Mobile Application Sandboxing

Tommy Thorsen

10th January 2017

As mobile devices replace old-fashioned computers for more and more of our daily tasks, they become very attractive targets for criminals. Despite efforts to detect and block malicious applications from entering the app stores, all the major stores contain significant amounts of malicious apps. It is up to the security mechanisms implemented in the mobile platforms to keep the users safe from fraud and identity theft. One common security mechanism that can be found in all the major mobile platforms, is *application sandboxing*.

In this paper, we will take a closer look at how application sandboxing works in the two largest platforms, Android and iOS. We will see examples of how malicious apps can break out of the sandbox, and we will have a look at various proposals for improving the sandboxes.

1 Introduction

The term *sandbox* in a computer security context, refers to an enclosed environment in which one can run software, in order to prevent said software access to the rest of the system. Sandboxes have been around for a long time, in various forms. *Virtual Machines* are common types of sandboxes. They allow us to run an untrusted application inside a separate virtual system. The disadvantage is that they are quite resource-demanding, both requiring a separate copy of the operating system, and extra CPU cycles spent on managing the virtual environment. More light-weight forms of sandboxing can be implemented using fine-grained access control mechanisms such as `SELinux`. System utilities such as `chroot` and FreeBSD's `jail` are also commonly used ways to confine a process to an smaller area of the file system.

A system where all applications ran within a completely watertight sandbox, would presumably be quite secure, but not very useful. Most applications need to access common files and resources, and communicate with the operating system and with other applications. A document editor would need access to the user's document folder, and an image viewer needs access to any image the user wants to view. We have also come to expect the ability to copy text from one program and paste it into another.

Most modern mobile platforms rely heavily on giving the user the ability to download *apps*, which are most often created by a third party. To protect agains a malicious app reading files or executing code that belongs to other apps, or to the platform itself, apps are run inside a sandbox.

2 Attacking Sandboxes

In this section, we will take a closer look at how the sandboxes on both Android and iOS work. We will look at what mechanisms exist for the apps to communicate with resources outside of the sandbox, and what possible attacks exist against these mechanisms. First we will look at Android, and then iOS.

2.1 Android

The Android mobile operating system has quickly become the most popular system in the market. This has made Android a very attractive target for attackers and malware creators. A significant amount of the applications in the Android Play Store can be categorized as malware or spyware. It is therefore important for the users that the Android system has security measures in place to protect them against resource misuse and privacy intrusions. We'll take a look at what measures are in place, and what known security loopholes currently exist in the Android platform. First, let's have a quick look at how the Android platform is put together.

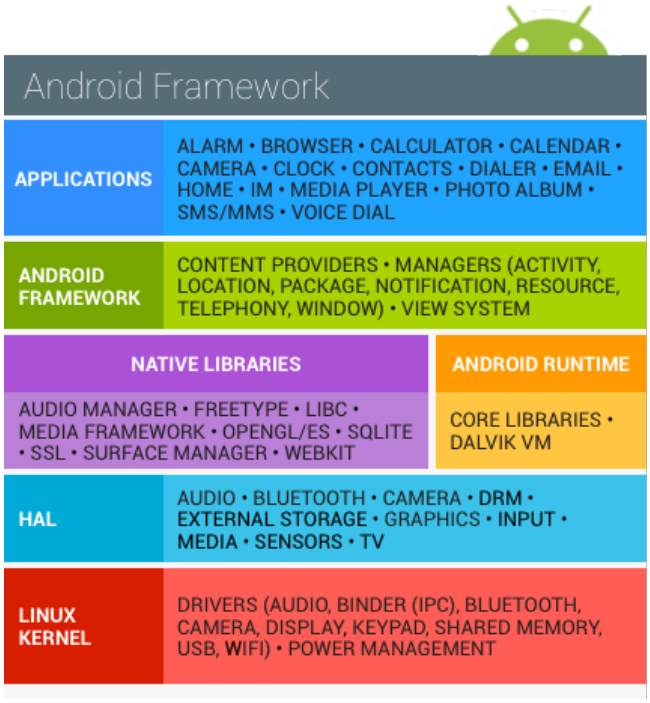

Figure 1: The Android Framework [Google, 2016b].

2.1.1 Platform Overview

The Android mobile platform is built up of several components or

Portions of this page are reproduced from work created and shared by the Android Open Source Project and used according to terms described in the Creative Commons 2.5 Attribution License.

layers, as shown in figure 1. At the very bottom sits a fairly regular Linux kernel. A hardware abstraction layer (HAL) sits right on top of the kernel, and provides general interfaces to the various hardware components on the device, such as camera, sound chip, input devices etc. Above this again, is a Java Virtual Machine (JVM), inside which all the apps are run. Although the JVM does provide a form of sandbox, this is not sufficient protection, since the apps may contain native libraries that communicate with the Java counterpart using the Java Native Interface (JNI). The JVM can therefore not be considered a proper security boundary. The JVM does, however, provide some protection against memory-related attacks against the java parts of the apps.

Android implements sandboxing through the mandatory access control mechanisms provided by SELinux. Upon installation, each app is given a unique User ID. When the app is launched, it runs as this special user, which only has access to a very limited set of resources. It is not allowed to access any files or folders belonging to other apps. If access to system resources such as networking or the camera is required, these permissions have to be stated in a manifest file. The user of the phone has to approve these permissions before the app is able to make use of them. In earlier Android versions, the permissions had to be accepted as a part of the installation process, but the latest Android version (version 6.0) changes this so that the app must as for permission at the time of use. The latest Android version also lets the user allow the app some permissions, but deny it others.

2.1.2 Privilege Escalation Attacks

The Android platform security sounds good in principle — both Linux and SELinux are mature products that are considered relatively secure, and should be up to the task. However [Davi et al., 2011] claims that "*Android's sandbox model is conceptually flawed and actually allows privilege escalation attacks.*" Specifically, it is possible to perform certain privilege escalation attacks. The paper states the problem as: "*An application with less permissions (a non-privileged caller) is not restricted to access components of a more privileged application (a privileged callee).*"

This weakness can be exploited through the *intent* mechanism, which is a mechanism used to pass events between applications. Some applications will, upon receiving a specific intent, perform an action which makes use of a permission that the application has been granted. For instance, the Phone application will start a phone call when it receives the `android.intent.action.CALL` intent with a phone number as the parameter. The problem here is that any application without the `PHONE_CALLS` permission can freely start unauthorized phone calls through the Phone application.

There exists no mechanism that would let the Android platform these indirect resource accesses. In principle, it is possible for apps like the Phone app to perform the necessary security checks upon receiving an intent that required accessing restricted resources. This would delegate the responsibilities of reference monitoring to the individual apps, which is not likely to work well. It might be possible to do this for the system apps that Google has control over, but to expect third parties to both care about this and to be able to implement this correctly, seems unrealistic.

The paper goes on to describe an actual proof-of-concept attack. The attack makes use of an application called *Android Scripting Environment* (ASE) which is not a core application, but is developed by Google developers and as such looks trustworthy. Since the application is a scripting environment, it has extensive permissions. A vulnerability in ASE is exploited along with a heap overflow vulnerability in order to mount the attack. We will not recount the full attack here, but we will instead move on to the topic of return-oriented programming on Android, which is the technique the paper uses to exploit heap overflow vulnerabilities.

2.1.3 Return-Oriented Programming

Version	Codename	API	Distribution
2.2	Froyo	8	0.1%
2.3.3 - 2.3.7	Gingerbread	10	2.7%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	2.5%
4.1.x	Jelly Bean	16	8.8%
4.2.x		17	11.7%
4.3		18	3.4%
4.4	KitKat	19	35.5%
5.0	Lollipop	21	17.0%
5.1		22	17.1%
6.0	Marshmallow	23	1.2%

Figure 2: The Android Framework [Google, 2016a].

Earlier versions of the Android platform did not provide *address space layout randomization* (ASLR), and as such was vulnerable to return-oriented programming (ROP). Google added its first implementation of ASLR in Android 4.0 which was released in October 2011 [Liebergeld and Lange, 2013]. The initial implementation only randomized the stack address and location of shared libraries. Support for *position independent executables* (PIE) was added in Android 4.1 in June 2012, along with a randomized linker for better ASLR support.

Subsequent releases of Android add an increasing amount of security measures to the platform, which illustrates the continuous battle between Google and the Android hackers around the world. Many exploits outlined in various papers (like the ROP part of the attack in [Davi et al., 2011]) might not be possible on the latest version of Android. This does not mean that the exploit has become irrelevant. The fragmented world of Android devices has become known for its slow uptake of new versions. Many manufacturers do

not bother with updating older devices to newer versions of Android, and as a result, there are still plenty of targets for an attacker with an outdated exploit. Figure 2 shows the distribution of users on the different platform versions as per February 2016.

Even on fully updated devices, there may be ways to do ROP. [Lee et al., 2014] claims to have found a critical vulnerability in the way processes are created on Android, that effectively cancels out all the benefits of ASLR. This vulnerability stems from a performance optimization that has existed in Android since the early days of the platform. Since the creation and initialization of new processes is quite expensive in terms of CPU, especially on low-end devices, the designers of the Android platform chose to let all new processes be forks of an already instantiated mother process, called Zygote. This technique drastically cuts down on the apps' startup time. However, since all processes are clones of the Zygote process, they also have more or less identical memory layout. All apps also share the same loaded libraries as the Zygote process, including the `libc` library.

The result is that the Zygote forking technique largely skips the memory layout randomization of ASLR, giving attackers an easy way to perform return-oriented programming and other memory-related attack techniques. The authors of [Lee et al., 2014] do not seem to have performed an actual attack using this method, but they do present a couple of detailed and realistic attack scenarios. It's not unlikely that we will see some real attacks in the wild, that make use of this memory layout weakness.

2.2 iOS

The second largest mobile operating system is iOS. Unlike Android, which is a relatively open system, used by many manufacturers, iOS is proprietary and only used on Apple's own devices. It was developed for the original iPhone, which was released in 2007.

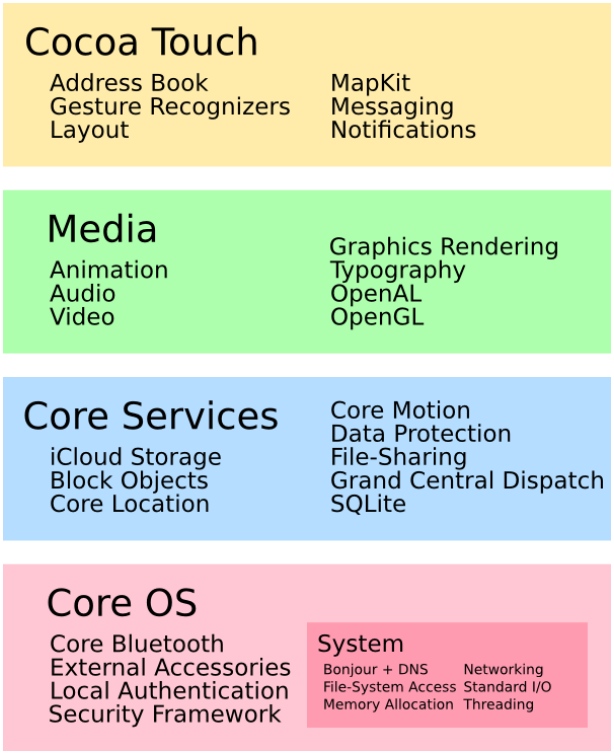

Figure 3: The iOS Framework [Apple, 2016].

2.2.1 Platform Overview

The iOS operating system is loosely based on the Mac OS X operating system, and the sandboxing mechanism is also taken from OS X. It uses the TrustedBSD mandatory access control framework to control the behavior of the apps. iOS has had ASLR since version 4.3 which was released in 2011, and *data execution prevention* (DEP) since version 2.0.

iOS does not give each app an individual User ID, like Android does. Instead it uses a special *Apple ID*, which is composed of *Team ID* which is unique to each developer along with a *Bundle ID* which identifies the app [Xing et al., 2015]. In a similar manner to Android, iOS applications that require permissions beyond the default ones, can request these by listing a set of *entitlements* within its property file.

Apple's iOS ecosystem has a reputation of being much stricter than the more open and developer-friendly Android environment. It is generally only possible to install apps that come from Apple's App Store, onto an iOS device. All of these apps have undergone some form of review by Apple, before it gets signed with Apple's private key, allowing for distribution through the App Store and installation to devices. While this has resulted in less malware on iOS devices than on Android devices, it's not a completely bulletproof system, and it is possible to fool Apple's review procedure.

2.2.2 Scheme Hijacking

One attack, that the authors of [Xing et al., 2015] had been able to upload to the App Store, is a so-called *scheme hijacking* attack. Like the privilege escalation attack for Android, described above, the scheme hijacking attack makes use of a mechanism for communicating between apps. An iOS app may register a *scheme* which lets other apps invoke it with a set of parameters. For instance, the Yelp app may want to register the `yelp://` scheme to let other apps pass search terms to Yelp. If another app invokes the url `yelp://search?terms=coffee`, the Yelp app would be invoked and instructed to launch a search for coffee nearby.

A scheme in iOS can only belong to a single app, and if multiple apps register the same scheme, it turns out that the last app that registered the scheme will be the final owner of that scheme. This conflict resolution strategy has some obvious drawbacks, as it allows a malicious app to take over a scheme belonging to another app. The author of that paper were able to perform the following attack against the popular Facebook and Pinterest apps, and even successfully uploaded it to the App Store:

> *As an example, we exploited this weakness and successfully obtained the victim's Facebook token for Pinterest, the most famous personal media management app. Specifically, Pinterest and other apps all support the SSO login through the Facebook app. Whenever the user clicks on "continue with Facebook" in these apps, the Facebook app is invoked to ask for the user's permission to let the authentication go through and also grant Pinterest (and other apps) access to some of her Facebook data. With the user's consent, Facebook triggers a scheme* `fb274266067164://access_token=CAAAAP9uIENwBAKk&X=Y`

> *to deliver a secret access token to the app. In our research, our attack app registered* `fb274266067164://` *and took over this scheme. As a result, Facebook unwittingly launched our app and passed to it Pinterest's access token.*
>
> —[Xing et al., 2015]

2.2.3 Jekyll Apps

We've mentioned earlier how Apple relies on their app review process to keep malicious apps out of the ecosystem. The exact procedure behind this review is not publicly known, but the reviewers are known to look for such things as apps that access private system apis, and apps that access private and sensitive data. [Wang et al., 2013] outlines a novel way to circumvent the app review process.

The idea behind this attack is to create an app, which does not contain any malicious code, but which does contain a remotely exploitable vulnerability / backdoor. The app can also contain a set of code gadgets that can be easily chained together to create a malicious code-path. This way, it is not possible for the App Store reviewers to detect any malicious behavior from the app.

Once the app has been deployed, it opens up a range of possibilities for the attacker. First of all, the app may access private APIs, which in many cases allow access to private data, or internal functionality, not intended for third party apps. The authors of [Wang et al., 2013] have developed a dynamic analysis tool that lets them see what internal functions are called in several different scenarios. This allowed them to find functions for posting Twitter tweets, and sending emails and SMSes without any user interaction, although some of these did not work on all platforms.

The Jekyll app also opens up more possibilities for exploiting vulnerabilities in drivers and other kernel modules. The authors were able to successfully exploiting a known vulnerability in iOS 5.x, using this method. Finally, the Jekyll app can be used to launch attacks on other applications on the device. As a proof-of-concept, the authors used the scheme invocation mechanism to send the mobile Safari app to a malicious web address, which then exploited a known vulnerability in Safari.

3 Improving Device Security

There are two main approaches for protecting the users of a mobile device. The first approach is to detect malicious apps before they get to the phone altogether, primarily through app store admission reviews. We have seen that Apple employs this strategy to a greater extent than Google. The second approach is to harden the security on the device, so that malicious apps that manage to get installed, can not do any damage. In this section, we will take a look at proposals that exist for enhancing the security on the mobile devices, and in the next section we will look at possible ways to detect malicious behavior as part of a review process.

3.1 Privilege Escalation Attacks

Earlier in this paper, we've looked at so-called *privilege escalation attacks*. We remember that this kind of attack is possible because many apps require an extensive set of permissions (see figure 4). Many of these apps also expose IPC interfaces for communicating with other apps, and in some cases, these interfaces can be exploited by less privileged apps in order to access resources that these apps would not normally have access to.

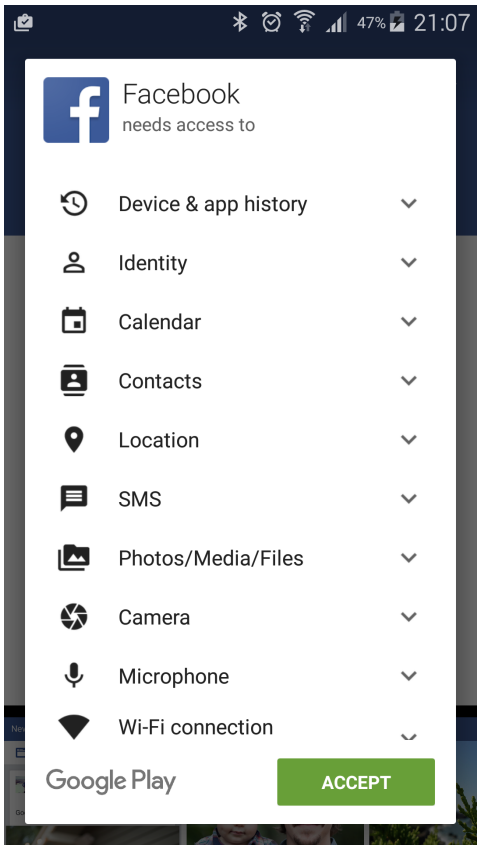

Figure 4: Example of permission requirements for a popular app on Android.

A proposal for a new mechanism to deal with this kind of attack is given in [Felt et al., 2011]. This new mechanism is called *IPC inspection*. Two things are required in order to implement IPC inspection. First of all, it must be possible to monitor the IPC (short for inter-process communication) events passed between applications. Secondly, we must be able to modify the permissions of a running application. With these two capabilities, we can detect when one app sends a request to another app, and reduce the permissions of the receiving app to the intersection of the permissions of the two apps. This will ensure that the requesting app is unable to use the receiving apps permissions to access resources it normally would not be permitted to access. In case of two-way communication, the permissions of both apps will be reduced.

Although there are some challenges with implementing this mechanism, it fulfills some very important criteria of such a security system. The most important criteria might be that it does not require any diligence from the individual app developers. Time and time again, it has been shown that app developers are not motivated to spend any extra time or resources adding extra security to their app, and any mechanism that relies on cooperation from the app developers is doomed to fail. By sitting on the outside of the apps, monitoring events and manipulating permissions, the IPC inspection mechanism can defend against privilege escalation attacks without the knowledge of the apps. This also makes the mechanism work regardless of how the apps are implemented — whether they are java or native apps.

The authors of [Felt et al., 2011] have created a proof-of-concept implementation on Android. This implementation consists of modifications to the two system apps `PackageManager` and `ActivityManager`. The `PackageManager` application is responsible for installing

applications, and managing permissions, and can be used to fetch permissions for the various apps and to perform the permission reduction. The `ActivityManager` application handles launching applications and communication between applications. When the modified `ActivityManager` application detects one of several predetermined actions, it will notify the modified `PackageManager`, which will reduce permissions appropriately.

Prior to implementing this prototype, the authors had done an investigation on the system applications in Android, and found 15 vulnerabilities in 5 of the applications. All of these vulnerabilities were found to be impossible to exploit with the IPC inspection mechanism in place.

3.2 TaintDroid

The *TaintDroid* tool presented in [Enck et al., 2014] is a tool for doing live *taint tracking* of data within a live app. Taint tracking is a technique for tracking data flow, where the data we're interested in is marked as *tainted*. Any data that is derived from tainted data, such as function arguments or strings created by concatenating tainted strings, is also marked as tainted. In the case of TaintDroid, privacy sensitive data is marked as tainted, and if tainted data reaches the network, we can consider it a privacy breach.

TaintDroid is implemented as a modified version of the Dalvik VM, where extra space is allocated to each stack variable in memory in order to keep track of the various categories of taint. The so-called *taint tags* are interleaved with the original data values. Messages sent over IPC are also given a single taint tag to represent the data passed within the message. This allows taint flow to be tracked between applications. Files are also given a single taint tag per file. This taint tag is stored in the filesystem's extended attributes, and updated on file write. All data read from a file is tagged with the taint tag of the file. Although this may lead to false positives, it keeps malicious code from circumventing TaintDroid by channeling the sensitive data through a temporary file.

> *Our study of 30 popular applications shows the effectiveness of the TaintDroid system in accurately tracking applications' use of privacy sensitive data. While monitoring these applications, TaintDroid generated no false positives (...). The flags raised by TaintDroid helped to identify potential privacy violations by the tested applications. Half of the studied applications share location data with advertisement servers. Approximately one third of the applications expose the device ID, sometimes with the phone number and the SIM card serial number.*
>
> —[Enck et al., 2014]

TaintDroid, being implemented in the VM, is not able to monitor native code. For some functions that are built into the VM, the authors made modifications to enable taint propagation. `JNI` functions are handled simply by setting the taint tag on the return value to the union of the taint values of the parameters. This does not consider

the contents of the native functions at all, and while some cases may be covered by this implementation, the case where tainted data is sent over the network from native code, is not detectable by TaintDroid.

A test was run on a set of 30 randomly chosen apps that all required the `INTERNET` permission along with one or more permissions required to access sensitive information such as location, camera or microphone. These were all tested manually, in order to go through all the necessary steps to make use of the main functionality of the apps. Along with the logs from the taint tracking system, WiFi packet traces were also taken for verification reasons. The manual testing also allowed the authors to evaluate whether any leaks of sensitive information was sufficiently notified in warning dialogs or privacy statements.

2 out of the 30 apps were found to send the device's phone number and IMSI to the app developers' server. One of these apps would even send this information right after installation, before the first use of the app. 9 apps sent the IMEI to the server, although two of these did specify this in the EULA, which arguably makes it acceptable. Half of the 30 apps sent information about the geographical location of the device to the servers. The WiFi packet traces showed that this data was in fact sent to various advertising backends. Although two of these apps did display a EULA on the first launch, neither of these EULAs made any mention of this.

One of the main implementation goals for TaintDroid was to implement a system that can be run in real time without adding overhead so severe as to hinder the usability of the monitored apps. The whole TaintDroid system was measured to cause a runtime overhead of about 14%. The added memory allocated for the taint tags causes around a 4.4% memory overhead. Although these overheads may be noticeable, they are not excessive, and should probably be acceptable for the security-conscious user

3.3 Boxify

The *Boxify* sandboxing solution presented in [Backes et al., 2015] aims to create a application virtualization sandbox for Android, in the form of a regular app. The Boxify app does not require any modifications to the firmware, nor does it require a rooted device.

It makes use of a little known feature in Android, called an *isolated process* to run other apps inside a controlled process. Isolated processes in Android have no permissions, and run with a separate UID from the rest of the app, causing them to have little filesystem access as well. Not many apps currently make use of this feature, with the exception of the Chrome browser, which uses isolated processes to run layout and javascript code. Since these are complex pieces of code, which interfaces heavily with input from untrusted sources, it makes sense to run them in an unprivileged process, so that any exploits will have limited impact.

Boxify, however, wants to use this isolated process feature to run entire apps. Having effectively removed all means for an app to communicate or access any resources, they use IPC redirection and libc hooking to re-establish communication and resource access mechanisms through a component called the Broker. The Broker is a reference monitor

that acts as a mandatory proxy for all I/O operations from the sandboxed application. Any call going through the Broker is evaluated against a security policy, and either granted or denied. Calls from the Android platform to the sandboxed app are likewise relayed by the Broker.

> *Android apps are tightly integrated within the application framework, e.g., for lifecycle management and inter-component communication. With the restrictions of an isolated process in place, encapsulated apps are rendered dysfunctional. Thus, the key challenge for Boxify essentially shifts from constraining the capabilities of the untrusted app to now gradually permitting I/O operations in a controlled manner in order to securely re-integrate the isolated app into the software stack.*
>
> —[Backes et al., 2015]

In order for the Boxify app to be able to control other apps, the other apps need to be launched through Boxify, and not through the regular launcher on the device. It would be possible to integrate Boxify into a custom launcher, in order to make the sandboxing mandatory and seamless to the user.

A test run of Boxify was done, using 1079 of the most popular apps in the Google Play Store. While most of these ran fine, a small percentage of the apps experienced crashes. Most of these crashes turned out to be caused by usage of exotic APIs that had not yet been implemented in Boxify. Since Boxify blocks everything by default, and implements overrides of all access methods through the Broker, it is vital that the implementation of Boxify covers every API that exists in the Android platform, if unexpected crashes are to be avoided.

The paper promises that the Boxify framework will be published as open source, available freely for academic and non-commercial use, but at the time of writing this, their webpage says: "Because of ongoing business negotiations, the public release of the source code has been postponed by a few weeks." Time will tell whether these ideas will be turned into a product available to smart phone users.

3.4 AppCage

The *AppCage* sandboxing solution presented in [Zhou et al., 2015] aims to solve many of the same problems as the Boxify sandbox we looked at above. AppCage is a *hybrid sandbox*, which means that it consists of two different sandboxes — one for java code and one for native code. Like Boxify, AppCage requires no changes to the Android platform, nor does it require root access to the device.

For each app that is installed to a device, AppCage will generate a wrapper app which contains the entire original app plus an instrumented version of any native code contained in the app. This wrapper app will request the same permissions as the original app. After this wrapper app is generated, the original app may be uninstalled. All of this is handled by a small utility AppCage app that continuously monitors the system

looking for newly installed apps, and when a new app is detected, requests permission from the user to replace it with a generated wrapper app.

In addition to this utility app, AppCage also consists of a permission manager app for configuring access policies. For each access attempted by the code inside the sandboxes, the permission manager is consulted, and can respond with allow, deny or prompt-to-user. It's also possible to make it return fake results, for instance fake locations, in order to keep apps that require GPS from crashing, but still keeping them from being able to extract sensitive information about the user. All permissions default to prompt-to-user, giving the user a dialog every time an app tries to access a new resource.

> *Android has become the leading mobile platform with nearly 85% market share in the second quarter of 2014. This trend is accompanied by the vigorous increase of third-party Android apps that are available for users to download and install. However, recent studies reveal that third-party apps, including popular ones, could leak private information without user consent or awareness.*
>
> —[Zhou et al., 2015]

The java sandbox, called a *dex sandbox* in this paper, is implemented by using framework API hooking. This is considered a more robust solution than techniques such as rewriting the sandboxed app's bytecode, since this will also work for dynamically loaded bytecode.

> *Our evaluation shows that AppCage can successfully detect and block the attempts to leak private data or perform dangerous operations by malware and invasive apps, and it also has an acceptable overhead, especially for apps without native code.*
>
> —[Zhou et al., 2015]

The native sandbox contained within AppCage uses a technique called *software fault isolation* to enforce control over the sandboxed app. It keeps the sandboxed native code from writing to memory outside of the sandbox. This is important, because otherwise malicious native code might tamper with the java sandbox. The native sandbox also heavily regulates access to system calls.

The security effectiveness of the AppCage solution was evaluated using malware samples from the Android Malware Genome Project along with an online malware repository. In addition, testing included several apps that are not considered malware, but still aggressively access sensitive information in order to deliver targeted advertising. AppCage was found to be able to intercept all malicious behavior tested.

A lot of effort went into ensuring that the performance of AppCage would be acceptable to the user. The extra size added by the wrapper code amounts to less than a 1% increase in disk space usage for apps with no native code. The increase in disk space for apps with native code is around 30% of the

size of the native code. Installation time and complexity is also somewhat increased. Runtime performance of apps with no native code is negligible, but native code is subject to a 10% performance loss.

3.5 XiOS

We have now presented three different security security solutions for Android, but where are the iOS enhancements? Is iOS so secure that it needs no hardening of its sandbox? While this is unlikely to be true, the vast majority of recent malware is aimed at the Android platform [F-Secure, 2014], which shows that the Android platform has the greatest need of enhanced security solutions. The last paper we will present in this section, however, does address the sandboxing solution on iOS, and propose a number of improvements.

The paper [Bucicoiu et al., 2015] presents the *XiOS* service, which promises fine-grained application sandboxing for the iOS platform. Like the solutions we've seen for Android, XiOS does not require a rooted (jailbroken) device. It does, however, require that the individual app developers submit their apps to a service that implants the XiOS reference monitor, prior to submitting the app to the App Store. While it seems unrealistic to expect most developers to do this, the paper also mentions the possibility of integrating XiOS into an enterprise mobile device management system, in order to harden all applications before deployment to employee devices. This would seem to be an attractive option for security-oriented companies.

> *Recent attacks have demonstrated that the current design of iOS is vulnerable to a variety of attacks that undermine the iOS sandboxing model leading to the invocation of private APIs (e.g., sending text messages in background). While previous attacks rely on specific assumptions such as the availability of a public framework, we showed that the default iOS application structure by itself can be easily exploited to invoke dangerous private APIs.*
>
> —[Bucicoiu et al., 2015]

The main objective of XiOS, is to prevent malicious apps access to private APIs, using methods such as the Jekyll App approach which we've discussed earlier. It is also a goal to be able to enforce a set of fine-grained access control rules. To accomplish this, XiOS uses binary instrumentation techniques performed by a static binary rewriter.

XiOS has proven to be able to prevent attacks such as the Jekyll App attack. There is some performance overhead, which typically amounts to a less than 5% increase. The biggest challenge with this approach is likely to be the deployment, as it relies on developers adding the extra sandbox to their app on their own accord. This seems unlikely to succeed, unless Apple would start requiring the presence of such a sandbox in order to be allowed into the App Store.

4 Catching Malicious Apps

Finally, we will look at a slightly different way of using sandboxes to improve security on mobile devices. We remember that both Android and iOS devices can download apps from an official app store. On iOS, downloading from Apple's Play Store is pretty much the only way to get new apps installed. On Android, it's not mandatory to use Google's Play Store — apps can be downloaded from third-party app stores or even regular web sites — but still, the majority of apps installed on Android devices come from the official app store.

If we were able to keep the official app stores completely free of malware, it is likely that the entire device ecosystems would also be very close to malware-free. One way to do this, is to use a similar sandbox as the ones that run on the devices, inside a virtual environment. By looking at all the access attempts made by the app for resources outside of the sandbox, we might be able to detect malicious behavior from the app before even allowing it into the app store. Let's look at a few proposed solutions for such malware detection systems.

4.1 AASandbox

[Bläsing et al., 2010] proposes a solution called *AASandbox* for detecting malicious Android apps as a cloud service. AASandbox consists of two components — a static analysis tool that scans the binary for malicious patterns, much like a traditional anti-virus tool, and a dynamic analysis tool that runs the application inside a sandbox. This sandbox is implemented using the regular Android emulator usually used for development and debugging.

> *In this paper, we propose an Android Application Sandbox (AASandbox) which is able to perform both static and dynamic analysis on Android programs to automatically detect suspicious applications. Static analysis scans the software for malicious patterns without installing it. Dynamic analysis executes the application in a fully isolated environment, i.e. sandbox.*
>
> —[Bläsing et al., 2010]

Monitoring of the sandboxed application, is performed using a *loadable kernel module* (LKM) which is placed in the kernel inside the Android emulator. Part of the reason for this is that monitoring of both java and native code is desired. Also, putting the monitoring code into the kernel, makes it harder for a malicious app to detect that it is being monitored. This is important, because if it is easy to detect the sandbox, then a malicious app might disable its malicious behavior while being monitored. Since most apps require some user interaction in order to exhibit all of its behavior, the Android Emulator is paired up with the Android MonkeyRunner tool, which is a tool for generating pseudo-random streams of input to the app.

The static analysis tool takes apart the whole APK (the packaged app) by first decompressing it and then decompiling the code using a tool called Baksmali. The resulting set of files are then scanned for suspicious patterns such as: usage of Java Native Interface, usage of `System.getRuntime().exec()`, usage of reflection etc.

Finally, the results from the static analysis is then combined with the log from the dynamic analysis run. Usage of any of the suspicious interfaces does not disqualify an application outright, but the full result is placed in a signature file, which can be compared with the usage patterns of regular apps in order to detect apps with suspicious behavior.

The full AASandbox system may be run as a cloud service, where it may be used for instance by an app store as a part of a pre-deployment verification process. It might also be possible for a local malware detection tool on an Android device, to contact this cloud service in order to trigger a scan of a newly installed app before allowing it to run.

4.2 Mobile-Sandbox

The *Mobile-Sandbox* solution proposed by [Spreitzenbarth et al., 2013] is another hybrid sandboxing solution, which, like AASandbox, combines static and dynamic analysis techniques. This paper, being three years newer than [Bläsing et al., 2010], builds and expands upon the existing work.

The implementation of both the static and the dynamic analysis modules have much in common with the solution we saw used for AASandbox. The static analyzer decompresses, decompiles and scans the resulting files for patterns. It also scans for usage of suspicious functions such as `sendTextMessage()`, which can be used to send SMS messages to premium phone numbers, which can be a way to extract money from the victim. The static analyzer also looks to see if the app requests more privileges than it strictly needs, or if it is underprivileged, which can be a sign that something odd is going on. Finally, the static analysis tool collects a list of all timers and broadcasts the app reacts to. This list will be used during dynamic analysis, to try to trigger any malicious behavior.

> *Within these 40,000 samples our system detected 4,641 malicious applications and additionally 5 suspicious samples which try to hide their malicious action inside native code. This insight clearly indicates that current analysis systems are overlooking important potential threats.*
>
> —[Spreitzenbarth et al., 2013]

The dynamic analysis module uses the regular Android emulator to execute the apps. In order to improve the logging capabilities of the emulator, it has been enhanced with the *TaintDroid* [Enck et al., 2014] and *DroidBox* [Lantz et al., 2012] tools. Unfortunately, these tools build upon the Dalvik Java VM, and are unable to see and log calls from native code. For tracking any native code that may exist in an app, a modified version *ltrace* was used. In order to trigger the malicious functionality that may exist in the sandboxed app, the list of event listeners from the static analysis run is used, along

with the MonkeyRunner tool also used in AASandbox.

In addition to the function call logs from the static and dynamic analysis runs, Mobile-Sandbox also uses the network capture functionality in the Android emulator to create a PCAP file with all the network traffic.

> *Within the Asian set samples that use native code, we found five samples that were hiding their malicious actions inside the native part of their code. When uploading these samples to VirusTotal we got a detection rate of 0%. This again makes clear how important it is to monitor and analyze native code.*
>
> —[Spreitzenbarth et al., 2013]

A set of representative malware from different malware families was collected and used to test the performance of the system. Mobile-Sandbox was able to recognize all of these malicious apps, although at first it seemed as if some of the malicious behavior supposedly existing within these apps was not recognized. The authors realized only later that these missing behaviors were due to missing external stimuli such as botnet control servers etc.

The biggest challenge with the Mobile-Sandbox tool is the runtime performance. Most apps take between 9 and 14 minutes for a full scan on a decently powerful computer. Most of this time is spent interacting with the Android ARM emulator, which is known to be slow. Another challenge is the detectability of the sandbox. There are several values in the system, which are queryable by the app, that reveals that the app is running inside an emulator. While some of these values can be safely changed to mimic any real device, two variables (`ro.kernel.qemu` and `ro.hardware`) were not possible to change without causing the emulator to crash.

An extensive test was done by downloading around 80,000 apps from various Asian markets and analyzing them automatically. This took around 14 days on a single system. 641 malicious apps were detected, which amounts to a malware percentage of 1.78%.

The team behind this paper have continued their work on Mobile-Sandbox with [Spreitzenbarth et al., 2015], where they add machine-learning techniques to further enhance the detection quality of the sandbox.

4.3 PiOS

PiOS is a static analysis tool for iOS apps, presented in [Egele et al., 2011]. Its purpose is to detect privacy leaks, i.e. sensitive data leaving the phone without the user's consent. To this end, the authors behind the paper have developed a set of novel techniques for constructing program control flow graphs (CFG) from binary iOS applications. We will not delve too deeply into these techniques in this paper — the interested reader is encouraged to check out [Egele et al., 2011] for himself.

The main challenge with creating a CFG on iOS, is the way object method calls are handled in Objective-C. These method invocations are performed as a message passed through a central dispatch function, to the instance. This means that resolving the right

destination instance from a method invocation in binary code is non-trivial. The authors were, however, able to find a method that works in the majority of cases. Based on this CFG, PiOS looks for connections between sensitive data, and functions for communicating with the network. Any such connection represents a potential privacy leak.

To verify that a connection between sensitive data and the network really constitutes a data leak, an additional data flow analysis step is done. This analysis is based on *taint tracking*, which is a technique we have already looked at in relation to the TaintDroid tool.

> *We have analyzed more than 1,400 iPhone applications. Our experiments show that most applications do not secretly leak any sensitive information that can be attributed to a person. This is true both for vetted applications on the App Store and those provided by Cydia. However, a majority of applications leaks the device ID, which can provide detailed information about the habits of a user.*
>
> —[Egele et al., 2011]

The act of downloading an app, and extracting it in order to inspect the binary, is not straight-forward on iOS. Apps are downloaded from the App Store in an encrypted form, and the decryption key for the app is downloaded through a separate channel, and added to the secure key chain on the device. In order to obtain access to the decrypted app code, a jailbroken device is required. The method used by PiOS is to download and install apps to a jailbroken device, and then launch the app with a breakpoint set at the program entry point. When the breakpoint is triggered, the unencrypted program code can be copied from memory.

PiOS has been evaluated against a set of 1,407 of free apps from both the App Store and from the third party BigBoss store which is available to jailbroken phones using Cydia. An unsuspected finding was that a large number of apps contained the exact same kind of privacy breach. Many free apps contain a library for displaying advertisements, and it turns out all of these leak the device ID. 55% of the apps tested, leaked the device ID through either an advertisement or tracking library. Another interesting find, was that the apps available through the BigBoss store were not more malicious than the ones available through the App Store, even though the BigBoss store does not do any vetting or verification of its apps.

A leak of address book information was found in a social networking application called *Gowalla*. This app slurps the address book, and sends the entire thing over the network, to the developer of the app. The reason being that they want to cross-check the address list with their user database, to be able to auto-suggest friends for the new user. Several other apps displayed similar behavior, although some did bother to warn the user at some point before doing this.

However, with the exception of the device ID leaks, the vast majority of apps, both in Apple's App store and on Cydia, do not leak sensitive information.

5 Discussion

While we seem quite willing to trust applications on our PCs, and expect them to be allowed to make use of our full range of system permissions, we trust mobile applications much less. We have come to accept and expect mobile applications to be run within restrictive sandboxes in order to limit the damage that can be done if one app should turn out to contain malicious behavior.

This suspicion of mobile apps may certainly turn out to be well founded. Over the past few years, multiple cases of malware have been discovered in both Google and Apple's app stores. Considering how much sensitive information, and how many possibilities for fraud, that can be found inside an average mobile device, we would do well to secure ourselves as best we can. But the existing sandboxes are not completely bullet proof. Although we want to restrict the apps as much as we can, we also need them to be able to access filesystems, communicate over the network and send messages to other apps. In these intentional portals to the world outside the sandbox, malicious apps can find loopholes and vulnerabilities that allow them permissions we did not intend them to have.

> *The vast majority of the malicious Android samples we analyzed were Trojans of one kind or another. Even though most of these don't technically fall in the families explicitly focused on SMS-sending (e.g., SMSSender), almost 83% of the Trojans performed surreptitious SMS-sending anyway, making it by far the most common objectionable activity. One interesting development related to this is the introduction of a notification prompt for SMS messages sent to premium-rate numbers in the 4.2 (Jelly Bean) update for the Android operating system. The user is given the option to allow or block this action - a change likely to put a major crimp in an SMS-sending Trojan's operation.*
>
> —[F-Secure, 2014]

In this paper, we have looked at mobile device sandboxes from three different angles, and we've looked at several pieces of technology for both breaking and strengthening security. Through these papers, we have also gained an insight in the differences between the two largest smart phone platforms, which are Apple's iOS and Google's Android operating system. We have seen that Apple enforces a tighter control over both its app store and over its devices, which Google's Android system is a more open system. It is hard to speculate whether this is why most of today's mobile malware is aimed at Android, but it could very well be a part of the reason. However, the openness of the Android system also leads to easy development of new software, and we've seen that there is a plethora of propositions in the form of scientific papers describing novel new ways to strengthen the security.

Of course, neither Apple nor Google are standing passively on the sideline in this

battle. The platform security is a never-ending arms race between the system developers and the criminals, and many of the ideas in these papers may be already adopted by Apple or Google in some form, or they may be obsolete. One particular event that may have rendered some of these papers obsolete, is the release of Android 6.0 (Marshmallow). Many of the older papers above have complained about the old permission system in Android, where users were given a huge permission list to grant at application install time. Since there was no fine-grained control over the permissions, it was not possible to accept some and deny others. Denying the permission grant request meant simply that the app would not be installed at all. The new solution in Android 6.0 forces apps to request permissions as they are about to be used, which should give users a better chance to make an informed decision of whether to accept or deny. It is also possible to accept some and deny other permissions to an app.

This war is by no means over, and it is certain that the future will bring both inventive new malware, and amazing security products. For the time being, it is a good idea to watch one's step in the world of mobile devices. There is plenty of malware to be found in the app stores, but there are also promising security solutions that can help us defend ourselves. The author is personally keeping an eye open for any development on Boxify [1] which looks like it may develop into a real product.

References

[Apple, 2016] Apple (2016). ios technology overview. https://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/Introduction/Introduction.html, Accessed February 22th, 2016.

[Backes et al., 2015] Backes, M., Bugiel, S., Hammer, C., Schranz, O., and von Styp-Rekowsky, P. (2015). Boxify: Full-fledged app sandboxing for stock android. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 691–706.

[Bläsing et al., 2010] Bläsing, T., Batyuk, L., Schmidt, A.-D., Camtepe, S. A., and Albayrak, S. (2010). An android application sandbox system for suspicious software detection. In *Malicious and unwanted software (MALWARE), 2010 5th international conference on*, pages 55–62. IEEE.

[Bucicoiu et al., 2015] Bucicoiu, M., Davi, L., Deaconescu, R., and Sadeghi, A.-R. (2015). Xios: Extended application sandboxing on ios. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 43–54. ACM.

[Davi et al., 2011] Davi, L., Dmitrienko, A., Sadeghi, A.-R., and Winandy, M. (2011). Privilege escalation attacks on android. In *Information Security*, pages 346–360. Springer.

[1] https://infsec.cs.uni-saarland.de/boxify/

[Egele et al., 2011] Egele, M., Kruegel, C., Kirda, E., and Vigna, G. (2011). Pios: Detecting privacy leaks in ios applications. In *NDSS*.

[Enck et al., 2014] Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. (2014). Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5.

[F-Secure, 2014] F-Secure (2014). Mobile threat report q1 2014.

[Felt et al., 2011] Felt, A. P., Wang, H. J., Moshchuk, A., Hanna, S., and Chin, E. (2011). Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium*, volume 30.

[Google, 2016a] Google (2016a). Android dashboards. http://developer.android.com/about/dashboards/index.html, Accessed February 20th, 2016.

[Google, 2016b] Google (2016b). Android security. https://source.android.com/security/, Accessed February 20th, 2016.

[Lantz et al., 2012] Lantz, P., Desnos, A., and Yang, K. (2012). Droidbox: Android application sandbox.

[Lee et al., 2014] Lee, B., Lu, L., Wang, T., Kim, T., and Lee, W. (2014). From zygote to morula: Fortifying weakened aslr on android. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 424–439. IEEE.

[Liebergeld and Lange, 2013] Liebergeld, S. and Lange, M. (2013). Android security, pitfalls and lessons learned. In *Information Sciences and Systems 2013*, pages 409–417. Springer.

[Spreitzenbarth et al., 2013] Spreitzenbarth, M., Freiling, F., Echtler, F., Schreck, T., and Hoffmann, J. (2013). Mobile-sandbox: having a deeper look into android applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1808–1815. ACM.

[Spreitzenbarth et al., 2015] Spreitzenbarth, M., Schreck, T., Echtler, F., Arp, D., and Hoffmann, J. (2015). Mobile-sandbox: combining static and dynamic analysis with machine-learning techniques. *International Journal of Information Security*, 14(2):141–153.

[Wang et al., 2013] Wang, T., Lu, K., Lu, L., Chung, S. P., and Lee, W. (2013). Jekyll on ios: When benign apps become evil. In *Usenix Security*, volume 13.

[Xing et al., 2015] Xing, L., Bai, X., Li, T., Wang, X., Chen, K., Liao, X., Hu, S.-M., and Han, X. (2015). Cracking app isolation on apple: Unauthorized cross-app resource access on mac os. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 31–43. ACM.

[Zhou et al., 2015] Zhou, Y., Patel, K., Wu, L., Wang, Z., and Jiang, X. (2015). Hybrid user-level sandboxing of third-party android apps. *Memory*, 2200(0500):0e00.