Inside Apple's MDM Black Box

INTREPIDUS GROUP
MOBILE SECURITY

David Schuetz, Senior Consultant
david.schuetz@intrepidusgroup.com
@schuetzdj

black hat usa+2011
EMBEDDING SECURITY

Feedback!

INTREPIDUS GROUP
MOBILE SECURITY

Please complete the
Speaker Feedback Surveys

This will help speakers to
improve, and can help Black Hat
make better decisions regarding
content and presenters for
future events.

Introduction

- Who am I
  - Background: Computer science, help desk
  - SysAdmin led to INFOSEC roles in mid 90s
  - Like building tools, solving puzzles
- Joined Intrepidus Group in August 2010

First a little bit about me. My degree is in Computer Science, but instead of becoming a programmer I got sucked into the exciting world of help desk and system administration. In about 1995, I transitioned into the first of many roles more focused on information security. Since then I've spent time supporting security assessments, done some R&D for ways to detect malicious insiders, and provided technical support, guidance, and testing services for certification and accreditation work.

Last summer (not long after DEF CON), I joined Intrepidus Group. I've worked several different kinds of projects since joining, including penetration testing, and web and mobile application testing, but most of the last 6+ months I've spent supporting large-scale enterprise-wide iPad deployment.

Why am I here?

- Have customers who need MDM
- Ask questions like "How secure is it?"
  - We'd like to be able to answer them
- To answer, need to understand it
- Protocol wasn't published
  - So we thought we'd share.

So why am I here today?

In a nutshell, we have customers who need mobile device management capabilities, especially for iOS platforms. As security consultants helping them to design the most secure deployment possible, they naturally ask us "How secure IS this system?" And, frankly, we'd like to be able to give them a good answer.

But to answer that question well, we have to understand just how MDM works. Unfortunately, the protocol hasn't been openly published, so not many people really understand it that well. We understand it (at least, we feel like we do), and so we thought we'd share so that others can build on our knowledge and better help their organizations and customers.

Why are you here?

INTREPIDUS GROUP MOBILE SECURITY

- Learn how MDM actually works
- Review configuration settings
  - Entered directly on device
  - Imposed via profiles
  - How these differ
- Dig into the MDM protocol
  - Enrollment, commands, responses
  - Write your own server (or use the one I wrote)
  - Bugs, weaknesses, Good Things
- New twist on an old attack

Here's what I hope you'll get out of this talk: Though the broad strokes of MDM are well understood, you may not know much about the lower-level details. Hopefully, you'll come away from this understanding how it actually works.

We'll start off by reviewing how configuration settings work on iOS devices -- especially how user-entered and profile-controlled settings differ. Then we'll look at the MDM protocol itself, how it enrolls devices, and how devices receive and respond to commands. I'll also present a do-it-yourself server that can be used for further MDM research, and discuss some of the good things, bad things, weaknesses, and bugs I've encountered.

FInally, I'll show you a new twist on an old attack.

No NDA Here!

INTREPIDUS GROUP MOBILE SECURITY

- Lion now officially released
  - But didn't use it anyway
- iOS 5 not officially released
  - Nothing here relates to iOS 5 specifically
  - All demonstrations, tests, etc., use iOS 4
  - Haven't even installed iOS 5 yet :(

I'd also like to take a moment to point out that this talk represents work with published, openly available information, software, and devices. Though Lion has now officially been released, the use of Lion in the research and development of this talk was extremely limited.

Likewise, iOS 5 has been available to developers for some time, but nothing here even touches on iOS 5. All demonstrations, test, and research, were performed using version 4.x -- I haven't even installed iOS 5 on anything yet.

Basic iOS Controls

To understand just what MDM is actually providing for us, let's talk for a moment about some basic iOS controls.

End User Controls

- Basic stuff
- Passcodes, restrictions, etc.
- Entered by user, can be removed by user
- May password protect settings
  - But limits additional "personal" controls

At its most basic level, iStuff are controlled by the primary user of that device. This is basic stuff -- locking the device with a passcode, restricting use of certain applications (so your kids can't buy apps, for example), or configuring email or VPN accounts.

These settings are entered by the user, so they can be removed by the user. The user (or their IT department) may restrict access to certain settings with a password, but then that may also limit other things the user may do.

Demo:
Basic
Settings

So let's run through a few settings on the screen first. We've got a camera on a tripod, and an iPad on a stand, and hopefully they'll play nice together. Nothing special, just a few quick examples of how settings work at the end-user level.

Passcodes

- Basic settings only
  - Simple (4 numbers) or complex
  - Require immediately or after grace period
  - Erase all data after 10 attempts
- Picture Frame mode
  - Shows photos when locked
  - Anything sensitive in album?

General	Passcode Lock
Turn Passcode Off	
Change Passcode	
Require Passcode	Immediately >
Simple Passcode	ON

A simple passcode is a 4 digit number.

| Picture Frame | ON |
| Erase Data | OFF |

Erase all data on this iPad after 10 failed passcode attempts. Data protection is enabled.

First, we look at passcode settings. There's not much the user can configure here -- simple (4-digit numbers) or not-simple (full-keyboard), whether to erase after 10 invalid attempts, etc. Also of note is the Picture Frame mode, which might escape attention: When active, anyone holding the device can activate a slideshow even when it's locked. So if there's any photos with sensitive information in them in your photo album, they might appear here.

Managing Restrictions

- Can disable
  - Selected applications
  - Certain features or functions
  - Changes to accounts
- Also (not shown):
  - Parental Controls
  - Game Center features
  - In-App Purchase disable

Allow:

Safari		ON
YouTube		OFF
Camera		ON
FaceTime		ON
iTunes		OFF
Ping		OFF
Installing Apps		OFF
Deleting Apps		OFF

Allow Changes:

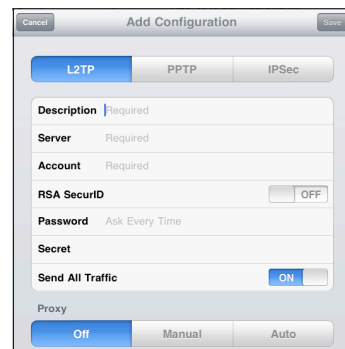|	
Location	>
Accounts	>

Next up is the restrictions panel. In here you can disable certain applications or features, can prevent changes to accounts (like adding email accounts), and so forth. Also there are parental controls available, various Game Center features, things like Ping and FaceTime, and also the ability to disable in-app purchases.

Other Settings

- Email accounts
  - Plus calendars, notes, etc.
- Wi-Fi networks
- VPN settings
- etc., etc.

Cancel	Add Configuration	Save

L2TP	PPTP	IPSec

Description	Required
Server	Required
Account	Required
RSA SecurID	OFF
Password	Ask Every Time
Secret	
Send All Traffic	ON

Proxy

Off	Manual	Auto

Finally, there are all the other general built-in settings, like email accounts (including calendars, notes, etc.), Wi-Fi networks, VPNs, etc. All these are set by the user, but in many cases, it'd be very useful to control them remotely, at an enterprise level. For one, that allows for "instantaneous" updates when settings are changed, and for another, that eliminates a lot of the "tap here, then enter this" sort of handholding from the help desk.

So how do we do that?

Pre-Built Profiles

We build profiles ahead of time, and shove 'em out to phones as needed.

Configuration Profiles

- Good for IT management
- Install standard profile on many devices
- Can be locked with password
  - But won't preclude other personalizations
- Most-restrictive superset of settings
  - If user chooses "simple passcode" ...
  - But management profile requires complex....
  - ....management wins!

One of the best reasons to consider standardized profiles is because they can make things much easier for management. Rather than relying on users to update policies, or having admins manually enter policy changes on devices, a single profile can be saved to disk and then loaded onto devices.

These profiles can also be locked, so the user can't remove them. But unlike locking settings within the Settings application, this lock only applies to the settings within the profile. So the user is still free to make other personalizations that don't conflict with the installed profile.

In cases where there are conflicts, the most secure setting wins out in the end.

Increased Flexibility

- iPhone Configuration Utility (IPCU)
  - Windows, OS X
  - Install profiles, certificates
  - Transfer profiles via USB
- Many new controls not available on device
  - More supported VPNs
  - Extensive passcode controls
  - Safari controls

The easiest way to create standardized profiles is via the iPhone Configuration Utility. This free application lets you create and install profiles, install provisioning profiles (for application development and enterprise deployment), etc.

Profiles can be installed to devices via USB, or they can be distributed via a web server (user simply taps on a URL and it downloads and installs the profile).

Additionally, the IPCU provides access to controls and settings not available directly on the device, including better VPN support, extensive passcode controls, and some more detailed Safari security settings.

Demo: IPCU

A quick run-through of IPCU, showing how it works, and where some of the additional settings reside.

Passcodes

- **Significant additional functionality**
  - Minimum length
  - Non-alphanumeric
  - Aging and history
  - Change "erase after" rule
    - As low as 4 failures
    - As high as 16
    - (default is 10)

☑ **Require passcode on device**
Enforce the use of a passcode before using the device

　☑ **Allow simple value**
　Permit the use of repeating, ascending, and
　descending character sequences

　☑ **Require alphanumeric value**
　Require passcodes to contain at least one letter

　[5 ⇕] **Minimum passcode length**
　Smallest number of passcode characters allowed

　[1 ⇕] **Minimum number of complex characters**
　Smallest number of non-alphanumeric characters allowed

　[180 ⇕] **Maximum passcode age (1–730 days, or none)**
　Days after which passcode must be changed

　[2 minutes ⇕] **Auto-Lock (1–5 minutes, or none)**
　Device automatically locks when time period elapses

　[5 ⇕] **Passcode history (1–50 passcodes, or none)**
　Number of unique passcodes before reuse

　[Immediately ⇕] **Grace period for device lock**
　Amount of time the device can be locked without
　prompting for passcode on unlock

　[11 ⇕] **Maximum number of failed attempts**
　Number of passcode entry attempts allowed before all
　data on device will be erased

IPCU provides significantly enhanced controls for passcodes. For example, minimum length, minimum number of non-alphanumeric characters, and passcode aging and history retention. Also, the rule for whether to erase after 10 passcode failures can be modified, to erase the device after as few as 4 failures or as many as 16.

Restrictions

- **Again, significantly enhanced**
  - Lock out camera / FaceTime
  - Disable screen capture
  - Force encrypted backups
  - Block popups, restrict cookies

Device Functionality
Enable use of device features
- ☑ Allow installing apps
- ☑ Allow use of camera
  - ☑ Allow FaceTime
- ☑ Allow screen capture
- ☑ Allow automatic sync while roaming
- ☑ Allow voice dialing
- ☑ Allow In App Purchase
- ☑ Allow multiplayer gaming
- ☑ Allow adding Game Center friends
- ☐ Force encrypted backups

Applications
Enable access to applications on the device
- ☑ Allow use of YouTube
- ☑ Allow use of iTunes Music Store
- ☑ Allow use of Safari
  - ☑ Enable autofill
  - ☐ Force fraud warning
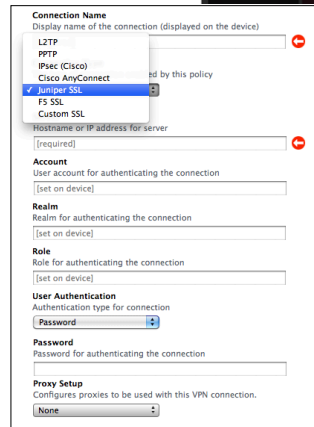 - ☑ Enable JavaScript
  - ☐ Block pop-ups

Accept cookies
[Always ⇕]

Similarly, the restrictions screen has some enhancements, including being able to disable the camera, disable screen captures, and force encrypted backups.

Additional restrictions are present for Safari, as well -- form auto-fill can be disabled, as can javascript and popups, and even some basic cookie management.

New VPN Types

- Cisco AnyConnect
- Juniper SSL
- F5 SSL
- Custom SSL

Connection Name
Display name of the connection (displayed on the device)

L2TP
PPTP
IPsec (Cisco)
Cisco AnyConnect
✓ Juniper SSL
F5 SSL
Custom SSL

Hostname or IP address for server
[required]

Account
User account for authenticating the connection
[set on device]

Realm
Realm for authenticating the connection
[set on device]

Role
Role for authenticating the connection
[set on device]

User Authentication
Authentication type for connection
Password

Password
Password for authenticating the connection

Proxy Setup
Configures proxies to be used with this VPN connection.
None

Finally, the VPN settings have support for four additional VPN types that aren't available through the Settings application. (I don't know if anyone's figured out how to make this work with OpenVPN yet...not sure whether that's possible or not.)

Over-the Air Delivery

- No USB connection necessary
  - No need to push a cart thru the building
  - Can use simple web links to download profile
- More secure OTA method
  - Documented by Apple
  - Uses SCEP to establish secure exchange
  - Not easy to implement
    - Apple suggests MS or Cisco SCEP servers

- Still need user to visit link and install profile

Of course, creating the standardized profile is only half the problem. For a large enterprise, the prospect of pushing a cart through the building and manually connecting each device via USB to load up current profiles is...well...pretty much a non-starter.

Distributing them through an online link is much better, but that still has some concerns (such as controlling access to the profiles, etc.). Some of these concerns are addressed in a more formal Over-the-Air system, as documented by Apple. This approach is more secure (using on-the-fly exchanges of certificates via Simple Certificate Exchange Protocol, for one), but because of the use of SCEP, its complexity level is pretty high.

And in either case, you still need for the end user to actively visit a link and install the profile. And if they know the profile is going to disable YouTube, how many do you think will willingly click on it?

Mobile Device Management

So this is where MDM comes in.

MDM Basics

- OTA + Push Notifications
- Sends profiles directly to device
- Can update enterprise in minutes
- No reliance on end-user
  - However, user can remove MDM
- Same configuration features as IPCU
  - Including extended configuration options
- Additional management and control features

In many ways, MDM is just the OTA framework, paired with Push notifications (with a few new features thrown in).

Using MDM, the server can push profiles directly to devices, potentially updating an entire enterprise in just minutes (device availability and network vagaries notwithstanding). Best of all, though, there's no reliance on the end user to initiate the updates. They all happen without the user's interaction (though some commands require the device to be unlocked first).

MDM also permits remotely querying the device for settings, listing of applications, etc., and adds remote controls like locking and wiping.

Commercial Solutions

INTREPIDUS GROUP
MOBILE SECURITY

- Available from several vendors
- Many go well beyond iOS MDM
  - Blackberry
  - Desktops
  - Asset Management, etc.
- Also: Now part of Lion Server ($49.99)
- Need a Push Notification (APNS) certificate
  - Previously only through Developer program
  - Now available with Lion Server

MDM server capabilities are available from several 3rd party vendors. Many of these offerings go well beyond just iOS MDM, and can manage Windows phones, Blackberries, and desktops, or even get into the realm of full-out asset management systems.

The newly-released Lion server can also get you a workable MDM solution for iOS only (and possibly OS X clients), for about $50.

In order to send push notifications to the client, the server needs a certificate/key from Apple. Previously, these were only available through the developer program (I believe only through the Enterprise Developer Program), but with Lion they appear to be available for free (or were during the beta program, at least).
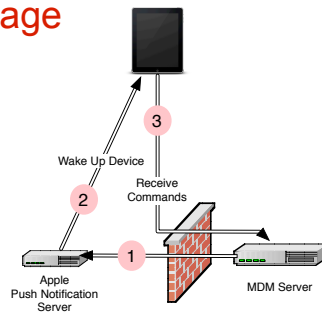
INTREPIDUS GROUP
MOBILE SECURITY

How does it work?

So now we come to the reason you're here. How does MDM work?

Overview

- Enroll device
  - Installs profile linking device to MDM

- MDM server pushes message
  - "Hey, call me"
- Device connects to server
  - "You rang?"
- Server provides command
- Device sends response

First, the device has to enroll with MDM (essentially, installing a configuration profile that links the device to the MDM server).

Once that's complete, the flow of control is like this:
* The server queues up a command for the device
* It then sends a notification to the device via the Apple Push Notification Service (APNS)
* The device receives the notification, and contacts the server
* Once connected to the server, it downloads (and acts upon) the queued command

Push - Client

- Device connects to Apple using TLS
- Two-way certificate validation
- Apple provides unique token to device
  - Apps forward token to service providers
- Device leaves connection open
- Notifications flow from Apple to Device

Before we get to the meat of the MDM commands, let's look more closely at APNS.

The APNS connection is the only long-term connection established for MDM, and serves as the conduit for all kinds of notification traffic to the device.

When creating the APNS connection, the device first checks Apple's TLS certificate. The Apple server also checks that the device itself has a valid certificate (which was provided to the device, by Apple, when the device was activated). So getting into the middle of this exchange should actually be pretty difficult.

Once the connection is established, applications request, through the connection, unique Device Tokens, which they then forward to their service provider.

Notifications - Server

- Short message
  - Payload <= 256 bytes
- Sent in JSON format
- Addressed by the DeviceToken
- Signed by originator and sent to Apple

Cmd	Token Length		Device Token	Payload Length		Payload (JSON encoded)
0	0	32	(binary data)	0	34	{"aps":{"alert":"You have mail"}}

(image source: Apple)

Once the provider has the Device Token, they can send push notifications to the device. The payload is JSON formatted, and total length (including JSON structure) limited to 256 bytes. The Device Token and some framing information are prepended to the payload, and the whole bit is wrapped and signed by the Push Notification certificate obtained from Apple.

Once received by Apple, the notification is sent to whichever device corresponds to the Device Token (presuming it's got an active APNS session), and thus, the notification is delivered.

MDM Notifications

- Does not have the "aps{}" field
- Instead, single string "mdm"
- Value is another token, "PushMagic"
- PushMagic, DeviceToken, and Certificate ID
  - Together authenticate MDM push message

Cmd	Token Length		Device Token	Payload Length		Payload (JSON encoded)
0	0	32	(binary data)	0	47	{"mdm":"040ac7bf-391a-4a36-a8ab-47bd380afd33"}

MDM notifications are pretty similar, but instead of the top-level "aps {}" dictionary, the payload contains only a top-level string named "mdm". The contents of that string are another token provided by the device during enrollment, the "Push Magic" token.

So the device needs to have match three items in order for a push notification to trigger an MDM response: The Device Token (without which the notification will never reach the device), and the Push Magic token (without which the MDM client will just discard the notification). Finally, the "Subject Name / User ID" field in the push notification certificate used to sign the notification must match the "Topic" field in the MDM profile.

These three items, especially the User ID on the certificate signed by Apple, together make it fairly difficult (if not impossible) to forge a push notification to the MDM client on the device.

Enrollment Profile

- Install enrollment profile
  - URLs for enrollment and server
  - Associate with MDM's push certificate
  - Provide identity certificate to device
  - Set MDM rights over device
 - Can install OTA, or via USB

I just mentioned PushMagic being sent during enrollment. So let's talk about enrollment.

The enrollment profile, pushed from a server to the device, defines some key items: The URLs needed to access the service, the topic (User ID) for the Push Notification certificate, a certificate to uniquely identify the device, and rights the device offers to the MDM service.

Upon receipt of this profile, the device then connects to the server and authenticates, then provides the necessary tokens to the server.

Enrollment - Network

- Authenticate
  - Identifies device to server
  - MDM server may decline
- TokenUpdate
  - Sends tokens to server
  - Enables push notification
  - Keys to unlock device

MDM Server | Mobile Device

Install Enrollment Profile

Authenticate

TokenUpdate
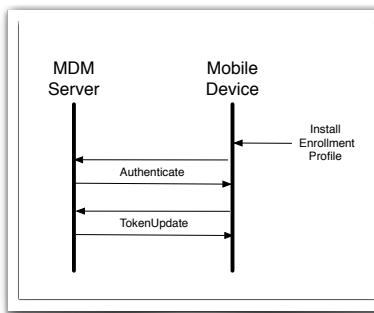
The enrollment profile can be installed through IPCU directly, or through an IPCU-created profile pulled off a web link, or by another process hosted on the MDM server.

Once the profile is installed, the device sends the Authenticate message, then in a separate connection, the TokenUpdate message.

Authenticate

- **HTTP PUT to "Check In" URL**

```
PUT: /checkin
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC
   "-//Apple//DTD PLIST 1.0//EN"
   "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
   <dict>
      <key>MessageType</key>
      <string>Authenticate</string>
      <key>Topic</key>
      <string>com.example.mdm.pushcert</string>
      <key>UDID</key>
      <string> [ redacted ] </string>
   </dict>
</plist>
```

The first step is the device authenticating itself to the server. It's not so much a secure authentication as it is a simple identification, but it provides the server a chance, based on the Topic and UDID provided, to accept or reject the enrollment request. This can be used to prevent just anyone from enrolling with the corporate MDM server. (additional steps at a more user-friendly level, such as web-based authentication panels with real usernames/passwords, are also possible, and can be used prior to creating an enrollment profile).

The data is presented to the server over HTTP (perhaps TLS-encryped), using PUT, to the URL specified in the enrollment profile's "Check In" field. The data itself is an Apple Property List (.plist) file, XML formatted. [key elements of the data are highlighted here in red]

Authenticate Response

- **Server can decline enrollment**
- **Valid "ok" response is a blank plist**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC
   "-//Apple Computer//DTD PLIST 1.0//EN"
   "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
   <dict></dict>
</plist>
```

If the server accepts the enrollment request, it doesn't need to provide anything beyond a simple, blank .plist file.

Token Update

- Via HTTP PUT to /checkin
- Blank plist also a valid response

```
<dict>
    <key>MessageType</key>
    <string>TokenUpdate</string>
    <key>PushMagic</key>
    <string> [ uuid string ] </string>
    <key>Token</key>
    <data> [ 32 byte string, base64 encoded ] </data>
    <key>Topic</key>
    <string>com.example.mdm.pushcert</string>
    <key>UDID</key>
    <string> [ redacted ] </string>
    <key>UnlockToken</key>
    <data> [ long binary string encoded in base64 ] </data>
</dict>
```

The next step is for the client device to provide some additional special data to the server. This is the source of the aforementioned PushMagic string, as well as the DeviceToken provided by Apple when connecting to APNS, and the Push Notification "Topic" string. These three items permit the server to send MDM notifications to the device, and without it, MDM simply won't work.

The last, most interesting item here is the UnlockToken. As best as I can determine, this (about 2kb) binary stream includes an actual key to an escrow keybag on the device. This token is used to clear the passcode on a locked device.

Again, this message is sent via HTTP PUT over the "Check In" URL. And, again, all the server needs to respond with is a blank .plist.

Command List

- Configuration
  - Install and Remove Configuration Profiles
  - Install and Remove Provisioning Profiles
- Status
  - Device Information, Security Info, Restrictions
  - List Apps, Certs, Profiles, Provisioning Profiles
- Control
 - Device Lock
  - Clear Passcode
  - Erase Device

There are a total of 14 commands available to the MDM service. Roughly, I've grouped them into 3 categories: Those which modify the configuration of a device, those that query the device for some kind of status or informational data, and those that actually remotely control the device.

Command Flow

- First, device connects to APNS server
- Four connections per command:
  - Server to APNS
  - APNS to Device
  - Device to Server
    - Receives Command
  - Device to Server
    - Provides response
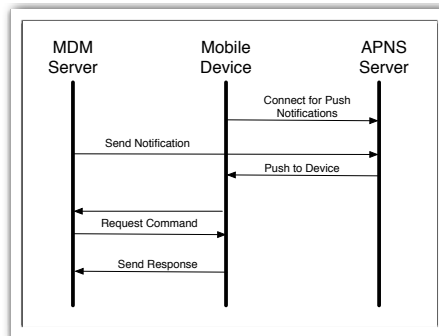
First, the device must make the persistent connection to Apple, to receive push notifications. Then, for every command the MDM server sends, four total connections are created: The server sending the message to Apple, Apple sending it down (the already established) connection to the device, then the device asking for (and receiving) the command from the server, and finally the device providing an response to the server.

Client Initial Connection

- Simple "status" message
- Identifies self via UDID
- Triggers server to provide command

```
<plist version="1.0">
<dict>
   <key>Status</key>
   <string>Idle</string>
   <key>UDID</key>
   <string> [ redacted ] </string>
</dict>
</plist>
```

Now that the device is fully enrolled, it can respond to push notifications by contacting the "Server URL" address. When connecting, the client PUTs a simple message identifying "Status = Idle", and the server responds with a command, again in a .plist format.

The client executes the command, and responds with a simple acknowledgement message, or with appropriate data as requested by the server.

Command Format

- Command in "RequestType" field
- CommandUUID field mandatory
  - Content optional
  - Allows matching of commands and responses

```xml
<dict>
   <key>Command</key>
   <dict>
      <key>RequestType</key>
      <string>DeviceLock</string>
   </dict>
   <key>CommandUUID</key>
   <string>9bbef8e8-f9ef-49fb-92f4-792faa1b6b04</string>
</dict>
```

The commands are framed in .plist format. A top-level key "Command" is made up of a dictionary of key/value pairs. Usually this will include only the single key "RequestType," which has as its value the actual command being sent. In some cases, additional parameters will get added as peers to the RequestType key.

The command also includes a UUID string that uniquely identifies the instance of the command. This is useful for matching up responses with the command that caused them to be sent, especially when several commands may be queued up but aren't necessarily delivered, or processed, in order.

In practice, this field can actually be left blank, however it's probably best practice to fill it with some random (or controllable) UUID.

Optional Parameters

- Parameters added to Command dict
- Most commands have no parameters

```xml
<dict>
   <key>Command</key>
   <dict>
      <key>RequestType</key>
      <string>ClearPasscode</string>
      <key>UnlockToken</key>
      <data>[ redacted ] </data>
   </dict>
   <key>CommandUUID</key>
   <string>69c4597f-4640-437c-8fac-b341af659212</string>
</dict>
```

This shows the ClearPasscode command, with the additional UnlockToken parameter. That token contains the key to unlock the device when a passcode has been set, and was sent by the device during enrollment.

When the device receives this command, it can use the data within the UnlockToken to open (what I believe to be) an escrow keybag on the device, which then allows access to the passcode keybag. Or perhaps it simply contains all the master keys, just like the passcode bag, and so can delete and re-create the passcode keybag (just without a passcode set). The exact mechanics haven't been documented, and I didn't dig too deeply.

It's enough to know what if you have this token, and can get the command interpreted by the MDM client on the device, you can clear the device's passcode and unlock it.

Generic Response

- Returned for many commands
  - Basically, anything other than a query cmd
- If not Acknowledged, likely will get error msg

```
<plist version="1.0">
<dict>
   <key>CommandUUID</key>
   <string>69c4597f-4640-437c-8fac-b341af659212</string>
   <key>Status</key>
   <string>Acknowledged</string>
   <key>UDID</key>
   <string> [ redacted ] </string>
</dict>
</plist>
```

The standard response is simply an acknowledgment of the command. If the command caused an error, then an error code and strings (localized and in US English) are returned in the plist data. For those commands which require a more detailed response, a more detailed plist file is returned.

Response - Device Info

```
{'CommandUUID': '105ac7bd-5bce-430c-a675-38ea890526f6',
 'QueryResponses': {'AvailableDeviceCapacity': 12.210990905761719,
            'BluetoothMAC': '--redacted--',
            'BuildVersion': '8J3',
            'CarrierSettingsVersion': '10.0',
            'CurrentMCC': '000',
            'CurrentMNC': '00',
            'DataRoamingEnabled': False,
            'DeviceCapacity': 14.020820617675781,
            'DeviceName': 'IG Chantilly',
            'IMEI': '--redacted--',
            'IsRoaming': True,
            'Model': 'MC823LL',
            'ModelName': 'iPad',
            'ModemFirmwareVersion': '07.11.01',
            'OSVersion': '4.3.3',
            'ProductName': 'iPad1,1',
            'SIMCarrierNetwork': 'AT&T',
            'SerialNumber': '--redacted--',
            'UDID': '--redacted--',
            'WiFiMAC': '--redacted--'},
 'Status': 'Acknowledged',
 'UDID': '--redacted--'}
```

And here's the result from just such a call: The "DeviceInformation" command. Several parameters are available to be queried, and those that pertain (or have a value) to the device queried are returned in this structure. This includes some identifying information, such as IMEI, Serial Number, UDID, MAC addresses, Phone Number, and so forth.

Believe it or not, we now know enough to create our own server.

DIY MDM Server

DIY Server

- Fewer than 300 lines of python
- Uses standard libraries, plus:
  - web.py
  - APNSWrapper.py
    - OpenSSL (command-line)
- Simple command and response console
  - Send a command
  - Look at the response

DIY Server

- Very rough:
  - Implements all commands
  - But not all responses
  - Only one device at a time -- "last in wins"
- No guarantees
- Good starting point for experimentation
- White paper on conference CD
  - More complete documentation
  - Complete source code

It's a pretty rough server, designed only for testing MDM and researching how it works. It implements all the commands, but not all the responses (for example, it doesn't provide, nor respond to, error messages).

It's also only capable of tracking a single device at a time -- so the last device that gets enrolled, that's the one you're controlling. :)

The source for this is available in the white paper (which should be on the conference CD), and should also be available on my company's website.

Demo: Mini MDM

MDM Limitations

- **User can terminate MDM relationship**
  - Will lose whatever MDM installed
  - Corporate profile settings, etc.
- **Doesn't address co-mingling of data**
  - Will probably require multi-user model
- **Can't detect jailbreak**
- **Needs push notification certificate**
  - Much lower barrier with Lion server

The MDM system is not without some limitations, however. First, the user can terminate the MDM relationship at any time by simply deleting the MDM profile. Any profiles which had been installed by MDM (web clips, account information for corporate email, etc.) also get deleted. MDM also doesn't address the co-mingling of data, such as easily dragging messages from the corporate account to a personal email account.

There was talk that Apple had included a jailbreak detection command in an earlier version of MDM, but it's not there now. Third party services can install their own clients to supplement the MDM feature set, and some of these do their own jailbreak detection, but there's nothing built into the MDM system to do so.

Finally, you need the push notification certificate. This is now a much lower barrier, though.

Missing Features

- **Change (not just erase) passcode**
- **Can't disable microphone**
- **Some basic settings not available in MDM**
  - PictureFrame restriction
  - Can't lock down accounts
- **Geolocation not available**
  - Find My iPhone appears to use different system

Beyond the limitations, there are some simple features that'd be nice to have. For example, it'd be great if the MDM could not just clear the passcode, but set a whole new one. This would be useful for a corporate device where the owner is leaving the company, and we want to lock them out of the device but not wipe it quite yet.

You also can't lock down the creation of accounts, though that feature does exist in the Settings application.

It'd be nice to disable the microphone, for some environments, and finally, geolocation does not appear to be available through MDM. You need the Find My iPhone system to do that.

Interesting Bugs?

- Clearing passcode delay bug
  - Doesn't immediately interrupt delay
  - Doesn't clear failure count
    - Don't mistype your new passcode the first time
    - Otherwise count continues, you're delayed again

First, after several passcode failures, a delay gets added between attempts. If MDM clears it, the device won't immediately open -- the current passcode entry delay must first timeout (which can be up to an hour).

Worse, the failure count doesn't appear to be cleared. So you get unlocked, into the device, and set a new passcode. Then if you lock it, and immediately mistype the passcode, you're back where you started -- if you had 7 failures before, then that's # 8, and you now have to wait 15 minutes to try again.

Lock Out Bug

- Clear simple passcode with panel up

Another bug: when you have a simple passcode, and it's cleared while the numeric keypad is up. When that happens, entering any random 4-digit code will let you in, while hitting "cancel" kills the panel, but also kills the "swipe to unlock". So the device is locked, at a (mostly) blank screen, and you can't do anything at all other than power-cycle it.

Security Concerns

- No command authentication
  - "Sign message" option not enforced
- TLS authentication
  - Appears to accept any cert with 'trusted' root
  - MITM likely possible
- Configuration not protected on filesystem

The profile has the option to "sign messages," but even when that's set the device accepts plain, unsigned commands with no real authentication.

It also appears that there's not much authentication for TLS -- it just checks for a validly rooted cert. So MITM attacks are likely possible.

Finally, the MDM configuration is not protected by the FileProtection system -- so it's readable without the passcode. Several techniques exist whereby a locked device can be booted into a special mode, providing access to the filesystem, which means that MDM details can be read, or potentially even changed.

Possible Attacks

- Can't forge APNS, so need to MITM
- Downgrade security requirements
- Multi-stage malware install
  - Push provisioning profile
  - Push webclip
  - User clicks webclip & installs program
    - Combine with social engineering encouragement
- Denial of Service

You could downgrade security requirements on the device, such as reducing passcode complexity requirements.

A multi-step attack could install a profile that includes a signed provisioning profile, and a webclip placed on the user's home screen. When the user taps on the webclip, it installs a custom app, which might include malware payloads.

Finally, great mayhem: If you can manage to position your MITM server "just outside" the legitimate MDM server (that is, MITM at the enterprise level, not just against a single targeted device) ... then you could conceivably just intercept all the devices doing a daily checkin and make them all..er...erase themselves.

Good Security Findings

- Things you can't do:
  - Install profile on locked device
  - Read installed profile details
    - List profiles gives meta-data only
  - Make locked device re-send unlock token

There are a couple of very good points, though: You can't install a profile on a locked device (if there's a passcode), nor can you induce a locked device to resend you the unlock token. (Again, my theory is that the token is a key to an escrow keybag, and once the device is locked, such keys are wiped from memory, so it should be impossible for the device to provide the unlock key when it's locked). Also, though MDM can list the profiles installed on a device, it can't actually read the contents of those profiles. So it might see a profile named "Corporate Email Settings," but the details like the server name, userids, passwords, etc., won't be returned via the MDM call.

One more thing...

Overwrite MDM Profile

- Can't install profile on locked device
- Can install when device unlocked
- If you reinstall the MDM profile....
- ....it re-enrolls in MDM.
- Which sends a new UnlockToken

As I was finishing up the first draft of these slides, I was testing some limitations of profile installation. And accidentally installed the MDM profile (instead of the profile I used in the demo). Imagine my shock when I saw new tokens appear in my debug window.

Evil Maid Attack

1. Get device to talk to your sever
2. Install copy of MDM profile
3. Receive new UnlockToken
4. Send ClearPasscode
5. Copy device, leave

The "Evil Maid" attack has been known for some years now. It exemplifies the dangers associated with physical access to a device -- if you can touch it, you can own it. In the traditional attack, a bribed hotel maid (or someone disguised as one) boots a computer using full disk encryption from a USB fob, installing malware to the root partition / boot sector. That malware eventually collects the password to decrypt the drive, providing the attacker with access to the data. This is simply an iOS version of the attack, using MDM as the main attack vector.

Man in the Middle

INTREPIDUS GROUP
MOBILE SECURITY

- Use standard MITM techniques
  - Wi-Fi trickery, forge SSL cert, etc.
- Or, since you have physical access to device:
  - Get onto device
  - Add entries to /etc/hosts
  - Install your own certificate

There are many ways to get into the middle of the MDM conversation. If the MDM server uses HTTPS (and it should), then some SSL trickery may be required, either to exploit weaknesses in certificate validation, or to somehow get a forged (but apparently signed) cert, or to place a rogue CA cert onto the device.

Duplicate MDM Profile

INTREPIDUS GROUP
MOBILE SECURITY

- Gain access to device filesystem
  - Use DFU / tethered boot magic
- Retrieve copy of MDM profile "stub"
  - In /var/mobile/Library/ConfigurationProfiles
- Extract original MDM profile from stub
  - It's in there, base-64 encoded

The tricky bit is that the profile being re-installed must match *exactly* with what the device already has. This is made much easier by the fact that a copy of the profile is stored on the device, and can be easily retrieved.

Alternatively, if every single device in an organization uses the same profile to install MDM (using SCEP services instead of bundling a device identity cert), then it might be possible to simply acquire a copy from another device. (This could make the "Evil Lackey" variant described later much easier to accomplish.)

Push Profile to Device

- Can't send the push message
- But can instruct device to poll
  - MDMOutstandingActivities.plist
    - The "Status: Idle" message
  - Place into ConfigurationProfiles folder
  - Device will poll MDM server
- If command is "InstallProfile":
  - Device responds "Not Now"
  - Waits until unlocked, then tries again

No matter what we retrieve from the device, it's unlikely we'll be able to forge the push messages through Apple. So the attacker could simply wait for the legitimate MDM server to send a push message, or we could force a poll of the server by adding a special file to the device.

Once the device sees that file, it contacts the server. If the server then responds with the installation command for our MDM profile copy, the device will say "Not Now" and refuse to install. But as soon as the device is unlocked, it will remember that it deferred a profile, contact the server, and complete the command.

Re-Installing MDM

- If device receives new MDM
  - Fails: "Device already managed by MDM"
- Unless new profile exactly matches old
  - Then, the device silently re-enrolls
  - Sends Token, PushMagic, and UnlockToken
- Since we're MITM, those come to us
- Can't use the APNS tokens
  - Don't have right push cert

Normally, installing a new MDM profile will result in an error from the device. But as mentioned, if the profile exactly matches, then the device will simply "re-enroll" in the same service. It does this silently, without any indication to the user at all.

Since the MDM server is being impersonated by the attacker, now the attacker gets the new UnlockToken. (They also get a new APNS Token and PushMagic string, but those are useless without the right APNS certificate).

Send Clear Passcode

- Wait for another chance at device
- DFU boot, force MDM poll
- Evil server sends ClearPasscode
- Now device fully unlocked
  - Browse everything
  - Better yet, backup entire device and run

Now the attacker has the unlock token for the device. They need only to wait until the device is left unattended again. Then they get back into the device (through the DFU tethering magic), force the MDM server poll, and send the ClearPasscode command with the Unlock Token. At this point, the device is unlocked, and all protected data (email, account passwords, etc.) are available to the attacker.

Covering Your Tracks

- But now there's no passcode!
  - The victim will suspect
- Replay TokenUpdate to real MDM
  - Now the corporate server can talk to it again
  - Most importantly, it can unlock the device again
- Set a bogus passcode
  - User returns to device, can't unlock it
  - Eventually calls for help
  - They unlock it remotely
  - Everyone forgets this ever happened

Looking ahead at what happens next, one might realize "But wait! There's no passcode left on the device. Won't the victim know something's wrong?" So you set a bogus password, and replay the TokenUpdate command to the device's legitimate, real MDM server. Now the MDM server has the current push and unlock tokens, and when the user can't get in, they simply call the help desk and get it unlocked remotely. Chances are pretty good nobody will ever remember this even happened....

Scenario

- Important CEO target at conference
- Left iPad in hotel room
  - Break in, DFU device, retrieve MDM profile
  - Configure MITM server to install profile
  - Hide server in room (under bed, in HVAC, etc.)
- CEO returns, checks email
  - When device unlocked, we get the tokens!
- CEO goes to hotel bar
  - Re-enter, DFU device, force poll, clear passcode
  - Dupe device, cover tracks, leave

Isn't this a bit crazy?

- Yes. Yes, it is.
- For a high-value target, not unheard of.
- Levels of difficulty:
  - Access via DFU: *Pretty difficult*
  - Putting server in middle: *Moderate*
  - Executing the commands: *Easy*
  - Duplicating the unlocked device: *Trivial*
  - General logistics: *Moderate to Difficult*

  Crazier than "Normal" Evil Maid?
  Yes. But maybe not by much.

This is not an easy attack to pull off. It requires a fair degree of technical sophistication, especially around the DFU-based access to the filesystem. That's not to mention the logistical difficulties of gaining access to the room (and making sure the device is even in there to begin with!).

Demo:
Evil Maid

Evil Lackey

- Change:
  - "Hotel" to "Office"
  - "Maid" to "Passed-over Deputy to the Deputy"
  - "Hotel Bar" to "Office Gym"
- Attacker enters target's office
- Places stack of papers on desk
- Leaves with iPad hidden in stack
- ....etc...

Fixes?

- First, protect MDM configuration
  - FileProtectionComplete for relevant profiles
- Better use of TLS
  - Remember certificates (connections, commands)
  - Refuse connection if web certs don't match
  - Refuse commands if signatures don't match
    - (or if there's no signature at all)
- Require user acknowledgment for re-enroll
  - So tokens don't get sent silently
- Perhaps additional auth for EraseDevice

Fortunately, I think that fixes to this (and other MDM issues) can be pretty easily accomplished by Apple.

First, protecting the MDM installation profile (if it's even necessary to retain on the device at all) would make acquiring the required profile more difficult. Better use of TLS-based authentication, both at the client to server level, and at the command level, would also raise the bar for MITM attacks. Finally, whenever the device enrolls in MDM (and thus sends an Unlock Token to a 3rd party), the user should be directly notified, especially if the device already appears to be enrolled.

It'd probably be nice to add some kind of authentication to the EraseDevice call...perhaps requiring the UnlockToken. Though there may be valid reasons why the current method was chosen.

So... "How secure is it?"

- Actually, not too bad.
- Some limitations and some holes
  - No serious conceptual flaws
  - Issues should be reasonably easy to fix
- Protocol pretty straightforward
  - Now openly documented
- Hope this serves as catalyst for research

Bibliography / Links

INTREPIDUS GROUP
MOBILE SECURITY

- Apple docs:
  - Troubleshooting Push Notifications
  - Local and Push Notification Programming Guide
  - Over-the-Air Profile Delivery and Configuration
- APNS library for Python (google code)
- White paper on CD & conference site

- White paper, slides, and current code:

  www.intrepidusgroup.com/insight

Feedback!

INTREPIDUS GROUP
MOBILE SECURITY

Please complete the
Speaker Feedback Surveys

This will help speakers to
improve, and can help Black Hat
make better decisions regarding
content and presenters for
future events.