A Case Study in Building Security into “Off-The-Shelf” Smartphones

A. Stavrou (GMU)
J. Voas, T. Karygiannis, S. Quirolgico (NIST)

Recent advancements in hardware have increased the computing power, memory, storage, and wireless connectivity of handheld mobile devices. Smart-phone devices are used for everyday activities that range from maps and geo-location tagging to banking. Indeed, these new hand-held devices are capable of carrying significant amount of both personal and professional data including documents thus extending the operations that we can perform from desktop to smaller devices. Unfortunately, this reliance on hand-held devices has made them an attractive target for applications and new mobile application markets have been spawned for the different types of devices. At the same time, these new devices have become the target of malicious attackers that have shifted their attention from desktop systems to malware and malicious software for hand-held mobile devices.

Most mobile smart-phone devices today are equipped with a phone, web browser, music player, camera, and a wide range of other applications and services. Google Android, NeoFreeRunner, Nokia Maemo, iPhone OS, and Windows Phone operating system are some noteworthy hand-held device platforms capable of performing most of the functions previously found only in full-fledged desktop operating systems. Usability of such devices is further enhanced by the availability of third-party applications that can be purchased or freely downloaded by users from online application stores (appstores) or developer websites. This possibility for greater functionality and convenience, however, also exposes the user to a greater risk from malicious software programs. While mobile applications that can be downloaded from un-trusted third-party sites are commonly regarded as the main source of such malicious software, security risks can also come from vendor-certified app stores, as it is difficult for the vendors to thoroughly test thousands of applications whose behaviors could potentially be made time and location dependent. To address time and location dependent behaviors, a combination of operational, field testing, along with in-laboratory app testing for specific well publicized security flaws, serves to better substantiate the argument that an app has received sufficient scrutiny to enter an appstore. Further, even the built-in applications that come pre-installed on wireless mobile devices may contain security flaws that still need to be addressed. It is not enough to only vet the apps that are downloaded from the marketplace.

Hand-held wireless devices are different than conventional computing platforms, such as desktop computers, in several aspects. Firstly, such devices have very limited resources in terms of computing power, batteries, memory, to name a few. A faulty or malicious program targeting these weaknesses, can easily abuse the unregulated access to the battery through excessive use of CPU or radio communication. Secondly, hardware parts are tightly integrated, which makes it difficult to identify the source of an abnormal condition. For instance, if a program is able to increase the GPS intervals of communication, it is not easy for the end-user to identify the cause of power consumption and take action. Third, due to mobile nature of such devices, and lack of built-in protection against malicious software through the underlying operating system or through third-party anti-virus programs, malicious programs are more easy to spread and cause damage before the user is aware of their presence.

The danger of malicious applications making it to the official markets is increasing as more and more developers from unknown sources upload their applications. During the first week of September 2011, the Android market had more than 276,000 Android applications while Apple's application store provided Apple users with access to almost double as much: a staggering half a million applications. A majority of these applications are freely available or supported by mobile advertisements. As a result, the users are presented with a plethora of applications to choose from satisfying their needs for a mobile office to casual entertainment. Most of these applications let the user download a free, trial version before they have to purchase the full version enabling the user to get a "taste" of the product before making a decision.

The federal government and the military is trying to take advantage of this successful and agile model of mobile computing which offers access to thousands of existing applications and small but powerful handheld devices for both civilian and tactical environments. The end goal is to leverage existing off-the-shelf hand-held devices retrofitted with software and hardware accessories that will enable them to meet the security and operational requirements for a wide-range of use cases.

Over the past eighteen months, researchers from NIST and George Mason University (GMU) have been working under the auspices of a DARPA project to understand the security risks and challenges of maintaining a secure supply chain of "off-the'shelf" mobile devices and software. It was clear from the beginning that, to be able to enforce any policies on these devices, the researchers had to focus on two major areas: vetting the mobile applications and strengthening the security of the mobile device. In this article, we will focus on our initial efforts towards building a comprehensive mobile-app vetting service.

To enforce policies for mobile security and software reliability, the application market model that was introduced by Apple and later adopted by Google was appealing: unlike the existing PC application purchase and distribution process, it allowed for a single entry point to a web portal where both application developers could be authenticated and their submitted application code tested. To achieve that, NIST and GMU designed a web portal where applications can be tested, analyzed, and rated in terms of code vulnerabilities, functionality risk, and power consumption before being approved by the individual civilian or military agencies for their use.

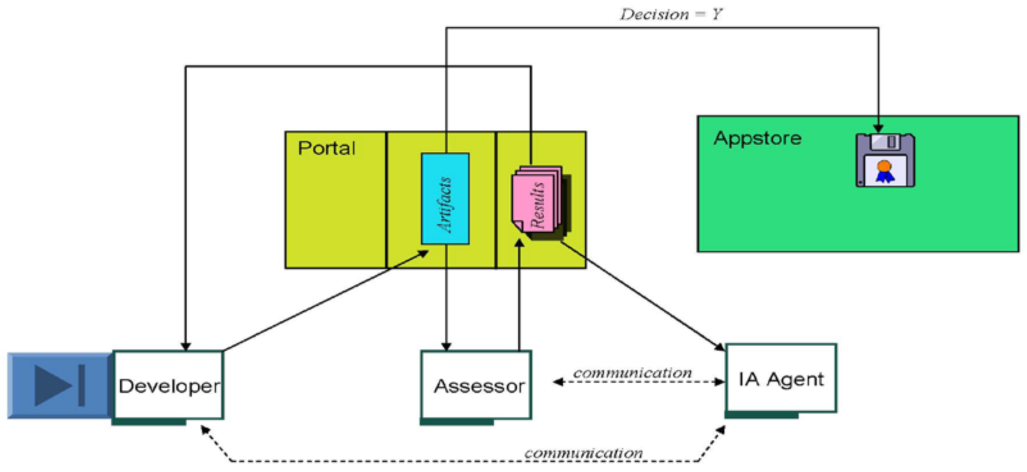

Figure 1: **Application testing as a Web Service; the Developer submits *Artifacts* (e.g., app source code, app binaries, app libraries, app documentation, Developer documentation) to the application testing Portal. If the analysis *Results* from the *Artifacts* are satisfactory (*Decision = Y*) to the IA Agent (with possible verbal communications from the Assessor), the app is eligible for incorporation into the application store (Appstore). Otherwise, information in the *Results* indicating potential security or other behavioral risks is returned as feedback to the Developer, and the app is not yet eligible for incorporation into the Appstore. (Dotted lines indicate communications that may or may not occur.)**

In contrast to the existing commercial app markets, when the developer submits their source code or application package, the application portal has the capability of providing a detailed analysis of the functionality of an application as a result of the testing tools connected to the portal that perform various security and reliability analyses. This means that a mobile-app assessor or buyer can receive various forms of risk analysis results that an application may present to users. The researchers on this project made a conscious choice to focus on apps for which source code is readily available due to the more thorough risk analysis that can occur, however we understand that such access may not be granted for other organizations who wish to vett third-party apps. Part of the reason for this choice was that the Android Dalvik engine uses Java syntax and libraries, and Dalvik bytecode can be readily decompiled back to source code using tools that are publicly available including apktool among others. Further, the Android platform was the platform selected by the agency that funded this work.

The Android Operating System (OS) permits applications to invoke native code, by calling existing library functions through the Java Native Interface (JNI) framework, the native code is forced to execute inside the Davlik VM (Virtual Machine). This results in the smartphone’s security framework still being applied even though native code is being run. Moreover, our analysis can detect and prevent all JNI calls removing dependencies to native code that can be a potential security risk.

An informed reader might be wondering why not just apply the native operating system and Dalvik enforced permissions?

Indeed, Android smartphones use a permission-based model to limit the behavior of an application and to inform the user of the application’s potential behavior. An application declares the permissions that it uses in its AndroidManifest.xml file. The user is presented with the list of permissions that an application uses when the application is to be installed. The user gets to make the choice whether or not to install the application based on the list of permissions it requires. The user cannot selectively allow or disallow individual permissions that the application requests. The installation of the application is done on an all-or-nothing basis in regard to the permissions. Once an application is installed, the permissions that it has remain static.

The AndroidManifest.xml contains information about the application's permissions, broadcast receivers, intent filters, activities, required features, content-providers, libraries the application uses, and any instrumentation the application uses. These high-level attributes form a profile of the application. A static analysis program could examine this profile and determine if the actual behavior of the program is in accord with its AndroidManifest.xml file. This verification process can help to vet applications.

The use of the important resources on these smartphones is controlled by the use of permissions. For example, to use the GPS resource on phone, the application must possess the ACCESS_COARSE_LOCATION permission or the ACCESS_FINE_LOCATION permission. If an application requests the GPS resource without having the appropriate permission, then the operating system may throw a Security Exception or simply not grant the resource, which was requested. These permission-protected resources are accessed through the Android API (Application Programming Interface). There exists a mapping for permission to the API calls that can be used as a result of having the corresponding permission. Having the ACCESS_FINE_LOCATION permission in an application’s AndroidManifest.xml file will give the application access to a number of API calls that use the GPS resource.

However, developers tend to require more permissions than necessary for their mobile applications. This might be due to their lack of understanding for the existing policies and permissions, due to mere convenience, or in expectance of added functionality in the future. In all cases, the result is the same: the application has a loose security model which renders it vulnerable to local and remote attacks including reflection type of attacks that can use the permissive application as a stepping-stone. We have identified thousands of applications built for these smartphones that request to be granted permissions that are excessive and do not correspond to functionality the app implements or requires.

Furthermore, the all-or-nothing permission model of an application's requested permission list is restrictive and does not take into consideration environments where the permissions for an application might dynamically change based on context. We have been considering methods for allowing certain permissions while denying other permissions. The choice of which permissions are acceptable would be at the discretion of the user. This would be done by redirecting certain API calls so they would be handled in a special way. For example, if the application requests network access and the user does not want to grant network access, then the OS could inform the application that there is no network access even though network access is available and the application has the INTERNET permission. In addition, undesired access to the contact list, via the READ_CONTACTS permission, could be redirected to a contact book full of false entries. This redirection to contrived resources allows a user to selectively allow or disallow individual permissions that the application requests.

In addition, it is possible to modify the source code to achieve a finer-grained security policy in regard to permissions. Extra permissions can be integrated in the operating system and the Dalvik VM by modifying the source code in order to add in extra permission checks. This would be fairly straightforward to do, and it would deliver tighter control over the current default permission model. This granularity can allow an organization to develop and enforce a custom security policy. Certain organizations may require a very stringent security policy, which is based on their operational requirements. This model can afford granularity or coarseness since it is also possible to delete the permission checks that occur in the java files of the source code.

Furthermore, static and dynamic application analysis currently performed at GMU and NIST can identify excessive privileges and hidden functionality generating feedback reports with the appropriate level of details to the developer allowing for program fixes based on the findings. These analyses are based on a variety of commercial and proprietary tools. We believe that by quantifying the applications' functionality, and by enforcing a more fine-grained permission model, we can assist an analyst (or in some cases, the end-user) to make an informed decision about the risk that an application might pose to her phone and personal data. Our approach can identify and thus thwart a large class of malware. Of course, this is not a panacea or a security "silver-bullet" but it a step closer to having better security and software reliability for mobile devices.