University of Amsterdam

MSc System and Network Engineering

Research Project 1

Jailbreak/Root Detection Evasion Study on iOS and Android

Authors:

Dana Geist
Dana.Geist@os3.nl

Marat Nigmatullin
Marat.Nigmatullin@os3.nl

Supervisor:
Roel Bierens
rbierens@deloitte.nl

August 23, 2016

Abstract

Compromised devices are a major security issue. If a device is compromised, users might bypass security controls that protect their information. Therefore, many applications attempt to identify these devices. However, detection is not a trivial task, as there are cloaking techniques being developed to evade them. In this paper, we investigated root/jailbreak detection and evasion methods used for Android and iOS, compared them and evaluated their strength. In addition, we made an analysis on the latest trends in the field and studied in depth AppMinder which is an ARM assembly based jailbreak detection tool. In this report, we described how AppMinder variant B works and presented an algorithm and implementation to bypass it. We also showed that assembly level techniques can be used to bypass other types of jailbreak detection. Finally, we assessed Android and iOS current detection state and showed that detection methods are not effective enough.

Contents

Chapter 1

Introduction

1.1 Motivation

Mobile devices are ubiquitous and due to increasing computational power, they are used to perform a large number of tasks for both personal and business purposes [12]. Usage can vary from very simple tasks such as sending emails to managing corporate expense accounts. Those activities usually involve handling sensitive data. Therefore, security and privacy have become important for users as well as for companies providing mobile platforms and applications.

In this context, compromised devices are a major security issue. By "compromised" we refer to both rooted (Android) and jailbroken (iOS) devices. A detailed description about the difference between the two terms can be found in Appendix A.

Compromised devices have been altered in order to provide low-level system access to applications and users that by design were not supposed to. With these type of privileges, users might be able to bypass security controls that vendors or companies put in place to protect information. Moreover, malware may also be able to access and abuse private data. Therefore, many business applications attempt to identify compromised devices, specially in some fields such as finance. Detection mechanisms in these applications can prevent data disclosure by informing users that their device is compromised.

However, there is a race in which companies are trying to protect data through root/jailbreak detection, and cloaking techniques are being developed as their counterpart. Cloaking techniques try to conceal the fact that mobile devices are compromised.

In this context, it is important to investigate root/jailbreak detection methods currently being used, and evaluate their strength. To do so, it is also necessary to research evasion techniques.

As we discuss in the "Related work" section, this is a recent field of research and there is still room for improvement. By conducting this research project, we intend to provide an analysis about the current state of the field, and shed light on different detection techniques and their evasion counterparts.

The rest of the paper is structured as follows: in the reminder of this chapter we describe the research questions in Section 1.2, related work in Section 1.3, scope in Section 1.4 and approach in Section 1.5. Chapter 2 discusses existing detection and evasion methods for Android and iOS. Chapter 3 presents the detection evasion implementation on iOS and the analysis of the results. Chapter 4 shows a summary of the results. Finally, the conclusions are presented in Chapter 5, and future work is discussed in Chapter 6.

1.2 Research Questions

Our work aims to answer the following research questions, in order to bridge this aforementioned knowledge gap:

- RQ1: Which techniques are currently being used in order to detect compromised devices and evade detection in both Android and iOS?
- RQ2: Are there any relevant differences between the techniques used for each of the platforms? Are the root/jailbreak detection methods they present effective?
- RQ3: What are the latest trends and/or most advanced techniques applications use for detection?
- RQ4: Could those latest trends be circumvented? If so, is it possible to create evasion methods and implement them?

1.3 Related Work

Detection/evasion on Android. Evans et al. [29] show in their study that root detection on Android is done directly from the Java code in most applications, with only a few using native code to do so. They state that the blind spot for these root-evasion applications is native code, as the frameworks they are based on, do not support interception of native function calls. In order to bridge that gap, they created a small proof of concept that uses library interposition to evade all the checks that they identified for both Java and C/C++ (native code).

In addition, according to Sun et al. [30] every root detection method discovered during their study can be evaded. The group collected data on the use of various detection techniques, and used it in order to categorize the different methods. They also created an evasion tool called RDAnalyzer which was used to prove the techniques found could be evaded. In their work, they conclude that a reliable root detection method should be provided by the Android operating system. This way, root detection logic could be implemented in the trusted parts of the system, such as integrity-protected kernels or external trusted execution environments.

Detection/evasion on iOS. Jailbreak detection checks are typically implemented in Objective-C or C. These checks can be easily circumvented using publicly known techniques, such as runtime hooking or C function hooking (see [36] and [34]). These methods hook into the applications and inspect the system calls being made. Once these methods identify requests that are usually related to detection, they act as a man-in-the-middle (between the application and the operating system) and change responses to hide certain system characteristics present in jailbroken devices.

To thwart these hooking techniques, NESO Security Labs developed a free prototype for jailbreak detection, based on ARM assembly code called AppMinder [18] [17]. This solution provides defensive protections to be integrated into Enterprise iOS Applications. The protections help detecting whether an attacker has compromised a device, and increase the attackers overall time and complexity required to bypass these checks in an automated fashion .

General detection/evasion ad-hoc methods. There is a considerable amount of ad-hoc work published on the web, that describes different mechanisms (for iOS and Android) that can be used to both detect when a mobile device is compromised [16] [33] [11] and evade detection [5] [14]. The scenarios and methods shown are extensive. For instance, in the field of evasion,

techniques can go from very simple changes such as renaming a file, to implementing complex frameworks that interact with the application in real time and intercept their checks [36] [25].

Our take. It can be seen from the discussed related work that most of the research is focused on Android. The aforementioned publications shed light on the current state of detection and evasion on that particular operating system. The studies agree on the fact that detection methods being developed are not effective against evasion techniques that can be used against them. This is the reason why our practical work is iOS oriented. By studying iOS methods more in depth both platforms can be compared.

In order to accomplish this goal, we decided to analyze AppMinder's method as we considered it to be innovative. Therefore, we wanted to assess whether it could be circumvented. In this research we also wanted to study different evasion methods that can be more effective than the already existing ones. As there is already a lot of effort invested in high level (Java and Objective-C) and in native code (C/C++) evasion solutions, we decided to look into lower level techniques that involve working with ARM assembly.

1.4 Scope

The focus of the project was to research general techniques used by applications in order to detect compromised devices and to evade detection for both Android and iOS. The objective was to understand, categorize and compare them according to different dimensions such as effectiveness and reach.

In addition, we analyzed specific iOS techniques that AppMinder uses to implement advanced detection mechanisms. Studied them, and determined whether these techniques can be circumvented and how.

Finally, we created a proof of concept (PoC) to evade AppMinder's techniques. This PoC consists of two parts. The first one, is a simple iOS application implementing AppMinder. The second one, is an appliance that attaches to that application and implements necessary evasion techniques. It is important to note that the intention of building the PoC is to show how certain advanced techniques such as AppMinder could be evaded. However, by no means should be considered as a general solution.

Discussing how devices can be rooted/jailbroken, and the consequences of doing so, is out of the scope of this paper.

1.5 Approach

The research consists of two parts. First of all, we focused on the theoretical aspect of the existing root/jailbreak detection and evasion methods for the iOS and Android operating systems. This first phase helped us understand the topic and enabled us to find vulnerabilities or improvements on the analyzed mechanisms. It also included comparing the techniques studied between iOS and Android. The approach we took was to start with a general analysis of the state of the field, and narrow down the research to those techniques found more relevant.

For the second part, we focused on the practical aspect which included the analysis of existing advanced detection techniques, specifically AppMinder. Another objective was to relate the findings to the observations performed in the first phase of the research, and draw conclusions on the strength or effectiveness of the studied techniques.

Chapter 2

Studying Android and iOS Detection and Evasion Methods

2.1 Methodology

In order to investigate which techniques are currently being used to detect compromised devices and evade detection (RQ1); and determine if there are any relevant differences between iOS and Android (RQ2); we studied existing detection and evasion methods from various sources. First of all, we focused on the primary literature discussed in the "Related Work" section [30] [29] [18]. Secondly, we looked into existing root/jailbreak evasion frameworks such as RootCloak [25], RootCloak Plus [26] and xCon [36]. In addition, we explored several popular forums such as XDA Developers [9], The iPhone Wiki [33], OWASP [23] and Stack Exchange [10]. Finally, we used the collected information to decide what are the latest trends and/or most advanced techniques applications use for detection (RQ3).

2.2 Taxonomy of Android Root Detection Methods

Presence of files. One of the most common checks involves verifying if certain files such as the "su" binary or the "superuser" [7] APK are present in the system. While it is a simple method, there are actually multiple checks that can be performed.

Packages: Some applications check for the presence of common root packages installed on the system (e.g.,"com.chainfire.supersu", "com.noshufou.android.su"). These checks can be performed using Android API's as well as by executing "pm list packages".

Applications: Most modern methods of rooting utilize a root access management application. This is the reason why some root checks look for different APKs on the system under a path such as "/system/app/Superuser.apk". Some sample applications are:

- SuperSU [6]: An application that is installed along with the su binary for users to regulate root access.
- Rooting applications: Applications that exploit privilege-escalation vulnerabilities to root the device (e.g., One Click Root [27], iRoot [15]).

- Root applications: Applications that require root privileges for their functions, such as BusyBox [28] and SetCPU [13].
- Root cloakers: Applications used for evasion (e.g., Root Cloak, Root Cloak Plus).
- API hooking frameworks: Libraries that provide API hooking functions (e.g., Cydia Substrate [34], Xposed Framework [39]).

PackageManager is an Android Java class for retrieving various kinds of information related to installed application packages. This class is commonly used to check whether a specific application package that requires root privileges is installed on the device. For instance, getInstalledPackages and getInstalledApplications return a list of all packages or applications that are installed on the device. Other commonly used APIs are getPackageInfo and getApplicationInfo which retrieve similar information.

Files: On Android devices, binaries are typically located in a few known paths (e.g., /system/bin/, /system/xbin/,/sbin/). Some root checks simply hard-code these paths and issue an open system call. This technique works in many cases, but is easily circumvented by moving the binary. It can be done by extending the PATH variable. This is the reason why some root checks parse the PATH variable, appending "/su" to each entry and attempt to open each of them.

The File.exists Java method and the C API can be used to check the existence of a file within the file system.

Build settings. Some applications use certain build 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 applications 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. The following code is used by applications to test this condition: *if (!android.os.Build.TAGS.equals("release-keys"))*.

Build version: There are some specific checks that look into the setting "ro.modversion", which can be used to identify certain custom Android ROMs (such as Cyanogenmod [32]).

File permissions. Some rooting tools make certain root folders readable (e.g.,/data) or writable (e.g., /etc, /system/xbin, /system, /proc,/vendor/bin) to any process on the device. The File.canRead and File.canWrite Java APIs, or access C API can be used to check whether such condition exists.

Shell commands execution. Shell commands are commonly used to detect rooting traits. Applications can use Runtime.exec Java API, ProcessBuilder Java class, or execve C API to execute a specific command in a separate process, and then examine the output of the shell command to detect rooting traits. Commonly employed shell commands are:

su : If the su binary exists, this shell command will exit without error; otherwise, an IOException will be thrown to the calling application.

which su : This check involves executing the Linux "which" command with parameter "su" and verifying 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.

id : This check is based on executing the command "id", and verifying the UID to determine if it corresponds to root (uid=0 is root).

Runtime characteristics. Other root checks rely on verifying various aspects of the system at runtime.

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). There are mainly two variants of this check. The first simply runs the mount command and looks for a "rw" flag. The second attempts to create a file under "/system/" or "/data/". If the file is successfully created, it implies the mount is "rw".

Ability to mount: This method attempts to mount the "/system" partition with the command "mount -o remount,rw /system", and then checks if the return code was 0.

Check Processes/Services/Tasks: This check consists of finding whether a specific application that requires root privileges is running on the device. The ActivityManager.getRunningAppProcesses method returns a list of currently running application processes. Similarly, getRunningServices or getRecentTasks APIs are also used by applications to retrieve a list of current running services or tasks.

System properties. If the value of system property ro.secure equals "0", the adb shell will be running as root instead of shell user. System.getProperty Java API can be used by applications to examine this property value. In addition, examining ADB source code reveals that the adbd daemon is also running as root user if both "ro.debuggable" and "service.adb.root" properties are set to "1".

2.3 Taxonomy of iOS Jailbreak Detection Methods

Presence of files. Applications check the device's file system in order to find files that are usually present in jailbroken devices. For instance, they can check for paths like /Applications/Cydia.app/, /Library/MobileSubstrate/, /bin/bash, /usr/sbin/sshd, /etc/apt and /private/var/stash. Most often, these are checked using the *-(BOOL)fileExistsAtPath:(NSString*) path* method in NSFileManager. However, there are also applications that use lower-level C functions like fopen(), stat(), or access().

File permissions. This method consists of checking the permissions of specific files and directories on the system. More directories have write access on a jailbroken device than on a non compromised one. One example of this is the root partition, which originally has only read permission. If it is found it has read and write permissions it means the device is jailbroken. There are different ways of performing these checks such as using NSFileManager and C functions like statfs().

Process forking. Sandboxd does not deny applications the ability to use fork(), popen(), or any other C functions to create child processes on jailbroken devices. However, sandboxd explicitly

denies process forking on non jailbroken devices. Therefore, by checking the returned pid on fork(), an application can tell if a device is compromised.

SSH loopback connections. Due to the very large portion of jailbroken devices that have OpenSSH installed, some applications attempt to make a connection to 127.0.0.1 on port 22. If the connection succeeds, it means OpenSSH is installed and running, which proves the device is jailbroken.

Executing privileged actions. Calling the system() function with a NULL argument on a non jailbroken device will return "0"; doing the same on a jailbroken device will return "1". This is since the function will check whether /bin/sh can be accessed, and this is only the case on jailbroken devices. Another possibility would be trying to write into a location outside the application's sandbox. This can be done by having the application attempt to create a file in, for example, the /private directory. If the file is successfully created, it means the device is jailbroken.

Calling dyld functions. This method consists of checking whether certain dynamic libraries exist. The dynamic loader [22], dyld, is a shared library that programs use to gain access to other shared libraries. Functions like _dyld_image_count() and _dyld_get_image_name() can be used to determine which dylibs are currently loaded.

System properties. This method involves checking for different operating system characteristics that are usually changed when a device is jailbroken. Many jailbreaking tools modify file /etc/fstab (which contains mount points for the system) by adding entries to it, changing its file size. Therefore, if the size of this file is modified it means the device is jailbroken. In addition, some directories are originally located in the small system partition, but this partition is overwritten during the jailbreak process. The data must be relocated to the larger data partition. Because the old file location must remain valid, symbolic links are created such as Library/Ringtones, /Library/Wallpaper, /usr/arm-apple-darwin9, /usr/include and /usr/libexec.

AppMinder Jailbreak Detection. AppMinder is a prototype jailbreak detection tool for the Apple iOS platform based on ARM assembly code. It is mainly designed and developed for Enterprise iOS Applications and implements the fork check mentioned before.

2.4 Root/Jailbreak Evasion Methods

2.4.1 Basic methods

Hiding su binary (Android): Many detection methods are adequate at discovering the su binary on a rooted Android device as mentioned before. However, this check can be evaded by simply renaming the su binary. 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.

Runtime checks (Android): This evasion method consists of intercepting certain calls that are commonly use for detection, and change their behavior. It can be done by using Android system interception tools or library interposition. This way, calls to "mount" and "id" can be intercepted and their results modified. In the case of "mount" the evasion method would make

it appear that (for instance) the "/system" partition is not mounted read/write but read only. In the case of "id", the evasion method would arbitrarily change the return code.

Modify source code (Android): One of the methods that could be implemented is modifying the source code. This would consist of decompiling the application, locating the root detection controls, modifying them and recompile (this could also involve re-signing). The changes that can be made vary from commenting out the controls to altering result values of certain functions.

Binary patching (iOS/Android): Formally a patch in computer science is known to be a piece of software designed to update a computer program or its supporting data, to fix or improve it [38]. In this paper by "patch" we refer to a series of instructions that describe how to modify a file or a set of files. In this context, patching means the binary file is going to be modified permanently to include the necessary changes. The first step of the process consists of locating the instructions that want to be bypassed in the binary. Then, a method for changing the instructions needs to be chosen (for instance replacing some instructions with a no operation). The final step is to modify the binary accordingly and place it back on the phone.

Disable root/jailbreak privileges temporary (iOS/Android): This method consists of disabling root/jailbreak privileges. However, doing so prevents all applications from having low level permissions, even the ones that need those to function properly. In addition, applications such as SuperSu cannot hide themselves so they can be detected more easily.

2.4.2 Frameworks

RootCloak (Android): RootCloak is a module of the Xposed Framework. It allows to run applications that detect root in compromised devices without being detected. It performs different techniques such as hiding the su binary, superuser/supersu APKs or processes run by root. In order to do so, it hooks into basic SDK [3] level calls, such as exec() and getInstalledApplications (). As Xposed framework does not operate on a native level, RootCloak can not intercept native calls (Android NDK [2]) such as access() and fopen().

RootCloak Plus (Android): RootCloak Plus is based on the same idea as RootCloak, but it uses Cydia Substrate instead of the Xposed Framework. Cydia has deeper integration within Android via native calls. Therefore, it has more reach and can successfully bypass root detection on more applications than RootCloak.

xCon (iOS): xCon is a collaborative project that aims to be an evasion solution. It hooks known methods and functions responsible for informing an application of a jailbroken device, and changes their behavior.

2.5 Android vs. iOS: Method Comparison

The analysis presented shows there is no substantial difference between the methods used by Android and iOS. Most detection methods available try to find certain characteristics in the system that indicate the device is compromised. There is a wide range of possibilities that include finding files, checking for directory permissions or even executing privileged commands. Even though some detection methods are unique for each operating system (such as process forking for iOS and executing su binary for Android), they are based on the same idea; looking

for non-standard features. The evasion methods, in turn, are even more similar. They identify those functions and system calls that are used for detection, and hook into them so as to change their behavior. Both platforms present standard ways of implementing evasion through solutions such as RootCloak Plus or xCon.

2.5.1 Detection Methods Comparison

This section is intended to compare the detection methods shown. In order to do so, we defined different dimensions:

Level of abstraction: There are three main abstraction levels in which root/jailbreak detection mechanisms could be programmed: Java/ObjectiveC, C/C++ and ARM Assembly. Detection methods are easier to evade if they are placed in upper levels of abstraction. This is due to the fact that it is easier to hook on high level language methods. If controls are implemented in assembly level, checks get more difficult to bypass as the common hooking techniques can not be used.

Types of checks: There are several detection methods that look for certain system characteristics. We can categorize them depending on where they look for anomalies or non-standard behavior. Some of them look into operating system features such as directory permissions or presence of symbolic links. Others, look for the existence of packages, applications or ports. Some of them even check for dynamic libraries.

2.5.2 Evasion Methods Comparison

The shown evasion methods can be categorized according to different dimensions:

Level of complexity: There are several evasion techniques that can be implemented which vary from very simple tricks to using comprehensive frameworks. In this context, we have the following categorization:

Simple techniques: These are usually focused on one aspect of the detection and try to implement measures to evade it. Some examples could be renaming files (such as superuser.apk) or changing binary's behavior (let su binary do nothing and hide the real file on another location in the system). Even though these mechanisms are simple, they are flexible as they can be combined with each other to build a more robust evasion system.

Use of applications: Some evasion methods are based on using applications like SuperUser to hide certain system characteristics. One example of what these applications could do is to block the "su" command in order to hide its presence. The advantage of such an application is that it is easy to install and use. However, it has a major drawback: some detection techniques check for the existence of the application itself. This situation could be hard to conceal.

Use of frameworks: Different tools such as RootCloak, xCon and RootCloak Plus can be used to implement evasion. As discussed earlier, these frameworks provide methods for bypassing different detection checks. In addition, they require little configuration. However, they don't provide support for every existing application. This means that even though they are more comprehensive than the previously mentioned mechanisms, they can not be considered to be a general solution.

Static vs. dynamic: The difference between static and dynamic methods is that the static focuses on solving the issue before the application runs. In contrast, the dynamic handles the problem while the application is executing.

Static: This group includes techniques that rename files, change executables to behave differently, and even binary patching mechanisms. The disadvantage is that these are less flexible as everything needs to be prepared before the application runs. The advantage is that in general they are faster, as nothing needs to be performed during execution.

Dynamic: This group is mainly formed by those applications and frameworks that can be installed on mobile devices for evasion. They run in the "background" and when the application calls a function for root/jailbreak detection they act on it. The main advantage is that they are more comprehensive, as they can capture several controls. However, they intercept certain methods and/or system calls and change their behavior on the fly, which could slow down execution.

2.6 Latest trends

Previous research showed that most applications implement detection checks using high level programming languages such as Java (Android) and Objective-C (iOS). Evan et al. [29] mention that most of the applications they studied did not leverage native code (C/C++) for the purpose of detecting root access. Instead, they rely on Java code, making them vulnerable to known root evasion programs. As their counterpart, evasion techniques capable of hooking into those high level functions were implemented to block them (for instance RootCloak). Despite the fact that most of the controls are found in high level languages, we observed there is a trend in which some applications are trying to implement controls in native language so as to make their detection methods more effective. Sun et al. [30] found four out of thirty applications that perform rooting detection in native code, instead of Java. Even though those are more difficult to bypass, several root/jailbreak evasion techniques provide capabilities of hooking into native functions (such as RootCloak Plus and xCon).

At this point, in which the situation seemed to be beneficial for the evasion developers, NESO Security Labs created AppMinder as a jailbreak detection solution. This technique goes one step further. Besides of being implemented in assembly, it obfuscates the code and supports some level of polymorphism. It changes the rules in the jailbreak detection/evasion game. If an application implements this mechanism, it is not possible to use traditional hooking methods for evasion. As this is one of the most advanced and innovative jailbreak detection methods, we were interested in studying it and determine if it can be circumvented.

Chapter 3

Implementing iOS detection evasion

3.1 Methodology

In the previous chapter we analyzed different detection methods, and found AppMinder to be one of the most advanced and innovative. Therefore, we studied the solution and analyzed whether it could be bypassed and how (RQ4). In addition, we investigated high level jailbreak detection methods within Cordova Plugin [19] and created a strategy based on ARM assembly to bypass them.

In order to accomplish this goal we followed these steps:

1. Create an iOS application.
2. Integrate jailbreak detection code (AppMinder and Cordova Plugin) into the application.
3. Inspect application's behavior.
4. Perform static analysis by looking at the assembly code structure and functions implemented.
5. Perform dynamic analysis by inspecting memory, register values and execution flow.
6. Identify patterns.
7. Design a strategy to bypass detection techniques.
8. Implement design.

3.2 Materials

The tools used are the following:

- Macbook laptop (version 10.11.3 El Capitan) and XCode IDE version 7.2 [4] were used to develop and deploy the testing application on the mobile device.
- AppMinder code.

- GNU Debugger (gdb) version 6.50.3 [24].
- Jailbroken iPhone 5 with iOS 8.1.2 (OpenSSH and gdb installed).
- To establish a SSH connection via USB cable between the laptop and the iPhone we used libimobiledevice-1.2.0 and usbmuxd-1.1.0 libraries. For the libraries to work properly, packages libusb, libusbmuxd, libplist, openssl and python-dev should be installed on the laptop prior to the compilation of the libraries.

3.3 AppMinder

3.3.1 What is it?

AppMinder is a prototype jailbreak detection tool for the Apple iOS platform based on ARM assembly code. The protections provided by AppMinder help to detect whether an attacker has compromised a device, and increase the attacker's overall time and complexity required to bypass these checks in an automated fashion [18].

3.3.2 How does it work?

To use AppMinder in an iOS application, it is necessary to obtain its code from the NESO Security Labs' web page [18]. After following the steps outlined, one will be provided with a jailbreak detection code, as well as with instructions on how to integrate it into the application. Sample code can be found in Appendix B.

The code consists basically of 5 functions which will be called inside of the jailbreak detection method in the application. All functions contain ARM assembly code intended to check whether the device is jailbroken. If the code detects the device is compromised, the application is aborted.

3.3.3 Why is it difficult to bypass?

There are different reasons that make AppMinder difficult to evade. Some of them are listed below:

- All AppMinder jailbreak detection checks are implemented in ARM assembly code in order to thwart common hooking techniques. This means no traditional methods or frameworks can be used to bypass it.
- Every time a new code is generated, the assembly instructions vary (polymorphism) so it is not possible to look for specific static sequences of code.
- Function names and parameters are obfuscated in order to make it more difficult to read.
- AppMinder includes self integrity checks which make static binary modification not easy to achieve.
- Assembly code is added "inline" which means functions will not have a specific location and label within the compiled assembly code. Instead, they will be unrolled every time they are used. Therefore, it is more challenging to track them and identify where and when they are being called.

3.3.4 What are the limitations?

AppMinder Jailbreak Detection tool presents some limitations:

- Only one criterion is used to determine whether the device is compromised (the return value of the *fork* system call is evaluated).
- Only one tamper-response mechanism is available (if a jailbreak is detected, the application is terminated).
- Only a subset of the possible anti-debugging measures, self-integrity checks and code obfuscation techniques has been implemented so far.
- AppMinder only works for iOS (checking for the result of the *fork* system call is not a traditional method used for detection of rooted Android devices).

3.4 AppMinder Analysis & Bypassing

3.4.1 What technique should be used to tackle AppMinder?

There are a number of different ways to manipulate an iOS application. The most popular and expedient method involves using tools such as Cycript [8] to manipulate the Objective-C runtime. This enables one to hijack the program flow of the application, create new objects and invoke methods within an application. However, these type of tools only work well at a high level, as they can hook into Objective-C/C/C++ functions [31]. Therefore, they are not suitable to evade AppMinder as it lies at assembly level.

Another technique that can be used is binary patching, which we described in Section 2.4. Even though it is a valid and effective technique, it can be difficult to implement if the application performs integrity checks. It also requires calculating the right offset and modifying data displayed in hexadecimal/binary format. This process could be cumbersome, specially if there are several changes to be implemented.

A different approach could be to use debugging tools such as GNU Debugger (gdb). GNU Debugger offers extensive facilities for tracing and altering the execution of applications [37]. It allows one to attach to an application, and not only be able to examine the high level language code, but to disassemble it. GNU Debugger also makes it possible to observe the low level assembly instructions, set breakpoints, manipulate the values in the registers and hence change the application flow. This method was used to inspect the application and bypass its detection mechanisms.

3.4.2 Static analysis

The purpose of what we call "static analysis" is to take a look at the disassembled application and gather information about the program's structure and organization. The goal is to find a pattern in the way AppMinder performs its checks.

Code comparison

The first step in this process, was to compare the disassembled code obtained with gdb, with the original code provided by AppMinder. Based on the observations we were able to determine that:

- Both SWI and SVC instructions in AppMinder's code are translated into SVC instructions in the real disassembled code. SVC instructions are supervisor calls used to change the execution mode into a protected mode for the operating system [20]. SVCs are usually used to instruct the operating system to execute a privileged system call such as fork, exit, read or write.
- Inline addition of AppMinder assembly code into the Objective-C code complicates analysis as the instructions of each function are unrolled instead of being called like a regular function (which would be executed as a branch to a certain address/label).
- Even though there are minor differences between the two codes, the majority of the instructions in AppMinder can be found in the compiled assembly code. This helped us to map each of the functions and determine the differences between them.

Fork and exit patterns

AppMinder checks for the result of the fork system call, and depending on it decides whether to abort the application. If the call is successful the application crashes as it assumes the device is jailbroken, otherwise it will continue with its execution. Based on that knowledge we identified two ways of bypassing the detection. The first one is to address the fork execution and act on it to pretend the device is not jailbroken. The second one is to handle the exit part so as to avoid application termination.

The possibilities considered to bypass the fork check were:

- Avoid execution: replace fork by a no operation (nop) or jump to the next instruction.
- Alter return value: let the fork execute but alter the return value to be the one AppMinder code is expecting.

When the exit is executed the program terminates immediately. Therefore, in this case the only option left was to try to skip it.

Once the objectives were defined, the code was analyzed to find traces of fork or exit executions. In order to aid this task we decided to create a simple C program containing the fork() system call and cross compile it to ARM assembly. The goal was to compare it with the AppMinder code to find the fork instruction pattern. This method did not work as the pattern could no be easily recognized. Appendix C shows fork code (C and assembly).

According to [21] AppMinder executes the fork system call via SVC 0x80 instruction. The code places an integer value corresponding to the fork system call in register r12, and then issues the SVC instruction for the operating system to execute it. However, this description was not sufficient to find fork patterns as there was some missing information. First of all, we found supervisor calls can be executed by different instructions such as SVC 128 (decimal version of SVC 0x80), SVC 17 and SVCNE 128. Secondly, r12 needs to "2" in order to execute fork and "1" to execute exit. This value depends on the operating system [35].

Focusing on register R12 and SVCs

To detect fork and exit operations we monitored the value of register r12.

For fork we found that AppMinder uses different ways of setting r12 to the value of "2". Sample codes are shown below:

Assign "2" to r12 directly:

```
mov r12, #2;
add r0, r0, #62;
b L273165;
cmp r0, r10;
L273165:;
mov r4, pc;
ldr r4, [r4, #0];
svc 0x80;
```

Assign to r12 a value different than "2" and the operate on it:

```
mov r12, #256;
asr r12, #7;
svc 0x80;

mov r12, #64;
mov r3, r3;
asr r12, #5;
mov r4, pc;
ldr r4, [r4, #0];
svc 0x80;

mov r12, #4;
sub r12, r12, #2;
mov r0, r0;
add r0, r0, #19;
mov r2, r2;
mov r4, pc;
ldr r4, [r4, #0];
svc 0x80;
```

Assign "2" to a different arbitrary register and then assign that register value to r12:

```
mov r1, #2;
b L505572;
stmdb sp!, {r0-r12};
L505572:;
mov r12, r1;
svc 0x80;
```

As the sequence of instructions varies, it is difficult to find a static pattern that determines when a fork is executed. This is not only because AppMinder assigns value "2" to register r12 in different ways, but because it also interleaves different assembly instructions between the assignment and the SVC call.

For exit, the pattern is simpler:

```
mov r12, #1;
swi 0x11;

mov r12, #1;
svc 0x80;
```

At this point, we determined that static analysis was not enough to find patterns, as there are many values determined at execution time that need to be considered. This is the reason why we also performed a dynamic analysis described in Section 3.4.3.

3.4.3 Dynamic analysis

The strategy we used in order to conduct the dynamic analysis, was to place breakpoints at instructions we considered important such as SVC calls, so we could analyze the state of the system before and after they were executed. Step by step execution was also used in specific cases.

Exit system call

Based on the static analysis results, handling the exit part seemed more promising than addressing the fork part. Therefore, we decided to focus on it first. However after several executions of the program we saw the following:

- AppMinder's code does not only use the r12/SVC technique we described before to exit the application. It also implements this behavior by generating deliberate invalid memory accesses or bad signals. This makes program termination difficult to detect.
- The code is designed to follow a correct pre-defined execution path. If it detects the mobile device is jailbroken it will prepare to crash, independently of the exit call being bypassed.

Given the evidence collected during the experiment, we ruled out further analysis on the exit and focused on the fork.

Fork system call

As mentioned earlier, we put different breakpoints at the SVC instructions and analyzed the behavior of the program. We defined the pre and post conditions that held for the fork call execution. As per the pre conditions r12 needed to be "2" before the SVC execution. The post conditions were determined empirically. We found that after the execution of the fork register r0 was modified. Experiments showed that r0 contained child's process ID (PID) when executed in a jailbroken device and "1" otherwise.

This allowed us to determine the values expected by the application and rule out the "avoid execution" strategy mentioned before. The choice made was to implement the "alter the return value" strategy. This one consisted of changing the value of r0 to be "1" after fork execution.

3.4.4 Bypassing AppMinder: putting strategy to a test

To implement the bypassing strategy described in the previous section we used gdb debugging tool. The system architecture is shown in Figure 3.1.

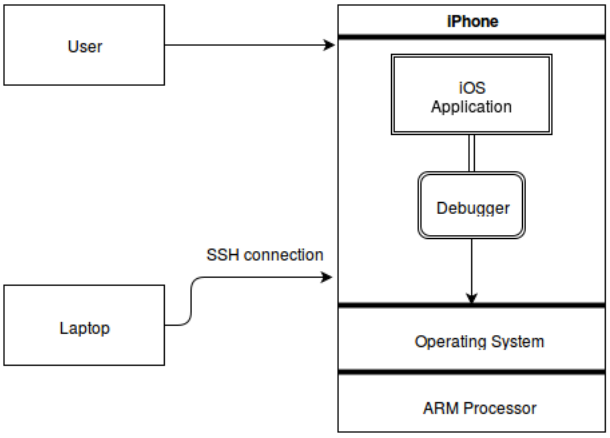

Figure 3.1: System Architecture

The bypassing strategy consisted of attaching gdb to the application and performing the required fixes dynamically. Figure 3.2 illustrates the interactions between the different system components (including a supervisor call execution).

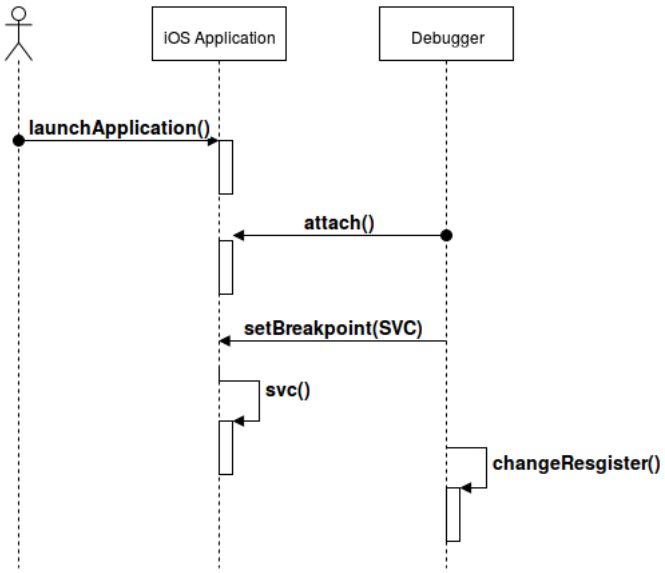

Figure 3.2: Interaction Diagram

First approach: interactive

In this approach we show how to bypass AppMinder by inspecting the ARM assembly code statically and executing the sequence of gdb commands to change the flow of the application manually. For that we followed these steps:

- Start jailbreak detection application manually.
- Start gdb on the mobile device and attach it to the running application.
- Put breakpoints at all SVCs and their consecutive instructions.
- When a SVC breakpoint is reached, check the value of register r12 to make sure that it is a fork system call (otherwise it should be skipped as we don't want to alter any other system call outcome).
- If it is a fork, execute the SVC instruction and continue to the consecutive one.
- Before executing the consecutive instruction set register r0 to be "1" and continue.

We found AppMinder's 5th function has a different functionality, so the assembly code differs from the other functions. Therefore, it needed to be analyzed separately. To bypass it, we targeted specific compare instructions, and altered their input register values.

After all fork and specific compares were addressed, AppMinder was bypassed using the above algorithm. It is important to note that if the program goes through the correct execution path, no exit instructions are reached.

Second approach: automating gdb steps

The objective of the second approach we proposed was to create a script that executes all gdb commands required for evasion. This strategy still requires some manual work (look at the disassembled code and set the breakpoints), but the bypassing is performed faster as all it takes is to run the gdb script (instead of typing gdb commands interactively).

Third approach: semi-automatic solution

We wanted to explore the possibility of fully automating the proposed algorithm, at least for the first 4 function calls which follow the exact same pattern (r12=2 and SVC call). The idea would be to examine every assembly instruction, and detect when the fork system call is executed in order to act accordingly. Even though this solution seems reasonable at first glance, it places a great burden on the application as each assembly instruction needs to be examined and compared. Efficiency could be drastically affected by this form of automation, specially for large scale applications that contain millions of assembly instructions. This is the reason why we decided to create a method that could combine manual and automated work in order to get the best from both worlds. Sacrifice a complete automated solution in order to gain performance.

The method we created consists of a parser that takes as an input the assembly lines for the functions implementing AppMinder's detection, and outputs a gdb script that can be used to bypass it.

The user needs to identify all functions in the application that implement jailbreak detection, disassemble them and put those in a file. Once the input file is ready, the parser needs to be run so as to generate the final gdb script. The output will be used in oder to automate the evasion. Figure 3.3 illustrates this flow.

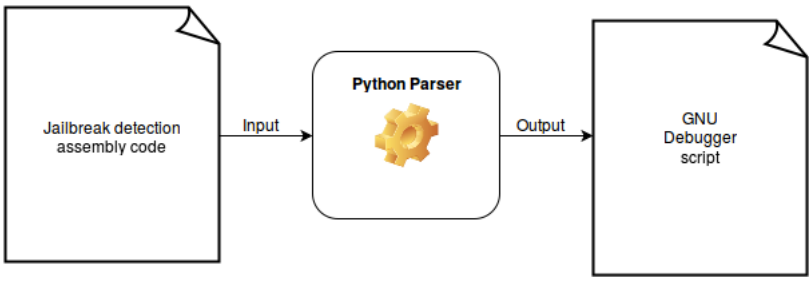

Figure 3.3: Semi-automatic solution

Anti-debugging measures

AppMinder also gives the possibility of including anti-debugging measures. It is important to take those measures into consideration as applications could (and should) implement them. Many iOS applications use anti-debugging techniques to prevent malicious users from using a debugger to analyze or modify their behavior [1].

A common method of implementing anti-debugging measures is using ptrace. This is a system call that is primarily used to trace and debug applications. It is defined as follows:

```
int ptrace(int request, pid_t pid, caddr_t addr, int data);
```

The first argument (request) specifies the operation to perform. One of the operations is called PT_DENY_ATTACH and has the value of "31". When the request is set to that value all other arguments are not used and set to zero. The application then, informs the operating system that it doesn't want to be traced or debugged. Any attempts to trace the process will be denied and the application will receive a segmentation violation.

In order to evade anti-debugging measures, we needed to locate ptrace executions. From previous analysis it is known that AppMinder implements system calls by setting register r12 to a specific value and then executing a SVC instruction. According to the iOS system call list mentioned earlier [35], ptrace value is "26".

Ptrace execution should be detected and avoided; otherwise the program exits. This can be done by replacing ptrace system call by another one such as fork. If ptrace was detected, value of register r12 was modified to be "2" (so fork was executed instead of ptrace), and the value of register r0 was altered to be "1".

We improved our parser so it could also bypass anti-debugging measures using the mentioned strategy. The code can be found in Appendix D.

3.5 Exploring alternative jailbreak detection methods

This section describes how we bypassed jailbreak detection techniques within Cordova Plugin. The main detection function could be found in Appendix E.

The code implements known detection methods mentioned in Section 2.3, such as checking for the existence of certain directories, files or packages.

The code was added to our iOS application and once integrated, its corresponding assembly was analyzed. To accomplish evasion, we focused on the "if" statements. Those are generally mapped to the "cmp" instruction in assembly. To bypass them we made the comparison equal or unequal depending on the branch instruction that followed (branch equal or branch not equal) in

the code sequence . In Appendix F we show the python script we used to generate a bypassing gdb script. Table 3.1 shows the comparison of the first "if" statement between ARM assembly and its Objective-C equivalent.

Assembly code	**Objective-C**
call to OS to check the existence of the file	[[NSFileManager defaultManager] fileExistsAtPath:@"/Applications/Cydia.app"]
ldr r0, [sp, #164] //load return value of the call sxtb r1, r0 // convert value to 32 bit cmp r1, #0 //compare return value with 0 [0 means file does not exist]	if file /Applications/Cydia.app exists
movs r0, #1 // load the return value of the method into r0 [YES] strb.w r0, [r7, #-5] //store return value in memory b.n *address // branch to the end of the method	then return YES
beq.n *address // branch to the next check	else { next check }

Table 3.1: Code Comparison

3.6 Limitations

There are certain limitations on the methods and tools we provided:

- Our work was focused on studying AppMinder's variant B.
- The bypassing algorithm is expected to work properly in any application for the first four AppMinder function calls. However, as the fifth function call exhibits different behavior it needs to be analyzed on a case by case basis.

3.7 Ethical considerations

All methods and tools showed in this research project were studied and developed for academic purposes. The objective was to evaluate how strong advanced detection techniques on iOS are, and investigate whether it is possible to bypass those. Jailbreak checks that applications implement are intended to protect users and companies from malicious attacks. Therefore, it is not recommended to use the provided bypassing methods. Detection methods should only be bypassed under special circumstances such as conducting a penetration test on an application under the owner's consent.

Chapter 4

Results

4.1 Assembly Bypassing

4.1.1 Results

As shown in Chapter 3 we created a method in order to evade AppMinder variant B. We also implemented a solution that can help with automating its evasion to certain extent. In addition, we showed assembly mechanisms can be used to evade different techniques such as the ones implemented by Cordova plugin.

If Android applications were to implement assembly detection techniques, evasion mechanisms similar to the ones we presented could be used. The reason for that is that the solutions we provided are based on ARM assembly, which both platforms use.

4.1.2 Method Comparison

This section is dedicated to compare our bypassing method with the existing ones according to different dimensions, as per Table 4.1. The objective of this comparison is to show the advantages and disadvantages exhibited.

	RootCloak/[29]/ [30]	**RootCloak Plus/ xCon** /[29]/[30]	**Our Method**
Technology	Java	C/C++	ARM assembly
Abstraction Level	High level	Native Level	Low Level
Supported OS	Android	iOS(xCon)/ Android(rest)	iOS
Exception tracking	Yes	Yes	No
Evasion potential	Java methods	Java/Objective-C/C/C++	All
Automated	Yes	Yes	Partially for AppMinder

Table 4.1: Method Comparison Table

The main advantage of our method is that it has the potential to evade different root/jailbreak detection techniques. This is due to the fact that lower level of abstraction gives more control

over the application. However, there is a trade off that needs to be considered. The method is more comprehensive but more complicated and less flexible. Whereas dynamic frameworks such as xCon and RootCloak Plus can handle a variety of applications in an automatic way, our method requires a case by case study.

4.2 Root and jailbreak detection state comparison

As we discussed before, it has been shown in several studies [30] [29] that all existing Android root detection methods can be bypassed.

Throughout our research, we found there are several iOS bypassing techniques. Most of them are focused on bypassing controls implemented in high and native languages. In our study we showed a new method for bypassing jailbreak detection techniques based on ARM assembly.

Our results suggest that even though Android and iOS are different operating systems with distinctive characteristics, the state of root/jailbreak detection they currently have is very similar. Both platforms present detection techniques which are not effective enough as they can by evaded.

Chapter 5

Conclusions

In this research we studied a variety of root/jailbreak detection and evasion techniques for both Android and iOS. We categorized and compared them based on different aspects such as functionality, level of abstraction and complexity. Our conclusion is that despite some minor differences, detection and evasion methods for both operating systems are very similar.

We also analyzed which were the latest trends in detection. Throughout that analysis we were able to determine many applications were starting to implement low level detection techniques, so as to be more effective. Among the latest trends available for iOS, one interesting exponent is the solution created by NESO Security Labs called AppMinder. We consider it to be innovative as it is assembly based. This characteristic makes the solution stand out from the rest that are usually implemented either in high level (Objective-C) or native level (C/C++) languages.

In this report we explained how AppMinder variant B works, and showed an algorithm to evade its checks. In addition, we presented a proof of concept implementing this algorithm. Moreover, we showed that assembly evasion techniques could also be used to bypass high level detection methods implemented in Objective-C.

Finally, we compared Android and iOS root/jailbreak detection state. Even though they are different operating systems with distinctive characteristics, our observations suggest that the state of root/jailbreak detection the exhibit is comparable. The detection methods they present are not effective enough. However, there are certain techniques such as AppMinder, which are more difficult to evade.

Chapter 6

Future Work

As part of future work, we would like to address the limitations of our current study. First of all, it would be important to study AppMinder variant A, and analyze the possibility of circumventing it as well. This would allow us to evaluate AppMinder as a whole with both variants in place. In this context, it would also be interesting to analyze in depth the possibility of implementing an efficient fully automated bypassing solution to evade AppMinder.

Secondly, we would like to study different detection mechanisms (high/native levels) for both Android and iOS and try to find patterns so as to be able to automate their evasion in assembly.

Chapter 7

Acknowledgments

We would like to thank Roel Bierens (Deloitte) and Cedric van Bockhaven (Deloitte) for their help and input throughout the research project. In addition, we would like to thank our SNE shepherds Junaid Chaudhry and Konstantin Urysov for their guidance. Finally, we would like to give a special thank to our reviewers Romke van Dijk and Loek Sangers for their helpful feedback, which allowed us to improve our report.

Chapter 8

Responsible Disclosure

The findings of this research were communicated to the developers of the AppMinder tool, by means of a responsible disclosure procedure carried out by the OS3 Ethical Committee. Below there is a discussion of what we consider to be the main points that were brought up by the tool developers.

The developers said that variant A and variant B of AppMinder should be considered complementary instead of mutually exclusive. In this context, they argue that both variants should be included as each of them implements different mechanisms that are meant to thwart different attack strategies. They expressed that 'variant A implements various different countermeasures, including but not limited to breakpoint detection, that were designed to precisely prevent those debugging attacks as outlined in the report'.

When analysing AppMinder, we found that the tool provided two different variants. It was recommended to use at least one variant A and one variant B in the code but there was no hard requirement enforcing that, so we decided to pick variant B as we only had a limited amount of time of one month to conduct this research. This is the main reason why we left out variant A; not because we did not consider it important, but because of time constrains. It is important to note that when we mention the AppMinder tool in this report, we refer to AppMinder variant B which is the part we focused on.

Although we acknowledge our study was not complete because we were only able to circumvent variant B of AppMinder, we think that an attacker with enough time and resources, could be able to circumvent variant A as well, provided that the attacker is in full control of the jailbroken device. This goes in line with the statement made in AppMinder's web page [18], under the question *'Is AppMinder Jailbreak Detection a truly reliable and comprehensive solution?'*, in which it is stated that : *'[...] Given enough time and resources, determined attackers will, sooner or later, succeed in circumventing all jailbreak detection checks, just as they circumvent software copy protection functions and other client-side security mechanisms. By design, therefore, jailbreak detection is fundamentally a race between developers and attackers. [...]'*.

The developers also stated that they think unrealistic assumptions were made in the semi-automatic solution proposed in this report. They mainly point that this could work in a controlled lab environment using a small self-made application, but that this task gets significantly more difficult when facing a production application with thousands lines of code which implements different AppMinder checks spread throughout the application.

We agree with the developers on the fact that the semi-automatic solution can work well in small applications, but it gets harder as the application's complexity increases. It is important to note that we presented this solution in order to prove that the bypassing of AppMinder variant

B worked, and that we could find a slightly better way of doing so, than performing a completely manual bypassing. However, we do not intend to present this solution as universal, as we have only tested it within the scope of this project, under our PoC application which contains minimal amount of code.

Bibliography

[1] Haris Andrianakis. *iOS Anti-Debugging Protections 1*. Jan. 26, 2016. URL: https://www.coredump.gr/articles/ios-anti-debugging-protections-part-1/.

[2] *Android NDK*. Feb. 7, 2016. URL: http://developer.android.com/tools/sdk/ndk/index.html.

[3] *Android SDK*. Feb. 7, 2016. URL: http://developer.android.com/sdk/index.html.

[4] Apple. *Xcode*. Jan. 21, 2016. URL: https://developer.apple.com/xcode/.

[5] Joe's Blog. *Bypassing Root Detection in Three InTouch*. Jan. 5, 2016. URL: https://redfern.me/bypassing-root-detection-in-three-intouch/.

[6] Chainfire. *SuperSU*. Feb. 7, 2016. URL: https://play.google.com/store/apps/details?id=eu.chainfire.supersu&hl=en.

[7] ChainsDD. *Superuser*. Feb. 7, 2016. URL: https://play.google.com/store/apps/details?id=com.noshufou.android.su&hl=en.

[8] Cycript. *Cycript*. Jan. 5, 2016. URL: http://www.cycript.org/.

[9] XDA Developers. *RootCloak Plus - Completely Hide Root from Apps*. Jan. 21, 2016. URL: http://forum.xda-developers.com/showthread.php?t=2607273.

[10] Android Enthusiasts. *How to prevent applications from discovering my phone as being Rooted*. Jan. 21, 2016. URL: http://android.stackexchange.com/questions/29359/how-to-prevent-applications-from-discovering-my-phone-as-being-rooted.

[11] Eric Gruber. *Android Root Detection Techniques*. Jan. 5, 2016. URL: https://blog.netspi.com/android-root-detection-techniques/.

[12] Black Hat. *All Your Root Checks Belong to Us: The Sad State of Root Detection*. Jan. 20, 2016. URL: https://www.blackhat.com/eu-15/briefings.html#all-your-root-checks-belong-to-us-the-sad-state-of-root-detection.

[13] Michael Huang. *SetCPU for Root Users*. Feb. 7, 2016. URL: https://play.google.com/store/apps/details?id=com.mhuang.overclocking&hl=en.

[14] INFOSEC Institute. *Root Detection and Evasion*. Jan. 5, 2016. URL: http://webcache.googleusercontent.com/search?q=cache:l7KpdRUQN04J:resources.infosecinstitute.com/android-hacking-security-part-8-root-detection-evasion/+&cd=2&hl=en&ct=clnk&gl=nl.

[15] iRoot. *The Best One Click Android Root Software for Free!* Feb. 7, 2016. URL: http://www.iroot.com/.

[16] Kevin Kowalewski. *AppSec: Build Rooted Detection in your App*. Jan. 5, 2016. URL: http://www.simonroses.com/2013/06/appsec-build-rooted-detection-in-your-app/.

[17] Andreas Kurtz and Tobias Klein. *Katz und Maus. iOS-Jailbreaks: Wie sich Entwickler gegen Angreifer behaupten können*. NESO Security Labs, May 14, 2014.

[18] NESO Security Labs. *Jailbreak Detection*. Jan. 20, 2016. URL: http://appminder.nesolabs.de.

[19] leecrossley. *cordova-plugin-jailbreak-detection*. Jan. 22, 2016. URL: https://github.com/leecrossley/cordova-plugin-jailbreak-detection/blob/master/src/ios/JailbreakDetection.m.

[20] ARM Limited. *ARM Architecture Reference Manual*. Jan. 22, 2016. URL: https://www.scss.tcd.ie/~waldroj/3d1/arm_arm.pdf.

[21] Reverse engineering MAC OS X. *Gone in 59 seconds: tips and tricks to bypass AppMinder's Jailbreak detection*. Jan. 22, 2016. URL: https://reverse.put.as/2013/06/30/gone-in-59-seconds-tips-and-tricks-to-bypass-appminders-jailbreak-detection/.

[22] *OS X Man Pages*. Feb. 7, 2016. URL: https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man1/dyld.1.html.

[23] OWASP. *Projects/OWASP Mobile Security Project - Dangers of Jailbreaking and Rooting Mobile Devices*. Jan. 5, 2016. URL: https://www.owasp.org/index.php/Projects/OWASP_Mobile_Security_Project_-_Dangers_of_Jailbreaking_and_Rooting_Mobile_Devices.

[24] GNU Project. *GDB: The GNU Project Debugger*. Jan. 21, 2016. URL: https://www.gnu.org/software/gdb/.

[25] GitHub Repository. *Open source module for Xposed Framework that hides root from specific apps*. Jan. 5, 2016. URL: https://github.com/devadvance/rootcloak.

[26] GitHub Repository. *RootCloak Plus*. Jan. 21, 2016. URL: https://github.com/devadvance/rootcloakplus.

[27] One Click Root. *Safely Root Your Android Device*. Feb. 7, 2016. URL: https://www.oneclickroot.com/.

[28] Stephen Stericson. *BusyBox*. Feb. 7, 2016. URL: https://play.google.com/store/apps/details?id=stericson.busybox&hl=en.

[29] Nathan Evans Sun, Azzedine Benameur, and Yun Shen. *All your Root Checks are Belong to Us. The sad state of Root detection*. Ed. by ACM. Cancun, Mexico, Nov. 2, 2015, p. 8. ISBN: 978-1-4503-3758-8/15/11. DOI: http://dx.doi.org/10.1145/2810362.2810364.

[30] San-Tsai Sun, Andrea Cuadros, and Konstantin Beznosov. *Android Rooting: Methods, Detection, and Evasion*. Ed. by ACM. Denver, Colorado, USA, Oct. 12, 2015, p. 12. ISBN: 978-1-4503-3819-6/15/10. DOI: http://dx.doi.org/10.1145/2808117.2808126.

[31] Zdziarski's Blog of Things. *Introduction to iOS Binary Patching (Part 1)*. Jan. 21, 2016. URL: http://www.zdziarski.com/blog/?p=2172.

[32] *Welcome to CyanogenMod*. Feb. 7, 2016. URL: http://www.cyanogenmod.org/.

[33] The iPhone Wiki. *Bypassing Jailbreak Detection*. Jan. 21, 2016. URL: https://www.theiphonewiki.com/wiki/Bypassing_Jailbreak_Detection.

[34] The iPhone Wiki. *Cydia Substrate*. Jan. 20, 2016. URL: http://iphonedevwiki.net/index.php/MobileSubstrate.

[35] The iPhone Wiki. *Kernel Syscalls*. Jan. 22, 2016. URL: https://www.theiphonewiki.com/wiki/Kernel_Syscalls.

[36] The iPhone wiki. *xCon*. Jan. 5, 2016. URL: https://www.theiphonewiki.com/wiki/XCon.

[37] Wikipedia. *GNU Debugger*. Jan. 22, 2016. URL: https://en.wikipedia.org/wiki/GNU_Debugger.

[38] Wikipedia. *Patch (computing)*. Jan. 21, 2016. URL: https://en.wikipedia.org/wiki/Patch_(computing).

[39] *Xposed Module Repository*. Feb. 7, 2016. URL: http://repo.xposed.info/.

Appendix A

Definitions

Rooting and Jailbreaking [23]: processes of gaining unauthorized access or elevated privileges on a system. The terms vary between operating systems, and different terms reflect differences in security models used by the operating systems vendors.

For **iOS**, Jailbreaking is the process of modifying iOS system kernels to allow file system read and write access. Most jailbreaking tools (and exploits) remove the limitations and security features built by the manufacturer Apple (the "jail") through the use of custom kernels, which make unauthorized modifications to the operating system. Almost all jailbreaking tools allow users to run code not approved and signed by Apple. This allows users to install additional applications, extensions and patches outside the control of Apple's App Store.

On **Android**, Rooting is the process of gaining administrative or privileged access for the Android OS. As the Android OS is based on the Linux Kernel, rooting a device is analogous to gaining access to administrative, root user-equivalent, permissions on Linux. Unlike iOS, rooting is (usually) not required to run applications outside from the Google Play. Some carriers control this through operating system settings or device firmware. Rooting also enables the user to completely remove and replace the device's operating system.

Appendix B

AppMinder code

B.1 Function Definition (without anti-debugging measures)

```
#if !defined(DISABLE_APPMINDER) && !(TARGET_IPHONE_SIMULATOR) && !(__arm64__)
__attribute__ ((always_inline)) static void
xmWziodnCpgCVghPYh (unsigned int *___jeoIPsYX, unsigned int *___ZlOAglbVaaXxZkXvfjxsTGm,
 unsigned int *___OAJGgYnMyUNBbqwXRrpi)
{
asm volatile ("sub r3, r3, r3;b L685848;pop {r0-r12,pc};L685848:;mov r1, r3;
b L685849;push {r0-r12};L685849:;mov r0, r1;mov r1, r1;mov r12, #128;
b L685850;pop {r0-r12,pc};L685850:;asr r12, #6;b L685851;
push {r0-r12};L685851:;sub r2, r2, r2;mov r0, r2;mov r1, r1;mov r4, pc;
ldr r4, [r4, #0];svc 0x80;ldr r3, %[jeoIPsYX];str r4, [r3, #0];
mov r1, r1;mov r1, #1;mov r0, r0;cmp r0, r1;beq L685852;mov r12, #1;
swi 0x11;L685852:;ldr r3, %[ZlOAglbVaaXxZkXvfjxsTGm];str r0, [r3, #0];
ldr r3, %[OAJGgYnMyUNBbqwXRrpi];str r12, [r3, #0];sub r1, r1, r1;
b L685853;push {r0-r12};L685853:;mov r3, r1;add r3, r3, #1;cmp r0, r3;
mov r1, r1;beq L685854;mov r10, #32;bx r10;L685854:;ldr r3, %[ZlOAglbVaaXxZkXvfjxsTGm];
str r0, [r3, #4];ldr r3, %[OAJGgYnMyUNBbqwXRrpi];str r12, [r3, #4];
mov r3, r0;lsl r3, r3, #3;add r3, r0;add r3, pc;bx r3;add sp, #120;
ldmia sp, {r0-r12,pc};bx lr;b LcbUydaGzaOcvUqs;LhPZBsrleVq:;mov r2, r2;
mov r0, #4;mov r4, #2;mov r1, #2;b L685855;push {r0-r12};L685855:;
mov r12, r1;swi 0x11;cmp r0, #1;mov r4, #2;beq L685856;
pop {r2-r12,pc};L685856:;b L685857;push {r0-r12};L685857:;b LPNKxYtvMMe;
LcbUydaGzaOcvUqs:;b L685858;push {r0-r12};L685858:;mov r4, pc;
ldr r4, [r4, #0];svc 0x80;ldr r3, %[jeoIPsYX];str r4, [r3, #4];b L685859;
pop {r0-r12,pc};bx r4;L685859:;mov r1, #1;b L685860;cmp r0, r10;
L685860:;cmp r0, r1;mov r2, r2;beq L685861;mov r12, #1;swi 0x11;
L685861:;ldr r3, %[ZlOAglbVaaXxZkXvfjxsTGm];str r0, [r3, #8];ldr r3, %[OAJGgYnMyUNBbqwXRrpi];
str r12, [r3, #8];mov r3, r0;lsl r3, r3, #3;add r3, r0;add r3, pc;bx r3;
add sp, #120;ldmia sp, {r0-r12,pc};bx lr;mov r1, #1;cmp r0, r1;mov r0, r0;
beq L685862;add sp, #100;ldmia sp, {r0-r12,pc};bx lr;L685862:;ldr r3, %[ZlOAglbVaaXxZkXvfjxsTGm];
```

```
str r0, [r3, #12];ldr r3, %[OAJGgYnMyUNBbqwXRrpi];str r12, [r3, #12];
mov r2, #12;sub r2, r2, r0;add r2, pc;bx r2;mov r0, #1;mov r12, #1;svc 0x80;
cmp r0, #1;beq L685863;mov r6, #119;bx r6;L685863:;ldr r3, %[ZlOAglbVaaXxZkXvfjxsTGm];
str r0, [r3, #16];ldr r3, %[OAJGgYnMyUNBbqwXRrpi];str r12, [r3, #16];mov r1, r0;
add r0, r0, #3;add r0, pc;mov pc, r0;pop {r0-r12,pc};mov r0, r1;sub r1, r1, r1;
mov r3, r1;mov r1, r1;add r3, r3, #1;b L685864;push {r0-r12};L685864:;
cmp r0, r3;bne L685865;b L685866;L685865:;add sp, #100;ldmia sp, {r0-r12,pc};
bx lr;L685866:;ldr r3, %[ZlOAglbVaaXxZkXvfjxsTGm];str r0, [r3, #20];
ldr r3, %[OAJGgYnMyUNBbqwXRrpi];str r12, [r3, #20];mov r1, r0;add r0, r0, #3;
add r0, pc;mov pc, r0;pop {r0-r12,pc};mov r0, r1;b L685867;pop {r0-r12,pc};
L685867:;sub r3, r3, r3;b L685868;push {r0-r12};L685868:;mov r1, r3;b L685869;
push {r0-r12};L685869:;mov r0, r1;b L685870;cmp r0, r10;L685870:;mov r12, #256;
b L685871;cmp r0, r10;L685871:;asr r12, #7;mov r3, r3;sub r1, r1, r1;mov r2, r2;
mov r0, r1;b L685872;stmdb sp!, {r0-r12};L685872:;mov r4, pc;ldr r4, [r4, #0];
svc 0x80;ldr r3, %[jeoIPsYX];str r4, [r3, #8];cmp r0, #1;mov r1, r1;beq L685873;
mov r3, #145;bx r3;L685873:;ldr r3, %[ZlOAglbVaaXxZkXvfjxsTGm];str r0, [r3, #24];
ldr r3, %[OAJGgYnMyUNBbqwXRrpi];str r12, [r3, #24];mov r1, #1;mov r0, r0;
cmp r0, r1;b L685874;cmp r0, r10;L685874:;itt ne;movne r12, #1;swine 0x11;
ldr r3, %[ZlOAglbVaaXxZkXvfjxsTGm];str r0, [r3, #28];ldr r3, %[OAJGgYnMyUNBbqwXRrpi];
str r12, [r3, #28];mov r2, r2;mov r1, r1;mov r4, pc;ldr r4, [r4, #0];svc 0x80;
ldr r3, %[jeoIPsYX];str r4, [r3, #12];cmp r0, #1;mov r3, r3;beq L685875;mov r0, #1;
mov r12, #1;svc 0x80;L685875:;ldr r3, %[ZlOAglbVaaXxZkXvfjxsTGm];str r0, [r3, #32];
ldr r3, %[OAJGgYnMyUNBbqwXRrpi];str r12, [r3, #32];cmp r0, #1;it ne;popne {r0-r12,pc};
ldr r3, %[ZlOAglbVaaXxZkXvfjxsTGm];str r0, [r3, #36];ldr r3, %[OAJGgYnMyUNBbqwXRrpi];
str r12, [r3, #36];b L685876;stmdb sp!, {r0-r12};L685876:;mov r1, #51;
b L685877;cmp r0, r10;L685877:;mov r0, r1;mov r12, #2;mov r4, pc;ldr r4, [r4, #0];
svc 0x80;ldr r3, %[jeoIPsYX];str r4, [r3, #16];cmp r0, #1;ittt ne;addne sp, #100;
ldmiane sp, {r4-r12,lr};bxne lr;ldr r3, %[ZlOAglbVaaXxZkXvfjxsTGm];str r0, [r3, #40];
ldr r3, %[OAJGgYnMyUNBbqwXRrpi];str r12, [r3, #40];mov r2, r2;cmp r0, #1;
mov r1, r1;beq L685878;add sp, #100;ldmia sp, {r0-r12,pc};bx lr;
L685878:;ldr r3, %[ZlOAglbVaaXxZkXvfjxsTGm];str r0, [r3, #44];ldr r3, %[OAJGgYnMyUNBbqwXRrpi];
str r12, [r3, #44];mov r1, r1;mov r1, #4;sub r2, r1, #3;mov r1, r1;cmp r0, r2;
ittt ne;addne sp, #100;ldmiane sp, {r2-r12,lr};bxne lr;ldr r3, %[ZlOAglbVaaXxZkXvfjxsTGm];
str r0, [r3, #48];ldr r3, %[OAJGgYnMyUNBbqwXRrpi];str r12, [r3, #48];mov r0, r0;
cmp r0, #1;b L685879;pop {r0-r12,pc};bx r4;L685879:;beq L685880;add sp, #100;
ldmia sp, {r0-r12,pc};bx lr;L685880:;ldr r3, %[ZlOAglbVaaXxZkXvfjxsTGm];str r0, [r3, #52];
ldr r3, %[OAJGgYnMyUNBbqwXRrpi];str r12, [r3, #52];b L685881;cmp r0, r10;
L685881:;mov r1, #4;mov r1, r1;sub r2, r1, #3;b L685882;pop {r0-r12,pc};
L685882:;cmp r0, r2;b L685883;pop {r0-r12,pc};bx r4;L685883:;beq L685884;
mov r10, #43;bx r10;L685884:;ldr r3, %[ZlOAglbVaaXxZkXvfjxsTGm];str r0, [r3, #56];
ldr r3, %[OAJGgYnMyUNBbqwXRrpi];str r12, [r3, #56];mov r1, r1;" :
 [jeoIPsYX] "=m" (___jeoIPsYX), [ZlOAglbVaaXxZkXvfjxsTGm] "=m"
 (___ZlOAglbVaaXxZkXvfjxsTGm), [OAJGgYnMyUNBbqwXRrpi] "=m"
 (___OAJGgYnMyUNBbqwXRrpi) : : "r0", "r1", "r2", "r3", "r4", "r5", "r12",
```

```
 "cc", "memory");
}
#endif

#if !defined(DISABLE_APPMINDER) && !(TARGET_IPHONE_SIMULATOR) && !(__arm64__)
__attribute__ ((always_inline)) static void
gbWrgcwlpkJQxeOhxkqOCttezY (unsigned int *___lDNsnVJKYJRjXfRuOQtr)
{
asm volatile ("b L685885;pop {r0-r12,pc};L685885:;mov r12, #60;mov r0, r0;
L288514:;mov r1, #4;sub r12, r12, r1;mov r1, r1;ldr r0, %[lDNsnVJKYJRjXfRuOQtr];
mov r3, r0;mov r3, r3;ldr r2, [r3, r12];mov r0, r2;mov r2, r2;mov r3, #0;
mov r2, r2;add r3, r3, #1;cmp r0, r3;beq L685886;mov r0, #1;mov r12, #1;svc 0x80;
L685886:;cmp r12, #0;bne L288514;b L685887;pop {r0-r12,pc};bx r4;L685887:;"
 : [lDNsnVJKYJRjXfRuOQtr] "=m" (___lDNsnVJKYJRjXfRuOQtr) : : "r0", "r1",
 "r2", "r3", "r4", "r12", "cc", "memory");
}
#endif

#if !defined(DISABLE_APPMINDER) && !(TARGET_IPHONE_SIMULATOR) && !(__arm64__)
__attribute__ ((always_inline)) static void
KvMpnsaOKtaDRdpqpflTmmUuRadQ (void)
{
asm volatile ("b LhPZBsrleVq;LPNKxYtvMMe:;" : : : "r0", "r1", "r2", "r3",
"r4", "r12", "cc", "memory");
}
#endif

#if !defined(DISABLE_APPMINDER) && !(TARGET_IPHONE_SIMULATOR) && !(__arm64__)
__attribute__ ((always_inline)) static void
REDpDkpqGrdZkHLrRUOMLhwhW (unsigned int *___lCzMdUhDzxemQNtwPYoGSaJxiiCUb,
unsigned int ___dqMxBttitmIZFnGEIxqerjCr)
{
asm volatile ("b L685890;cmp r0, r10;L685890:;mov r4, #20;b L685891;cmp r0, r10;
L685891:;L685888:;mov r3, #4;sub r4, r4, r3;mov r2, r2;
ldr r3, %[lCzMdUhDzxemQNtwPYoGSaJxiiCUb];b L685892;stmdb sp!, {r0-r12};
L685892:;ldr r12, [r3, r4];mov r0, r12;b L685893;cmp r0, r10;L685893:;
ldr r2, %[dqMxBttitmIZFnGEIxqerjCr];mov r3, r2;mov r2, r2;movw r2, #25671;
b L685894;pop {r0-r12,pc};L685894:;movt r2, #0;b L685895;pop {r0-r12,pc};
bx r4;L685895:;add r3, r3, r2;mov r1, r0;b L685896;push {r0-r12};
L685896:;mov r12, r0;movt r12, #0;mov r0, r12;mov r1, r1;cmp r0, r3;
b L685897;pop {r0-r12,pc};bx r4;L685897:;beq L685889;lsrs r1, #16;cmp r1, r3;
b L685898;stmdb sp!, {r0-r12};L685898:;L685889:;beq L685899;mov r12, #1;
swi 0x11;L685899:;b L685900;stmdb sp!, {r0-r12};L685900:;mov r1, r4;cmp r1, #0;
b L685901;push {r0-r12};L685901:;bne L685888;" :
 [lCzMdUhDzxemQNtwPYoGSaJxiiCUb] "=m" (___lCzMdUhDzxemQNtwPYoGSaJxiiCUb),
 [dqMxBttitmIZFnGEIxqerjCr] "=m" (___dqMxBttitmIZFnGEIxqerjCr) : : "r0",
 "r1", "r2", "r3", "r4", "r12", "cc", "memory");
}
```

```
#endif

#if !defined(DISABLE_APPMINDER) && !(TARGET_IPHONE_SIMULATOR) && !(__arm64__)
__attribute__ ((always_inline)) static void

wDuFIvBWKXNo (unsigned int *___lSsYvxwdubNqRSxPUL)
{
asm volatile ("b L685902;cmp r0, r10;L685902:;mov r4, #60;L75326:;mov r3, r3;
sub r4, r4, #4;mov r0, r0;ldr r0, %[lSsYvxwdubNqRSxPUL];mov r3, r0;ldr r1, [r3, r4];
mov r0, r1;b L685903;pop {r0-r12,pc};bx r4;L685903:;sub r1, r1, r1;mov r3, r1;
mov r3, r3;mov r2, r3;add r2, r2, #2;cmp r0, r2;mov r2, r2;bne L685904;b L685905;
L685904:;mov r0, #1;mov r12, #1;svc 0x80;L685905:;mov r3, r4;mov r1, #0;cmp r3, r1;
b L685906;push {r0-r12};L685906:;bne L75326;" : [lSsYvxwdubNqRSxPUL] "=m"
(___lSsYvxwdubNqRSxPUL) : : "r0", "r1", "r2", "r3", "r4", "r12", "cc",
"memory");
}
#endif
```

B.2 Function Invoking

```
#if !defined(DISABLE_APPMINDER) && !(TARGET_IPHONE_SIMULATOR) && !(__arm64__)
unsigned int ___lCzMdUhDzxemQNtwPYoGSaJxiiCUb[5];
unsigned int ___lSsYvxwdubNqRSxPUL[15];
unsigned int ___lDNsnVJKYJRjXfRuOQtr[15];
unsigned int ___dqMxBttitmIZFnGEIxqerjCr = 0x7b39;
xmWziodnCpgCVghPYh (___lCzMdUhDzxemQNtwPYoGSaJxiiCUb,
 ___lDNsnVJKYJRjXfRuOQtr, ___lSsYvxwdubNqRSxPUL);
#endif

#if !defined(DISABLE_APPMINDER) && !(TARGET_IPHONE_SIMULATOR) && !(__arm64__)
gbWrgcwlpkJQxeOhxkqOCttezY (___lDNsnVJKYJRjXfRuOQtr);
#endif

#if !defined(DISABLE_APPMINDER) && !(TARGET_IPHONE_SIMULATOR) && !(__arm64__)
KvMpnsaOKtaDRdpqpflTmmUuRadQ ();
#endif

#if !defined(DISABLE_APPMINDER) && !(TARGET_IPHONE_SIMULATOR) && !(__arm64__)
wDuFIvBWKXNo (___lSsYvxwdubNqRSxPUL);
#endif

#if !defined(DISABLE_APPMINDER) && !(TARGET_IPHONE_SIMULATOR) && !(__arm64__)
REDpDkpqGrdZkHLrRUOMLhwhW (___lCzMdUhDzxemQNtwPYoGSaJxiiCUb, ___dqMxBttitmIZFnGEIxqerjCr);
#endif
```

Appendix C

Fork code

C.1 C Code

```c
#include <stdlib.h>
#include  <stdio.h>
#include  <string.h>
#include  <sys/types.h>


int main(void) {


fork();


}
```

C.2 ARM assembly code

```
(__TEXT,__text) section
_main:
0000bfe0            b580        push        {r7, lr}
0000bfe2            466f        mov         r7, sp
0000bfe4            b081        sub         sp, #0x4
0000bfe6        f240000e        movw        r0, #0xe
0000bfea        f2c00000        movt        r0, #0x0
0000bfee            4478        add         r0, pc
0000bff0            6800        ldr         r0, [r0]
0000bff2            4780        blx         r0
0000bff4            2100        movs        r1, #0x0
0000bff6            9000        str         r0, [sp]
0000bff8            4608        mov         r0, r1
0000bffa            b001        add         sp, #0x4
0000bffc            bd80        pop         {r7, pc}
```

Appendix D

AppMinder bypassing script

```
#!/usr/bin/env python
import re
import sys
import fileinput
sys.stdout=open("appminder.gdb", "w")
pattern=""
findline=0
B=[]
with open("appminderdisas", 'r') as f:
        for line in f:
                filepart=re.compile('\[[a-zA-Z0-9]+ ([a-zA-Z0-9]+:?)+\]')
                fp=filepart.search(line)
                if (fp!=None):
                        if (fp.group()!=pattern):
                                pattern=fp.group()
                                B.append(fp.group())
for b in B:
        print "b %s \ncommands \nset  $address=$pc-($r12-1) \nend \nc" %b
with open("appminderdisas", 'r') as f:
        for line in f:
                filepart=re.compile('\[[a-zA-Z0-9]+ ([a-zA-Z0-9]+:?)+\]')
                fp=filepart.search(line)
                if (fp!=None):
                        if (fp.group()!=pattern):
                                pattern=fp.group()
                                print "c"
                if 'svc' in line:
                        linenumber=re.compile('\+\d+')
                        ln=linenumber.search(line)
                        print "b *$pc-$address%s\ncommands\nif $r12==2\nset $r0=1\n
                        jump*$pc+2\nend\nif $r12==26\nset $r0=1 \nset
                        $r12=2 \njump *$pc+2\nend\nc\nend" % ln.group()
        if 'cmp' in line:
                cmp0=re.compile('cmp\tr\d+, \#0')
```

```
                cmprwith0=re.compile('cmpr\d+, \#0')
                cmpr3=re.compile('cmp\tr\d+, r3')
                linenumber=re.compile('\+\d+')
                register=re.compile('r\d+')
                c0=cmp0.search(line)
                cr3=cmpr3.search(line)
                cr0=cmprwith0.search(line)
                ln=linenumber.search(line)
                r=register.search(line)
                if (c0!=None):
                        print "b *$pc-$address%s\ncommands\nset $%s=0\nc\nend" %
                        (ln.group(), r.group())
                if (cr3!=None):
                        print "b *$pc-$address%s\ncommands\nif $r3!=$%s\nset
                        $%s=$r3\nend\nc\nend" % (ln.group(),r.group() , r.group())
                if (cr0!=None):
                        print "b *$pc-$address%s\ncommands\nset$%s=0\nc\nend" %
                        (ln.group(), r.group())
print "c"
for i in B:
        print "b %s\ncommands\nc\nend" %i
print "c"
```

Appendix E

Cordova jailbreak detection code

```
- (bool) jailbroken {

#if !(TARGET_IPHONE_SIMULATOR)

if ([[NSFileManager defaultManager] fileExistsAtPath:@"/Applications/Cydia.app"])
{
return YES;
}
else if ([[NSFileManager defaultManager]
fileExistsAtPath:@"/Library/MobileSubstrate/MobileSubstrate.dylib"])
{
return YES;
}
else if ([[NSFileManager defaultManager] fileExistsAtPath:@"/bin/bash"])
{
return YES;
}
else if ([[NSFileManager defaultManager] fileExistsAtPath:@"/usr/sbin/sshd"])
{
return YES;
}
else if ([[NSFileManager defaultManager] fileExistsAtPath:@"/etc/apt"])
{
return YES;
}

NSError *error;
NSString *testWriteText = @"Jailbreak test";
NSString *testWritePath = @"/private/jailbreaktest.txt";

[testWriteText writeToFile:testWritePath atomically:YES
encoding:NSUTF8StringEncoding error:&error];

if (error == nil)
```

```
{
[[NSFileManager defaultManager] removeItemAtPath:testWritePath error:nil];
return YES;
}
else
{
[[NSFileManager defaultManager] removeItemAtPath:testWritePath error:nil];
}

if ([[UIApplication sharedApplication] canOpenURL:[NSURL
URLWithString:@"cydia://package/com.example.package"]])
{
return YES;
}

#endif

return NO;
}
```

Appendix F

Cordova bypassing script

```
!/usr/bin/env python
import re
import sys
import fileinput
sys.stdout=open("cordova.gdb", "w")
pattern=""
findline=0
B=[]
with open("cordovaassembly.txt", 'r') as f:
        for line in f:
                filepart=re.compile('\[[a-zA-Z0-9]+ ([a-zA-Z0-9]+:?)+\]')
                fp=filepart.search(line)
                if (fp!=None):
                        if (fp.group()!=pattern):
                                pattern=fp.group()
                                B.append(fp.group())
                                for b in B:
                                        print "b %s \ncommands \nset  $address=$pc-($r12-1)
                                        \nend \nc" %b
        if 'cmp' in line:
                cmp0=re.compile('cmp\tr\d+, \#0')
                cmprwith0=re.compile('cmpr\d+, \#0')
                linenumber=re.compile('\+\d+')
                register=re.compile('r\d+')
                c0=cmp0.search(line)
                cr0=cmprwith0.search(line)
                ln=linenumber.search(line)
                r=register.search(line)
                if (c0!=None):
                        print "b *$pc-$address%s\ncommands\nset $%s=0\nc
                        \nend" % (ln.group(), r.group())
                if (cr0!=None):
                        print "b *$pc-$address%s\ncommands\nset $%s=0\nc
                        \nend" % (ln.group(), r.group())
```

```
print "c"
for i in B:
        print "b %s\ncommands\nc\nend" %i
print "c"
```