2011

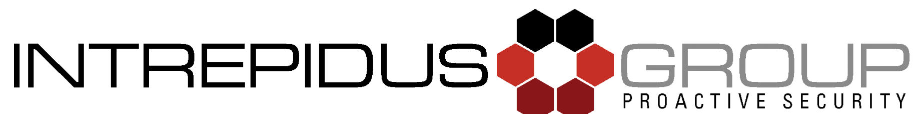
Intrepidus Group, Inc.

By David Schuetz
Senior Consultant

THE IOS MDM PROTOCOL

Abstract: Mobile Device Management (MDM) has become a hot topic as organizations are pressured to bring iStuff into their organization. Mobile devices are invading every level of corporate society, making the need to remotely manage and control them increasingly urgent. Apple has provided some enterprise management features, first via over-the-air configuration profiles, and beginning in 2010, full MDM support. Unfortunately, the exact features available through MDM, as well as details of the protocol itself, have not been publicly released by Apple.

This paper describes how Apple's MDM system works. It details the method by which an MDM server initiates a connection to a managed device, how the device enrolls with the server, and the various commands available to the system. Full parameters are provided for each command, as well as details for specialized responses from the device. Finally, source code is provided for a very simple MDM server, that will permit basic experimentation with the MDM protocol using actual iOS devices.



Introduction

The use of iOS devices, such as the iPhone, iPad or iPod, can present a serious risk to an organization. These devices are powerful computers with high storage capacity and can potentially exfiltrate data beyond corporate control and release it, deliberately or accidentally, to unauthorized third parties. Many protections available for iOS focus on the device itself, such as the use of passcodes to prevent a 3rd party from accessing the data on the device. However, managing such "iStuff" presents a serious challenge to the large enterprise, and has historically been a complicated and cumbersome process.

The easiest way to configure a device is through the iOS Settings application. Of course, this requires direct physical access to the device, which becomes increasingly impractical as the installed base of iOS devices grows.

To ease this, Apple created the iPhone Configuration Utility (IPCU), which directly installs custom .mobileconfig files over USB. An extension to this capability allows Over the Air (OTA) configuration -- letting the end user click on a link to fetch and install new profiles.

But even the OTA configuration method has its drawbacks, not least the requirement of end-user interaction. Then, in 2010, Apple introduced Mobile Device Management (MDM) services for iOS, a solution to the problem of iOS MDM, targeted at the enterprise. This system features remote installation of profiles, querying of device settings, and certain remote controls: lock, unlock, and remote wipe of a device.

Unfortunately, documentation of the underlying protocol has never been freely available. Obviously, third parties selling MDM servers were provided access to the documentation by Apple, but it's not been available for researchers or smaller development shops. This hampers risk analysis for enterprises making use of MDM. In order to aid such risk assessments, and to enable and encourage future research, this project was born.

The goal is not to create a simple, turn-key, stand-alone MDM server, nor to probe the protocol for weaknesses or hidden features, but simply to document as much of the protocol as possible. It is hoped that future researchers may build upon this documentation to better understand the security of MDM, and in particular, of various implementations of MDM servers and clients.

Disclaimer

This work was performed without use of any Apple confidential material subject to Non-Disclosure Agreements. At the time of writing, OS X 10.7 (Lion) and iOS 5 were both available in Developer Preview, but neither system was utilized in the deciphering of the MDM protocol. Lion's contribution to this research was limited to the acquisition of an Apple Push Notification Service (APNS) certificate, while iOS 5 hasn't even been downloaded, let alone installed on any devices. All work was performed on OS X 10.6 (Snow Leopard), and tested with multiple iOS devices (running a mix iOS 4.2.1, 4.3, and 4.3.1).

Background

Basic user configuration changes are made in the Settings application. Many of these settings are stored as Property List (`.plist`) files on the device, in `/var/mobile/Library/ConfigurationProfiles`, along with profiles installed by IPCU or MDM. For example, the file `UserSettings.plist` may contain the following:

```
<plist version="1.0">
<dict>
      <key>assignedObject</key>
      <dict/>
      <key>restrictedBool</key>
      <dict>
            <key>allowAccountModification</key>
            <dict>
                  <key>value</key>
                  <true/>
            </dict>
            <key>allowAddingGameCenterFriends</key>
            <dict>
                  <key>value</key>
                  <true/>
            </dict>
            <key>allowAppInstallation</key>
            <dict>
                  <key>value</key>
                  <true/>
            </dict>
            <key>allowAppRemoval</key>
            <dict>
                  <key>value</key>
                  <true/>
            </dict>

[...remainder not shown...]
```

In this case, we can see (for example) that installing applications is permitted by the local user. If the user were to enter Settings, and navigate to General -> Restrictions to turn off the App Store, then the value for “`allowAppInstallation`” would be changed to “false.”

These configuration files are not normally visible to the end user, but can be accessed on a jailbroken device. Understanding how these are formatted and interpreted by the operating system is useful to decoding how MDM works as a whole.

If the user, or their IT Admin Staff, installs a new profile via IPCU that restricts certain items, that profile is also stored in this same folder, but in a different `.plist` file. An excerpt from a profile might include the following:

```
[.....]

<plist version="1.0">
<dict>
      <key>InstallDate</key>
      <date>2011-07-05T17:49:37Z</date>
      <key>MCProfileIsRemovalStub</key>
      <true/>
      <key>PayloadContent</key>
      <array>
            <dict>
                  <key>PayloadDescription</key>
                  <string>Configures device restrictions.</string>
                  <key>PayloadDisplayName</key>
                  <string>Restrictions</string>
[.....]
                  <key>allowAddingGameCenterFriends</key>
                  <true/>
                  <key>allowAppInstallation</key>
                  <false/>

[...remainder not shown...]
```

So now, the device has two profiles -- the locally-created one (`UserSettings.plist`) which permits the App Store, and one (installed by IPCU) which disables it. In such a situation, it's been observed that the more restrictive setting "wins": in this case, the App Store would be disabled. The merged settings are cached in `ProfileTruth.plist`, and additional files in the `PublicInfo` subdirectory.

MDM Protocol Overview

Thus far, all the configuration profiles discussed were installed manually -- either through the device's Settings application or via locally-initiated USB or wireless download of a profile. A key feature of MDM is that it allows administrators to push profiles to the device without any manual intervention.

Basics

The MDM service essentially consists of three elements:

1. The device being managed (iPhones, iPads, iPod Touches)
2. The server doing the management (various MDM servers)
3. A method by which the server wakes up the device (APNS)

Enrollment

MDM servers support over-the-air (OTA) enrollment, as documented in Apple's "Over-the-Air Profile Delivery and Configuration" developer guide[1]. The OTA enrollment can implement various challenge and response systems, both to authenticate the user and to ensure that only desired devices are enrolled in the system. However, the OTA system relies on a Simple Certificate Enrollment Protocol (SCEP) server, and is beyond the scope of this research.

Enrollment is also possible through a profile created by the iPhone Configuration Utility. The user specifies the network address of the MDM server, provides some additional information, and installs the profile directly on the device to be managed.

During enrollment, the device provides unique identifying information to that server, which is used by the server to send messages through the Apple Push Notification Service. Long term connections from server to client, or client to server, do not exist with the design of MDM -- only the connection to APNS. This long-term APNS connection is part of the Push Notifications framework supporting multiple iOS applications, not just MDM.

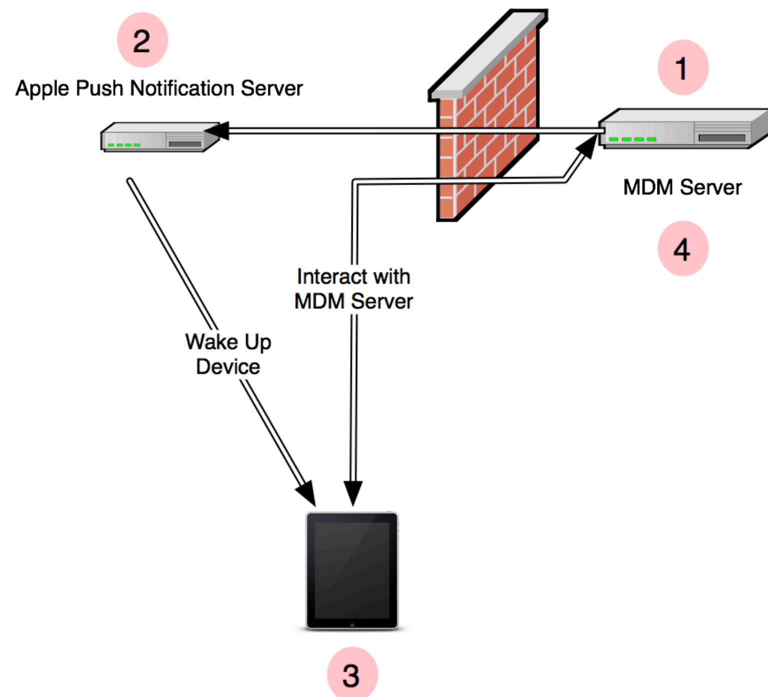
After enrollment, each interaction between client devices and the MDM server consists of four elements [see also diagram on the next page]:

1. Server requests push notification through Apple
2. Apple pushes notification to device
3. Device connects to server
4. Server and client exchange commands and responses

[1] See Appendix C for referenced documents.

Push Notification

When the MDM server needs to communicate with a device, it queues up the desired command, then sends a very simple push notification message through APNS. No information other than an identifying token is present in this message. Upon receipt of the notification, the device contacts the server, which then provides the queued command to the client. Upon completion of the command, the client responds with an appropriate acknowledgment, and the connection between client and server is closed.



Client / Server Interaction

The device interacts with the server by connecting to a designated URL and exchanging XML formatted data, in the form of an Apple Property List (`.plist`) file. The first message a client sends upon connecting to the server is a simple "`Status: Idle`" notification, indicating that the device is ready to receive commands from the MDM server.

Upon receipt of this message, the server sends whatever command may be waiting for that device. This command is also presented as an XML formatted `.plist` file, and in most cases is only a simple command (with perhaps some associated arguments).

The device acts upon the command, and may then respond with another `.plist`, providing either a simple acknowledgement of the command, an error message, or a detailed response (in the case of device inventories and similar commands).

Creating a Simple MDM Server

A very basic MDM server can be created in fewer than five pages of Python, using only basic libraries. This section will describe such a server in detail, and full source code will be provided in Appendix B.

Requirements

APNS: The most important requirement is acquisition of an APNS push certificate. In the past, such a certificate was only available to members of the Apple Enterprise Developer program (with annual fees of $300). However, with the delivery of Lion, basic MDM services are bundled with Lion Server, available for only $49.99.

Python: This demonstration server was built on OS X 10.6 (Snow Leopard), using Python version 2.6. Libraries used include `web.py`, `pprint`, `plistlib`, and `APNSWrapper`. Some of these are standard libraries, others are readily available on the net. The `APNSWrapper` library itself depends on the commonly-installed OpenSSL package.

Network Connectivity: To send push notifications, the server needs to communicate with Apple's APNS server: This requires outbound TCP connectivity to `gateway.push.apple.com` on port 2195. The devices being managed likewise need connectivity to APNS, via outbound TCP port 5223. Finally, the device needs to be able to contact the MDM server itself on whatever port is defined in the MDM enrollment profile.

Caveats and Disclaimers

Again, the intended use of this tool is not to replace actual MDM server products. It is intended to allow for further research into the MDM system, and potential vulnerabilities, in order to better understand the risks it may present, or be used to mitigate, in an enterprise. It does not make use of many features which are arguably required for a production service. Some limitations:

- No transport layer security: all transactions occur over HTTP, though MDM and the utilized `web.py` framework both support SSL
- No authentication: though a client identity certificate is created, it is not used, nor are any of the server commands signed
- Only manages one device at a time
- Enrollment is accomplished through IPCU-installed profiles, however, because there's no authentication, it's trivial to recreate this profile
- Any new device can therefore enroll with the server and "kick out" the device currently being tested

In other words: This tool is for research purposes only. Don't even think about exposing it to the open Internet, or using it to actively manage devices with important data!

Setting up APNS

First, an Apple Push certificate must be acquired. This is what allows the MDM server to communicate with the client, and without it (and the appropriate certificate, provided by Apple) the MDM service will not work. At this time, the easiest way to get an APNS certificate is to purchase OS X Lion Server, though obviously this requires Mac hardware. Within the Server application, the administrator can request and install push notification certificates directly from Apple. Once installed, the certificate can easily be extracted from the keychain.

Open the Keychain Access program, and locate the APNS certificate. In my case, it was named "`APSP:<uuid>`", where uuid looks like a long random string of hex digits (and, in fact, is essentially just that). Highlight the certificate, and export it to a `.p12` file (for example,

PushCert.p12). The program will prompt for a password to protect the private key.

Next, open a Terminal window, navigate to the folder where the certificate was saved, and convert the file to .pem format: For example, using openssl pkcs12 -in PushCert.p12 -out PushCert.pem. The password used when exporting the key from Keychain will be needed to open the .p12 file, and a new password will be required to protect the .pem file.

Since this .pem is encrypted, the tool will prompt for the passphrase each time you attempt to send a push notification. To avoid this, first create a copy of the key with no passphrase: openssl rsa -in PushCert.pem -out PlainKey.pem. Then in a text editor, replace the encrypted RSA Private Key section in PushCert.pem with the contents of PlainKey.pem. Once this is complete, the script will be able to access the certificate and key to send notifications without using a passphrase. Of course, be sure that this file is well protected from disclosure, or others will be able to forge your push notifications. (It's probably also a good idea to acquire a push certificate strictly for testing purposes, and not to use any production credentials with this tool).

Mobile Device Management

Server URL
The URL of the Mobile Device Management server
http://192.168.1.1/server

Check In URL
The URL the device will use for check in during installation
http://192.168.1.1/checkin

Topic
Push notification Topic for management messages
com.example.mdm.pushcert

Identity
Cryptographic credential used for authentication
mdm-ipad.p12

Sign messages

Access Rights
Access rights granted to remote administrators

Query device for
General settings
Security settings
Network settings
Restrictions
Configuration Profiles
Applications
Provisioning Profiles

Add / Remove
Configuration Profiles
Provisioning Profiles

Security
Change device password
Remote wipe

MDM Enrollment Profile

The next step is to create a profile in the iPhone Configuration Utility[2] (IPCU) that will instruct the device to connect to the MDM server. Within IPCU, create a new profile, and select the MDM payload. In the Server field, enter the URL for the server (for example, http://192.168.1.1/server. This will be the main URL that devices use to poll the server for MDM commands. Optionally, a different URL can be entered in the Check In field (for initial connection to the MDM server). The example server should be able to support Check In over the server URL, but in testing this was separated to http://192.168.1.1/checkin.

[2] Currently available at http://www.apple.com/support/iphone/enterprise/

The Topic field needs to contain the User ID listed in the Subject Name section of the APNS push certificate. Finally, an identity certificate for the device needs to be generated, installed via the IPCU Certificates payload, and selected here. This can be a simple certificate generated using the Keychain Access "Certificate Assistant," but keep in mind that it may need to be signed by a trusted entity. If not, then the Certificate Authority (CA) used to sign the identity may need to be installed on the device in order for the device to recognize it. (Though, as mentioned before, this certificate isn't even used by the test server).

Once all is finished, save the configuration, and it's ready to be copied via USB (or over-the-air methods) to the iOS device being used for the server testing.

The Enrollment Exchange

As the MDM profile is being installed, the device will attempt to connect to the MDM server. It first sends a basic "`Authenticate`" request to the Check In URL, providing the device's Universal Device Identifier (UDID) and the Push Notification topic.

This exchange provides the server with an opportunity to accept or deny the enrollment request, based on the Topic and (more likely) the UDID.

```
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>
```

To proceed, the server can respond with 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>
```

After receiving this response, the client sends the server a "`TokenUpdate`" request, containing three key pieces of identifying information:

- `PushMagic` - a unique token the MDM server sends with each push request
- `Token` - a unique token that identifies the device to the APNS service, and

- `UnlockToken` - an escrow key used to clear the passcode on the device.

All three tokens have been redacted in the following listing. The `PushMagic` token is a hexadecimal string, likely a simple random UUID (in UUID format). The `Token` is a 32-byte binary value, presented as Base64 encoded text, and `UnlockToken` is a much longer binary string (closer to 2 kilobytes), also encoded in Base64.

```
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>TokenUpdate</string>
    <key>PushMagic</key>
    <string> [ redacted uuid string ] </string>
    <key>Token</key>
    <data> [ 32 byte string, base64 encoded, redacted ] </data>
    </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, redacted ]
    </data>
</dict>
</plist>
```

Again, the server doesn't need to provide any response beyond a simple blank plist. Once this is complete, the server retains copies of the tokens, and the device is now fully enrolled.

Sending Push Notifications

When the MDM server needs to contact a device, it sends the device a notification via APNS. This uses a special format of the push notification, where the top-level "`aps {}`" list is replaced by a single "`mdm`" value. Unfortunately, the default APNSWrapper library doesn't support this alternate form, and produces messages with both keys. Fortunately, current versions of iOS (at least from 4.2.1 onwards) don't seem to care, though they do log an error on the console.

The push notification needs both the device's APNS `Token`, and the mdm `PushMagic` token. Sending the notification, using `APNSWrapper`, is straightforward:

```
    wrapper = APNSNotificationWrapper('PushCert.pem', False)
    message = APNSNotification()
    message.token(my_DeviceToken)
    message.appendProperty(APNSProperty('mdm', my_PushMagic))
    wrapper.append(message)
    wrapper.notify()
```

The resultant APNS notification's payload will look approximately like this:


```
{"aps":{},"mdm":"996ac527-9993-4a0a-8528-60b2b3c2f52b"}
```


Once sent to the APNS server, the notification should be quickly relayed to the device (provided it's currently connected to APNS), and the device will then contact the MDM server for further instructions.

Device Response to Push

After receiving the push notification, the client contacts the server to receive instructions. From here forward, it uses the "Server" URL (instead of the "Check In" URL used during enrollment), but everything remains sent over `HTTP PUT`. First, the device sends a simple "`Status: Idle`" message:


```
PUT: /server
<?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>Status</key>
    <string>Idle</string>
    <key>UDID</key>
    <string> [ redacted ] </string>
</dict>
</plist>
```


The server then responds with whatever command has been queued up for the device. In all cases, the command response includes a "`CommandUUID`" field that should include a unique random UUID. This allows the server to match commands sent with responses received from the client, but is not necessary (and may currently be left blank).

Most commands are simply the command name, but a few require additional parameters. Details are provided in the command list in Appendix B. For a quick example, we'll look at two commands: `DeviceLock` and `ClearPasscode`.

Device Lock

The DeviceLock command requires no parameters, and is simply the following:

```
<?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>
      <key>Command</key>
      <dict>
            <key>RequestType</key>
            <string>DeviceLock</string>
      </dict>
      <key>CommandUUID</key>
      <string></string>
</dict>
</plist>
```

When this is provided to the device, as the response to the Status: Idle command, the device immediately locks. The response from the device is the standard acknowledgement message:

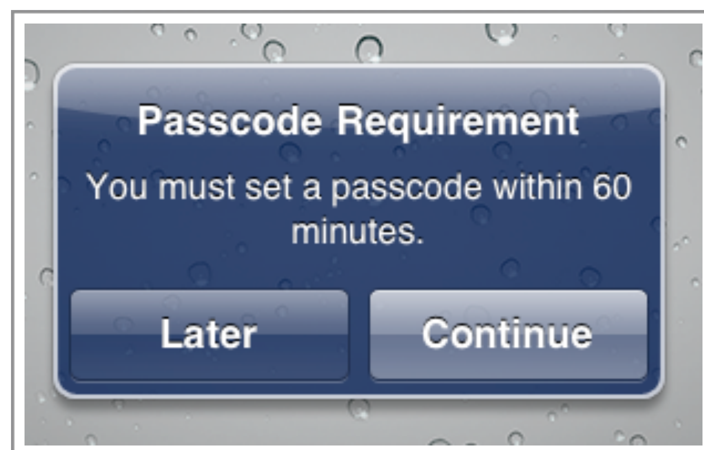
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
      <key>CommandUUID</key>
      <string></string>
      <key>Status</key>
      <string>Acknowledged</string>
      <key>UDID</key>
      <string> [ redacted ] </string>
</dict>
</plist>
```

Clear Passcode

The `ClearPasscode` command requires the device's `UnlockToken` (which was provided to the server during the enrollment phase, in the `UpdateToken` message):

```
<?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>
	<key>Command</key>
	<dict>
		<key>RequestType</key>
		<string>ClearPasscode</string>
		<key>UnlockToken</key>
		<data>
	[ redacted ]
		</data>
	</dict>
	<key>CommandUUID</key>
	<string></string>
</dict>
</plist>
```

If the `UnlockToken` matches what was created during enrollment, then the device should clear the passcode. It will remain locked, but no passcode will be required to unlock. If a policy exists that requires a passcode, the user will be given a grace period to enter a new passcode.



This warning will also appear if a new policy is installed that changes the passcode requirements.

Using the Example Program

To use the example MDM test program, first follow the steps listed above in "**Creating a Simple MDM Server**". The default port will be 8080, so be sure to use that in the manual MDM enrollment profile created in IPCU (or adjust accordingly).

In order to test installation and removal of profiles, create a basic Configuration Profile or Provisioning Profile in IPCU and save them as "`test.mobileconfig`" and "`test.mobileprovision`" in the same directory as the server. Once created, the UUID for the Provisioning Profile will need to be updated within the source, or demonstrating removal of the test Provisioning Profile will not be possible.

Once running, simply open a browser to the server's port (such as `http://localhost:8080`). A very simple interface should appear:

First, you'll need to enroll the test device with the server. Connect the device via USB and use IPCU to install the manual MDM configuration profile. The device should immediately connect to the server, issue the `Authenticate` and `TokenUpdate` commands, and complete installation. Nothing will be displayed in the browser, but the terminal in which the server process runs should show logs of the traffic reaching the server.

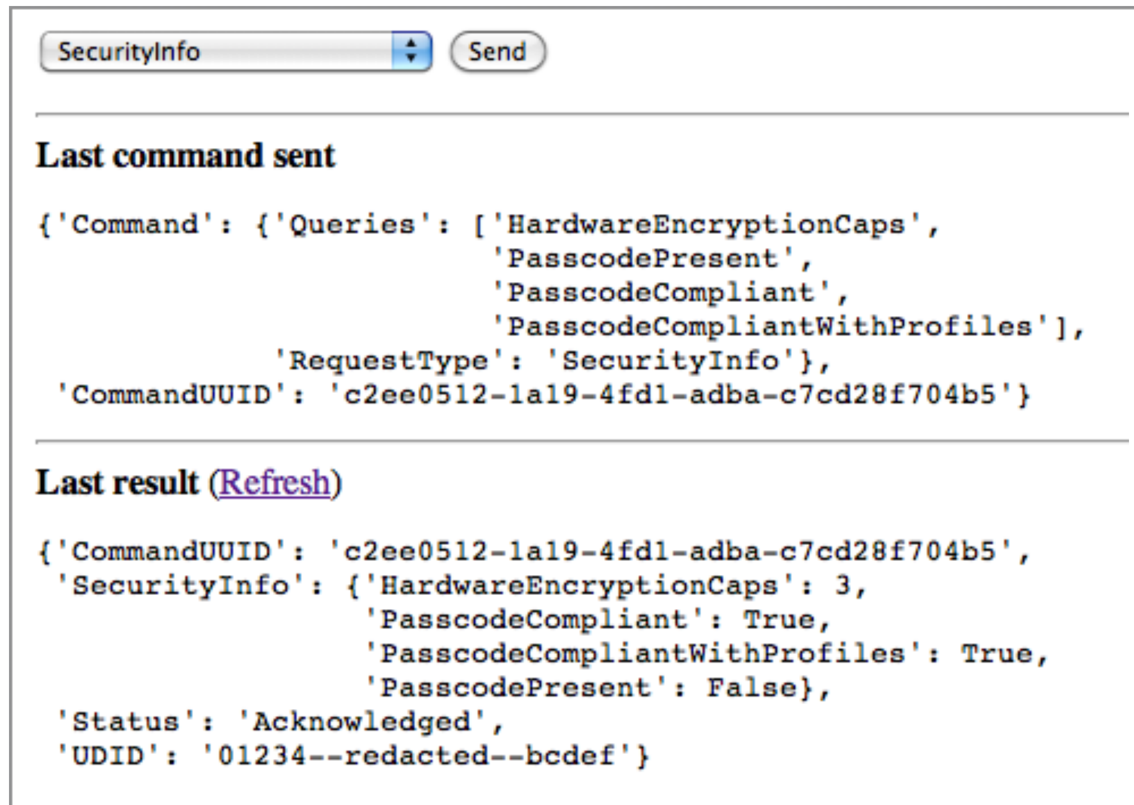
Once the device has been enrolled, the USB connection can be removed -- all further communication will occur over the air.

Select a command to send from the drop down. Click "Send," and the command data will be displayed in the window as a JSON-formatted string. At this time, the server will send the push notification to the device through Apple's APNS system.

Within a few seconds, the device should connect to the server. The first thing it will do is submit a "`Status: Idle`" message, which tells the server the device is ready to accept a command. The server will reply with the command selected, and within a few seconds, the device should provide the response to the command via another `HTTP PUT`.

Once the response has been received, click "Refresh" in the lower half of the browser window to display the result, which again has been formatted in JSON for easy interpretation.

Shown below is the command, and response, for a SecurityInfo command.



Conclusion

Overall, Apple's MDM protocol is reasonably straightforward, and not difficult to implement. Appendix A outlines as much of the protocol as has been deciphered to date, while Appendix B provides the source code to the MDM test server described in this paper.

Though this documentation is reasonably complete, there are some known shortcomings:

- An exhaustive documentation of error conditions and responses, both from the server and from the client, has not been undertaken
- Additional controls may be present in the protocol, but exist as optional parameters, and so have not been discovered (nor documented)
- Setup interactions with the APNS server, including the derivation of the Device Token, have not been investigated
- A detailed investigation of SSL-secured MDM interactions, including the potential for signed commands and responses, has not yet commenced

Despite these shortcomings, it is hoped that this effort may inspire further research into the security and reliability of iOS Mobile Device Management, leading to a better understanding of the risks iOS devices may present to large enterprises, and, ultimately, to more secure systems and networks.

Appendix A - Command Listing

Control Commands	• Device Lock • Erase Device • Clear Passcode
Device Queries	• Security Information • Installed Application List • Device Information • Certificate List • Profile List • Provisioning Profile List • Restrictions List
Device Configuration	• Install Profile • Remove Profile • Install Provisioning Profile • Remove Provisioning Profile
Device to Server Commands	• Authenticate • Token Update

Overall Format

All commands are sent as Apple Property List (`.plist`) files. Each includes a top-level key called "`CommandUUID`", containing a UUID string to uniquely identify the command instance, and a top-level key "`Command`", which is a dict containing additional information.

```
<plist version="1.0">
<dict>
      <key>Command</key>
      <dict>
            <key>RequestType</key>
            <string>[command name]</string>
            [... additional parameters as needed ...]
      </dict>
      <key>CommandUUID</key>
      <string></string>
</dict>
</plist>
```

Each command is listed below with a short description, and the required parameters.

Responses

The device responds to many commands with a simple acknowledgment:

```
<plist version="1.0">
<dict>
    <key>CommandUUID</key>
    <string></string>
    <key>Status</key>
    <string>Acknowledged</string>
    <key>UDID</key>
    <string>[device UUID]</string>
</dict>
</plist>
```

The "Status" field may contain "Acknowledged", "Error", "CommandFormatError", or "NotNow" (see below for details on the error fields).

Where commands elicit a more extended response (such as for DeviceInformation queries), details of those responses are given below. Typically, these commands add a top-level field (such as InstalledApplicationList) which has as its value the extended data, stored as a string, dict, or array of other elements. For example:

```
<plist version="1.0">
<dict>
    <key>CommandUUID</key>
    <string></string>
    <key>SecurityInfo</key>
    <dict>
        <key>HardwareEncryptionCaps</key>
        <integer>3</integer>
        <key>PasscodeCompliant</key>
        <true/>
        <key>PasscodeCompliantWithProfiles</key>
        <true/>
        <key>PasscodePresent</key>
        <false/>
    </dict>
    <key>Status</key>
    <string>Acknowledged</string>
    <key>UDID</key>
    <string>[device UUID]</string>
</dict>
</plist>
```

Error Messages

The general format for an error message is the same as the acknowledgement, with "Status" changed to "Error" and an additional array of dicts added as "ErrorChain":

ErrorCode	(integer) A unique identifying error code
ErrorDomain	(string) Category of error
LocalizedDescription	(string) Error message, translated to a localized language
USEnglishDescription	(string) Standardized version of error message

A special error message is "NotNow", which is seen when a command cannot be requested because the device is locked with a passcode (such as requesting Security Information or installing a profile). When that occurs, the device will attempt to re-connect with the MDM server as soon as the device is unlocked, in order to retry the command.

A command sent with invalid or missing parameters returns the "CommandFormatError" status.

Device Lock

Immediately locks the device. If a passcode is present, that passcode will be required to unlock the device.

RequestType	DeviceLock

Erase Device

Immediately wipes the device memory and resets it to a "clean from factory" state. Requires connection to iTunes to restore from backup or configure as new.

RequestType	EraseDevice

Clear Passcode

If a passcode is present on the device, this command will clear that passcode. If a passcode is required by other configuration controls, the user will be given a grace period in which to set a new passcode.

RequestType	ClearPasscode
UnlockToken	(data) UnlockToken data, base-64 encoded

Security Information

Lists specified security-related settings for the device, including hardware encryption capabilities, and whether a passcode is present (and if so, whether it is compliant with configuration). If the passcode is present, the device must be unlocked for this command to execute.

RequestType	SecurityInfo
Queries	(array of strings): “HardwareEncryptionCaps”, “PasscodePresent”, “PasscodeCompliant”, “PasscodeCompliantWithProfiles”

The response is based on the general acknowledgement response, with an additional dictionary named “`SecurityInfo`”:

HardwareEncryptionCaps	integer
PasscodePresent	boolean
PasscodeCompliant	boolean
PasscodeCompliantWithProfiles	boolean

Installed Application List

Lists all the applications currently installed on the device. Includes the overall persistent storage used by the application, expressed in bytes, along with the application’s name, version, and bundle identifier. Does not list applications installed via jailbreaking methods.

RequestType	InstalledApplicationList

Additional response information, in key “`InstalledApplicationList`”, is an array of dict items:

BundleSize	integer
DynamicSize	integer
Identifier	string
Name	string
Version	string

Device Information

Retrieves specified general information about the device, including MAC addresses, IMEI, phone number, software version, model name and number, serial number.

RequestType	DeviceInformation
Queries	(array of strings): "AvailableDeviceCapacity", "BluetoothMAC", "BuildVersion", "CarrierSettingsVersion", "CurrentCarrierNetwork", "CurrentMCC", "CurrentMNC", "DataRoamingEnabled", "DeviceCapacity", "DeviceName", "ICCID", "IMEI", "IsRoaming", "Model", "ModelName", "ModemFirmwareVersion", "OSVersion", "PhoneNumber", "Product", "ProductName", "SIMCarrierNetwork", "SIMMCC", "SIMMNC", "SerialNumber", "UDID", "WiFiMAC", "UDID"

The response is a dict named "`QueryResponses`" including the above-listed items as keys. Responses that would be null (for example, the `PhoneNumber` field from an iPod Touch) are simply omitted. `AvailableDeviceCapacity` and `DeviceCapacity` are real number fields, while `DataRomingEnabled` and `IsRoaming` are boolean values. All the rest are returned as strings.

Certificate list

Lists all certificates currently installed on the device.

RequestType	CertificateList

The response includes a "`CertificateList`" array of dict values:

CommonName	string
Data	base-64 cert information
IsIdentity	boolean

Profile List

Lists configuration profiles installed on the device. Includes Common name, whether a remove passcode is required, whether removal is disallowed, unique identifiers, and other similar information.

RequestType	ProfileList

The response key "`ProfileList`" contains an array of dict items:

HasRemovalPasscode	boolean
IsEncrypted	boolean
PayloadDisplayName	string
PayloadIdentifier	string
PayloadRemovalDisallowed	boolean
PayloadUUID	string
PayloadVersion	integer
SignerCertificates	array of data items, each with base-64 cert info
PayloadContent	array of dicts, each with PayloadDisplayName, PayloadIdentifier, PayloadType, and PayloadVersion keys.

Provisioning Profile List

Lists provisioning profiles installed on the device (similar to the Profile list).

RequestType	ProvisioningProfileList

The response includes a "`ProvisioningProfileList`" key, which contains an array of dict values:

ExpiryDate	date
Name	string
UUID	string

Restrictions List

Lists restrictions currently in effect on the device. For example, lists disabled applications, whether backup encryption is forced on, etc.

RequestType	RestrictionsList

The response includes "`GlobalRestrictions`", which is a dict containing detailed list of restrictions, most presented as boolean values. The exact content and structure depends on the restrictions in place on the device.

Install Profile

Given a base-64 encoding of a `.mobileconfig` profile (as created by the IPCU or other tools), installs the profile on the device.

RequestType	InstallProfile
Payload	(data) IPCU .mobileconfig file, base-64 encoded

Remove Profile

Given a payload identifier (which is typically shown as a reverse-DNS identifier such as "`com.example.cfg.restrictions`"), removes the profile from the device.

RequestType	RemoveProfile
Identifier	(string) Profile identifier

Install Provisioning Profile

Given a base-64 encoding of a `.mobileprovision` profile (as created by the IPCU or other tools), installs the profile on the device.

RequestType	InstallProvisioningProfile
Payload	(data) IPCU .mobileprovision file, base-64 encoded

Remove Provisioning Profile

This command removes the provisioning profile from the device, given the profile's UUID.

RequestType	RemoveProvisioningProfile
UUID	(string) Provisioning profile UUID

Authenticate

This is a client command, sent by the client to initiate enrollment. Can be used by the server to permit or deny enrollment based on the device's UDID. NOTE - Does not follow same format as server-to-client commands. Has no `CommandUUID` field nor the `Command` dict structure -- all parameters are top-level items in the main property list dict.

MessageType	Authenticate
Topic	(string) Subject Name: User ID on APNS push certificate used by server
UDID	(string) Device UDID

Token Update

This is a client message, sent by the client during enrollment. Provides the server with tokens used to contact device via APNS, as well as a key to unlock the device through the Clear Passcode command. NOTE - Does not follow same format as server-to-client commands. Has no `CommandUUID` field nor the `Command` dict structure -- all parameters are top-level items in the main property list dict.

MessageType	Token Update
PushMagic	(string) UUID-like string
Token	(data) 32-byte APNS device token, base-64 encoded
Topic	(string) Subject Name: User ID on APNS push certificate used by server
UDID	(string) Device UDID
UnlockToken	(data) Device unlock key, base-64 encoded

Appendix B - Source Code

server.py

```
import web, os, pprint, json, uuid, sys
from plistlib import *
from APNSWrapper import *
from creds import *

#
# Simple, basic, bare-bones example test server
# Implements Apple's Mobile Device Management (MDM) protocol
# Compatible with iOS 4.x devices
# Not yet tested with iOS 5.0
#
# David Schuetz, Senior Consultant, Intrepidus Group
#
# Copyright 2011, Intrepidus Group
# http://intrepidusgroup.com
#

###########################################################################
# Update this to match the UUID in the test provisioning profiles, in order
#   to demonstrate removal of the profile

my_test_provisioning_uuid = 'REPLACE-ME-WITH-REAL-UUIDSTRING'

###########################################################################

last_result = ''
last_sent = ''

global mdm_commands

urls = (
    '/', 'root',
    '/queue', 'queue_cmd',
    '/checkin', 'do_mdm',
    '/server', 'do_mdm',
)


def setup_commands():
    global my_test_provisioning_uuid

    ret_list = dict()

    for cmd in ['DeviceLock', 'ProfileList', 'Restrictions',
        'CertificateList', 'InstalledApplicationList',
        'ProvisioningProfileList']:

        ret_list[cmd] = dict( Command = dict( RequestType = cmd ))

    ret_list['SecurityInfo'] = dict(
        Command = dict(
            RequestType = 'SecurityInfo',
            Queries = [
```

```
                'HardwareEncryptionCaps', 'PasscodePresent',
                'PasscodeCompliant', 'PasscodeCompliantWithProfiles',
            ]
        )
    )

    ret_list['DeviceInformation'] = dict(
        Command = dict(
            RequestType = 'DeviceInformation',
            Queries = [
                'AvailableDeviceCapacity', 'BluetoothMAC', 'BuildVersion',
                'CarrierSettingsVersion', 'CurrentCarrierNetwork',
                'CurrentMCC', 'CurrentMNC', 'DataRoamingEnabled',
                'DeviceCapacity', 'DeviceName', 'ICCID', 'IMEI', 'IsRoaming',
                'Model', 'ModelName', 'ModemFirmwareVersion', 'OSVersion',
                'PhoneNumber', 'Product', 'ProductName', 'SIMCarrierNetwork',
                'SIMMCC', 'SIMMNC', 'SerialNumber', 'UDID', 'WiFiMAC', 'UDID'
            ]
        )
    )

    ret_list['ClearPasscode'] = dict(
        Command = dict(
            RequestType = 'ClearPasscode',
            UnlockToken = Data(my_UnlockToken)
        )
    )

# commented out, and command string changed, to avoid accidentally
# erasing test devices.
#
#    ret_list['DONT_EraseDevice'] = dict(
#        Command = dict(
#            RequestType = 'DONT_EraseDevice',
#        )
#    )

    if 'test.mobileconfig' in os.listdir('.'):
        my_test_cfg_profile = open('test.mobileconfig', 'rb').read()
        pl = readPlistFromString(my_test_cfg_profile)
        ret_list['RemoveProfile'] = dict(
            Command = dict(
                RequestType = 'RemoveProfile',
                Identifier = pl['PayloadIdentifier']
            )
        )
    else:
        print "Can't find test.mobileconfig in current directory."
        sys.exit()

    ret_list['InstallProfile'] = dict(
        Command = dict(
            RequestType = 'InstallProfile',
            Payload = Data(my_test_cfg_profile)
        )
    )
```

```
    if 'test.mobileprovision' in os.listdir('.'):
        my_test_prov_profile = open('test.mobileprovision', 'rb').read()
    else:
        print "Can't find test.mobileprovision in current directory."
        sys.exit()

    ret_list['InstallProvisioningProfile'] = dict(
        Command = dict(
            RequestType = 'InstallProvisioningProfile',
            ProvisioningProfile = Data(my_test_prov_profile)
        )
    )

    ret_list['RemoveProvisioningProfile'] = dict(
        Command = dict(
            RequestType = 'RemoveProvisioningProfile',
    # need an ASN.1 parser to snarf the UUID out of the signed profile
            UUID = my_test_provisioning_uuid
        )
    )

    return ret_list

class root:
    def GET(self):
        return home_page()

class queue_cmd:
    def GET(self):
        global current_command, last_sent
        global my_DeviceToken, my_PushMagic
        i = web.input()
        cmd = i.command

        cmd_data = mdm_commands[cmd]
        cmd_data['CommandUUID'] = str(uuid.uuid4())
        current_command = cmd_data
        last_sent = pprint.pformat(current_command)

        wrapper = APNSNotificationWrapper('PlainCert.pem', False)
        message = APNSNotification()
        message.token(my_DeviceToken)
        message.appendProperty(APNSProperty('mdm', my_PushMagic))
        wrapper.append(message)
        wrapper.notify()

        return home_page()


class do_mdm:
    global last_result

    def PUT(self):
        global current_command, last_result
        i = web.data()
        pl = readPlistFromString(i)

        if pl.get('Status') == 'Idle':
```

```
            rd = current_command

        elif pl.get('MessageType') == 'TokenUpdate':
            rd = do_TokenUpdate(pl)

        elif pl.get('Status') == 'Acknowledged':
            rd = dict()

        else:
            rd = dict()

        out = writePlistToString(rd)

        if pl.get('UDID'):
            pl['UDID'] = '--redacted--'
        q = pl.get('QueryResponses')
        if q:
            redact_list = ('UDID', 'BluetoothMAC', 'SerialNumber', 'WiFiMAC',
                'IMEI', 'ICCID', 'SerialNumber')
            for resp in redact_list:
                if q.get(resp):
                    pl['QueryResponses'][resp] = '--redacted--'

        last_result = pprint.pformat(pl)
        return out


def home_page():
    global mdm_commands, last_result, last_sent, current_command

    drop_list = ''
    for key in mdm_commands:
        if current_command['Command']['RequestType'] == key:
            selected = 'selected'
        else:
            selected = ''
        drop_list += '<option value="%s" %s>%s</option>\n'%(key,selected,key)

    out = """
<html><head><title>MDM Test Console</title></head><body>
<form method="GET" action="/queue">
  <select name="command">
  <option value=''>Select command</option>
%s
  </select>
  <input type=submit value="Send"/>
</form>
<hr/>
<b>Last command sent</b>
<pre>%s</pre>
<hr/>
<b>Last result</b> (<a href="/">Refresh</a>)
<pre>%s</pre>
</body></html>
""" % (drop_list, last_sent, last_result)

    return out
```

```
def do_TokenUpdate(pl):
    global my_PushMagic, my_DeviceToken, my_UnlockToken, mdm_commands

    my_PushMagic = pl['PushMagic']
    my_DeviceToken = pl['Token'].data
    my_UnlockToken = pl['UnlockToken'].data

    mdm_commands['ClearPasscode'] = dict(
        Command = dict(
            RequestType = 'ClearPasscode',
            UnlockToken = Data(my_UnlockToken)
        )
    )

    out = """
# these will be filled in by the server when a device enrolls

my_PushMagic = '%s'
my_DeviceToken = %s
my_UnlockToken = %s
""" % (my_PushMagic, repr(my_DeviceToken), repr(my_UnlockToken))

    print out

    fd = open('creds.py', 'w')
    fd.write(out)
    fd.close()

    print "Device enrolled!\n"

    return dict()


mdm_commands = setup_commands()
current_command = mdm_commands['DeviceLock']


if __name__ == "__main__":
    app = web.application(urls, globals())
    app.run()
```

creds.py

```
# these will be filled in by the server when a device enrolls

my_PushMagic = ''
my_DeviceToken = ''
my_UnlockToken = ''
```

Appendix C - References

iPhone in Business: Over-the-Air Enrollment and Configuration: http://images.apple.com/iphone/business/docs/iPhone_OTA_Enrollment_Configuration.pdf

Over-the-Air Profile Delivery and Configuration: http://developer.apple.com/library/ios/documentation/NetworkingInternet/Conceptual/iPhoneOTAConfiguration/iPhoneOTAConfiguration.pdf

Local and Push Notification Programming Guide: http://developer.apple.com/library/ios/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/RemoteNotificationsPG.pdf

Troubleshooting Push Notifications: http://developer.apple.com/library/ios/#technotes/tn2265/_index.html

APNS library for Python: http://code.google.com/p/apns-python-wrapper/

Apple iOS Enterprise Support, including the iPhone Configuration Utility for OS X and Windows: http://www.apple.com/support/iphone/enterprise/