All your Root Checks are Belong to Us:

The sad state of Root detection

Nathan Evans
Symantec Research Labs
Herndon, VA
USA
nathan_evans@symantec.com

Azzedine Benameur
Symantec Research Labs
Herndon, VA
USA
azzedine_benameur@symantec.com

Yun Shen
Symantec Research Labs
Dublin
Ireland
yun_shen@symantec.com

ABSTRACT

Today, mobile devices are ubiquitous; a facet of everyday life for most people. Due to increasing computational power, these devices are used to perform a large number of tasks, from personal email to corporate expense account management. It is a hassle for users to be required to maintain multiple mobile devices to separate personal and corporate activities, but in the past this was a commonplace requirement. Today, the Bring Your Own Device (BYOD) revolution has promised to consolidate personal and business applications onto one device for added convenience and to reduce costs. As business applications move to personal devices, a clear problem has arisen: how to keep business data secure and personal data private when they reside on the same device. Many solutions exist, both for increasing the security of mobile devices as well as BYOD and Mobile Device Management (MDM) software to allow access to business applications and data while keeping it secure.

One chink in the armor for both security and business applications is "rooted" devices. These devices have been unlocked, providing low level system access to users and applications. With root access, users may be able to bypass BYOD mechanisms in place to protect data, and malware may be able to access both private personal and business data on devices. As such, security applications and business applications often attempt to identify rooted devices and report them as compromised. In this paper, we analyze the most popular Android security focused applications along with market leading BYOD solutions to discover how "rooted" devices are identified. We dissect the aforementioned applications with commonly available open source Android reverse engineering frameworks to demonstrate the relative ease of circumventing these root checks. Finally we present AndroPoser, a simple tool that can subdue all the root checks we discovered, allowing "rooted" devices to appear "non-rooted".

MobiWac'15, November 02-06, 2015, Cancun, Mexico
ACM 978-1-4503-3758-8/15/11 ...$15.00
DOI: http://dx.doi.org/10.1145/2810362.2810364

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: [Android Security, Root Detection, Library Interposition]

1. INTRODUCTION

The wide proliferation of smartphones in recent years has led to a boom in Android (Linux based) mobile devices. Android is so popular that, as of the end 2014 it was installed on over 70% [2] of smartphones and tablets sold globally. Following the same trend, personal devices penetrated enterprises; putting corporate crown jewels alongside personal applications, data and threats. This led to the wide spread adoption of Mobile Device Management (MDM) and Bring Your Own Device (BYOD) in corporate security solutions to protect their assets against malicious applications and malicious or unsuspecting users.

Android is successful, in part, due to its open nature that avail users, enterprises, governments and telecommunications providers of numerous customization options. However, Android's openness (and Linux ancestry) has also given rise to applications that leverage "root" access to modify the most intricate parts of the operating system. Such low-level and overprivileged access is often seen as a security risk [6, 8] as it provides complete access to the system that can be leveraged by malware targeting Android devices. For this reason, mobile software security vendors often include checks for rooted devices, either to warn the user, provide added functionality or simply record and report it.

Even without considering the possibility of enabling malware running on a rooted device to do more harm, it is difficult if not impossible for enterprises to guarantee that their corporate policies can be enforced for rooted BYOD devices. Typically corporate policy prohibits the use of certain dangerous apps, requires the use of some security and policy enforcing apps, and may include device tracking and remote shutdown/wipe capability. However, rooted devices threaten these policies because users or malicious apps that gain access can manipulate the underlying operating system. Also, apps used for business purposes often encrypt their data at rest, but a user or application that can observe the memory and library usage of these apps are able to access this data when unencrypted in memory.

Approaches currently exist to turn Android into a trusted platform [3], which allow sensitive functions such as encrypt-

ing and decrypting data, storing encryption keys, etc. in a trusted computing base, but these are not widely deployed. Currently the barriers to entry are that a trusted platform complicates development, requires partnerships with chipmakers, and the functionality of a trusted base is rather limited. Therefore, both security focused applications and BYOD solutions alike embed checks for the sole purpose of detecting if a device is "rooted". Like many security measures root detection and avoidance has quickly become a cat-and-mouse game where applications have emerged to hide traces of "rooting" [4, 1], and application developers have developed more and more checks. While it is well known in the community that checks for "rooted" devices are in place, as well as countermeasures (and counter-counter measures), no systematic analysis of these methods and their usage across security and BYOD/MDM applications exists as far as we know. To fill that gap, in this paper we provide an analysis of the state of the art in root detection across security and BYOD/MDM applications based solely on static analysis of application packages freely available for Android. Our analysis reveals that, unfortunately, root detection remains a tricky task, with no method being able to detect rooted devices in the presence of a determined user. In addition, we provide details on how devices are rooted in the first place, and how privilege escalation to root works on Android. We also discuss which root detection methods are more difficult to evade than others, as well as which ones are better at avoiding detection by simple static analysis.

The remainder of the paper is organized as follows. Section 2 presents related work. Section 3 describes how root access works on mobile devices along with a detailed analysis of the mechanisms used to detect such access amongst popular security focused applications and BYOD/MDM applications. Section 4 presents AndroPoser; a simple tool that despite its simplicity can circumvent all of the checks we encountered, causing apps to behave as if installed on a device that is non-rooted while it in fact is "rooted". Finally Section 5 summarizes our work and concludes the paper.

2. RELATED WORK

MDM In [13] the authors present an enhanced root management system for protecting rooted Android phones by providing a fine-grained policy and more contextual information about applications that are requesting root privileges. Wang *et al.* proposed DeepDroid [15], a dynamic enterprise security policy enforcement scheme on Android devices using dynamic memory instrumentation of a small number of critical system processes without any firmware modification. Rhee *et al.* [12] proposed a new threat modeling methodology to analyze and identify threat agents, assets, and adverse actions against MDM systems.

Root exploits In [6] the authors analyze the incentives of mobile malware. The authors encountered multiple instances of mobile malware leveraging root exploits and questioned the incentives of rooting a phone for malware writers and smartphone owners. RiskRanker [7] aims to scalably detect whether a particular app exhibits dangerous behavior like launching a root exploit. Similar efforts can also be found in [17, 16, 5]. Suarez-Tangil *et al.* proposed Alterdroid [14] for detecting hidden or obfuscated malware components distributed as part of an app package by analyzing the behavioral differences between the original app and repackaged versions. Interestingly, Yeongung Park *et al.* proposed RGBDroid [10] to protect the system by effectively responding after an attacker has already attained root-level access.

Root evasion Applications have emerged to evade root detection; [1] allows users to hide the presence of the **su** binary by renaming or removing it. Others such as [4] use Java reflection to intercept standard API calls and change their behavior. The main limitation is that if the check happens to be done using native code it cannot be bypassed.

Misc. Ongtang *et al.* proposed Secure Application INTeraction (Saint) [9] to govern install-time permission assignment and their run-time use according to security policies defined by the app authors. Rastogi [11] evaluated the state-of-the-art commercial mobile anti-malware products for Android to test how resistant they are against various common obfuscation techniques which are usually leveraged by malware to hide root exploit code.

In our study of the state of the art we were unable to find a paper that focuses on the mechanisms employed by applications to detect the presence of "rooted" phones. Thus we believe this to be the first paper to investigate these methods and uncover their simplicity.

3. ROOT CHECKS

3.1 How does Root work

There are two common ways that root access is attained on Android: either a custom Android image (aka ROM) is installed that provides privilege escalation or an application exploits a flaw in the operating system to add a privilege escalation binary to the system partition.

Unlike standard Linux systems the invocation of "su" is commonly not enough to elevate privileges to root. Following the conventions of the Android ecosystem root privilege is managed through intents. With these intents, an application must ask for permission to escalate privileges and this request must be granted by a supervisor process. Figure 1 depicts the life-cycle of an application requesting root privileges. First the application invokes the **su** binary, which emits a root request intent that gets verified by a supervisor application (e.g. SuperUser) against an authorization policy engine before returning the grant/deny status back to **su**. Finally, commands which need to run as root are forwarded to the **su_daemon** process for execution. The **su_daemon** process is started at device boot time by *init*, and therefore runs under user id 0 (in the root context). This rather intricate system is a recent change to cope with new security features that have been added since Android 4.3.

3.2 ToolBox and Workflow

Android applications are distributed as Android Application Packages (APKs). The APK is a container that contains an app's code along with its resources, assets, certificates, and manifest file required for installation and execution. The program's code is encapsulated into a Dalvik Executable format, Dalvik being the Virtual Machine that interprets said bytecode. The Android build process starts from Java,

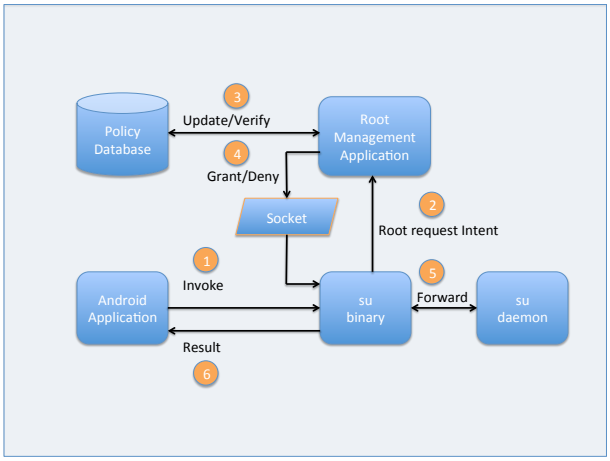

Figure 1: How Root works

compiles it to Java bytecode and converts it to Dalvik instructions as necessary. Thus, the process of converting the Android bytecode back to Java bytecode has gained a lot of interest as the last step from Java bytecode back to Java source can leverage the already well populated space of Java decompilers. Therefore for our research we decided to go, whenever possible, from Dalvik Executable (DEX) format back to Java source. To that end we leveraged the following set of tools:

- Apktool: decompiles android byte code to an intermediate language (in case the Java source code was not fully recovered or the analysis was inconclusive).
- Dex2Jar: converts android bytecode to Java Archive (JAR).
- JD-core: converts JAR to Java source code.
- Custom Scripts: automates the process and searches for obvious Java calls and broad references to rooted phone features described in Section 3.3.

3.3 Common Root Discovery

In our analysis of the most popular security and BYOD apps on Android, we discovered a number of standard ways developers use to determine whether or not a device is rooted. These methods fall into three main categories which we explore in the following section.

3.3.1 Presence of files

The most common checks for root that we encountered during our analysis was the verification of the presence of the "su" binary or supervisor APK. While it is a simple check there are actually multiple methods used that we uncovered in the wild.

Static PATH: On Android devices, binaries are typically located in a few places (e.g., /system/bin/, /system/xbin/, /sbin/). Some root checks simply hardcode these paths (such as, /system/bin/su) and issue an **open** call such as shown in Figure 2.

Dynamic PATH: Assuming that the **su** binary exists in a few hardcoded paths works in many cases, but is easily circumvented by moving the binary, extending the **PATH** variable, etc. We encountered some root checks that parsed the **PATH** variable, appending "/su" to each entry and attempted to open each in a loop. This method is slightly better than checking for static paths.

System PATH: Similar to the previous check, this executes the Linux **which** command with parameter "su" and checks if the result is 0 (indicating the **su** binary was found). This is essentially the same as parsing the **PATH** variable, but requires less work for the caller. However, it relies on the **which** command and the **PATH** variable being untampered with.

Execution: Some methods bypass checking for the **su** binary manually and just attempt to execute it as a subprocess. If the return code is not -1, then the binary is assumed to have been found and executable.

All four variants of this method have the same result; they are typically adequate at discovering the **su** binary on a rooted Android phone. However, we must note how simple this type of check is to evade. Simply renaming the **su** binary to anything else causes all variants to fail completely. Less drastic evasion techniques are moving the **su** binary out of the normal system **PATH** or even altering the **PATH** to exclude wherever **su** is located. In fact, these methods are employed by root hiding apps in the market today, making these specific checks unreliable at best.

Root ACL Program: Most modern methods of rooting utilize a root access management app (described in detail in Section 3.1). Some root checks assume that the APK for this app exists on the system under the path "/system/app/Superuser.apk". While this is the common location of this app, this method can be similarly evaded by moving, renaming or deleting this file.

Setuid: We found one app with an interesting check; the presence of binaries on the system that were **setuid** root, or able to be executed as root (uid 0) by normal users. While standard **su** binaries are setuid root, we are not sure if this is a legitimate check for root as programs could be setuid root for other reasons.

Installed Packages: Some apps check for the presence of common root packages being installed on the system (e.g., "com.chainfire.supersu", "com.noshufou.android.su"). We saw both checks using Android API's as well as by **exec**'ing "pm list packages" and parsing the results from a shell manually. This is another check that is specifically covered by RootCloak to hide the presence of root.

3.3.2 General Device Settings

Some apps we examined also assume certain settings to indicate that a device is rooted. The two most common settings are found in the "build.prop" file:

Test keys: If a custom kernel is used on a device the build version shows that "test-keys" are used instead of "release-keys". Some apps assume "test-keys" means the device is

rooted, which is not always the case. Also, the presence of "release-keys" does not indicate the device is *not* rooted.

Build version: We encountered specific checks of the setting "ro.modversion" as well, which can be used to identify certain custom Android ROMs (such as Cyanogenmod).

Checks related to reading the ".prop" files on Android can also be relatively easily circumvented by using tools such as the Xposed framework, Cydia substrate or even interposing at the library or kernel level. The technique would be to create "non-root" versions of these files (these versions have all the settings a vanilla or stock Android would have) and then intercept the **open** calls to these files and replace them with a handle to the "non-root" versions.

3.3.3 Runtime Capabilities and Characteristics

Some other root checks rely on checking various aspects of the system at runtime or of the process that is being executed.

System mounted: Normally the "/system" partition on an Android device is mounted "ro" (read only). Some rooting methods require this partition to be remounted "rw" (read/write). We saw two variants of this check; the first simply runs the **mount** command and looks for a "rw" flag, the second actually attempts to create a file under "/system/" or "/data/". If the file is successfully created, it implies the mount is "rw".

Ability to mount: A related method we discovered actually attempts to mount the "/system" partition with the command "mount -o remount,rw /system", and then checking if the return code was 0.

User ID: A curious check we found in one case was the app getting the current user id (UID) of the app as it was running and checking if it was running as root (UID 0). This is curious because as far as we know, even on a rooted phone any app started by Zygote (the Android process tasked with actually launching apps) gets it's own unique (non 0) UID. However, it is possible that an app would request root access via intent (Section 3.1) and *then* issue the UID check.

As with the other checks, using the Android system interception tools or library interposition makes it straightforward to evade them. Calls to **mount** or **id** can be intercepted and their results modified to make it appear that (for instance) the "/system" partition is not mounted read/write, and/or arbitrarily change the return code from these processes.

3.4 Security Focused Applications

In this section we present our results for the most popular security focused applications available in the Google Play store at the time of writing. These apps provide multiple functions, but the primary focus for them all is detecting malware installed on the device and evaluating the devices' overall security posture. There are two main reasons that we target security applications in our analysis. The first reason is that security focused applications are in place to warn about security issues including that a device is "rooted" whereas most other categories of applications do not need to differentiate between rooted and non-rooted devices. The motivation for this is that rooting a device potentially opens it up to more damage from malware than non-rooted devices, because all aspects of the system from the lowest level can be accessed and modified. Second, some of the vendors that provide security applications also provide BYOD solutions and we were interested in comparing the strength of the security mechanisms between their different applications to see if the corporate centric apps have stronger detection or protection since they are often used for enforcing compliance to policies.

In our analysis, we shockingly discovered that the majority of the checks found are in the form of the code snippet depicted in Figure 2. These checks correspond to the techniques described in Section 3.3.1.

Table 3.4 presents our detailed results, using the following columns:

- **Company Name** Name of the company that created the app
- **Static su** Checks for **su** binary using hardcoded paths (e.g., /sbin/su)
- **Relative su** Checks for **su** binary using **PATH** search
- **Test Keys** Checks for custom build of Android
- **ACL Prog** Checks for privilege escalation package
- **App List** Searches for known superuser packages installed
- **Mount /system** Tests whether "/system" is mounted **rw**
- **UID 0** Checks if running as root (or can escalate)
- **Total Checks** Count of distinct methods used to check for root
- **App Installs** Google play reported count of number of installations

It is interesting to note that none of these applications leverage native code for the purpose of detecting root access (or, if they do, it is not immediately apparent from static analysis of the included binaries). Instead they rely on Java code, making them vulnerable to known root evasion programs [4, 1]. Also only one mobile security application (provided by Avast) takes advantage of root access to offer advanced features to users (a firewall using iptables).

There are a few notable findings from our investigation using static analysis that we detail. The most interesting root check capability in a mobile security application is present in Kaspersky's "Kaspersky Internet Security". First of all, it has the most comprehensive Java based root check of all the applications we analyzed, including each of the three types of checks documented in Section 3.3. It also uses the "safest" check for the **su** binary, utilizing the **which** command to automatically search all executable paths. But the most interesting part of the Kaspersky root check is that it is intentionally and cleverly hidden from our basic static analysis checks. Specifically, the code used for the root check

```
public static boolean d()
  {
    String[] arrayOfString = new String[3];
    arrayOfString[0] = "/system/bin/su";
    arrayOfString[1] = "/system/xbin/su";
    arrayOfString[2] = "/sbin/su";
    int i = arrayOfString.length;
    int j = 0;
    while (j < i)
    {
      String str = arrayOfString[j];
      if (new File(str).exists())
              return true;
          j += 1;
    }
        return false;
  }
```

Figure 2: Example of the most commonly used code used to assess if a device has root access.

is not directly included, but extracted as a ".jar" file at runtime, which is then dynamically loaded and called via reflection. In order to find the ".jar" and ".dex" to decompile, we actually needed to load the app and copy the files from the application data directory on a phone/emulator at runtime. This finding is also interesting because it implies that some developers at least are taking note of the simplicity of static analysis as well as the ease of circumventing the most common root checks today.

It is likely that in the future application developers will make it more difficult to determine statically exactly how they are checking for root access. Currently, it is relatively simple in almost all cases to decompile an APK to discover how a root check is done, and easily circumvented due to the root check being done using the standard Java/Android APIs. Also, while many of these apps have some level of obfuscation in place to make analysis of the decompiled Smali/Java more difficult, it is not sufficient to hide strings (such as "Superuser.apk" and "/system/xbin/su"), or program flow in a way to actually hide what the code is doing. We believe that more advanced code obfuscation will become the norm, as well as more complex checks for root and other security issues on Android as workarounds continue to be used.

3.5 BYOD Solutions

We next turn our focus to mobile device management (MDM) apps for Android, and how they approach rooted devices. In can be argued that MDM apps should provide stronger checks for root. MDM apps indicate that a device is used for business purposes. In this context, the data and communications on those devices may not belong exclusively to the end-user, but also a company, government, etc. A rooted device provides users and attackers a greater ability to circumvent restrictions placed on apps to secure data and communications, and possibly misuse, leak or steal sensitive information.

Unfortunately, the same methods (covered in Section 3.3) used for security apps are the ones employed by most MDM apps. We collected and analyzed the top Android MDM applications (as of the time of writing) in the same way as for security apps detailed in Section 3.4. In addition to well known MDM application vendors, we also attempted wherever possible to analyze MDM solutions from companies that also have security apps, to see if they exhibit any difference in root detection behavior between the two types of apps. In total we analyzed 19 MDM apps, with 5 from companies that also provide security apps for Android. The results of our analysis are provided in Table 2. We should note that the count of app installs for these MDM solutions are those provided by the Google Play store, and thus may not accurately represent actual install counts, because entities that use them likely distribute the APKs to users directly, or may preinstall MDM on managed devices.

Our analysis led us to a number on interesting findings related to root detection in the enterprise MDM solutions.

Lack of root detection: Based on static analysis alone, we failed to find any of the common root detection code in the MobileIron, Deutsche Telekom and Panda MDM apps. However, we found references in the code to compliance settings for MobileIron. It relates to the Cisco AnyConnect VPN package that includes a root detection check. It is possible that MobileIron is leveraging this code to identify whether or not a device is rooted. The code is apparently checking whether a root only version of the app is installed (com.cisco.anyconnect.vpn.android.rooted), which would not be a very robust check to use. It is also possible that MobileIron's source is sufficiently obfuscated to evade our straightforward root detection checks, though we did not notice much sophistication in the obfuscation used (unencrypted strings and straightforward decompilation). MDM apps require enrollment with a server and it is highly possible that a policy for "root" check gets pushed only once the device is enrolled in the case of MobileIron.

Native code: For VMware's airwatch MDM agent, we found references to a root detection check that is called by a native library. The library **libcoredevice.so** is not particularly difficult to reverse and sheds light on the checks used. The bulk of the checks are in the method **getDeviceState(JNIEnv *, jobject *)** where the presence of the binary **su** is checked for with an **exec** along with a search of the installed packages. Airwatch also retrieves a list of installed applications as part of the root check. In addition to the VMware native code detection, we found that the MDM solution provided by Excitor also leverages native code. In this case, the library used was easily reversed, giving us a clear picture of the detection methods used. It uses three of the well known root checks that we have covered, attempting to find the **su** binary and privilege escalation package in standard (static) locations. From this perspective, the native root check is not much more useful, though it would manage to work around the method RootCloak uses to hide root. These native code checks in some sense make the reversing of root checks harder for attackers.

Breadth: We were impressed by the apparent effort that went into making IBM's MDM solution as rigorous and in-depth as possible. To begin with, none of their checks are static, meaning they search for binaries and packages in an extensible manner. Unlike *nearly all* other solutions we observed, the code searches for binaries other than **su** that could provide escalation, and checks for each in multiple locations dynamically. Furthermore, more than one root ACL

Company Name	Static su	Relative su	Test Keys	ACL Prog	App List	Mount /system	UID 0	Total Checks	App Installs
AVAST		✓				✓		2	100M-500M
Lookout	✓		✓	✓				3	100M-500M
Cleanmaster	✓							1	100M-500M
Qihoo		✓						1	100M-500M
AVG	✓							1	100M-500M
McAfee								0	10M-50M
Norton	✓							1	10M-50M
NQ Labs	✓	✓				✓		3	10M-50M
Kaspersky		✓	✓	✓		✓		4	10M-50M
Trustlook		✓	✓	✓				3	10M-50M
Avira	✓		✓	✓				3	10M-50M
Trend		✓					✓	2	1M-5M
ESET		✓	✓	✓				3	1M-5M
CY Security	✓					✓		2	.5M-1M
Panda								0	.5M-1M
Sophos								0	.1M-.5M

Table 1: Summary comparison of the root checks in place for the top security applications from the Google Play store.

management package is searched for, and in multiple possible places. Finally a list of packages that enable root are searched for, as well as common packages used to hide root. While we were impressed with the number and rigorous nature of the detection, we were luckily unimpressed with the code obfuscation. Once decompiled, we were able to easily reconstruct the source and strings to identify how root checking was done. Obfuscating the source and possibly encrypting strings in the binary would go a long way to foiling simple reverse engineering.

Security/MDM comparison: We looked at the solutions for five vendors that supply both Android security and MDM apps, and discovered a few small surprises. First, one of the five (McAfee) has no checks for root we were able to find in their security offering, but we did find them in their MDM offering. As stated previously, root detection is likely more important for MDM solutions, so this makes some sense. Both Kaspersky and Symantec use the same code across their solutions. However, Kaspersky's code for root detection is hidden from trivial decompilation in the security app, but unobfuscated and easily found in the MDM solution. This indicates either a different build process (meaning they were not intending to hide root detection in the first place) or less concern that the MDM solution would be tampered with. Panda does not seem to provide root detection either for their security or MDM solution.

In summary, we found that most MDM solutions include some checks for rooted devices, as we expected. MDM apps are meant to enforce policy on BYOD devices, and thus it makes sense for them to attempt to detect rooted devices as they pose a security risk to enterprise apps and data. Even so, there were some apps in which we were unable to discover root detection code using static analysis due to use of native libraries and possibly some intentional obfuscation. In order to definitively find out if and how these apps are performing root detection we would need to turn to dynamic analysis of the apps while they are running. Unfortunately, MDM apps require a subscription or connection to a management service provided by the enterprise they are intended to work with. This makes it difficult to run these apps and trigger the full functionality to allow us to see what they are doing.

3.6 Summary

In this section we detailed how an Android device becomes "rooted", and how privileges are currently managed through the use of a supervisor application. We then presented an overview of a custom toolset we used for searching APKs automatically to determine what types of checks they use to determine the rooted status of a device. We then discussed in detail the commonly used checks for root, and how they can be evaded. Next we presented our findings from the use of our toolset to analyze the top security and BYOD/MDM apps in the Google Play store.

Our findings show that most apps use some combination of the different methods of detecting rooted devices, seemingly arbitrarily. Most root detection is done directly from the Java code used which makes up the bulk of most apps, with only a few using native code to do so. Most apps tested also do not obfuscate their code (native or Java), making static analysis and reverse engineering sufficient to discover what they are doing. For nearly all of the apps, common root hiding programs are sufficient to fool the root detection. There are a few recommendations we can make to improve root detection. First, checks should be performed natively to avoid the simplest avoidance methods. Second, code and strings should be obfuscated in code to foil naïve static analysis such as we presented in this paper. Third, as many checks as possible should be implemented. Root detection is used infrequently and the overhead of using one check or ten checks is insignificant. Finally, more advanced checks that are not as easily avoided should be put into place. For instance, listing installed apps as part of a root check or checking **su** apps against known signatures are rarely used, but can prevent against simple evasion techniques such as renaming packages or moving/renaming the **su** binaries.

Company Name	Static su	Relative su	Test Keys	ACL Prog	App List	Mount /system	UID 0	Total Checks	App Installs
MobileIron								0	1M-5M
VMware		✓		✓	✓		✓	4	1M-5M
Kaspersky		✓	✓	✓		✓		4	500k-1M
Citrix	✓		✓	✓				3	100k-500k
IBM	✓	✓	✓	✓	✓			5	100k-500k
SAP	✓	✓		✓				3	100k-500k
McAfee	✓	✓		✓				3	100k-500k
Excitor*	✓		✓	✓				3	50k-100k
AVG	✓							1	10k-50k
Symantec	✓							1	10k-50k
Deutsche TK								0	10k-50k
GLOBO		✓		✓				1	10k-50k
Tangoe	✓			✓		✓		3	10k-50k
Soti	✓		✓					2	10k-50k
Amtel	✓		✓	✓				3	5k-10k
Dell	✓		✓	✓				3	5k-10k
Wavelink								0	5k-10k
Good	✓		✓	✓	✓			4	1k-5k[1]
Panda								0	1k-5k

Table 2: Popular enterprise mobility management apps and those from companies that provide both MDM and security apps.

4. ANDROPOSER

In our research, most of our analysis was based on statically reverse engineering the applications. However we wanted to combine this with dynamic analysis to make sure our findings were correct and observable at runtime. For this, we initially created "AndroPoser". AndroPoser is a library we inject into Android processes leveraging a feature of the dynamic linker that allows us to transparently modify the runtime behavior of selected functions using LD_PRELOAD. This dynamic library interposition allowed us to hook functions and modify the data they manipulate and/or their return code. We realized that this could be used not only as a support tool for our analysis, but also to subdue any native code that checks for evidence of root access.

Implementation:

AndroPoser has been implemented in C and is a little over 300 lines of code. A code sample of the basic functionality for the "open" libc function call is illustrated in Figure 3, and the same basic pattern is used to hook other selected functions. It is important to note that in order to interpose on a function, the interposer code needs to have the exact same method signature. Once the method signature is matched the implementation is straightforward: obtain a pointer to the original symbol, then either invoke the real symbol's implementation and return its original value (as we show in our example), or perform some other operation before returning any arbitrary value. If we were to modify the example to evade root detection in Figure 3, we would check if "open" were called for "/system/bin/su", and instead the injected library would always return -1, indicating to the caller that the file does not exist, whether or not it actually is present. An exhaustive list of hooked functions required to circumvent all checks is omitted, but some oft used examples are libc functions such as "access", "execve", "open", "stat" and "system".

```
int  open(const char*  path, int  mode, ...)
{
  int ret;
  int (*real_open) (const char *, int, ...);
  real_open = dlsym(RTLD_NEXT, "open");
  __android_log_print(ANDROID_LOG_DEBUG,
        "AndroPoser: OPEN", "Path: {%s}", path);
  if (strcmp(path, "/system/bin/su") ==0)
        return -1;
  ret = real_open(path, mode);
  return ret;;
}
```

Figure 3: Example of AndroPoser hooks

On a Linux platform, setting LD_PRELOAD is simple: the command to be executed is simply prefixed with LD_PRELOAD. However, on Android it is slightly different due to the complicated execution process for an application. To set AndroPoser for a given package Android offers a way to do so with the **setprop** command and an example is depicted in Figure 4.

```
setprop wrap.com.package.id
        ''LD_PRELOAD=/data/androposer.so''
```

Figure 4: Setting AndroPoser for a given package

Evaluation: In order to evaluate AndroPoser, we included counter measures for all the root detection mechanisms presented in Section 3. The results are that AndroPoser was able to fool all known methods, presented in Section 3, into reporting that a device is non-rooted while it is. Moreover it succeeds in cases where commonly used root evasion applications failed, including native code checks. Table 3 com-

pares AndroPoser with two common root evasion applications. Since each application takes a different action (or no action) upon discovering a device is rooted, we implemented each of the root checks we encountered in a standalone Android app which simply reports whether root was detected or not. We then tested this app and each of the methods individually with Androposer in order to fill in Table 3.

Root Check Type	Andro-Poser	Root-Cloak	Hide My Root
Static Path	✓	✓	✓
Relative Path	✓	✓	✓
Native Code	✓	NO	NO
ACL Prog	✓	✓	NO
UID	✓	NO	NO

Table 3: AndroPoser Evaluation

5. CONCLUSION

In this paper we analyzed security focused applications as well as BYOD solutions that check for evidence that a device is "rooted". We dissected a sample of the most popular applications currently available for Android (using Google Play) and uncovered the shocking simplicity of most of them. We identified some applications that have been built to evade such checks, and work in most cases due to the simplicity of the checks. We also noted that the blind spot for these root-evasion apps is native code, as the frameworks they are based on do not support interception of native function calls. We then discussed a small proof of concept that can bridge that gap, using library interposition to evade all the checks that we identified for *both* Java and native code. While we strongly believe that "root" identification is crucial, especially for MDM and BYOD to enforce policies on enrolled devices, we demonstrated that even security focused applications fail to do so reliably and that data reported by such applications should not be blindly trusted.

As future work we plan on further improving our automated analysis to include triggering functionality in the Java and native code without needing to run the full app in order to improve our detection results. As part of this, we will extend our automated static analysis to include some automated dynamic analysis as well. Also for future work we would like to investigate the possibility of using of a Trusted Platform Module (TPM) on Android devices to collect reliable, trustworthy information reported from devices about their state.

6. REFERENCES

[1] Hide my root. URL: `https://play.google.com/store/apps/details?id=com.amphoras.hidemyroot`.

[2] Smartphone os market share. URL: `http://www.idc.com/prodserv/smartphone-os-market-share.jsp`.

[3] Trusted execution environment. URL: `https://www.trustonic.com/products-services/trusted-execution-environment/`.

[4] Xposed module repository. URL: `http://repo.xposed.info/module/com.devadvance.rootcloak`.

[5] Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2014.

[6] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '11, pages 3–14. ACM, 2011.

[7] Michael C. Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *MobiSys'12, Ambleside, United Kingdom - June 25 - 29, 2012*, pages 281–294, 2012.

[8] Tsung-Hsuan Ho, Daniel Dean, Xiaohui Gu, and William Enck. Prec: Practical root exploit containment for android devices. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, CODASPY '14, pages 187–198. ACM, 2014.

[9] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in android. In *Computer Security Applications Conference, 2009. ACSAC '09. Annual*, pages 340–349, Dec 2009.

[10] Yeongung Park, C Lee, Chanhee Lee, J Lim, Sangchul Han, Minkyu Park, and Seong-Je Cho. Rgbdroid: a novel response-based approach to android privilege escalation attacks. In *Proceedings of the 5th USENIX conference on Large-Scale Exploits and Emergent Threats, LEET*, volume 12, pages 9–9, 2012.

[11] 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.

[12] Keunwoo Rhee, Dongho Won, Sang-Woon Jang, Sooyoung Chae, and Sangwoo Park. Threat modeling of a mobile device management system for secure smart work. *Electronic Commerce Research*, 13(3):243–256, 2013.

[13] Yuru Shao, Xiapu Luo, and Chenxiong Qian. Rootguard: Protecting rooted android phones. *Computer*, 47(6):32–40, June 2014.

[14] Guillermo Suarez-Tangil, Juan E. Tapiador, Flavio Lombardi, and Roberto Di Pietro. Thwarting obfuscated malware via differential fault analysis. *Computer*, 47(6):24–31, 2014.

[15] Xueqiang Wang, Kun Sun, Yuewu Wang, and Jiwu Jing. Deepdroid: Dynamically enforcing enterprise policy on android devices. In *Proc. 22nd Annual Network and Distributed System Security Symposium (NDSS'15). The Internet Society*, 2015.

[16] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109. IEEE, 2012.

[17] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *NDSS*, 2012.