https://doi.org/10.48009/1_iis_2011_301-312

Issues in Information Systems

Volume XII, No. 1, pp. 301-312, 2011

COMPARATIVE ANALYSIS OF MOBILE APPLICATION DEVELOPMENT AND SECURITY MODELS

Joseph Packy Laverty, Robert Morris University, laverty@rmu.edu
David F. Wood, Robert Morris University, wood@rmu.edu
Frederick G. Kohun, Robert Morris University, kohun@rmu.edu
John Turchek, Robert Morris University, turchek @rmu.edu

ABSTRACT

While mobile application programming languages are based on traditional programming languages, there are differences in the methods that a mobile application is development, tested, deployed, and secured. Even when mobile platforms are based on similar programming languages, mobile platforms vary in the security model used to protect mobile devices from malware and other security attacks. This paper compares Android v 2x/3x, iOS 4x, Windows Phone 7, and Blackberry v6/7 application development and execution from a security perspective. Alternative mobile application development approaches are compared using the following criteria: a) application frameworks and programming language, b) native code support, c) development, packaging and deployment. Mobile application security models are also compared using these criteria: application sandboxing (isolation), application signing, installation, and application security permissions.

Keywords: Android Security, iOS Security, iPhone Security, Windows Mobile Security, Windows Phone 7 Security, Blackberry Security, Application Signing, Native Code

INTRODUCTION

Career opportunities for mobile application developers continue to grow. At the end of 2010 Dhanjani (2011) reported that over 100 million iPhones were in use, 15 million iPads , and over 10 billion mobile apps were downloaded [14, p.1]. Mobile application platform applications are developed using traditional programming languages, which have been modified for the requirements and restrictions of mobile platforms. Mobile platforms permit one active application, one application window, no user security, limited keyboards and display sizes, and limited hardware resources [16, pp. 3-8, 10, pp. 3-9,]. Due to these limitations, platform-specific, specialized mobile frameworks have modified traditional programming languages, e.g. Java, C/C++/C#, and VB.NET. For example, a mobile programming language is called Android, not Java.

While mobile application programming languages are based on traditional programming languages, there are differences in the methods that a mobile application is development, tested, deployed, and secured. Even when mobile platforms are based on similar programming languages, mobile platforms vary in the security model used to protect mobile devices from malware and other security attacks. This paper compares Android v 2x/3x, iOS 4x, Windows Phone 7, and Blackberry v6/7 application development and execution from a security perspective. Alternative mobile application development approaches are compared using the following criteria: a) application frameworks and programming language, b) native code support, c) development, packaging and deployment. Mobile application security models are also compared using these criteria: a) application sandboxing (isolation), application signing, installation, and application security permissions.

Android, an open source operating system and programming language, uses a component-based architecture that permits inter-process communication between applications. This component-based architecture is innovative and requires a more sophisticated security model. The availability of open source information permits a more detailed review of the Android mobile platform compared to other mobile platforms.

ANDROID APPLICATION DEVELOPMENT AND SECURITY

Android system is an open source operating system based on a modified Linux kernel. Initially created by Google and the Open Handset Alliance in 2005, maintenance and future releases will be provided by Android Open Source Project (AOSP). The Android operating system is written in three programming languages, e.g., C, Java, and C++, and is configured by million of lines of XML code. In early 2011, Android 3.1 was the current version, while 66% of Android mobile devices were based on Android version 2.2 [21].

Android Applications Technical Environment

Android mobile applications may be written in Android (Java), C/C++ (NDK), and J2ME. It is important to distinguish between Java and Android as an application programming language. Android is a Java-based programming language, but is not fully compatible with Java SE or J2ME standards or applications. While Java core libraries are used, Android provides additional APIs to integrate with the Android operating system, platform resources, and security model. APIs, Application Program Interfaces, are additional pre-written source code, often packaged in an application Framework, that may be used by application developers. In addition, the Android SDK and development tools includes a run-tine environment based on the Dalvik Virtual Machine, and JIT compiler, an application debugger, mobile device emulators, and compiled program libraries to provide for browser engine, SSL, SQLLite, media players and other services [21; 16, pp.. 15-19]. Android Java-based mobile applications may be developed by using: a) Android SDK plug-in installed in Eclipse, or b) command line text editor, Apache ANT and the Java Developments Tools (JDT), or other third-party IDEs [12, pp. 3-15; 3, pp. 17-18]. Windows, Linux and a MAC OS X platforms may be used to develop Android applications.

Android Framework

Developing an Android mobile application starts with the creation of a "project". A project is a hierarchical directory (folder) storage system (also called a namespace) that organizes one or more source code class files, compiled dex code files, and other application and development resources, e.g., graphics, required by the Android application. Each source code file is compiled into dex-code (intermediate code), executed and debugged[12]. Java applications compile, using javac.exe, to byte-code format compatible with Java Virtual Machine run-time environment. However, the Android framework compiles applications into a different type of intermediate code format, named dex-code. Dex code is compatible with the Darvik Virtual Machine run-time environment. Application Virtual Machines advantages include: isolation from other applications and operating system, b) execution compatibility with on multiple operating systems, c) automatic memory protection and garbage collection, d) and enforcement of security policies on a per-application basis [15].

Dex Code

Dex code is faced with the same security concerns as any intermediate program code, e.g. Java byte-code, .NET, MSIL code, etc. Intermediate program code is easy to be reversed engineer or disassemble. A reverse engineering attack means that hackers may view security mechanisms and attributes of mobile application source code. Code obfuscation is a technique that purposefully changes instruction paths and symbolic information to prevent successful reverse engineering attacks [16, p.164]. To provide limited protection Android provides a command line code obfuscation tool named dexdump [15].

Managed versus Native Code

Mobile application programming languages may be classified as either "managed code" or "native code". Managed code provides compilation and run-time environments that are type-safe, prevents overflows and code insertion attacks. Native code applications are normally written in C or C++. Exploitation of native code vulnerabilities may cause memory corruption, memory leaks, application crashes, breach of virtual machine protection, or operating system failures [29].

Android supports native code applications. CPU-intensive applications requiring high-performance, e.g., gaming, may benefit from native code applications. Android provides a Native Code Development Kit plug-in for Eclipse. Specialized native code protection APIs are provided to prevent common vulnerabilities. Native code application development is limited to user-developed libraries (compiled code) and has no access to the Android Java APIs. This means that an Android native code application has no user interface, nor can be used as a content provider or a service. Native code executable libraries may be invoked by a SDK Android application using JNI (Java Native Invocation) [25; 12, p.19].

Android Testing

An Android application may be tested with a mobile device emulator called an Android Virtual Device (AVD). Since there are many versions of the Android operating system, an AVD must be created for each minimum Android targeted version. An AVD permits testing on the development operating system, e.g., Windows, without the use of a physical mobile device. Eclipse also permits testing on USB connected phone.

Deploying Android Applications

To deploy or install an Android application it needs to packaged into an APK file. The package name is the unique identifier used for the Android application. An APK file is ZIP compressed files that contains the necessary contents of your Android project (application). The APK file is very similar to the Java JAR file. An APK package contains the AndroidMainfest.xml file, dex files, and application resources [15]. You may deploy (install) an Android application: a) through the Android Market, b) from within the Eclipse IDE using an USB connection and the "adb install" command line utility, or c) download an installation (adb) script and ADK package [20, 5]. During installation, the user may be informed or warned that the application requires access to protected Android applications, e.g., the browser, email client, camera, etc. Once approved and install the Android security model does not change. This presents a challenge to mobile enterprise systems, e.g., BES and Microsoft ActiveSync, which want to control security policies remotely.

Component-based Architecture

Android, like Java, is a component-based architecture. A Windows Mobile application is designed be self-contained and does not rely on any other third-party application. Component-based software architectures subdivide applications into independent modules, each having a single functional purpose. The ability of one application to use the services of another application is fundamental. Software components are designed to be reusable by other applications. An Interface Description Languages specify the rules that components may use to share data, e.g., an intent or message [16, p21].

GUI Interfaces

A Java application may one or more GUI user interfaces called Frames, e.g., extending the JFrame classes. Frames are not used in Android. Android applications may have one or interactive user interfaces called Activities, e.g., extending the Activity class. An Android Activity may be used to display a menu, game board, video or anything else that can be displayed. When an Android application starts the user interface, or Activity, the user interface is not automatically displayed. Individual applications do not manage what appears on the mobile device screen, the Activity Manger of the Android operating system does. To start (or display) an Activity (user interface) a request must be sent to the Activity Manger requesting to display the Activity using the startActivity method. The next question is "How does the Activity Manager know what screen is to be displayed?" Of course, the application developer tells it [16; pp. 20-28; 12, pp. 22-28]. An Intent is a specially formatted Android message, transmitted in clear text, between applications or Android framework components. In the previous example, a application developer would construct an Intent that will inform the Activity Manager which user interface to start, e.g., startActivity(myActivityIntent).

Activity Classes

An Activity class will normally contain the methods that react to events detected by Activity Manager, e.g., button-click, mouse over, etc. The event will generally trigger the execution an assigned method, e.g. to calculate, play music, retrieved stored data, etc. Assume that the application wants to start second Activity, e.g., display another screen. Mobile devices are limited to displaying one screen at a time. What happened to the state information from the original Activity when there was a switch to the second Activity, e.g., a content that you typed into a document? The Activity Manager maintains the state information even though the screen is not visible [16, 21-27; 12, p.18; 8].

Content-based Architecture

Content-based architectures also permit one application to use the resources of another application, e.g., another application's Activity. In this context, a mouse-click event in the first application's Activity may start or change the state of a different application's Activity. Assume that the second Activity now starts the Activity of a different application, e.g., three Activities are active. At this point, the Activity Manger is maintaining state and visibility for two screens of the first application and a third screen of a different application. As one's click on the forward and back buttons of your Android device, the Activity Manager must manage state as one switches between multiple screens and applications. We are accustomed to the forward and backwards navigation used by Internet Browsers, however, browsers do not manage the state of a web page not does it perform a context switch between applications.

Android Application Security

Android application security is a collaborative effort involving: a) application security APIs, b) Android Framework Components, e.g., Activity Manger Content Providers, Package Manager, etc., and c) the AndroidMainfest.xml file contained in the APK package. The AndroidMainfest.xml file is automatically created in the root directory of the

project when it is created. The XML formatted content of the AndroidMainfest File may include: a) the name of the application package and protection level, b) names of components (classes) within the package, e.g., activities, etc., c) internal and external security permissions necessary the execute components, d) the pre-compiled libraries that the application requires, and e) the minimum level of the Android API, e.g., v2.1, v2.2, etc., that the application requires [4, 16 pp. 23-27].

Security Permission

Assume that an application contains an Activity, named "MyBabyPicture", which will display your baby picture, This application also contains a second Activity named "MyGraduationPicture." The AndroidManifest must specify the list the name of each Activity and other secured components. Android security permission may or may not be assigned to the Activity. An Android security permission is a unique string associated with an application component, e.g.,, the MyBabyPicture Activity. An Android permission is simply a password that is required to call the Activity. When starting the Activity that has a security permission, the Intent must include: a) the name of the calling component, b) the name of the activity, and c) the permission string (password). Besides explicitly specifying the Activity name and assigning security permissions, an application developer has the option of using an intent-filter or exported attribute. Intent-filters and registers (publishers) act as alternative component identitities to the Activity Manager. A calling application using Intent-filer APIs may use this alternative component identity to access other components and applications. Intent-filters offer no security protection. If a component is exported, any application that has knowledge of your component and package name, can call the Android component if it provides the appropriate permission. To prevent other applications the use of an Android component, do not export your component. To secure access to applications and components designate security permissions for the component [16, pp. 22-30].

Android Framework

The Android Framework provides for other secured components and mechanisms. Service components operate in the background, e.g., listening for instant messages, playing music, etc. All android service names, permissions, and intent-filters, must be specified in the AndoridManifest.xml file. A service component may be started by any application using the same security rules previously specified for Activities [12, pp. 241-243, 16, p. 34, 9].

Persistent and Non-persistent Data

When an Android application stores non-persistent and persistent data, it may wish to share that data with other application components. Since Android applications are stored in separate directories, data stored in files is not accessible by default by other applications. Android provides two secure mechanisms to share data: s) Inter-Process Communication (Binder), and b) Content Providers.

Binder Interfaces

Data may be shared between two applications using the Binder Interface. The Binder Interface is Android's secure IPC mechanism which may be used by Android apps and native code libraries. The Binder Interface may also encrypt the contents of an Intent message and to determine the identity of the calling application. Content Providers are components, which are registered by the Activity Manager and makes data available to any component that provides the appropriate security permission. Content Provider data is organized in a relational format and individual rows can be accessed using pseudo SQL mechanisms. Additional AndroidManifest.xml Content Providers attributes may designate restrictions for reading and writing data. Do not confuse Content Providers with a stored database. Content Providers are a secured mechanism to share global data between multiple components [12, pp. 185-186; 16, pp. 35-36, 9].

Broadcast Receivers

Android's Broadcast Receivers are a type of component that listens for broadcasted messages. Broadcast Receiver use Intents to pass content between component Broadcast Receiver components are identified and secured in the AndroidManifest.xml.. After receipt of a broadcast, a Broadcast Receiver may perform some function. However, there are two security concerns associated with Broadcast Receivers: a) Does the Broadcast Receiver want to receive the Intent, and b) Does the sending component want to send the Intent to that Broadcast Receiver? To prevent unwanted messages a sending component must specify the Broadcast Receiver's permission in the Intent, else the Broadcast Receiver will not receive the Intent. The Broadcast Receiver will know the identity of the calling component by investigating the content of the intent. But, how will the broadcasting component know that only properly secured Broadcast Receivers are listening? If a Broadcast Receiver specifies no security permission, it can

listen to all broadcasts. To protect from unauthorized Broadcast Receivers, a receiver Permission may be required by the Broadcast Receiver in the Androidmaifest.xml that provides sender's password. This receiver permission password is required to listen [16, pp., 32-34; 9].

Signing

To deploy an Android application on a mobile device, an Android application must be packaged and signed. Signing an application is a two-part process: a) generate a private key and digital certificate, and b) signed the application with the private key. To sign an application you will need a digital identity, which is a private cryptographic key plus a digital certificate. Unlike other mobile platforms, Android permits self-signed applications for deployment. Other mobile operating systems may require digital certificates issued by the vendor, e.g., Microsoft, Apple, etc., or vendor certification before deployment. Signed application can also be used process to protect the intellectual property of the programmer and confirm that the original code has not been altered. But digital certificates and code signing, by itself, does provide protection from malware [19, pp. 412-414; 5].

APPLE IOS APPLICATION DEVELOPMENT AND SECURITY

The Apple iOS, formerly called iPhone OS, was based on the MAC OS X Tiger (Motorola-based processors) and MAC OS X Leopard (Intel-based processors) operating systems [16. p.51]. In 2011, the current version of Apple iOS was version 4.3. iOS applications are written in Objective-C, using Cocoa API and Touch Framework [30]. Apple mobile apps interact with four layers of the iOS operating systems: COCA Touch layer (user interface), Media Services layer, e.g., built-in camera, audio, and graphic applications [28], iOS Core Services layer, e.g., network, foundation, messaging, and cut-paste services [13], and iOS Core OS layer [27], e.g., memory and process management, file systems, security, etc.

Apple Xcode

One can download Apple's Xcode IDE v 4, which includes the iOS SDK and Interface Builder, at http://developer.apple.com/xcode/[32]. While Objective-C applications are used by both iOS and MAC OS X Tiger, there is one significant difference. The iOS version of Object-C does not support automatic garbage collection (automatic memory management), which means that application programmers must still use traditional coding methods, e.g., destuctors and finalization, to reclaim memory and prevent memory leaks [16, p. 51; 24].

Based on the C programming language, Objective-C is susceptible to the classic C vulnerabilities, e.g., buffer overflows, integer overflows, and format string attacks. The iOS SDK and Cocoa provides high-level APIs to reduce, but not eliminate, these classic C vulnerabilities from various attacks and exploits. C functions such as strcat() and strcpy() are the most often target of buffer overflow attacks. Buffer overflows occur when data is written to a fixed size memory spaces, overflowing around the destination location [16, 56-59].

Cocoa

Cocoa provides a family of functions called "strl" which provides protection from buffer overflows. Numeric and Integer Overflows are problems associated with storing a number too large for the allocated space. Similar attacks occur when one attempts to access an array beyond its allocation size. To protect from the vulnerabilities, Cocoa provides for classes, e.g., NSInteger, NSNumber, NSString, etc., which eliminates attacks which tries to write to the stack while an application is running, but provides no protection from reading from stack. Not all C vulnerabilities are prevented by using more secure libraries. Development of iOS code requires proper C coding protections, e.g., testing limits of storage and arrays before access, specifying how user input of strings should be formatted, and properly and consistently releasing objects created by the alloc() method (Double-free problems) [16, 56-59]. However, Xcode does include the Clang Static Analyzer tool to detect specific Objective-C vulnerabilities. In addition, Apple's certified signing process also verifies the code before permitting the code to be distributed in its App stores [16, p. 61].

Jailbreaking

"Jailbreaking" and "Carrier Unlocking" has been used to permit unsigned iOS code to be executed on an Apple mobile device and to permit service connections to other mobile carriers. Cydia, an unauthorized iOS application installer, has been distributing unapproved applications in competition to the Apple app store. While unsigned code may not be malicious, Apple's signing process does provide some assurance that classic C vulnerabilities have been addressed [16. p.51].

Application Developer Accounts

Before one may deploy iOS applications and download Xcode, one must sign up for a for an Application Developer Account. Development of an iOS app using Xcode begins by specifying the name of a project. Similar to other application development environments, an iOS app project is folder structure which contains subfolders named: Classes, Other Resources (non-Objective-C code), Resources (images, audio, xib files, etc.), Frameworks, and Products (the final compiled application). After an iOS application is signed and compiled, a deployment file using Project name appended with an .app file extension [24, pp. 367-401].

Plist Files

iOS projects contain a file called a plist file, e.g., MyApp.plist. A basic plist file contains the basic properties of the application, e.g., Bundle Name and Bundle Identifier (unique name of the application), version and mobile platform information. The plist file may contain other customized settings which will be presented to user either when the application is first started or every time an application is launched. Protected iOS applications the browser, camera, phone, Internet, etc. This notification is similar to the process of installing Adroidmanifest.xml file, except Android checks for permissions to access protected resources during installation [24, pp. 367-401].

Application Testing

iOS applications may be tested on iPhone emulators. To test and deploy iOS applications to a real device requires a digital signature. One must start by obtaining a development digital certificate from Apple. Using Xcode one first creates a Code Signing Request (CSR). The CSR contains developer and key information and is submitted the iOS Provisioning Portal. After the developer certificate has been approved, it is downloaded and added to the Keychain, a secured storage area for digital certificates and keys. To sign an application a provisioning profile must be created. A provisioning profile associates one or more development certificates, devices, and an app ID (iOS application ID). "To be able to install iOS applications signed with your development certificate on a device, you must install at least one provisioning profile on the device. This provisioning profile must identify you (through your development certificate) and your device (by listing its unique device identifier). If you're part of an iOS developer team, other members of your team, with appropriately defined provisioning profiles, may run applications you build on their devices." Developers may use either standard (up to 100 free certificates) or developer credentials ($99 per year) programs [22].

Signing

All iOS applications must be signed to be deployed on an Apple mobile device must be signed (except for Jailbreak devices). All iOS applications must pass a certification process before they may be distributed through the Apple App store. Enterprise applications may be distributed by Microsoft's Exchange with Active Sync. Signed applications can verify the identity of the application author, provide non-repudiation, protect intellectual property, and confirm that the original code has not been altered. But digital certificates and code signing, by itself, does not provide protection from malware.

WINDOWS MOBILE/PHONE 7 APPLICATION DEVELOPMENT AND SECURITY

Windows Phone 7 operating system is based on the Windows CE 6.2 kernel. Older Windows mobile device are based on older 5.x kernels. Windows CE 6.2 kernels supports dual core processors and 3G of RAM [12]. the most significant difference between the versions is the support of Silverlight, Microsoft's multimedia application platform similar to Adobe's Flash platform. User authentication and file permissions are not supported by Windows CE. Sensitive stored application data should be encrypted. The Windows registry, which stores device and application configuration data, is accessible by all applications [16, pp. 144-155].

Development Environment

Windows mobile applications may be developed using Visual Studio or Visual Studio Express. In addition to the Visual Studio IDE , Silverlight for Windows Phone, Microsoft Expression Blend (Design Screen), the XNA Game Studio 4, and Windows Phone Emulator may be used for the development of Windows mobile applications. A free version can be acquired at http://developer.windowsphone.com/windows-phone-7 [19, pp. 15-18]. Prior to 2010, the only mobile programming language supported by Visual was C#. VB.NET is currently supported. Window's Mobile recommends three main security protection technologies: StrSafe.h, IntSafe.h and Stack Cookie Protection when developing native code to be used on of Windows CE 5.x devices [16, pp. 100-104; 101]. As of early 2011, no native code support is available for the newer Windows 7 Phone Mobile .

Security Boundaries
Windows 7 Phone does not use a virtual machine, but uses security boundaries to isolate individual processes and to assign execution privileges[31]. Isolation does not mean a corrupt app can not bring down another app. There are four major security boundaries:

Trusted Computing Base (TCB) – TCB processes to have unrestricted access, can modify policy and enforce the security model, run kernel and kernel-mode drivers run, processes should be minimized - greatest security risk.

Elevated rights Chamber (ERC) – ERC are intended for services and user-mode drivers to be shared by ALL applications. It can access all resources except security policies.

Standards Rights Chamber (SRC) – Pre-installed Microsoft applications that do not provide device wide services, e.g., an email application.

Least Privilege Chamber (LPC) – All non-Microsoft applications (but must be signed.) The Least Privilege Container has 32 slots with an allocated size up to 32MB. This means that there is a limit of 32 concurrent applications.

Self Contained Applications
Windows 7 Phone does not support component based applications. Unlike the Android's security model, a LPC-type third-party application can not re-use components from, or call, another LPC application. A third-party application must be self-contained. Data between LPC applications may be shared by using isolated storage, e.g., files [19, p. 417]. However, a LPC component may use applications and services provided by higher level chambers, e.g., email applications. Higher level applications, e.g., TCB, ERC, and SRC applications, are provided by Microsoft or authorized vendors. Therefore, Windows Mobile provides no application permission system [31, 16].

Solutions
Using Visual Studio IDE, development of a Windows mobile application begins with creating a "solution", or project. A solution is a hierarchical directory structure that contains the source code, MSIL code, resources and configuration files. After testing and debugging a mobile application using various Windows Mobile Phone Simulator an application built into one or more assemblies and packaged for deployment. Windows Mobile Phone appls may be packaged using either the CAB er XAP file format. CAB files are used to deploy older Windows Mobile CE 5.x applications and XAP files are used to deploy newer Windows Phone 7 applications. CAB and XAP files are compressed archives that contains: a) shared and private assemblies b) _setup.xml, c) application provisioning information, d) and registry key information. The Windows Phone 7 XAP package may also contain Silverlight assemblies and the WMAppMainfest.xml file. The WMAppMainfest.xml file supports the submission of applications to the Windows Phone Marketplace, including certification, and installation of applications. The WMAppMainfest.xml file is automatically generated and specifies device capabilities necessary for deployment. Capabilities are an application requirement to access sensitive resource, e.g., location, phone, camera, networking, etc. [9, 16 pp 104-15].

Deployment of Mobile Applications
Windows mobile applications may be deployed by manual methods (copy CAB files directly to the device), PC-based deployment (cradling), OTA (SMS) deployment using Microsoft Exchange ActiveSync (Enterprise deployment), or Mobile2Market (Windows Phone Market Place). Visual Studio provides a set of security keys that may be used to test applications using an emulator [3, pp. 102-110]. Once deployed to a physical device, only signed applications will run. Visual Studio provides SDK keys and certificates that may be used to test applications using the Windows Phone emulator or a test deployment to a physical device. Most mobile devices contain certificates that permit the installation and execution SDK-signed CAB or XAP files applications [19, pp. 419-422]. Visual Studio also permits the creation a self-signed certificate to sign code. The simplest way to install a self-signed certificate on a Windows Phone 7 is to email it to the device. Once the certificate has been detected it will be automatically installed [19].

The process of deploying mobile applications from the Windows Phone Market Place requires "a structured application submission and certification process includes a suite of certification tests to identify certain behaviors that may be associated with problems and prevent those applications from being offered in the Windows Phone Marketplace" [26]. Self-signed certificates may not be used when deploying applications using the Windows Phone

Market Place. Most public certificate authorities may be used, e.g., Windows2Market, Verisign, and Thawte certificates. A CAB or XAP file must be submitted to Micorsoft and "certified" before it can be distributed in an App Hub. The digital certificate is used to confirm a digital identify and to protect from code alteration. The certification process is not designed to protect from malware. Protection from malware and malformed code is the objective of the Microsoft certification process [19].

BLACKBERRY 7 APPLICATION DEVELOPMENT AND SECURITY

The popular Blackberry OS 6.x is a proprietary mobile operating system developed by Research in Motion. It is designed for security and administration requirements of Blackberry mobile devices and applications to support enterprise-level Blackberry Enterprise Server (BES) architectures. Consumer-based Blackberry devices, called Blackberry Internet Services (BIS) are independent and not supported by BES, e.g., no pushed applications or settings. Newer blackberry devices introduced after 2011are based on Blackberry v.7 (QNX), a Unix-like embedded operating systems [11].

Blackberry Development Environment

Blackberry applications are Java-based, which uses Blackberry's APIs. Two GUI IDEs are provided by Blackberry: a) the Blackberry Development Environment (JDE), and b) the Blackberry JDE plug-in for Eclipse. Both can be downloaded at no charge at http://na.blackberry.com/eng/developers/javaappdev/devtools.jsp. Each IDE includes the javac.exe compiler (a JavaME compiler), pverify.exe tool (a class verifier), a RIM compiler that converts JAR files into RIM's proprietary format executable format called a COD file, various device emulators, code debugger, code signing and disassembly tools [16, pp. 125-129]. As the Blackberry mobile platform continues evolve additional SDKs APIs have become available, e.g., Blackberry Tablet OS.

Permission

Unlike Android, details concerning firmware and operating system internal security mechanisms are not publicly available. The Blackberry OS does provide a UNIX-like file permission system, has a hierarchical directory structure, but no root directory. URLs are used to identity files ,e.g. file:///store/home/user/pictures/go_steelers.png Blackberry OS uses simple file permissions. Depending on the device most developed apps are stored under file:///store/var/usr and the Blackberry OS and protected apps are stored under file:///store/ By default, unsigned applications can not write to files outside its installed home directory. While some Blackberry firmware was written in C/C++, the Blackberry platform does not support native code applications, Java Native Invocation (JNI) to connect to native C/C++ code or Java Reflection which can be used to circumvent public/private Java Class access restrictions. Blackberry's record of preventing memory corruption vulnerabilities is impressive [16, 5].

Application Development Environment

Similar to other mobile application development environments, a Blackberry app starts with a creating a project. Editing source code, adding resources, compiling, and debugging are similar to other Java application development environments. During initial testing Blackberry device emulators may used be used. Several methods are available to deploy a Blackberry app to a physical device, e.g., desktop connection, Blackberry Desktop Manger (BDM), a WAP push deployment from a Blackberry Enterprise Server (BES) or carrier server, downloading from a App store or website. All Blackberry applications are packaged into one or more COD files, a Blackberry proprietary format similar to a Java JAR file. COD files, which are ZIP files that contain application code and resources, are limited to a maximum file size of 64KB. Therefore, a Blackberry application may deploy more than one COD file.

Installation and Configuration

Blackberry application deployment requires one of two application installation/configuration files: a Java Application Description (JAD) file or ALX XML manifest file. OTA deployment using a Blackberry web browser requires a JAD configuration file. Deployment from a Blackberry Enterprise Server will use the ALX manifest file. Both application configuration files contain an application name, operating system requirements, application dependencies, and which COD files are required for the application [16, pp. 132-134].

API Signatures

Each type of RIM-controlled API requires a separate digital signature: a) RCR API signatures that protect cryptographic services, b) RRT API signatures for sensitive platform functions, e.g., APM, and c) RBB API signatures that protect Blackberry applications, e.g., the browser. Unlike other mobile platforms an Blackberry application may be signed by multiple keys. However, applications may not need to be signed. A simple gaming

application which uses no cryptographic or accesses Blackberry or other third-party applications does not require a digital signature or code signing. Blackberry provides a RIM-managed signing online service that may be used to provide a digital signature for each of the RCR, RRT, RBB signing authorities. When applications are installed on a Blackberry device, the loader compares any RIM-controlled API to an appropriate signature. During runtime every time an application invokes a RIM-controlled API, Blackberry Virtual Machine uses the signature to re-verify that the application has not been alter by a malicious application [16, 134-137].

Messaging Firewalls

Blackberry devices provide a messaging firewall. Message firewall settings may secure: a) communication connections, e.g., USB, Bluetooth, company network, internet, phone, etc. , b) interactions with Blueberry applications, and c) interactions between third-party applications, e.g., IPC, keystroke injection, etc. As appropriate to the security context, Application Permission Manager (APM) settings may be set to either Allow, Deny, Prompt or Customize. For example, an antivirus application should allow access to all applications but should not prompt the user for access [21, p 70-75]. Unlike the Android security model, application permissions are not set in the installation files, e.g., JAD or ALX files. Neither can Blackberry applications determine the identity or permission setting of a calling application. The firmware and operating system controls inter-process communication. However, APIs are also available for an application to read the message firewall settings before a call is made. If an application has been denied use of another application it can prompt the user to override the setting [16, 138-140]. Message firewall permission settings that control interaction between applications are determined by the device's defaults, settings WAP pushed from a BES server or user customization. The application programmer cannot set or change permissions for message firewall settings.

CONCLUSIONS

Table 1 summarizes application development and security comparisons of the Android v 2x/3x, iOS 4x, Windows Phone 7, and Blackberry v6/7 mobile platforms. All application development platforms provide for free IDEs and SDKs, phone emulators, deployment packaging, and multiple deployment methods. Native code vulnerabilities are most significant for the Apple iOS platform. Apple provides higher level APIs and a certification process to mitigate native code risk. Blackberry and Windows Phone 7 do not support native code applications. Android supports native code applications, but limits development to compiled support libraries.

All mobile platforms reviewed provide for file encryption, communication connection encryption, and application signing. All mobile platforms permit self-signed applications, except Apple iOS. Blackberry and Android platforms execute applications in an application virtual machine. Application virtual machines provide better isolation and stability. iOS and Windows Mobile use a generalized privilege execution model to provide security boundaries, but provide limited application isolation. Both Blackberry and Android platforms provide sophisticated fine-grained application permissions. Apple's iOS and Windows Mobile platforms do not support an application permission model, but rely on a very broad system of limited execution privileges to protect applications from other applications. To compensate Apple and Microsoft rely on their formal certification process and tools to prevent malformed code or malware.

As would be expected in open source environment, Android provides the most sophisticated application security model that will: a) support the flexibility and security protection required by component-based architectures, b) verify the identity of the calling and called application, and c) to encrypt inter-process communications between applications. Android application permissions are registered and centralized during application installation using Androidmanifest.xml file. Once installed Android application permissions can not be changed. A Blackberry mobile device relies on a message firewall, using the device's firmware and operating system, to protect applications from each other. Blackberry's message firewall is less robust than Anroid permissions and attributes of the AndroidManifest file. Blackberry application developers have no control over the message firewall during execution. Unlike Android, the ability to change message firewall settings after installation is an advantage.

The most significant Blackberry security mechanism is not what is built inside the mobile device, but the ability to remotely secure and managed Blackberry devices using the Blackberry Enterprise Server (BES). BES can remotely wipe the content and application on a device, change security settings, push new applications and upgrades, and monitor usage. iOS and Windows mobile devices to a lesser degree can be remotely managed by Microsoft Exchange Server using ActiveSync. Android has very limited support for enterprise-level management of mobile

devices. In May of 2011 Blackberry announced their plans to integrate the management of Android devices into their Blackberry Enterprise Server. Further research is needed to compare enterprise solutions to manage and secure mobile solutions.

REFERENCES

1 Android and iOS – Core Features and User Experience (n.d.). Retrieved on 3/6/2011 from: *http://blog.rootshell.ir/2011/02/android-and-ios-core-features-and-user-experience/*
2 Android Developers. (n.d.). Android NDK (Version 5). Retrieved on 2/12/2011 from: *http://developer.android.com/sdk/ndk/overview.html*
3 Android Developers. (n.d.). Android NDK (Version 5). Retrieved on 3/3/2011 from: *http://developer.android.com/sdk/ndk/overview.html*
4 Android Developers. (n.d.). The AndroidManifest.xml File. Retrieved on 2/12/2011 from: *http://developer.android.com/guide/topics/manifest/manifest-intro.html*
5 Android Ideas. (n.d.). How to deploy Android application? Retrieved on 3/17/2011 from: *http://androidideasblog.blogspot.com/2010/01/how-to-deploy-android-application.html*
6 Android Security and Permissions. (n.d.). Retrieved on 2/27/2011from: *http://developer.android.com/guide/topics/security/security.html*
7 Application Fundamentals. (n.d.). Retrieved on 2/12/2011 at *http://developer.android.com/guide/topics/fundamentals.html*
8 Application Fundamentals. (n.d.). Retrieved on 2/12/2011 from: *http://developer.android.com/guide/topics/fundamentals.html*
9 Application Manifest File for Windows Phone. (n.d.). Retrieved on 5/12/2011 from: *http://msdn.microsoft.com/en-us/library/ff769509(v=vs.92).aspx*
10 BlackBerry Guide: How to Install 3rd Party Applications on Your BlackBerry. (n.d.). Retrieved on 4/6/2011 from: *http://www.bbgeeks.com/blackberry-guides/blackberry-guide-how-to-install-third-party-applications-onto-your-blackberry-88400/*
11 Blackberry Playbook Specifications. (n.d.). Retrieved from *http://us.blackberry.com/playbook-tablet/BlackBerry_PlayBook_specs_US.pdf*
12 Burnette, E., (2010). *Hello, Android: Introducing Google's Mobile Development Platform*. 3rd Edition. The Pragmatic Bookshelf.
13 Core Services Layer Mac OS Development Library. (n.d.). Retrieved on 2/12/2011 from : *http://developer.apple.com/library/ios/#DOCUMENTATION/Miscellaneous/Conceptual/iPhoneOSTechOverview/CoreServicesLayer/CoreServicesLayer.html*
14 Dhanjani, N. (2011) New Age Application Attacks Against Apple's iOS and Countermeasures. Blackhat Barcelona 2011. Retrieved on 2/27/11 from: *https://media.blackhat.com/bh-eu-11/Nitesh_Dhanjani/BlackHat_EU_2011_Dhanjani_Attacks_Against_Apples_iOS-WP.pdf*
15 Disassembling DEX files. (n.d.). Retrieved on 4/27/2011 from: *http://mylifewithandroid.blogspot.com/2009/01/disassembling-dex-files.html*
16 Dwivedi, H., Clark, C. and Thiel, D. (2010). *Mobile Application Security*. McGraw-Hill.
17 Enck, P., MCDaneil, P. (2010). Understanding Android's Security Framework. Retrieved on 2/12/2011 at *http://siis.cse.psu.edu/slides/android-sec-tutorial.pdf*
18 Enck, P., MCDaneil, P. (2010). Understanding Android's Security Framework. Retrieved on 3/27/2011 from: *http://siis.cse.psu.edu/slides/android-sec-tutorial.pdf*
19 Henry, L. Chuvyrov., E. (2010). *Beginning Mobile 7 Development.* Apress
20 Hill, S. (2009). Simple Guide to Installing Android APK Retrieved on 3/17/2011 from: *http://www.brighthub.com/mobile/google-android/articles/37151.aspx*
21 Hoffman, D. (2007). *Blackjacking: Security Threats to Blackberry Devices, PDAs, and cell Phones in the Enterprise.* Wiley Publishing, Inc.
22 iOS Developer Library. (n.d.). Managing Devices and Digital Identities. Retrieved on 4/6/2011 from: *http://developer.apple.com/library/ios/#documentation/Xcode/Conceptual/iphone_development/128-Managing_Devices_and_Digital_Identities/devices_and_identities.html#//apple_ref/doc/uid/TP40007959-CH4-SW2*

23 Making BlackBerry device applications available to BlackBerry device users. (n.d.). Retrieved on 4/6/2011 from *http://docs.blackberry.com/en/developers/deliverables/23648/BlackBerry_Java_Plug-in_for_Eclipse-Development_Guide--1352506-1222024454-001-1.3-US.pdf*
24 Mark, D., Nutting, J, LaMarche, J. (2011). *Beginning iPhone 4 Development: Exploring the iOS SDK*. Apress
25 Paul, R. (n.d.). Android goes beyond Java, gains native C/C++ dev kit. Retrieved on 2/12/2011 from: *http://arstechnica.com/open-source/news/2009/06/android-goes-beyond-java-gains-native-cc-dev-kit.ars*
26 Security for Windows Phone. (n.d.). Retrieved on 5/12/2011 from: *http://msdn.microsoft.com/en-us/library/ff402533%28v=VS.92%29.aspx*
27 The iPhone iOS 4 Core OS Layer. (n.d.). Retrieved on 2/12/2011 from: *http://www.techotopia.com/index.php/The_iPhone_iOS_4_Core_OS_Layer*
28 The iPhone iOS 4 Media Layer. (n.d.). Retrieved on 2/12/2011 from: *http://www.techotopia.com/index.php/The_iPhone_iOS_4_Media_Layer*
29 Thorsteinson, P., Ganesh, G. (2004). *NET Security and Cryptography*. Pearson Education, pp 234-235.
30 What is Coca? Mac OS Development Library. (n.d.). Retrieved on 2/12/2011 from: *http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/CocoaFundamentals/WhatIsCocoa/WhatIsCocoa.html*
31 Windows Phone 7 Security Model. (2010). Retrieved from *http://www.microsoft.com/downloads/en/details.aspx?FamilyID=dfad6c2f-988a-4b09-9e3b-58bfc9ac0447*

Table 1: Application Development and Security Summary				
	Android	Apple iPhone	Windows Mobile	Blackberry
Operating System	Android 3.1 Android 2.2 ~66% of installations	Apple iOS 4.3 iPhone OS before March 2008.	Windows phone 7 - supports dual core and 3G of RAM	Blackberry 6.x Blackberry 7.x (QNX) – a Unix-like embedded operating systems.
Application Programming Languages	Android (java) C/C++ (native code) J2ME with third-part support	Objective C and COCA API and Touch	C#.NET, VB.NET C/C++ applications for older versions	J2ME and RIM APIs No native application code
Application Development Tools Frameworks	Android SDK (Android APIs) Android NDK (C/C++) Includes emulator Integrated in to Eclipse	Xcode (IDE) Version 4 and Interface Builder; Includes iOS SDK	Visual Studio\Express (IDE) Silverlight Expression Blend XNA Game Studio Windows Phone Emulators	Blackberry Development Environment (JDE) Blackberry JDE plug-in for Eclipse
Application Packages, Deployment and Security	APK packages; AdroidManifest installation resource security	APP files; no installation or application permission security	CAB and XAP files; Only XAP files (Windows Phone 7) has a capability list notification, AppManifest.xml	COD or JAR code files;. ALX or JAD configuration files; no installation or application permission security
Application Sandbox	Dalvik Virtual Machine	None, Use Execution Privilege Levels	None, Use Execution Privilege Levels	Proprietary Virtual machine; Protects OS, not applications. VMware can be installed.
Native Code Use Protection	Android NDK to develop libraries	Higher level APIs, Clang Static Code Analyzer Static	StrSafe.h, IntSafe.h and Stack Cookie libraries; No Native Code for Window Phone 7	No native code applications No JNI (Java Native Invocation) to OS
Local and Removable Storage Protection	Application home directory file permissions Removable storage not protected. May be encrypted	No file permission system. Encryption.	No file permission system. Encryption.	No file permission system. Encryption.
Application Security Model	Component-based security permissions and APIs, e.g., Activities, Services, Content Providers, Broadcast Receivers. Self-signed applications Centralized Android Manifest Security Permissions - set at installation No remote security policy and updates	Apple Application Signing and certification required for deployment. Remote security policy and updates - Microsoft Exchange ActiveSync Limited security information is available	Self-signed applications permitted. Windows Phone 7 permits self signed applications. Windows Certification and Public Signature required for Windows Phone Store deployment Execution Privilege Levels Remote security policy and updates - Microsoft Exchange ActiveSync	Application Signing required for RIM Controlled APIs Messaging firewall – permissions between app, may be changed locally or remotely Remote security policy and updates -BlackBerry Enterprise Server