AUDITING ENTERPRISE CLASS APPLICATIONS AND SECURE CONTAINERS ON ANDROID

The Limitations of Mobile Security in the Enterprise

Marc Blanchou of iSEC Partners

December 2012

marc[at]isecpartners[dot]com

https://www.isecpartners.com

Abstract

Today's corporations and governments must secure potentially sensitive information on mobile devices. In response, Mobile Device Management products claim to help enterprises secure these devices or even act as "secure containers". There is an increasing need to assess the security claims of such enterprise-class software vendors, but there is very little information on how their claims hold up to real-world threats. This paper covers research into those threats, with a focus on mobile devices running Android. By understanding the different attack vectors and the current mobile security models, this research paper aims to determine what should be protected, when it should be protected, and how commercial security solutions fit into the mix. The paper also discusses design concerns in popular Android security products and techniques used to assess them.

Executive Summary

Mobile platforms have risen in popularity and capabilities, making them a desirable work asset for enterprise networks. When used within the enterprise, these devices are likely to process sensitive information; this therefore raises the question of whether this data is secure with current mobile enterprise product offerings.

Current mobile security models market two flawed assumptions. First: they can use the same security model as laptops. Second: they are protected because compromising mobile platforms generally requires root privilege. These models are inefficient since devices are always turned on and there are limitations in user input that hinder the use of strong passwords. In the hands of an attacker, a powered-on device is susceptible to information disclosure via its flash memory (internal/external) and on some devices the RAM. Additionally, privilege escalation bugs and public exploits for rooting a phone are commonplace.

Mobile Device Management (MDM) software is the solution for enterprise policy enforcement. Some MDM products claim to be "secure containers", enforcing security policies set by the enterprise on their managed mobile devices, including separate encryption of corporate data. This paper investigates Good for Enterprise and Mobile Iron MDM products.

Partly due to the limitations of the Android devices, reviewed MDM products did not have the ability to properly enforce their policies. Detecting if a device is rooted, providing remote wipe or claiming that "corporate data is highly secure" with Federal Information Processing Standard (FIPS) compliant encryption – these claims are mostly marketing without any real security. Encryption provided by the solutions does not hold against real threats since cryptographic keys are retrievable in most cases. Remote Wipe can be circumvented and Rooting Detection uses weak checks that an attacker can easily bypass as well.

The use of MDM products coupled with a strong IT policy, including enforcement of strong passwords, would mitigate some attack vectors - especially for the Good for Enterprise solution - but a sophisticated attacker could still potentially access data on a compromised or stolen device.

The reviewed products delegate some aspects of their provided security to the underlying operating system. Prior versions of Android lack full disk encryption, and an attacker with physical access to a device protected by an MDM product can retrieve sensitive information stored on the device.

1. Introduction

Android is currently the fastest growing mobile platform[1], operating with thousands of new users every day. In 2011, IDC[2] showed that 69% of employees use mobile devices for business, implying most devices store sensitive data.

The mobile device security model is erroneously based on the security model of their technological predecessor: the laptop computer. Unlike the laptop, mobile devices are rarely shut down or hibernated. They are always turned on and are almost always connected, making the laptop security model insufficient. This connectivity also raises a new set of security risks with new threats and attack vectors.

---

[1] http://www.asymco.com/2011/11/06/the-us-smartphone-landscape/

[2] http://www.unisys.com/unisys/ri/report/detail.jsp?id=1120000970016710178

"Secure Containers" have been developed to facilitate the adoption of mobile devices into the enterprise and try to mitigate the risks inherent to the platform. Some of these applications claim to provide enterprise-grade security for Android devices. With the adoption of mobile devices into enterprises there is an increasing need to assess the security claims of such software.

This paper discusses the marketing claims made by software vendors such as Good for Enterprise and Mobile Iron on Android devices and how they mitigate current threats. It should be noted that these two products were chosen because they were two of the most popular MDM products at the time of testing (which started in 2011 and ended in March 2012), but they do not represent the current market and have been modified since initial testing (see "what is the state of these applications today").

2. Android Security Features

In Android, non-kernel applications are executed in a virtual environment provided by the Dalvik Virtual Machine. Android uses the Dalvik Virtual Machine (DVM) with just-in-time (JIT) compilation to run Dalvik executables translated from Java byte-code. The DVM is similar to the Java Virtual Machine (JVM) in concept and as such, many traditional Linux and Java application testing techniques can be applied to Android.

Android by design provides application sandboxing based on privilege separation. Each application runs with its own User ID (UID) and Group ID (GID). Similar to its Linux origins, Android prevents users from accessing resources of another UID. This results in having complete application segregation. Additionally, applications by default do not have permission to perform tasks that may negatively affect other aspects of the system. Instead, applications must explicitly request the permissions they need for additional capabilities not provided by the sandbox. These permissions include accessing or modifying user data such as other application's files, emails, contacts or using network resources. However, applications support running native code, which means the Dalvik Virtual Machine is not a security sandbox in itself. Running native code is usually used for performing time sensitive operations and this introduces the risk of classic low level language bugs.

Android version 4.0 (ICS) provides additional security features such as full disk encryption and the support of Address Space Layout Randomization (ASLR); however, very few devices have this version of the OS installed by default at the time of writing.

3. Android in the Enterprise

Android has very limited inherent features that would help keep it secure in the enterprise. The OS is currently focused toward the consumer market. It lacks reliable central management features such as granular application control, security policies, and device analytics. Additionally, no full disk encryption was provided prior to the release of Android 3.0. Android currently relies mostly on ActiveSync for enterprise mobile mail, and features supported on most common devices are limited[3].

4. Secure Containers

To address the limitations of the Android OS, enterprise class applications have been developed. They attempt in multiple ways to improve the security and manageability of Android devices. They introduce the ability to do remote analytics, remote wipe, password management and encryption of corporate data. Many of these features have been modeled after Blackberry Enterprise Server (BES), which has been a gold standard of enterprise mail for a long time.

Secure containers attempt to provide data segregation by separating personal and enterprise data. To create this separation, most encrypt enterprise mails, contacts, calendars and attachments on the device. These containers rely on a PIN to decrypt the data. This PIN is programmatically distinct from the Android PIN.

Good for Enterprise and Mobile Iron allow users to manage company emails, contacts, calendars and events for their end users. They claim to enforce policies, as well as perform remote lock and wipe of a device. Mobile Iron does not claim to be a secure container and relies on the full disk encryption provided by the device to secure the data. They both work on several platforms such as iPhone, Android and Windows Mobile and are both used by Fortune 500 companies and government agencies.

5. Threat Agents

It can be relevant to define the threat agents when assessing security measures and their associated risk.

One of main threat is the individual attacker. Using obfuscation in a product can potentially be enough to protect against the casual attacker but an experienced attacker could have specific skills dedicated to mobile exploitation such as the expertise to develop custom tools for application de-obfuscation and decryption of data using a large GPU cluster.

Another threat is corporate espionage. Several companies made the news after reporting incidents of industrial espionage[4]. Corporate spies may not have direct contact with targeted devices, but they have access to large financial resources and highly skilled computer experts.

In the same way, governments and law enforcement agencies may want access to information on mobile devices. They have frequent physical access to devices when individuals are going through airport or border security checks. They have access to many people with highly technical skills and large financial

---

[3] http://en.wikipedia.org/wiki/Comparison_of_Exchange_ActiveSync_clients

[4] http://en.wikipedia.org/wiki/Industrial_espionage

resources.

According to the Electronic Frontier Foundation (EFF)[5] 6,500 people travelling to and from the US had their devices searched in about two years - which is about 300 people per month - and almost half of them were US citizens. Moxie Marlinspike is a good example of one security researcher, among many, who was searched at the border[6]. Additionally, some states give no protection to contents of a phone during a search after a violation has been committed. In particular, Florida law used to treat a smartphone as a "container" for the purposes of a search. In addition, the "search incident to arrest"[7] would potentially allow warrantless search in California[8]. All in all, depending on the circumstances and the local court system, the police may be able to search a phone without a warrant.

6. How can the data be obtained?

A common way to obtain a phone's data is to leverage enabled development features such as USB debugging and PC mode. Getting root access on a phone using Android Debug Bridge (ADB) and publicly available exploits such as "rage against the cage" can be easily performed by an attacker without the need to reboot the device, in most cases making the internal storage and RAM accessible. There are numerous ways to access internal device storage, but it is more difficult to access RAM contents when the device does not have USB debugging enabled and is set up with a lock screen.

Some privilege escalation bugs allow for direct access of internal storage (an example is CVE-2012-0056 for the Linux kernel, implemented for android with mempodroid[9]). In addition, a recent study shows that more than more than half of all Android devices contain known vulnerabilities[10]. Alternatively, companies have publicly released hardware devices that can read the flash directly from the device over USB. One of the most popular of these devices is the UFED. One of the latest versions claims to support direct bit-by-bit flash copying as well as the ability to bypass the Android lock screen. They are currently limited to certain devices, but the list is growing, with Samsung supposedly next[11]. These devices are known to be used by government entities. Additionally, physically reading the flash would be possible, because as opposed to the RAM, there is no gradual memory degradation after memory chips are pulled. Another way to retrieve the internal storage and RAM (but after rebooting the device) is through a JTAG port. Most of the devices do not have a JTAG port, but some of them, including HTC Dream and HTC Magic[12] for instance, support the JTAG interface and can be retrofitted with one which will allow debug access to RAM.

One other attack vector is to be able to find the lock screen PIN (or lock screen pattern) on a stolen device. The lock screen being independent from the application lock, people may tend to have a weak phone PIN and a stronger PIN for the enterprise application, since it is used more often. Statistics show that people almost always chose a common PIN such as 0000 or 1234.[13] An attacker would then be able to access a device, find the PIN, activate USB Debugging and potentially find a way to bypass the "stronger" PIN of the container. In addition, a stolen device may have smudges on the touch screen surface that could help an attacker to find the PIN[14]. A recent device used by law enforcement, which can crack an iPhone's PIN in a few minutes, would also be able to crack an Android phone's passcode.[15]

Additionally, some manufacturers made the news due to bypass issues in the lock screen[16] that would allow enabling the USB debugging functionality to be enabled. A cold boot attack would also allow access to RAM, but would require some very specialized equipment and skills to de-solder memory chips from the board and place them into specialized readers.

The proliferation of Android malware is on the rise. For instance, one of the latest campaigns may have infected 5 million users,[17] often by masquerading as popular applications.[18] These can be distinguished as two groups: userland malware and malware with root access or exploiting OS level vulnerabilities. The former can use standard application permission vulnerabilities such as sensitive inter-process communication (IPC) mechanisms accessible to any application on the device, world readable shared preferences or use of external data storage. The latter group of malware could record keystrokes, access any file on the file system, trick the user into providing sensitive information and/or send files and encryption keys to a remote server. Even though a properly configured application can be relatively safe from userland malware, it is impossible for an enterprise application alone to properly protect against malware with root access or exploiting OS level vulnerabilities.

SSL has to be used by applications in order to protect sensitive data during transit. However, flaws such as improper certificate authenticity checks or a lack of proper hostname validation could

---

[5] https://www.eff.org/wp/defending-privacy-us-border-guide-travelers-carrying-digital-devices

[6] http://www.net-security.org/secworld.php?id=10187

[7] https://www.eff.org/issues/search-incident-arrest

[8] https://www.eff.org/deeplinks/2011/11/year-smartphone

[9] https://github.com/saurik/mempodroid

[10] http://threatpost.com/en_us/blogs/research-shows-half-all-androids-contain-known-vulnerabilities-091312

[11] http://www.cellebrite.com/news-and-events/press-releases/%20246-cellebrite-ufed-extends-forensic-capabilities-to-android-mobile-devices.html

[12] http://wiki.cyanogenmod.com/wiki/HTC_Dream_%26_Magic:_JTAG

[13] http://www.net-security.org/secworld.php?id=11167

[14] http://static.usenix.org/events/woot10/tech/full_papers/Aviv.pdf

[15] http://www.forbes.com/sites/andygreenberg/2012/03/27/heres-how-law-enforcement-cracks-your-iphones-security-code-video/

[16] http://www.bgr.com/2011/09/30/major-security-flaw-lets-anyone-bypass-att-samsung-galaxy-s-ii-security-video/

[17] http://www.computerworld.com/s/article/9223777/Massive_Android_malware_op_may_have_infected_5_million_users

[18] http://thenextweb.com/google/2012/10/05/over-60-percent-of-android-malware-comes-from-one-family-hides-in-fake-versions-of-popular-apps/

lead to Man in the Middle attacks (MITM) and would allow an active attacker to see all the network traffic including sensitive emails and credentials. A recent study showed that 8% of 13,500 applications examined contained code that was potentially vulnerable to MITM attacks.[19] Additionally, Certificate Authorities (CAs) have been compromised in the past, and applications on Android generally use the phone key's store to validate certificates; this means an attacker controlling a CA would be able to potentially get a certificate accepted by the application. To mitigate this risk, some applications internally store a whitelist of certificates known to be used by the server; this is called certificate pinning.[20]

7. Claims of the MDM Products

The claims below were found on the product websites. Good mentions that "Corporate data is highly secure" with Good for Enterprise, but also that[21]:

> "Governments have tested the product and approved it for their most sensitive deployments"

> "Over-the-air transmission and enterprise data at rest on the devices are secured with industry-leading AES-192 encryption."

This indicates that potentially very sensitive information is secure on these devices. But who is it secured from and when is it secured? When is a phone's data really "at rest?" Is it still secured when the phone is turned ON? How are encryption keys handled? It is also mentioned that it "Leverages a FIPS 140-2 certified cryptographic module to protect data-at-rest and data-in-transit" as well as:

> "The cryptography employed by Good has been successfully tested by NIST-approved labs and certified to be compliant with FIPS 140-2 Level 1"[22]

However, the FIPS level 1 specification is the lowest level of the FIPS "security levels". How are keys handled by the application and can they really be self-contained by the module? Zeroed when not in use? Additionally, in the level 1 specification, no hardware module is needed to store keys. A module contained within a cryptographic boundary also means that key information should not be passed outside of the boundary if implemented correctly.

Mobile Iron mentions in its literature that it enforces encryption (likely by leveraging the device's Full Disk Encryption on compatible devices) and password policies but does not really claim to encrypt the data with the product[23].

> "MobileIron combines traditional mobile device management capabilities with advanced security, data visibility, apps management, and access control powered by the Virtual Smartphone Platform Architecture. IT administrators can manage and secure the mobile device, data, and apps, from registration to retirement, and quickly get smartphone operations under control."

However in a "Mobile Device Security Features" section, "Encryption policy (phone, SD)" is mentioned as well as "Lockdown security (camera, SD, Bluetooth, Wi-Fi), Password enforcement, Remote lock and wipe" and "Password enforcement" as well as the following:

> "MobileIron can detect if an iOS or Android device has been compromised and can block the device from accessing corporate resources."

Good for Enterprise also mention the ability to remote wipe:

> "Performs remote wipe of enterprise data only"

Pulling the SIM card out of the phone would obviously allow an attacker to easily circumvent this feature, but the phone must be turned off in order to do so. Alternatively, if the phone needs to remain on in order to retrieve the data, an attacker could put the phone in a Faraday bag which would isolate the phone from any Radio Frequency (RF).

[19] http://www2.dcsec.uni-hannover.de/files/android/p50-fahl.pdf

[20] http://www.thoughtcrime.org/blog/authenticity-is-broken-in-ssl-but-your-app-ha

[21] http://www1.good.com/products/good-for-enterprise

[22] http://www.comdirect.ch/produkte/datenblaetter/doc_download/441-whitepaper-security-overview

[23] http://www.mobileiron.com/

8. Testing Android Applications

Reverse Engineering – Methodology Used

Dalvik Bytecode was decompiled using Backsmali, Dedexer, dex2jar and JD. Native code segments were reverse engineered using IDA Pro 6.0. Content of the RAM at different states of the phone was analyzed with raw memory dumps and heap profiling dumps. Structured memory analysis was performed with VisualVM or JHAT using heap profiling dumps generated by the Dalvik Virtual Machine. JDB was used to perform dynamic analysis allowing debugging of the applications. Modifying applications to bypass obfuscation was performed using apktool and jarsigner by decompiling, modifying, recompiling and re-signing the applications.

Dalvik Bytecode and Reverse Engineering

Most of the applications are written primarily in Java and compiled to Dalvik Bytecode, which makes applications easier to reverse engineer. Backsmali was used to decompile applications into Smali code, which is more readable and comprehensive than Dalvik Bytecode.

Annex 1 - Backsmali was used to convert the application into Smali code.

Static Analysis

Decompiling a dex file into Smali with the Backsmali tool[24] will give one of the most accurate representations of the original code. The author found it was better to use Backsmali in order to decompile a dex file; the other option is to use dedexer[25], which is a great tool, but does not allow recompiling the dex file afterward, which is always useful if some debugging is needed.

The syntax[26] and the Dalvik op-codes[27] need to be understood. For instance, p0 to p3 are used for parameters, p0 is usually the 'this' (refers to the object a function is a method of) and v0, v1, v2, etc. are local variables. The maximum number of variables is given with .locals at the top of the method body (with *--use-locals* option with backsmali). A method return type is given by the letter at the end of the method name (V means void, I integer, Z boolean for instance).

Apktool[28] is a tool facilitating the decompilation. It is an all-in-one tool which takes the apk of the application, decompresses it, and uses Backsmali on the dex file. It also retrieves the manifest and converts it into xml. The apk file can be decompiled with *"apktool decode *.apk"*.

The decompilation to Java using dex2jar[29] and JD[30] is not very reliable and is rather inaccurate (as some methods even fail to decompile), but can greatly help in order to have a quick overview of what an Application does.

Smali code can be recompiled using the Smali tool. The Smali code can be modified in order to help debugging, log some data, perform detailed traces, dump the memory heap after specific actions (see android.os.Debug.dumpHprofData()) and strip SSL. Once modified and recompiled with *"apktool build [DIR]"*, it needs to be signed.

Android applications are self-signed and can be modified with root access. The following steps are needed in order to run modified applications and resign them (not needed for system applications):

After installing the Java JDK, the "Java\jdk1.6.0_*\bin" directory contains two needed tools: jarsigner and keytools. Keytools allows creating a keystore, and jarsigner uses it to sign the APK with the private key generated.

Annex 2 - Decompilation issues – Example
Annex 4 - Example of obfuscated code (Mobile Iron)

Dynamic Analysis

DDMS, traceview and dmtracedump can be used to perform dynamic analysis. Native parts can be debugged with strace and gdb with remote debugging. JDB can do basic debugging tasks. It uses the JDWP protocol, but unfortunately not all of the JDWP requests are implemented by the Dalvik VM (which explains some crashes).

The Development Tools have to be installed on a rooted phone – on an emulator, they are installed by default. In the Development Tools menu, go to "Development Settings" and select the application that needs to be debugged; "Wait for debugger" option will also have to be enabled. Also, if the test is performed on a phone, applications must have the "debuggable" flag in the Application Manifest (the application can be recompiled if it needs to be added). Run the application, it should block, waiting for a debugger to attach. Example:

List JDWP processes:
adb jdwp
Set up port forwarding to connect to a JDWP process:
adb forward tcp:4200 jdwp:<pid>
Attach JDB to it – in this example, the emulator is on localhost:
jdb -attach localhost:4200

Memory Analysis

The goal of memory analysis is to find what is available in

[24] http://code.google.com/p/smali/
[25] http://dedexer.sourceforge.net/
[26] http://jasmin.sourceforge.net/guide.html
[27] http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html
[28] http://code.google.com/p/android-apktool
[29] http://code.google.com/p/dex2jar/
[30] http://java.decompiler.free.fr/?q=jdgui

memory and when. Unstructured memory analysis can be performed in order to search for strings or specific data that could have been spotted during the static analysis. Dalvik strings use UTF-16, which means a character is two (2) bytes, but there could be other types of strings. If there is a native code part, an application using a different encoding format can usually be examined with the Linux command *strings -a -e [encoding]*.

There are several methods to acquire memory dumps - either the Linux proc filesystem (procfs) with /proc/[PID] or the memfetch application could be used, both allowing for retrieval of a raw memory dump.

To perform a more structured analysis, a Java heap profile file could be dumped by the Dalvik VM itself. This dump file is stripped off Dalvik specific data and can be read with Java memory dump analysis tools such as JHAT, Jprofiler or VisualVM. The Garbage Collector of the Dalvik VM dumps a hprof file when sending a SIGUSR1 to the process (kill -10 [PID]). The hprof file can then be retrieved in /data/misc.

Intercepting SSL Traffic

The application can be statically modified and recompiled in order to accept any certificates; this would allow to proxy the traffic using a tool such a Burp Suite[31] for HTTPS traffic and Canape[32] or Mallory[33] if a binary protocol is used. Alternatively, SSL can be dynamically bypassed for any application on a device by a tool written by Justine Osborne of iSEC Partners called android-ssl-bypass.[34]

Testing SSL/TLS Client Certificate Validation

The certificate checks can be examined through static analysis, but using a tool is more convenient, exhaustive and reliable. TLSPretense[35] is a tool recently released by William (B.J.) Orvis, also of iSEC Partners, that allows for testing a number certificate validation issues. The tool generates a set of certificates containing specific flaws, and it presents the certificates to an application (which must be configured to trust a CA used by TLSPretense).

Testing for Specific Vulnerabilities

Android uses a number of inter-process communication (IPC) mechanisms that can be overly-permissive if incorrectly configured. Typically, applications declare IPCs that can be potentially interacted with by any other application on the device. These mechanisms include Activities, Services and Broadcast Receivers (talked to via messages named 'Intents'). These should be marked as not exportable in the Android Manifest unless there is a legitimate reason for it. Another IPC mechanism called Content Providers is used to export data by default. Implementations of Content Providers should be carefully reviewed in order to make sure they do not serve sensitive data and do not introduce security bugs such as SQL injection. More detailed information can be found in Jesse Burns' paper[36] and on the Android security best practice page[37].

Applications should avoid exposing sensitive data to other applications. Shared preferences, databases and files should not be world readable/writable if not needed; Using MODE_PRIVATE as the default file creation mode forces an application to create files with the correct, restrictive permissions.. Additionally, any data stored on the SD card is accessible to any application (since Android 4.1 they need the READ_EXTERNAL_STORAGE permission). Also, the application should never log potentially sensitive data in logcat (the Android logging system) as it can be read by other applications (with the READ_LOGS permission) on Android prior to 4.1.

In addition, application specific vulnerabilities can include common native language vulnerabilities if the application interacts with a native library or custom code such as custom cryptographic functions or any time sensitive functions.

Finally, a great number of applications use a WebKit WebView, which loads web content in an in-process browser. WebViews can be vulnerable to common web vulnerabilities such as cross-site scripting (XSS) and cross site request forgery (CSRF) issues if JavaScript is enabled. The WebView should be as restricted as possible,[38] and file system access, JavaScript and Plugin should be prohibited if not needed.

9. Good for Enterprise

Design concerns were found in version 1.7.1.157 of the application but were also present in version 1.6.5.146 and 1.6.3.138 which are the last supported versions for Android 1.6 and 1.5 respectively.

A. Ability to Retrieve Sensitive Data (when the device is locked and Powered ON)

The application maintains the master database key in memory even when the application has been locked for a long time. Being able to access this key on a stolen device compromises the security of all data stored by the application: email, contacts, appointments, and identifiers allowing an attacker to

[31] http://www.portswigger.net/burp/

[32] http://www.contextis.com/research/tools/canape/

[33] http://intrepidusgroup.com/insight/mallory/

[34] https://github.com/iSECPartners/android-ssl-bypass

[35] https://github.com/iSECPartners/tlspretense

[36] https://www.isecpartners.com/files/iSEC_Securing_Android_Apps.pdf

[37] http://developer.android.com/guide/practices/security.html

[38] http://labs.mwrinfosecurity.com/blog/2012/04/23/adventures-with-android-webviews/

impersonate the user.

Good for Enterprise is using a native library with functionality similar to the commercial library SQLite Encryption Extension (SEE), which itself has several functions very similar to the SQLCipher library. This was identified by decompiling the Good for Enterprise libraries with IDA Pro and looking at the functionality and function names since symbols were present. Commands specify the cipher and the key used with the database. Understanding how the library worked helped finding the key in memory dumps. Structured memory analysis showed that the master database key resides in a linked list in memory for use by a connection pool talking to a library that handles the encryption and calls to the database. This unique 192-bit key allows decrypting all the database files using AES-CBC.

The Annexes contain examples on how to easily get the master DB key from a memory dump using VisualVM and structured memory analysis.

Annex 6 - Master DB key from a memory dump using VisualVM
Annex 7 - It is also possible to retrieve the key in memory by looking for UTF-16 Strings
Annex 8 - Now the attacker has the ability to decrypt all the database files including emails

Exploit Scenario

An attacker obtains a device that has USB debugging turned on (or uses one of the techniques mentioned in section 6 to enable it). The device is put in a Faraday bag[39] to avoid remote wipe. The attacker then uses a publicly available exploit to become root, pull the master database key out of memory and decrypts the database files including emails, contacts and appointments.

B. Retrieve the User Password and Access the Data

The database key derives from the user's password. Verification of the password requires computing a symmetric key's hash using a Password Base Key Derivation Function (PBKDF2-SHA1 with a number of iterations, see PKCS #5 and PBKDF2 function) and one SHA1 operation. Good for Enterprise uses the PIN and an 8 byte salt. The hash and salt are obfuscated with an XOR between a static string and the hash as well as a great number of bitwise operations. This is then stored in an XML file located in */data/data/com.good.android.gfe/shared_prefs/GoodLockPrefs.xml*. Obfuscation could easily be bypassed - without having to reproduce de-obfuscation steps - by simply debugging the application and dumping the hash and salt, or modifying the application in order to dump the password hash in a log file when the application accesses them. Brute force analysis of the hash and salt can be performed relatively easily with a GPU cluster.

[39] http://www.teeltech.com/tt3/phoneshield.asp

Exploit Scenario

An attacker obtains a device and retrieves the flash memory (see section 6). This data is susceptible to brute force attacks because passwords on a mobile device are not likely to have more than 6 characters and will probably have fewer special characters than traditional passwords. Additionally, PBKDF2 functions on mobile devices use reduced iterations due to the CPU limitations.

Using the GPU outperforms using the CPU to generate hash values. The use of GPU to break PBKDF2 implementation has been demonstrated with the brute force of PBKDF2-HMAC-SHA1 256-bit keys with 4096 iterations, which is generally performed to attack WPA/WPA2 by computing Pairwise Master Keys. More than a hundred thousand keys per second can be generated using the already old Radeon HD 5970[40]. An attacker using a GPU cluster such as a FPGA or Amazon EC2 instances would have considerable computing power for a relatively cheap price. FIPS level 1 only requires a minimum of 1000 iterations, which would lead to hundreds of thousands of keys generated per second using a mere desktop graphic card.

C. Other

Jailbreak and Root Detection

Root detection is done by regularly verifying if superuser.apk is installed; the application also tries to run "su –c ls" and expects it to fail if the phone is not rooted. These verifications are weak and cannot protect against an attacker running a publicly available exploit to become root such as "rage against the cage." Additionally, just renaming 'su' in 'system/bin/su' on a rooted device when running Good would not be detected by the application. There is even an application called 'temp root remover' on the Android market that is used for this purpose.

No Anti-debugging And Plenty of Logs

No anti-debugging or anti-reverse engineering techniques were present in either application. Additionally, one of the applications makes it easy for an attacker to reverse engineer, as symbols and debug log messages were still present in the application.

An example of an inconsistent decompilation using dex2jar and jd-gui for the Dalvik Bytecode part and a decompilation of the native part using IDA Pro can be found in annex. It shows that symbols are present, and debug logs are filled with copious amounts of information that helps an attacker understand detailed functionality of the application.

Annex 3 - Detailed function names and a lot of information in the decompiled code.
Annex 5 - No obfuscation in the native code parts.

[40] http://erratasec.blogspot.com/2011/06/password-cracking-mining-and-gpus.html

10. Mobile Iron

Testing was done on Mobile Iron Version 4.5.2 (March 2012) and 4.1.7 (June 2011) through static analysis only.

Data exposure

In the last version tested, Mobile Iron version 4.5.2, user credentials can optionally be encrypted in a "config.properties" file. The application is obfuscated to some extent (e.g., function names have been stripped out). However, the AES-256 encryption key used is generated using SHA1 PRNG with android_id (UUID) as seed. The Android UUID is a constant number for the lifetime of the device and should not be used in encryption operations; the seed can easily be retrieved by an attacker or a rogue application (the UUID can be retrieved by any application) and the key could be generated. An attacker stealing the device can easily obtain this value, generate the key with the android_id, and retrieve user credentials and the database content. Additionally, if the same key were used to encrypt data on the SD card, any application installed on the device could potentially decrypt the files. Also, this AES 256-bits key is used in ECB mode. ECB is widely known to be insecure, and should not be used.

Unique Device ID is sent to the Mobile Iron server

The application collects the unique identifying Device ID and sends it to the company's Mobile Iron server during provision. As a best practice, the Device ID should not be stored; this identifier has been used by many applications as an authentication token in the past, which means that the company installing the Mobile Iron product would potentially be in possession of credentials for other applications. This raises the sensitivity of these tokens to a level not normally expected of a unique identifier. Public scrutiny has been focused on applications, largely on iPhone devices[41] but also on Android to some extent (with a presentation given by Lookout[42] at the Black Hat Security Conference in 2010).
This issue may be less significant in an MDM/corporate environment, as users have less expectation of privacy from their employer, and MDM products are implicitly expected to "track" mobile devices to some extent.

Jailbreak and Root Detection

Root detection is done by verifying the existence of "/system/bin/su" in version 4.1.7 and a few other places in version 4.5.2 (/system/xbin/su, /system/su and /system/xbin/sudo) which is not enough for promising root detection.

[41] http://www.pskl.us/wp/wp-content/uploads/2010/09/iPhone-Applications-Privacy-Issues.pdf

[42] https://blog.lookout.com/blog/2010/07/27/introducing-the-app-genome-project/

11. Can it be Secure?

It is currently difficult to properly secure application containers because of the mobile security model. It is also even more difficult to secure them without hardware modules integrated with a device for handling cryptographic keys[43] and a custom OS that would handle central management features. To this end, several fundamental security primitives have been defined (still as a draft) in a document released by NIST[44] and are intended to help the industry attempt to make devices more secure for the enterprise. However, there are a few solutions the products could implement to help mitigate some of the issues mentioned in this paper.

The key used to decrypt the data should not be stored on the device. Instead it should be retrieved from a server when the user enters the application's PIN. In addition, the key should be zeroed from memory after a certain amount of time when the device is locked, and it should not be possible to retrieve the user's encryption key even by brute force. This key should be retrieved directly from the server with a PIN, and the user's account should be locked after a certain number of unsuccessful attempts.
To be able to read emails offline (when no connection is available), the user should be able to keep the encryption key in memory for a certain amount of time, but this should be an optional 'unsafe/offline mode' so that the user would know that the data could be at risk if the phone was stolen at that point in time. However, this would obviously impact user experience, as the user would have to select this option in advance so that the key could be stored.
Another debatable solution would be to retrieve the key locally only when no network access is available, with a Password-Based Key Derivation Function with a very high number of rounds (high enough so that it takes a noticeable amount of time to retrieve the key on the phone) and an enforced strong password (different than the server PIN). The number of rounds would be enough to allow the user to report the stolen/compromised phone to IT and block email access before an attacker bruteforces the key. Doing this would make it more difficult for an attacker to retrieve the user's data on a stolen device as long as the phone is locked or turned off. This could be an acceptable mitigation without greatly diminishing the user experience.

Solutions for malware detection are primarily performed by searching for known cases of rootkits or root utilities - a form of blacklisting. The limitation of blacklisting is that it only is as good as the provided list of harmful applications. A preferable method to mitigation against malware gaining root access on a device is to have application whitelisting on a device; however, this is not currently available for Android.

[43] A few devices now integrate a Secure Element (SE) which is a tamper resistant smart card chip.

[44] http://csrc.nist.gov/publications/drafts/800-164/sp800_164_draft.pdf

12. What is the state of these applications today?

As they were discovered, iSEC disclosed the design concerns to the respective companies. It should be noted that Good and Mobile Iron were both responsive upon receiving these issues.

According to MobileIron, the data exposure issue (related to data persisting on the device) is not present in the default configuration because this persistence option is disabled by default. They stated that ECB is no longer used, key generation routines have been modified and jailbreak detection has been improved.

Good for Enterprise has released several versions of the product since the time of testing. Some obfuscation has been added to the product. According to Good, the key no longer persists in memory when the phone is locked in version 1.7.3.169. However, the ability to retrieve a user's PIN by brute force is still possible.

13. What can users do today?

User awareness is a critical component to mitigation of current attacks. Users must use caution when installing software and setting permissions on their phones. Users should disable unnecessary features that increase the attack surface, such as USB debugging or Bluetooth, refrain from rooting their phones, keep applications updated, and use complex PINsfor the lock screen instead of screen patterns. Users should also use different credentials for the MDM product than those used for the phone and the other applications. Users should be aware of the attack vectors, and understand that the data stored on a stolen device may be compromised by malicious applications gaining root access or using OS level exploits. Additionally, full disk encryption should be enabled on capable devices.

14. Conclusion

Marketing claims may be true in some cases, but are they always relevant? How can a phone's data be secured at rest if the data is never at rest? A user cannot rely solely on remote wipe or root detection in these applications. Strong passwords help - but will users really be willing to type a complex password each time they want to unlock their applications? And can it really be enough to secure the data from a sophisticated attacker? There are ways to mitigate the risk, but they rely on a network connection when unlocking the application, thus denying the user a smooth offline experience. For both the products tested, Good for Enterprise and Mobile Iron, user data can be retrieved on a stolen device and a user could potentially be impersonated.

A phone has a very large threat surface compared to most other devices; users and businesses must take this into account when storing sensitive data on mobile platforms. Enterprise solutions such as Good for Enterprise and Mobile Iron could implement better security practices and mitigate some attack vectors, but most importantly, companies must understand the implications of allowing mobile devices access to enterprise resources, even when using MDM products.

Acknowledgment

Special thanks to Mathew Solnik for the help with the initial research. Thanks to David Thiel for assisting with the disclosing process and the review of this paper throughout the writing process. Thanks to Alex Vidergar, Michael Reisinger, Shawn Fitzgerald and Aaron Grattafiori for their reviews. Additional acknowledgement should be given to Justine Osborne, Alex Stamos, Jesse Burns, Don Bailey, Paul Youn and iSEC Partners for their support.

Annexes

1. Backsmali was used to decompile the application into Smali code

```
.class Lcom/good/android/security/SecureDataAndPassword;
.super Lcom/good/android/security/SecureData;
.source "SecureDataAndPassword.java"

  # virtual methods
.method public hasPasswordMatch(Ljava/lang/String;)Z
    .registers 4
    .parameter "password"
    .annotation system Ldalvik/annotation/Throws;
        value = {
            Lcom/good/android/security/EncryptionException;
        }
    .end annotation
    .prologue
    .line 41
    iget-object v1, p0, Lcom/good/android/security/SecureDataAndPassword;->passwordHash:[B
    if-nez v1, :cond_a
    .line 42
    if-nez p1, :cond_8
    const/4 v1, 0x1
    .line 46
    :goto_7
    return v1
    .line 42
    :cond_8
    const/4 v1, 0x0
    goto :goto_7
    .line 44
    :cond_a
    invoke-direct {p0, p1}, Lcom/good/android/security/SecureDataAndPassword;->computeHash(Ljava/lang/String;)[B
    move-result-object v0
    .line 46
    .local v0, testHash:[B
    iget-object v1, p0, Lcom/good/android/security/SecureDataAndPassword;->passwordHash:[B
    invoke-static {v1, v0}, Ljava/util/Arrays;->equals([B[B)Z
    move-result v1
    goto :goto_7
.end method
```

2. Decompilation issues – Example

- Smali

```
.method protected Test()Z
    .registers 3
    .prologue
    iget v0, p0, ###/###/###;->value:I
    const/16 v1, 0x10
    if-ne v0, v1, :cond_8
    const/4 v0, 0x1
    :goto_7
    return v0
    :cond_8
    const/4 v0, 0x0
    goto :goto_7
```

- Java equivalent

```
protected boolean Test() {
    return (this.value == 16 ? true : false);
}
```

- From dex to Java
  - With dex2jar and JD

```
protected boolean Test()
{
  if (this.value == 16);
  for (int i = 1; ; i = 0)
    return i;
}
```

3. Detailed function names and a lot of information in the decompiled code (Good for Enterprise)

GxSecurity.class | SecureDataAndPassword.class | SetPasswordException.class | SecureData.class | PasswordStrength.class | ResetPassword.class

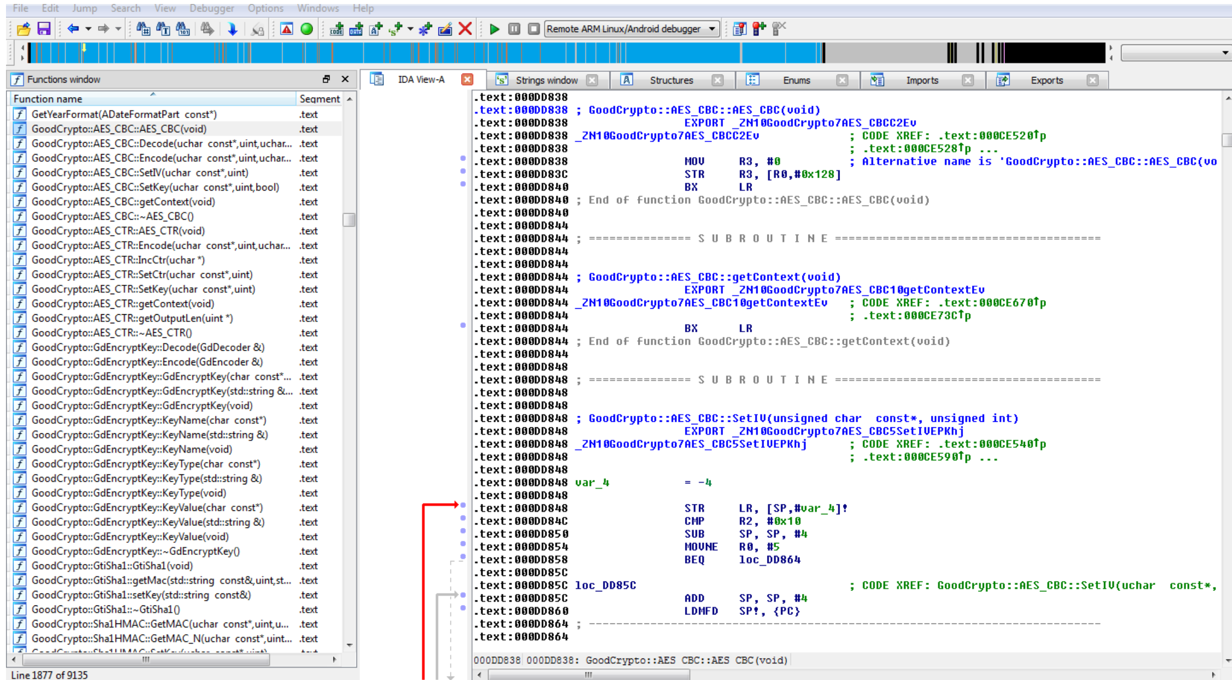
```
public boolean changePassword(String paramString1, String paramString2, boolean paramBoolean)
  throws ChangePasswordException, SetPasswordException, EncryptionException
{
  log.debug("changePassword");
  SecureDataAndPassword localSecureDataAndPassword = getSecureKeyAndPassword();
  if (localSecureDataAndPassword == null)
  {
    log.error("changePassword", "Trying to change password yet DB key has not been upgraded!?");
    throw new SetPasswordException("FATAL error: DB key has not been upgraded!");
  }
  if (paramBoolean)
    log.debug("changePassword", "Resetting password");
  byte[] arrayOfByte2;
  int j;
  try
  {
    while (true)
    {
      byte[] arrayOfByte1 = InitDB.getDatabaseKey();
      arrayOfByte2 = arrayOfByte1;
      if (($assertionsDisabled) || (arrayOfByte2 != null))
        break;
      throw new AssertionError();
      if (paramString1 == null)
      {
        log.debug("changePassword", "Old password is null");
        if (!localSecureDataAndPassword.isPasswordSet())
          continue;
        throw new SetPasswordException("Password is set so old password should not be null");
      }
      log.debug("changePassword", "Old password is not null");
      int i = 0;
      if (localSecureDataAndPassword.hasPasswordMatch(paramString1));
      try
      {
        byte[] arrayOfByte3 = localSecureDataAndPassword.decryptData(paramString1);
        i = 1;
        if (i == 0)
          throw new ChangePasswordException("Failed to verify password");
      }
```

4. Example of obfuscated code (Mobile Iron)

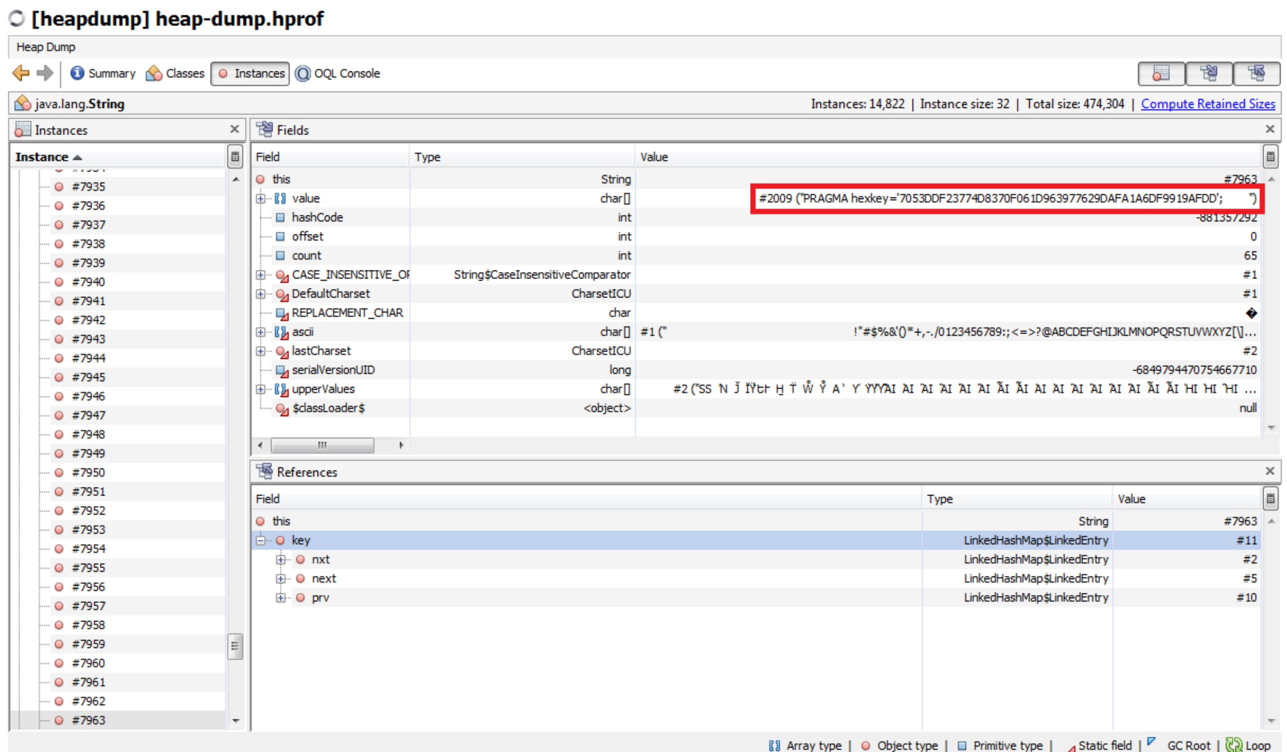
```
private void c()
{
  a locala = a.a();
  if (!g.a(ac.b().e()))
    j.c("", "");
  while (true)
  {
    return;
    long l1 = System.currentTimeMillis();
    d locald = new d();
    d.c(locald, l1 + 1000L * locala.n());
    long l2 = Math.min(1000L * locala.l(), this.d);
    this.d = (2L * this.d);
    if (this.d > 900000L)
      this.d = 900000L;
    j.c("", "" + this.d / 1000L + " sec");
    d.a(locald, l1 + l2);
    d.b(locald, 9223372036854775807L);
    d.c(locald, 9223372036854775807L);
    d.a(locald, b.b);
    this.b.add(locald);
  }
}

private void d()
{
  a locala = a.a();
  if (!g.a(ac.b().e()))
  {
```


iSECpartners part of nccgroup

5. No obfuscation in the native code parts (Good for Enterprise)



6. Master DB key from a memory dump using VisualVM (Good for Enterprise)



7. It is also possible to retrieve the key in memory by looking for UTF-16 Strings (Good for Enterprise)

```
 > ~/findstrings.sh heap-dump.hprof PRAGMA
's' option used with 'PRAGMA' on 'heap-dump.hprof'
GbhIDX_PRAGMA
PRAGMA
G`XHASH_PRAGMA
'S' option used with 'PRAGMA' on 'heap-dump.hprof'
GbhIDX_PRAGMA
Gc�PRAGMA
G`XHASH_PRAGMA
'b' option used with 'PRAGMA' on 'heap-dump.hprof'
'l' option used with 'PRAGMA' on 'heap-dump.hprof'
PRAGMA hexkey='A78D2017F53987E58350C438D028E5E890D884FB36684D15';
PRAGMA temp_store = 'MEMORY'
PRAGMA synchronous = 'NORMAL'
PRAGMA hexkey=
PRAGMA temp_store_directory =
PRAGMA default_cache_size = 96
'B' option used with 'PRAGMA' on 'heap-dump.hprof'
'L' option used with 'PRAGMA' on 'heap-dump.hprof'
 >
```

8. This results in being able to decrypt all the database files including emails (Good for Enterprise)

```
Good Admin<
Welcome to Good Messaging----------------
I. HANDHELD SETUP
----------------
This one-time setup takes
Vminutes depending on your specific handheld and network coverage.  Before you begin, ensure you are in a strong network coverage area.
Step 1: Using your handheld's browser, visit https://get.good.com.
Step 2: Download the OTA Setup program or the Good for Enterprise application depending on your handheld type.
Step 3: Locate and launch the application just downloaded on your handheld (see details below for your specific handheld).
Step 4: Log in with the following information (case insensitive):
        Email Address:
        PIN:
        ***Your PIN will expire: Never
Step 5: Follow the prompts to provision Good on your handheld.
------------------------------------------------------------
II. DETAILS FOR LOCATING OTA SETUP ON YOUR SPECIFIC HANDHELD
------------------------------------------------------------
ANDROID
1.      After download completes, the Good for Enterprise application will be automatically installed.  Once installation complet
\Kqc
~ /u[
he Good application.
2.      Enter your email address and PIN from Step 4 above to complete the setup process.
3.      After completing setup, you can place the Good application and the Good Widget on your home screen for quick access.
IPHONE
1.      After download completes, launch the Good application.
2.      Enter your email address and PIN from Step 4 above to complete the setup process.
PALM OS
1.      After OTA Setup download completes, select OK.
2.      OTA Setup will start automatically.
3.      Press the "start" button.
4.      Enter your email address and PIN from Step 4 above to complete the setup process.
SYMBIAN (NOKIA)
1.      After OTA Setup download completes, press the Applications key which is left of the joystick.
2.      Open the Apps folder and select the OTA Setup application.
3.      Enter your email address and PIN from Step 4 above to complete the setup process.  For handhelds with a numeric keyboard use th
e following PIN:
***N
 >
```