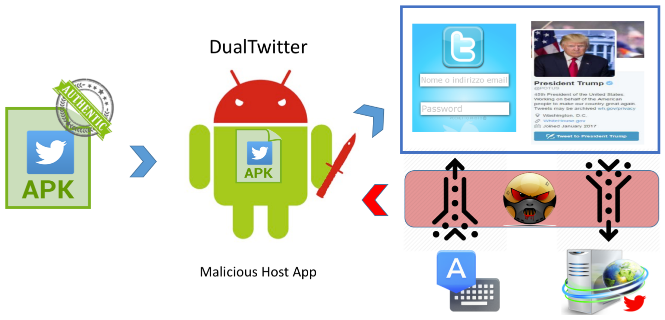


Figure 11: Dual Instance Malware: DualTwitter

4. SOLUTION

The trend in abusing of plugin technology is in massive scale, to prevent benign apps being the victim of this new type of attack, we propose a lightweight defense mechanism, `Plugin-Killer`. It is a Android SDK that helps benign apps detect whether its APK file is running in the virtual environment created by the plugin technology. We have also open-source our SDK that is available to be downloaded from GitHub.

4.1 Potential Solutions

Since we are the first to notice that plugin technology is abused, there is no existing solutions to mitigate the problem. We will share our thinking on the potential solutions and explain why we choose Plugin-Killer as our best solution.

- Blocking Plugin Technology. Obviously, Android plugin technology completely violates the security assumption of mobile platform and leads to multiple severe threats. Blocking this feature is a straightforward solution so that we can make the ecosystem [18] back to secure again. However, this solution is not practical due to the huge demanding from users who use benign apps powered by plugin technology in their daily life. In addition, it is not feasible to block every implementation of plugin technology since there is no strict standard.

- Detecting Malware powered by Plugin Technology. Security product such as firewall or antivirus software could warn users when they install this kind of malware. However, it is hard to distinguish between the malware and legal Android plugin app. It is because malware even use the same dual instance SDK as in the benign app. As the result, they both have a similar behavior. It will leads to a high false positive rate in the detection system.

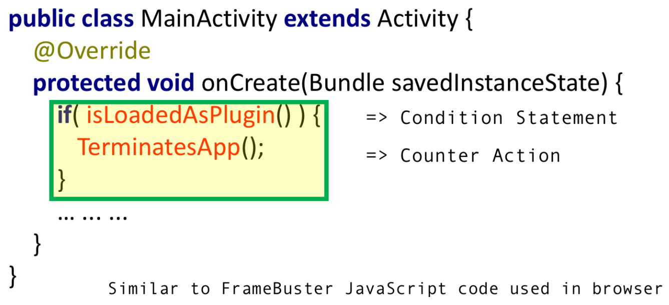
- Providing Plugin Technology by Android platform. The best solution is for the Android system to support a secure mechanism to launch multiple instances of an app, and we believe it is the ultimate solution to this new type of attack. In this solution, Android system can extend the existing Android security system to isolate the multiple instances of the same app, and design a interaction protocol for users to configure the apps that they want to launch multiple instances. By doing so, users won't need to reply on third-party apps as the host app, and the Android system is playing the role as host app. This solution is quite similar to what `AFrame` [21] proposed for isolating 3rd-party advertising libraries from the host app. However, it requires relatively longer period of time to design, release and update the new mechanism to all of the mobile devices (e.g. such as the compatibility issue to different Android versions). A lightweight and faster deployed solution is needed soon.

4.2 Our Solution: Plugin-Killer

The rational behind our solution is based on the fundamental problem we formulated: the authentic app does not aware of being launching as a plugin. In other words, the reason why the malware like DualTwitter can phishing the twitter app is because the original twitter APK file cannot distinguish whether it is launched as a plugin or not. Therefore, we would like to propose a method that allows Android apps to detect whether they are running in the virtual environment created by the plugin technology, and support an option for them decide whether stop running. Based on our analysis on the potential solutions, our solution cannot block the plugin technology and cannot involve any modification of the system.

Plugin-Killer Usage Scenario. Our solution, `Plugin-Killer`, is a defense mechanism that provides multiple ways to detect the virtual environment to prevent the potential threats. It is implemented as a library or SDK, and for benign apps that do not want to be a victim on this new type of phishing attack, they can embed our library in there app. As we just explained, it is very dangerous for a benign app to be launched as a plugin in a third-party app. Our library, PluginKiller, provides a way for the developers of benign apps to prevent being launched as a plugin.

Benign apps are the customers to use our library, which want to detect the potential threats when launched as a plugin. For example, Twitter's authentic app can embed PluginKiller library in their APK file, and they will gain the power to detect the potential threats when it was launched as a plugin by the `DualTwitter` malware. As figure 12 shows, the Twitter app just needs to add 3 lines of code at the beginning of its code, such as in the main activity's *onCreate* function.



```
public class MainActivity extends Activity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    if( isLoadedAsPlugin() ) {        => Condition Statement
      TerminatesApp();                => Counter Action
    }
    … … …
  }
}
        Similar to FrameBuster JavaScript code used in browser
```

Figure 12: How to use Plugin-Killer

The two functions (marked in red) are implemented by our PluginKiller library: the purpose of `isLoadedAsPlugin` function is to return a boolean value to tell the benign app whether running environment is virtual or not; the function `TerminatesApp` is another API implemented in our library to terminate the app or try to alert users to launch the app from launcher in the traditional way.

For people who are familiar with cyber security, it is easy to draw an analogy between clickjacking attack with Dual Instance attack. In clickjacking attack, the victim webpage can be loaded inside the iframe of malicious page. To defeat the attack, when the victim server generate the web page, it usually add a piece JavaScript code to detect whether the page is loaded inside the main frame or sub-frame. This technique, called `FrameBusting`, is utilized by majority of popular websites. We believe the Duel Instance malware shares the similar fundamental problem as the clickjacking attack. If mobile apps can bust from the virtual environment created by the malicious host app, they can successfully prevent them from this new type of attack.

Advantages of Plugin-Killer. Our solution and library is the only one to provide the opt-out option for users and app developers. Moreover, our solution has the following advantages:

- Only Solution. Our work is the first one to notice the massive abusing of plugin technology by malware, and PluginKiller is the best solution so far to help benign apps to not be a victim of this new type of attacks.

- Lightweight. Our approach does not requires any mobile system changes, and it works on any version of

Android system. Mobile apps does not need to make any changes to their code but adding the 3 lines of code we explained above.

- Compatibility. Plugin technology is supported by all the Android versions, and our solution should support all of them. Due to the method we designed to detect virtual environment is only rely on the common features supported by all of the Android versions, our solution is compatible to all of the Android versions

- Easy to Use. Plugin-Killer library is quite small since it only contains few function calls and few detection logics.

4.3 How to Detect Virtual Environment?

To defeat against being dynamically loaded as a plugin to an untrusted host app, we have to figure out a way for a mobile app to detect whether it is being loaded as a plugin. Our observation is that, although Android plugin technology creates a virtual environment to load and launch plugin, the virtual environment still has lots of behavior differences with the one created by the system in the traditional way.

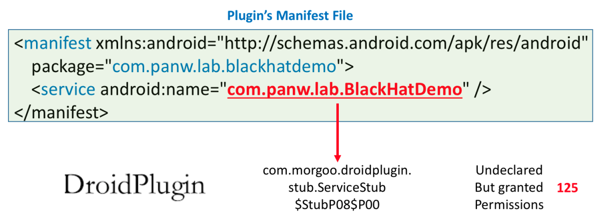
In our library, we systematically enumerates all of the potential methods to distinguish the behavior differences, including:

Detecting Mismatch in the Manifest. Every mobile app must have an AndroidManifest.xml file in its root directory. The manifest file provides essential information about the app to the Android system, which the system must have before it can run any of the app's code. During installation phase, system will parse the this file and record the information defined in it (e.g. declared permissions, components like activities or service). However, the plugin app has never been installed and it is launched dynamically by the host app. The manifest of plugin app and host app has many mismatched information. Our library will try to detect the virtual environment from finding the difference of plugin app and host app. We have listed some potential items defined in the manifest file that we can used to detect the behavior differences (Figure 13):

- `Permission`. Since users may launch any APK file as plugin, the host app usually declared most of the Android permissions (e.g. DroidPlugin declared 125 Android permissions). However, nearly for all of the host apps and libraries did not drop permissions that did not declared by plugin. Therefore, plugin will be granted more permissions then they declared in the manifest file. We can either leverage *PackageManager* to get the granted permission from the *PackageInfo* of the host app or try to access certain restricted resources that requires the permission not declared by plugin.

- `Package Name`. All Android apps installed to the system have a package name which uniquely identifies the app on the device. Since the guest app has never been installed, it can check whether its package name is registered to the system.

- `App component Name`. As we explained, DroidPlugin utilized the stub component to fool the AMS in order to create a component that not defined in the manifest file. Therefore, AMS records information of the stub component, not the actual plugin component. For example, if we use the API *getRunningServices* of *ActivityManager* to get the information of the running service, we will get the name of the stub service, such as *stub.ServiceStubStubP*08*P00* if using DroidPlugin.



```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="com.panw.lab.blackhatdemo">
   <service android:name="com.panw.lab.BlackHatDemo" />
</manifest>
```

Figure 13: Mismatch of host and plugin's manifest file.

Detecting Host App's Runtime Info. Due to the special way the plugin app is launched by the host app, some runtime information of the plugin will have slight difference with one launched by the system. Therefore, we can try to distinguish the virtual environment by checking the runtime information.

- `Process Info.` Unlike PID (Process ID) which is transient and keeps changing all the time, UID is assigned for each application at install time, stays constant as long as the application is not reinstalled. The UID should be unique to each application, except when the application explicitly requests to share a userid with another application. Since the plugin APK file has never been installed to the system, plugin package does not have an unique UID. Even though the host app can fork a new process to launch it, like what DroidPlugin did, the plugin process shared the different PID but same UID with the host app's process.

  To retrieve the running app's process information, we can declare the permission *GET_TASKS* (deprecated after Lollipop), use the API *getRunningAppProcesses* of *ActivityManager* class to get all of process information in a list, and iterate the list to read the process info from the *RunningAppProcessInfo* structure. For example, the process name to launch plugin in DroidPlugin library will be like {*host_app_pkg_name*}*:PluginP02.*

- `Internal Storage Info.` Android system supports an app to save files to internal storage, which directory is specified by the app's package name in a special location of the Android file system. The directory of the dynamically launched app is not specified by its package name but the host app's package name. For example, we can get the *dataDir* of the app from the *packageManager*, which is a directory assigned to the package for its persistent data. Usually, the path is '*/data/data/*{*pkg_name*}'. However, DroidPlugin needs to separate the data saved by different plugins, and each plugin will be assign the subdirectory of the host app's data directory as their *dataDir*. For example, the *dataDir* of the plugin in DroidPlugin is */data/data/*{*host_app_pkg_name*}*/Plugin/* {*plugin_pkg_name*}*/data/*.

Detecting App Components. App components are the essential building blocks of an Android app. A unique aspect of these app components is that any app can start another app's component. Android system supports asynchronous message mechanism, called intent, which binds individual components to each other at runtime. However, the dynamically launched guest app is transparent to android platform and other apps running in the same device. To support the interaction between the components between guest app and other apps, host app needs to customize the Intent mechanism. Due to the limitation of the customization, it leads to the behavior difference.

- `Number of Launched App Activity and Service`. App must declare its activity and service in the manifest file in order for it to be accessible to the system. However, since the guest app has never been installed, host app declared some stub activities and services as a dummy ones to fool the AMS. However, host app will only predefine certain number of stub components. In DroidPlugin, it pre-defined 10 stub activities and 10 stub services. If the plugin launched more than 10 services, it leads to a behavior difference.

- `Static Broadcast Receiver`. Android apps can send or receive broadcast messages from the Android system and other Android apps, similar to the publish-subscribe design pattern. Apps can register to receive specific broadcasts. When a broadcast is sent, the system automatically routes broadcasts to apps that have subscribed to receive that particular type of broadcast. App can register broadcast receiver statically or dynamically. In static way, app uses *receiver* and *intent-filter* to define a receiver in the manifest file. When the app is installed, system parsed the manifest and create the receiver for the app. App also can invoke the function *registerReceiver* to dynamically register an receiver instance, and the dynamic registration will be associated with lifecycle of the app.

  Since the plugin app has never been installed, the static receivers defined in the manifest file will not be parsed by the system. Therefore, DroidPlugin will parse the manifest file to retrieve the broadcast receivers defined by the plugin, and generate the code to dynamically register them at runtime. Since the static and dynamic registered receives are equivalent in terms of intercepting broadcasts. Plugin won't distinguish the different. However, if we manipulate the receiver at runtime, we can detect the change.

  For example, we can define a static broadcast receiver in plugin's manifest file, and unregister all of the dynamic receivers at runtime (use API *unregisterReceiver*). In the real Android environment, the static receiver is still effective; but in the virtual environment created by DroidPlugin, all of the static and dynamic receivers are gone. Later on, if we send a broadcast that should be intercepted by the static receiver, we can detect the whether current environment is virtual or not.

- `Change App Component Property at Runtime`. Another technique we can use is to change the property of an app component at runtime. When the app defined app components in the manifest file, it can also define the property of the components, such as component name, default status or labels. One of the special properties is *enable*, which tell the system whether to instantiate the component when launching the app. In normal launching procedure, the app have a change to enable the component at runtime by calling the API *setComponentEnabledSetting* with the component name. After enabling, the component will be alive and well-functioned.

  The tricky part is that, in the virtual environment, DroidPlugin cannot instantiate a disabled component. As figure 14 shows, we defined broadcast receiver in the plugin's manifest file, and set the *enabled* property of it as false by default. Then, we user the API *setComponentEnabledSetting* to enable this receiver at runtime. If it is running as a plugin, this runtime enabling will not take any effect, and the receiver is still in disable status. We can easily send a broadcast that matching this receiver to check whether it is enabled or not, and find evidence of whether the app is running as a plugin.

```
<receiver android:name=".AntiReceiver"
  android:enabled = " false ">
  <intent-filter>
    <action android:name="ANTI_STATIC" />
  </intent-filter>
</receiver>
```

```
ctx.getPackageManager().setComponentEnabledSetting(
     ComponentName, COMPONENT_ENABLED_STATE_ENABLED, …
)
```

Figure 14: Enable a Disabled Broadcast Receiver at Runtime.

Detecting Residues of Other Guest App.

- Shared Native Components. Some native components share the internal information among the instances within a same app. Even if the plugin is not running, the host app keeps the shared information anyway, leading to another form of data residue instance [20, 22]. For example, WebView [10, 16, 17], a web container implemented as a native C++ library, shares the browser states (e.g. cookies, saved form info) in a local database among the WebView instances within a same app. In dual instance case, the host app and guest apps are treated as a same app since they shared the same UID. Therefore, if user's login to a website in a WebView instance embedded in a guest app, other guest apps can embed a WebView instance to detect the login status and further determine if itself are launched in the virtual environment.

4.4 Evaluation of PluginKiller.

To check whether PluginKiller can detect different virtual environments cased by various of implementation, we takes several popular host apps and open-sourced libraries to evaluate it. It does not mean the popular host apps, such as Parallel Space, Gemini and etc, listed in Figure 1, are untrusted or malicious. The reason why we test it is because there is no such a standard of how to implement plugin technology. Attackers may reverse-engineering their code and use a similar way to implement their own malware, or even come up a new ways to implement plugin technology. Although we only detected the malware that utilize the plugin library *DroidPlugin* and *VirtualCore* at this time, we would like to show that it is highly possible that PluginKiller could detect every potential virtual environment to be created by malwares.

Figure 15 shows the evaluation result. Each row represents the detection result for a test case as we discussed earlier, and each column is one of the different virtual environments. We build a dummy APK file that only embeds PluginKiller library, and launch it as a plugin using different types of host apps to get detection result. The evaluation shows that PluginKiller can detect all of the current virtual environments.

	Droid Plugin	Go Multiple	Multiple Accounts	Parallel Space	Parallel Accounts	Parallel Box	Gemini
ServiceName Check	DETECTED				DETECTED	DETECTED	DETECTED
Undeclared Permission	DETECTED	DETECTED	DETECTED	DETECTED	DETECTED	DETECTED	DETECTED
SharedUID ProcessCheck	DETECTED	DETECTED	DETECTED	DETECTED	DETECTED	DETECTED	DETECTED
AppRuntimeDir Check	DETECTED	DETECTED	DETECTED	DETECTED	DETECTED	DETECTED	DETECTED
ReceiverFilter Check	DETECTED			DETECTED	DETECTED		
EnabledComp Check	DETECTED			DETECTED	DETECTED		

Figure 15: Evaluation of Plugin Killer for all virtual environment.

5. REFERENCES

[1] Android plugin technology analysis (in chinese). http://weishu.me/2016/04/05/understand-plugin-framework-classloader/.

[2] How parallel space helps you run multiple accounts on android. http://www.geekwire.com/sponsor-post/parallel-space-helps-run-multiple-accounts-android/.

[3] Rebert Abel. Pluginphantom trojan expoits android plugins to snoop. *SC Magazine*, Nov 2016.

[4] Catalin Cimpanu. Pluginphantom android malware uses novel approach to hide malicious behavior. Nov 2016.

[5] Zheng Cong and Luo Tongbo. Pluginphantom: New android trojan abuses "droidplugin" framework. *Blog of Palo Alto Networks Research Center*, Nov 2016.

[6] Valerio Costamagna and Cong Zheng. Artdroid: A virtual-method hooking framework on android ART runtime. In *Proceedings of the 1st International Workshop on Innovations in Mobile Privacy and Security, IMPS 2016, co-located with the International Symposium on Engineering Secure Software and Systems (ESSoS 2016), London, UK, April 6, 2016.*, pages 20–28, 2016.

[7] Wenjun Hu, Xiaobo Ma, and Xiapu Luo. Protecting android apps against reverse engineering. *Protecting Mobile Networks and Devices: Challenges and Solutions*, page 155, 2016.

[8] Wenjun Hu, Jing Tao, Xiaobo Ma, Wenyu Zhou, Shuang Zhao, and Ting Han. Migdroid: Detecting app-repackaging android malware via method invocation graph. In *Computer Communication and Networks (ICCCN), 2014 23rd International Conference on*, pages 1–7. IEEE, 2014.

[9] Yajin Zhou Xuxian Jiang. Detecting passive content leaks and pollution in android applications. In *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS)*, 2013.

[10] Xing Jin, Lusha Wang, Tongbo Luo, and Wenliang Du. Fine-grained access control for html5-based mobile applications in android. In *Proceedings of the 16th Information Security Conference (ISC)*, 2013.

[11] Eduard Kovacs. Pluginphantom android trojan uses plugins to evade detection. Nov 2016.

[12] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *NDSS*, volume 14, pages 23–26, 2014.

[13] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. Droidchameleon: evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 329–334. ACM, 2013.

[14] Trustlook Research. Trojan attempts to replace system launcher and collects confidential information. 7 2016.

[15] Threat Intelligence Team. Malware posing as dual instance app steals users' twitter credentials. October 2016.

[16] Luo Tongbo. *ATTACKS AND COUNTERMEASURES FOR WEBVIEW ON MOBILE SYSTEMS*. PhD thesis, Syracuse University, 2014.

[17] Luo Tongbo, Hao Hao, Du Wenliang, Wang Yifei, and Yin Heng. Attacks on webview in the android system. In *Annual Computer Security Applications Conference*.

[18] Meng Xu, Chengyu Song, Yang Ji, Ming-Wei Shih, Kangjie Lu, Cong Zheng, Ruian Duan, Yeongjin Jang, Byoungyoung Lee, Chenxiong Qian, Sangho Lee, and Taesoo Kim. Toward engineering a secure android ecosystem: A survey of existing techniques. *ACM Comput. Surv.*, 49(2):38:1–38:47, August 2016.

[19] Wenbo Yang, Yuanyuan Zhang, Juanru Li, Junliang Shu, Bodong Li, Wenjun Hu, and Dawu Gu. Appspear: Bytecode decrypting and dex reassembling for packed android malware. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 9404*, RAID 2015, pages 359–381, New York, NY, USA, 2015. Springer-Verlag New York, Inc.

[20] Xiao Zhang, Yousra Aafer, Kailiang Ying, and Wenliang Du. *Hey, You, Get Off of My Image: Detecting Data Residue in Android Images*, pages 401–421. Springer International Publishing, Cham, 2016.

[21] Xiao Zhang, Amit Ahlawat, and Wenliang Du. Aframe: Isolating advertisements from mobile applications in android. In *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC '13, pages 9–18, New York, NY, USA, 2013. ACM.

[22] Xiao Zhang, Kailiang Ying, Yousra Aafer, Zhenshen Qiu, and Wenliang Du. Life after app uninstallation: Are the data still alive? data residue attacks on android. In *NDSS*, 2016.

[23] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. Smartdroid: An automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, pages 93–104, New York, NY, USA, 2012. ACM.