PROMON

App Threat Report

The State of Game Security

Q1 - 2023

Index

Introduction

Welcome to our Q1 App Threat Report, Promon's quarterly analysis of current topics in mobile application security produced by our Security Research Team. This report reviews mobile gaming security, exploring how the top games by revenue protect themselves against hooking frameworks, repackaging, and rooted devices.

Mobile game downloads skyrocketed during the pandemic. While global consumer spending declined in 2022, the worldwide revenue generated still accounted for over 90B last year, according to a recent Newzoo report, with over 1.7B users playing downloaded games. In another report from Beyond Identity, 67% of mobile gamers reported that they had been hacked while playing at least once, losing $359 on average.

Game developers spend billions on developing their proprietary IP and want to protect that investment to protect their reputation and defend their revenue stream.

Our Q1 report explores the overall security level for mobile games. To assess that level, we checked more than 350 Android games to see how they fared against our repackaging attacks, deployment of hooking frameworks, and how they handled the detection of a rooted device. Initial results showed that most apps were vulnerable to our attacks.

We hope our findings help companies improve their security posture to protect their revenue, brand and IP, and users from harm.

Executive summary

Promon tested 357 of the world's highest revenue-generating mobile games for Android. These games represent more than $10.5 Billion in annual revenue according to SensorTower data.

We subjected each game to four tests: repackaging, hooking via Frida, hooking via LSPosed, and rooting detection test. 289 (81%) offered no defenses against repackaging or hooking and did not detect running on a rooted device.

Fifty-six apps (15.7%) deployed some form of repackaging detection, making it the most "defended against" attack vector.

To evaluate defense against hooking frameworks, we attached both Frida and LSPosed to apps. Thirty apps (8.4%) could defend against Frida, while only 16 (4.5%) detected LSPosed.

Only one app could detect the presence of a rooted device when we used all appropriate hiding methods, making it the security area of the least concern to game developers.

We also grouped the apps by revenue to see if apps with more significant revenue employed more defensive technologies. 33% of apps generating more than $100 million in annual revenue (source: SensorTower) prevented repackaging, more than any other revenue split, but none were able to detect LSPosed, the only split where no app detected the hooking framework.

Interestingly, apps generating less than $5 million per year had perhaps the most robust overall security posture. 16% could detect Frida (highest among all the revenue splits), 5% could detect LSPosed (tied for 2nd), 21% could prevent repackaging (tied for 2nd).

Overall, our automated, relatively straightforward attacks worked well across all the apps tested, underscoring a pressing need for all apps to improve their security level.

Repackaging

What is a repackaging attack?

Repackaging attacks modify or extend the code of an existing application and then package it into a new application.

While this report focuses on Android, attackers can also repackage iOS apps. Apple cites the risks of repackaging in its decision not to allow sideloading iOS apps. However, app developers may still find their apps on jailbroken iPhones, and sideloading is possible using, for example, enterprise distribution solutions. In some cases, applications can be repackaged, rebranded, and re-released to the Apple App Store, making sideloading unnecessary and negating attempts to downplay the risk of repackaging on iOS.

What are the risks of repackaging?

Malicious code injection carries significant risks for both app owners and users. Repackaging makes it possible to start and run or use a fraudulent version of an app after repackaging the original app.

It is possible to manipulate the app at rest, changing implemented code, checks, and security. An attacker could then provide an alternative app in the Google Play Store or "drive-by-download."

Malicious actors could add foreign code and features, like malware, or remove existing security flags. The repackaged original app would then, at startup, not notice changes.

Repackaging attacks can also allow other entities to leverage a company's proprietary IP or open a company up to brand and reputational damage.

Do app stores protect against repackaged apps?

Unlike major app stores, third-party app stores may not have such stringent policies. Also, attackers can use various methods, from ads to spam, to trick users into downloading a repackaged app. Pokemon Go in 2016, and Fortnite in 2018 were examples of games that were exposed to repackaging, and other apps like WhatsApp have also been successfully repackaged.

The new EU Digital Markets Act, effective in May 2023, will impact the mobile gaming market. This legislation aims to prevent large tech companies from "gatekeeping" the digital marketplace and instead encourages competition by enabling smaller players to enter the market and allowing third-party app stores and payment processors to operate outside of the Apple App Store and Google Play. For the mobile gaming industry, large publishers may have their own app stores and charge their users directly outside of the two traditional app stores. In contrast, smaller developers will have to depend on established app stores. However, this may lead to more fragmentation, which could increase security risks for users and developers due to hackers and fraudsters circulating repackaged or rooted apps in third-party app stores.

How to protect your app

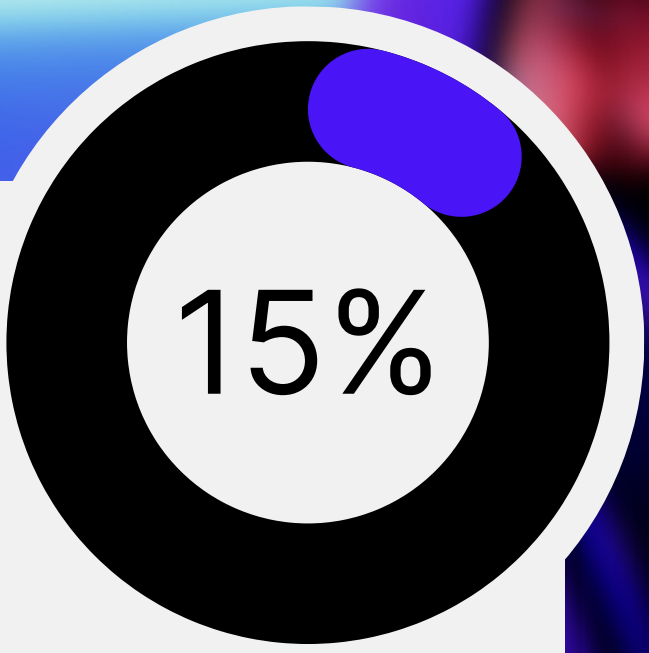
Robust repackaging detection and protection on multiple layers are required to mitigate this attack vector. The protection technology needs to understand how operating systems verify packages, how those are signed, and how they can be verified as original by the app publisher and not manipulated at rest.

Android and iOS developers can add these features to their apps:

1. Independent verification of the app's signature. O.S. verification will not work if it has been disabled (e.g., on a Jailbroken phone) or re-signed with a different but valid distributor key.
2. Verification of app resources before use.
3. Code integrity checks to detect tampering and binary modification.

Results

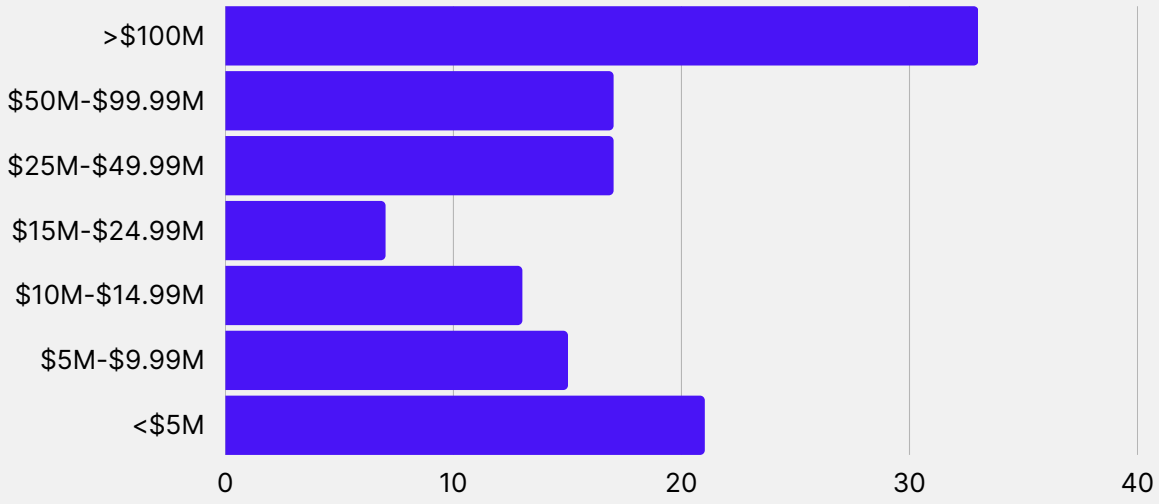
REPACKAGING ATTACKS



Repackaging was the most-prevented attack overall, yet Promon's tests successfully repackaged almost 85% of all the apps tested (301 apps in total) – meaning only 15% of apps successfully prevented it.

Our analysis also indicated that these apps could not mitigate or defend against such attacks and likely could not detect code injection and repackaging attacks.

Percentage of repackaged apps by annual revenue



33% of apps with $100M or more in annual revenue prevented repackaging. Interestingly, apps with under $5 million in annual revenue.

Hooking frameworks

What are hooking frameworks?

Hooking frameworks are tools used to intercept, modify, and redirect function calls and other events in a running mobile application. These frameworks allow developers and security researchers to monitor and analyze an application's behavior in real-time, which can help identify vulnerabilities and detect malicious activity.

Some popular hooking frameworks in mobile app security include Frida and LSPosed (see below for more on both).

While these hooking frameworks can be used for legitimate purposes, such as testing the security of an application or debugging a problem, they can also be used for malicious purposes, such as stealing sensitive information, understanding the inner workings of the application, manipulating the app during runtime, faking system calls and more. Attaching these hooking frameworks to an app is an essential step in developing and executing an attack against an app.

Frida

Frida is a dynamic instrumentation tool used for reverse engineering, debugging, and analyzing the behavior of applications on various platforms, including Android, iOS, Windows, and macOS.

Frida allows developers and security researchers to inject their own custom code into an application at runtime, which gives them unprecedented access to the application's behavior and underlying system. With Frida, you can monitor and intercept function calls, method invocations, network traffic, and system events, among other things.

Frida consists of two main components: a runtime library that is either injected into the application on disk via repackaging or injected into the application during runtime via code injection, and a Python-based command-line tool that you can use to communicate with the library and perform various tasks, such as injecting code, debugging, and tracing.

One of the most significant advantages of Frida is its ability to bypass common anti-debugging and anti-tampering measures that developers use to protect their applications from reverse engineering and hacking.

Frida does this by injecting its own code into the target process, which makes it difficult for the application to detect and prevent tampering.

Frida is an incredibly versatile tool that can be used for a variety of tasks, from analyzing malware to testing the security of your applications. It's a popular tool in the security research community used by both security research and malicious hackers.

LSPosed

LSPosed is a hooking framework that allows users to customize the behavior of their Android device. It is based on the Xposed Framework and requires root access to function. LSPosed offers a user-friendly interface for managing modules, which are packages of code that can be loaded into the framework to alter the behavior of the Android system and individual apps installed on the device.

While LSPosed and Frida have similar uses, Frida provides an extensive set of reverse engineering tools while LSPosed is more focused on providing the user the possibility to modify the Java code of apps during runtime by writing modules.

Why should developers protect against hooking frameworks?

Mobile game developers should protect against hooking frameworks for a few key reasons:

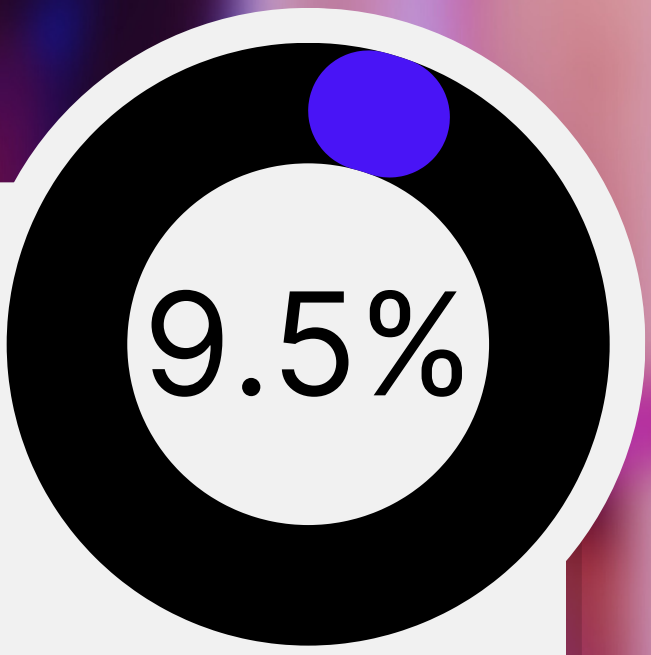
1. Cheating: Players can use hooking frameworks to cheat in games by modifying the game's code or data to gain an unfair advantage over other players. This can lead to an imbalance in the game's economy and frustrate legitimate players, leading to a decline in the game's popularity and erosion of its revenue base.
2. Revenue loss: Game developers can also suffer revenue loss from cheating because players who cheat are less likely to spend money on in-game purchases if they can obtain the same benefits through cheating.
3. Security risks: Hooking frameworks can also be used by malicious actors to reverse engineer and extract sensitive information from the game, such as proprietary game code, user data, or cryptographic keys. This can lead to IP theft, data breaches, and other security risks.
4. Reputation damage: If a game is widely known to be vulnerable to cheating or other security risks, it can damage the game developer's reputation and lead to lost trust among players.

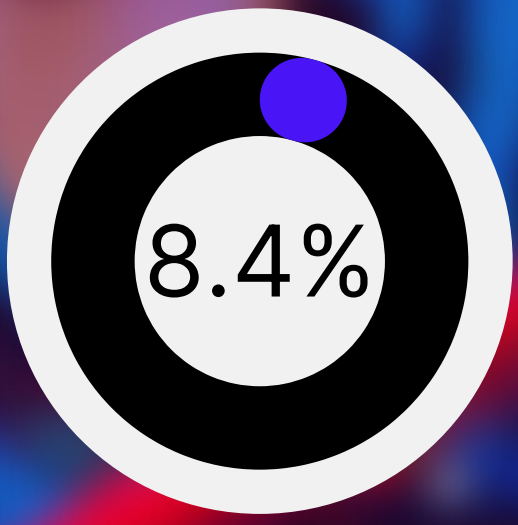
Results

HOOKING FRAMEWORKS

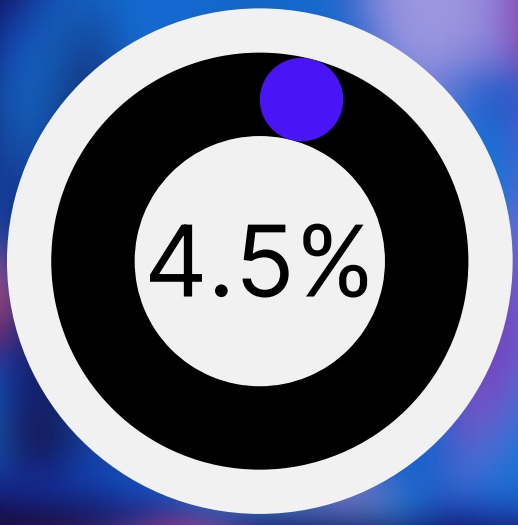
To test the security level against hooking frameworks, we attempted to hook apps using popular attack tools Frida and LSPosed.

Overall, 34 apps contained some hooking framework protection (9.5%). 30 of the 357 apps detected Frida (8.4%), while only 16 (4.5%) were able to detect LSPosed. Only 12 of the 357 apps tested could detect both Frida and LSPosed (3.4%).

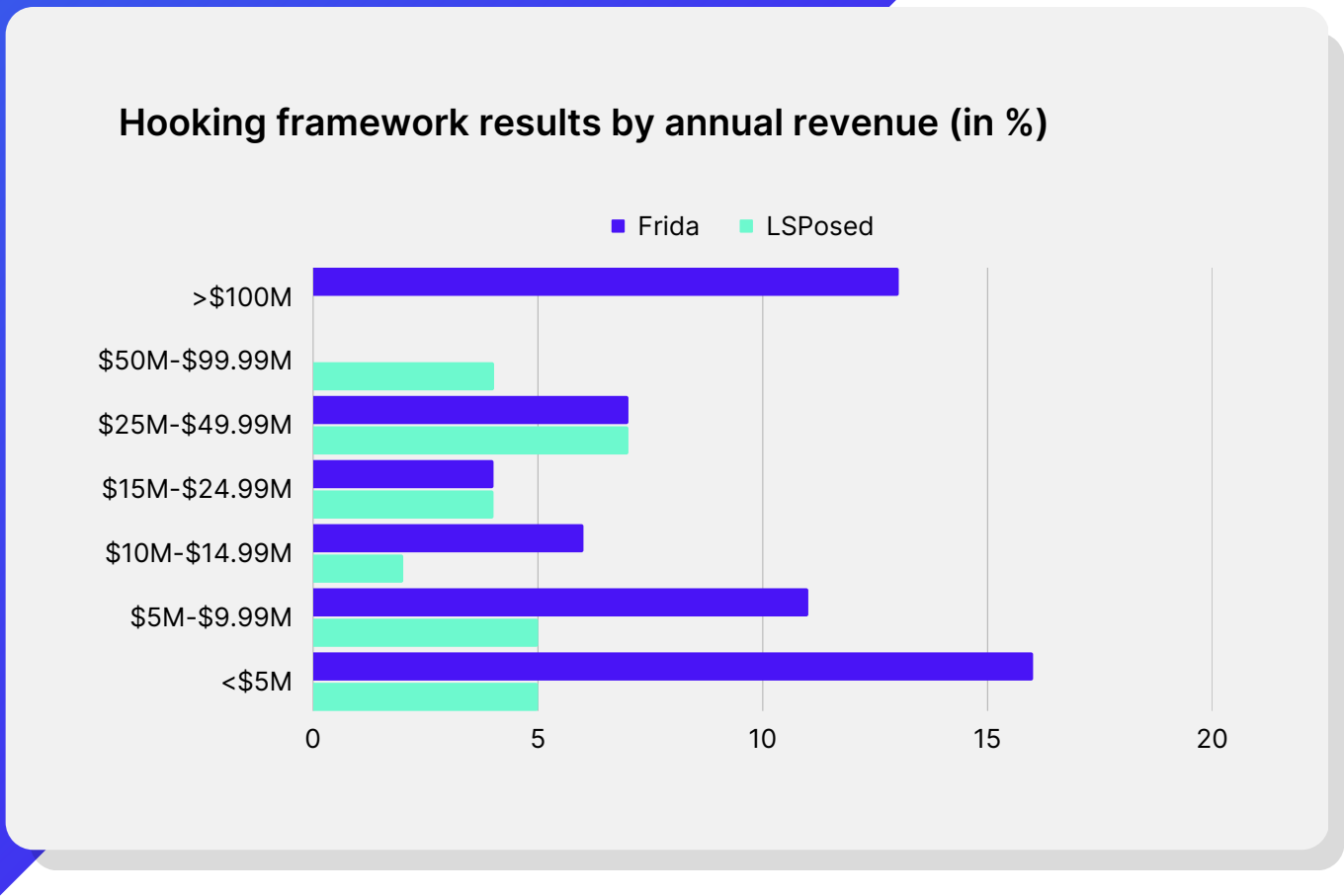




Frida



LSPosed



13% of apps with $100M or more in annual revenue could detect Frida, although none could detect LSposed. Curiously, apps with between $50M and $99M in annual revenue had no protection from Frida, and only one app detected LSposed. Apps with lower annual revenue fared better; 16% of apps with less than $5M in annual revenue could detect Frida, while 11% of apps with between $5M and $9.99M could. LSposed was detected by 5% of apps in both buckets.

How to protect your app

To protect against hooking frameworks, mobile game developers can implement various security measures, such as runtime protection and obfuscation techniques, to make it harder for hackers to reverse engineer the game's code. They can use server-side validation to detect and prevent cheating and integrate app shielding technologies to adopt a comprehensive app security posture. By implementing these measures, game developers can help ensure that their game remains fair, secure, and enjoyable for all players.

Rooting

What is rooting/jailbreaking?

Rooting and jailbreaking are privilege escalation methods used to bypass the security restrictions placed on mobile devices, allowing users to gain root or administrative access to the device's operating system. In the context of mobile apps, rooting and jailbreaking can allow users to access and modify the behavior of apps in ways that are not intended by the app developers or the operating system.

Rooting refers to the process of gaining administrative access to an Android device. By rooting an Android device, users can access system files and memory and modify the behavior of the operating system and individual apps. Rooting can be done by exploiting security vulnerabilities in the device's firmware or by using specialized software tools.

Jailbreaking refers to the process of gaining administrative access to an iOS device. By jailbreaking an iOS device, users can install and run applications that are not approved by Apple, access system files, and modify the behavior of the operating system and individual apps. Jailbreaking can be accomplished by exploiting security vulnerabilities in the iOS firmware and by using specialized software tools.

While rooting and jailbreaking can provide users with more control over their devices, they also introduce security risks and can compromise the stability and performance of the device and apps. Rooted or jailbroken devices are also more vulnerable to malware and other security threats, as security features provided by the operating system may be disabled or circumvented.

Is this a problem for gaming?

The question of whether mobile games on Android should detect running on a rooted device is complicated. On one hand, root access can be used to cheat in games, giving players an unfair advantage over others. Root detection can help prevent this by blocking access to the game for rooted devices.

On the other hand, many legitimate users rely on root access to perform essential tasks on their devices, such as customizing the OS or installing certain apps. Additionally, due to the diversity of the Android ecosystem, there are rare instances of devices that come pre-rooted from the factory. Root detection can prevent these users from playing the game, even if they have no intention of cheating.

For gaming, a potential solution could be to detect root presence, report it to a central server, and, afterward, restrict access to their account if any suspicious behaviors are detected from this user.

With Android being open source and highly customizable, it is also possible to release versions of the operating system that are impossible for any solution to detect whether are rooted or not. Because of this, one should never rely on root detection as the only security feature, but rather assume that the system is rooted, and secure the application with other security measures.

About Magisk

Our report used Magisk, a popular rooting tool for Android devices that allows users to gain root access to their devices.

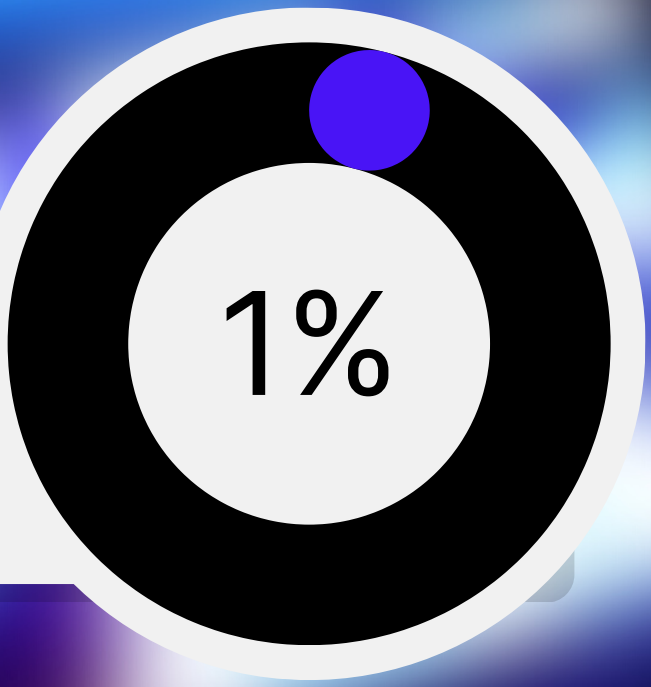
While Magisk itself may not look like much, it opens the door to a vast ecosystem of modules. These modules, with root privileges, can afterward do (almost) anything on the device. One of these modules is LSPosed, which this report also explores.

Magisk also includes a feature called Zygisk, which allows the user to modify the Zygote process (the process that every Android app forks from). This will enable Magisk to control the app before any of the app's code is executed. Magisk also has a DenyList, a list of apps that will not gain root access on the device. This is used to hide the Magisk presence from the app as all the Magisk files are stored in directories not accessible without root permissions.

Results

ROOT DETECTION

Only one app (<1%) could detect the presence of a rooted device, and it was within the third revenue split ($25M-$49.99M).



How to protect your app

To protect against rooted devices, mobile game developers can check for the presence of root or jailbreak detection frameworks. These tools search for 'markers' in the OS, such as files that are accessed when they usually would not be able to, reading/writing memory that should have access, and searching for apps that need rooted capabilities, for example. Open-source frameworks are available that can do trivial root and jailbreak detection. However, these are often trivially bypassed.

Methodology

Selection

Overall, Promon analyzed 357 unique Android apps. Apps tested were determined by finding the games with the most revenue over the past year on the Google Play Store, according to SensorTower.

Testing environment and process

Promon created a script to install and run the unmodified app to ensure it runs in the test environment. The application was run for up to 60 seconds, checking that it had not terminated. If the app did not terminate, the app was marked as valid for future testing. If the app crashed during this test, we assumed that the app or the test setup had a problem, and the app was excluded from future tests.

After passing the initial test, the apps underwent testing through an attack testing framework, which involves steps such as:

1. Initial app modification if needed (repackaging would go here)
2. App installation
3. Environment initialization (tasks like Magisk DenyList insertion would go here)
4. App launch
5. Continuous app monitoring for up to 60s with screenshot taking and log collecting
6. Screenshots are compared to detect potential app freeze
7. Logs are analyzed to determine test results

Afterward, all app screenshots that did not crash during these tests were checked manually for popups that would mention any kind of "hack detection."

There could be apps that launch successfully, do not freeze, do not show any "hack detected" popups but have detected the "attack," and do not (immediately) do something about it. This is challenging to determine but also far from ideal from a security standpoint. If the app knows that it is being attacked, it should not trust that the functionality it uses to prevent the app from running correctly has not been manipulated.

Process: Repackaging

Repackaging was done by decompiling the Java code of the app, inserting simple logging into it, then recompile and sign the app.

Any issues encountered during application modification resulted in the app being classified as not completing the test. Most often, the reason was the presence of certain features in these apps that were not supported by the decompilation/recompilation tools being used.

To be classified as being repackable the app had to not crash for up to 60s and not show a "hack detected," or a "go to play store and re-download the app" popup.

Process: Root detection

As Magisk is currently the main Android rooting framework, the focus was solely on it. A clean, rooted Android device was used, with all Magisk rooting artifacts removed. Magisk app hiding was enabled, as well as Zygisk and the enforcement of the DenyList. The Magisk DenyList is used to hide the presence of Magisk from the apps in the list. Before starting the test, the app and all of its processes were added to the DenyList.

Process: Hooking detection

For hooking, Frida was an obvious choice, as it is the most popular Android hooking/instrumentation framework right now. LSPosed was selected to complement Frida as it has a different "attack" pattern, and by testing with both, our tests could better understand the whole hook detection landscape.

– Frida

We tested Frida by running a Frida server on the device, starting the app with the Frida server attached, and hooking libc functions that are used in every app. The hook logged some data every time it was triggered. The logs were examined afterward for those hook messages, and if any were present and the app did not crash, the hooking attempt was marked as successful.

– LSPosed

For LSPosed hooking, an LSPosed generic module was written that hooked and logged some Java runtime methods. LSPosed hooking was enabled for the tested app, and the app was run normally. Later, same as with Frida, logs were analyzed for the hook messages.

PROMON

About Promon

A comprehensive Application Shielding solution can help reduce the risk of cheating, protect revenue, and defend your brand and intellectual property. Promon SHIELD™ combines advanced obfuscation and robust runtime protection to help protect apps and end-users from harm. Get in touch at promon.co to learn more.